

Ingegneria del Software



Ingegneria del Software

Capitolo 1 – Introduzione

1.1 Software: Prodotto e Processo

Con l'avvento di computer con hardware più sofisticati, si aprirono le porte alla progettazione di software estremamente più grandi. I primi insuccessi in questo nuovo campo fecero però capire che erano necessari metodi e tecniche nuove per la gestione e la produzione di essi. Oggi sono queste tecniche perfezionate inglobate nell'**Ingegneria del Software**.

L'**Ingegneria del Software** è una disciplina ingegneristica che si concentra nei costi di sviluppo dei sistemi software di grandi dimensioni, sviluppati tramite lavoro di gruppo. È possibile che vengono rilasciate più versioni del prodotto software. Tale attività ha senso per progetti di grosse dimensioni e di notevole complessità dove è necessaria la pianificazione.

L'ingegneria del software si occupa principalmente di alcuni aspetti fondamentali: i **metodi**, le **metodologie**, i **processi** e gli **strumenti** per la gestione professionale del software.

❖ Principi

I principi riguardano:

- il **rigore** e la **formalità**;
- l'affrontare separatamente le varie problematiche dell'attività;
- la **modularità**: cioè suddividere un sistema complesso in parti più semplici come la tecnica del *dividi et impera*. Inoltre, ogni modulo deve essere altamente coeso e tra i moduli deve esserci un basso accoppiamento per garantire l'indipendenza tra essi.
- **Astrazione**: si identificano gli aspetti cruciali in un certo istante ignorando gli altri. (*Information Hiding*)
- **Anticipazione del cambiamento**: la progettazione deve favorire l'evoluzione del software.
- **Generalità**: tentare di risolvere il problema nel suo caso generale.
- **Incrementalità**: lavorare a fasi di sviluppo, ognuna delle quali viene terminata con il rilascio di una release, anche se piccola.

❖ Metodi e metodologie

Un **metodo** è una particolare procedimento per risolvere problemi specifici, mentre la **metodologia** è un insieme di principi e metodi che serve per garantire la correttezza e l'efficacia della soluzione al problema.

❖ Strumenti, procedure e paradigmi

Uno **strumento** è un artefatto che viene usato per fare qualcosa in modo migliore.

Una **procedura** è una combinazione di strumenti e metodi finalizzati alla realizzazione di un prodotto.

Un **paradigma** è un particolare approccio o filosofia per fare qualcosa.

❖ Processo

Un **processo** è un particolare metodo per fare qualcosa costituito da una sequenza di passi che coinvolgono attività, vincoli e risorse.

Un **processo software** è l'insieme di attività che costituiscono la costruzione del prodotto da parte del team di sviluppo utilizzando metodi, tecniche, metodologie e strumenti. È suddiviso in varie fasi secondo uno schema di riferimento (il ciclo di vita del software). Descritto da un modello: informale, semi-formale o formale.

1.2 Cos'è il software?

Il **prodotto software** non è solo un insieme di linee di codice ma comprende tutti gli "artefatti" che lo accompagnano e che sono prodotti durante l'intero sviluppo come: il codice, la documentazione, i casi di test, manuali e varie specifiche e procedure di gestione.

Quindi il **software** è un prodotto invisibile, intangibile, facilmente duplicabile ma costosissimo: è un'opera dell'ingegno protetto dalle leggi. Inoltre, al software vanno attribuiti altre due keywords:

- **Requisito software:** funzione o proprietà controllabile (testabile) che deve possedere l'implementazione di un prodotto software. È importante per il cliente.
Esempio: l'utente deve poter registrarsi, aggiungere o togliere elementi al carrello, specificare indirizzi alternativi, pagare.
- **Feature software:** l'insieme di funzioni che permettono di usare un prodotto software in un servizio o business. È importante per il fornitore. *Esempio:* carrello per negozio elettronico.

1.3 Programma vs Prodotto

Un **programma** è una semplice applicazione sviluppata, testata e usata dallo stesso sviluppatore. In altri termini, l'autore è anche il cliente.

Il **prodotto software** viene sviluppato per terzi, è un software industriale che ha un costo circa 10 volte superiore ad un normale programma e deve essere corredato di documentazione, manuali e casi di test.

I **prodotti** possono essere **generici** oppure **specifici**:

- I **prodotti generici** sono dei software prodotti da aziende e utilizzati da un ampio bacino di utenza diversificato.
- I **prodotti specifici** sono software sviluppati ad-hoc per uno specifico cliente.

Il costo dei **prodotti generici** è maggiore rispetto a quello dei **prodotti specifici** ma il maggior sforzo di sviluppo è nei **prodotti specifici**.

1.4 Ecosistemi software

Gli **ecosistemi software** sono mercati, in cui si vendono prodotti (es. AppStore o PlayStore) o componenti e servizi. La caratteristica principale è quella di una collezione di prodotti software, su una piattaforma definita da un'azienda e che vengono sviluppati ed evolvono nello stesso ambiente.

1.5 Software libero

Il **"Software libero"** è software che rispetta la libertà degli utenti e la comunità e non riguarda il suo prezzo. In breve, significa che gli utenti hanno la libertà di eseguire, copiare, distribuire, studiare, modificare e migliorare il software. Un programma è software libero se gli utenti del programma godono delle 4 libertà fondamentali:

1. Libertà di eseguire il programma come si desidera, per qualsiasi scopo (**libertà 0**).
2. Libertà di studiare come funziona il programma e di modificarlo in modo da adattarlo alle proprie necessità (**libertà 1**). L'accesso al codice sorgente ne è un prerequisito.
3. Libertà di ridistribuire copie in modo da aiutare gli altri (**libertà 2**).
4. Libertà di migliorare il programma e distribuirne pubblicamente i miglioramenti da voi apportati, in modo tale che tutta la comunità ne tragga beneficio (**libertà 3**).

1.6 Costi e Manutenzione

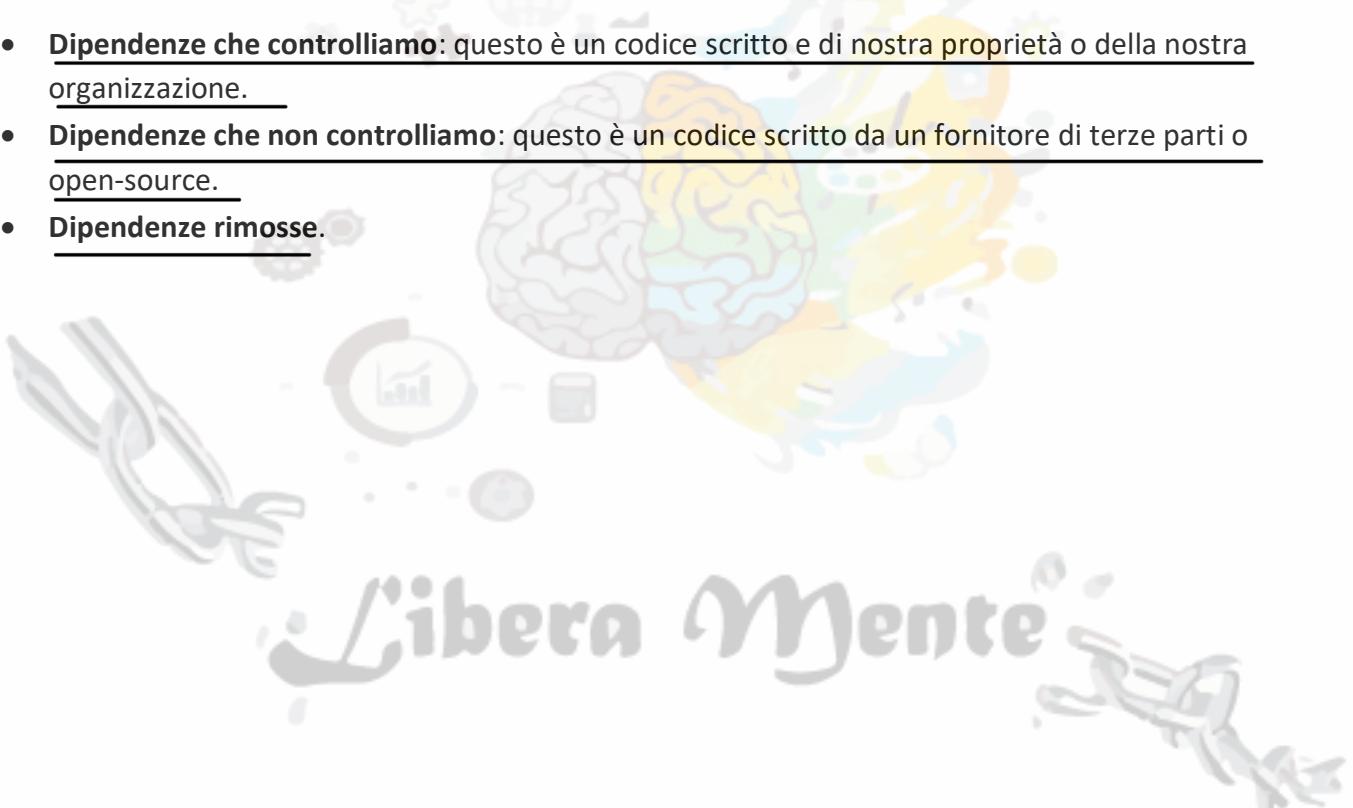
Il costo del software viene calcolato in base alle ore di lavoro, il software e l'hardware utilizzato e altre risorse di supporto. Il costo della manutenzione è più elevato rispetto a quello di produzione.

Il software dopo il suo rilascio, specie se lo stesso ha una vita lunga, ha bisogno di alcune fasi di manutenzione. Per **manutenzione** intendiamo sia la correzione di eventuali bug, sia l'estensione/modifica di alcune caratteristiche.

1.7 Dipendenze

Ogni prodotto software dipende da altri prodotti software, che a loro volta dipendono da altri software. Associamo a ciascun prodotto o sistema software un **grafo di dipendenze**. I nodi del grafo delle dipendenze sono pacchetti software (es. librerie) in diverse versioni. Abbiamo diversi tipi di dipendenze:

- **Dipendenze che controlliamo:** questo è un codice scritto e di nostra proprietà o della nostra organizzazione.
- **Dipendenze che non controlliamo:** questo è un codice scritto da un fornitore di terze parti o open-source.
- **Dipendenze rimosse.**



Capitolo 2 – Ciclo di vita del Software

2.1 Ciclo di Vita del Software: CVS

Il **ciclo di vita del software** è il periodo di tempo che inizia quando un prodotto software è concepito e termina quando il prodotto non è più disponibile per l'uso. Il ciclo di vita del software in genere include una fase concettuale, una fase dei requisiti, fase di progettazione, fase di implementazione, fase di test, fase di installazione e verifica, funzionamento e fase di mantenimento.

Queste fasi possono sovrapporsi o essere eseguite iterativamente.

Il **ciclo di sviluppo del software**, invece, è il periodo di tempo che inizia con la decisione di sviluppare un prodotto software e termina con la consegna del prodotto. Questo ciclo in genere comprende una fase dei requisiti, una fase di progettazione, fase di implementazione, fase di test e, talvolta, fase di installazione e verifica.

2.2 Modelli di CVS

Un **modello** del ciclo di vita del software (CVS) è una caratterizzazione descrittiva o prescrittiva di come un sistema software viene o dovrebbe essere sviluppato.

I modelli CVS devono avere le seguenti caratteristiche:

- Descrizione dell'organizzazione del lavoro nella software house.
- Linee guida per pianificare, dimensionare il personale, assegnare budget, schedulare e gestire.
- Definizione e scrittura dei manuali d'uso e diagrammi vari.
- Determinazione e classificazione dei metodi e strumenti più adatti alle attività da svolgere.

Le fasi principali di un qualsiasi CVS sono le seguenti:

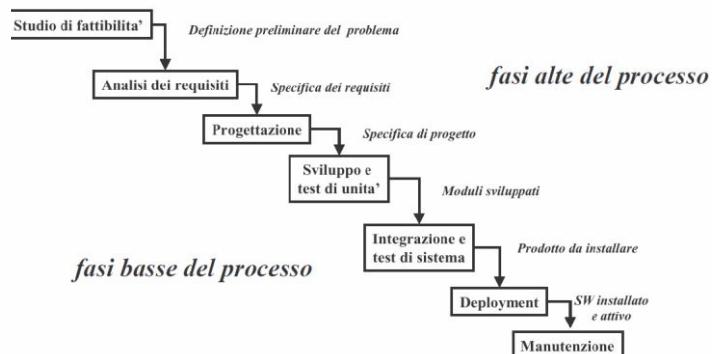
1. **Definizione** (si occupa del cosa). Determinazione dei requisiti, informazioni da elaborare, comportamento del sistema, criteri di validazione, vincoli progettuali.
2. **Sviluppo** (si occupa del come). Definizione del progetto, dell'architettura software, traduzione del progetto nel linguaggio di programmazione, collaudi.
3. **Manutenzione** (si occupa delle modifiche). Miglioramenti, correzioni, prevenzione, adattamenti.

Modello a Cascata (Waterfall)

Definisce che il processo segua una progressione sequenziale di fasi senza ricicli, al fine di controllare meglio tempi e costi. Inoltre, definisce e separa le varie fasi e attività del processo in modo da minimizzare la sovrapposizione tra di esse. Ad ogni fase viene prodotto un semilavorato con la relativa documentazione e lo stesso viene passato alla fase successiva. I prodotti ottenuti da una fase non possono essere modificati durante il processo di elaborazione delle fasi successive e vengono utilizzati come input per la fase successiva. È consigliato utilizzarlo in una situazione in cui i requisiti sono ben definiti.

Vantaggi: facile da comprendere e da applicare.

Svantaggi: l'interazione con il cliente avviene solo all'inizio e alla fine del ciclo. I requisiti dell'utente vengono scoperti solo alla fine. Se il prodotto non ha soddisfatto tutti i requisiti, alla fine del ciclo, è necessario iniziare daccapo tutto il processo.



Organizzazione sequenziale: fasi alte del processo

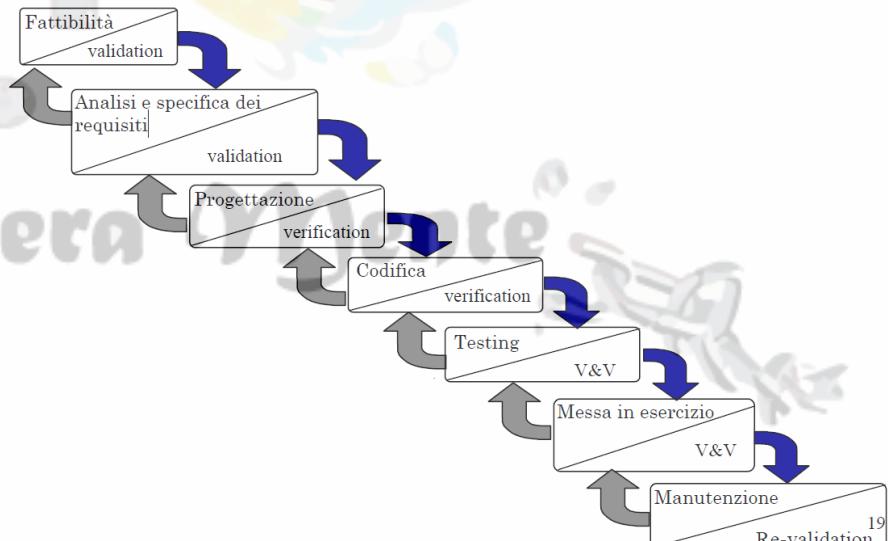
- **Studio di fattibilità:** effettua una valutazione preliminare dei costi e dei requisiti in collaborazione con il committente. L'obiettivo è quello di decidere la fattibilità del progetto, valutarne i costi, i tempi necessari e le modalità di sviluppo. Output: documento di fattibilità.
- **Analisi e specifica dei requisiti:** vengono analizzate le necessità dell'utente e del dominio d'applicazione del problema. Output: documento di specifica dei requisiti.
- **Progettazione:** viene definita la struttura del software e il sistema viene scomposto in componenti e moduli. Output: definizione dei linguaggi e formalismi.

Organizzazione sequenziale: fasi basse del processo

- **Programmazione e test di unità:** ogni modulo viene codificato nel linguaggio e testato separatamente dagli altri.
- **Integrazione e test di sistema:** i moduli vengono integrati tra loro e vengono testate le loro interazioni. Viene rilasciata una *beta-release* (release esterna) oppure una *alpha-release* (release interna) per testare al meglio il sistema.
- **Deployment:** rilascio del prodotto al cliente.
- **Manutenzione:** gestione dell'evoluzione del software.

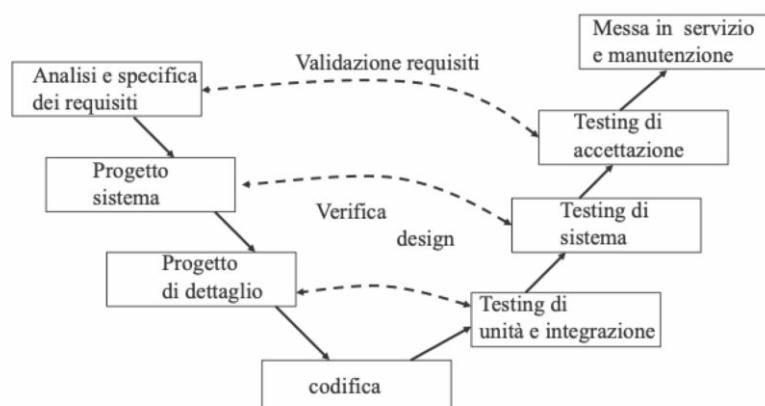
Modello verification e validation (V&V)

Questo modello è uguale al modello a cascata, ma con la differenza che **vengono applicati i ricicli**, ovvero al completamento di ogni fase viene fatta una verifica ed è possibile tornare alla fase precedente nel caso la stessa non verifica le aspettative.



Modello a V

Le attività di sinistra sono collegate a quelle di destra intorno alla codifica. Se si trova un errore in una fase a destra (es. testing di sistema) si ri-esegue il pezzo della V collegato.



Modello basati su prototipo

Viene usato un **prototipo** per aiutare a comprendere i requisiti o per valutare la fattibilità di un approccio. Il prototipo, quindi, è la realizzazione di una prima implementazione, più o meno incompleta da considerare come una ‘*prova*’, con lo scopo di:

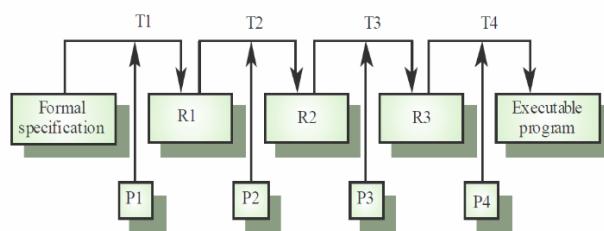
- accettare la fattibilità del prodotto;
- validare i requisiti.

Il prototipo è un mezzo attraverso il quale si interagisce con il committente per accertarsi di aver ben compreso le sue richieste, per specificare meglio tali richieste, per valutare la fattibilità del prodotto. Dopo la fase di utilizzo del prototipo si passa alla produzione della versione definitiva del Sistema Software mediante un modello che, in generale, è di tipo **waterfall** (a cascata). Abbiamo vari tipi di prototipazione:

1. **mock-ups**: produzione completa dell’interfaccia utente. Consente di definire con completezza e senza ambiguità i requisiti (si può, già in questa fase, definire il manuale di utente).
2. **breadboards**: implementazione di sottoinsiemi di funzionalità critiche del sistema, cioè i vincoli pesanti che sono posti nel funzionamento del sistema (carichi elevati, tempo di risposta, ...), senza le interfacce utente. Produce *feedbacks* su come implementare la funzionalità (in pratica si cerca di conoscere prima di garantire).
3. **Prototipazione “esplorativa”**: si inizia a sviluppare le parti del sistema che sono già ben specificate aggiungendo man mano nuove caratteristiche secondo le necessità fornite dal cliente.
4. **Prototipo usa e getta (*throw-away*)**: lo scopo di questo tipo di prototipazione è quello di identificare meglio le specifiche richieste dall’utente sviluppando dei prototipi che sono funzionanti. Non appena il prototipo è stato verificato da parte del cliente o da parte degli sviluppatori deve essere buttato via.

Modello trasformazionale (o trasformazioni formali)

Basato su un modello matematico che viene trasformato da una rappresentazione formale ad un’altra. Questo modello comporta problemi nel personale in quanto non è facile trovare persone con le conoscenze giuste per poterlo implementare. E’ consigliato questo modello quando si hanno requisiti stabili, sistemi critici per le persone o cose.



Modello di sviluppo a componenti

E’ previsto una repository dove vengono depositate le componenti sviluppate durante le fasi del ciclo di vita. Le componenti vengono prese dalla repository e riutilizzate quando necessario. Questo modello è particolarmente usato per sviluppare software in linguaggi Object Oriented.

Modello incrementale

Utilizzato per la progettazione di grandi software che richiedono tempi ristretti o costi alti. Vengono rilasciate delle release funzionanti (**deliverables**) anche se non soddisfano pienamente i requisiti del cliente.

Le fasi alte del processo sono completamente realizzate. Il sistema così progettato viene decomposto in **sottosistemi** (*incrementi*) che vengono implementati, testati, rilasciati, installati e messi in manutenzione secondo un piano di priorità in tempi diversi. Diventa fondamentale la fase di integrazione di nuovi sottosistemi con quelli già prodotti.

I **vantaggi** sono la possibilità di anticipare da subito delle funzionalità al committente, poiché ciascun incremento corrisponde al rilascio di una parte delle funzionalità; i requisiti a più alta priorità per il committente vengono rilasciati per prima e c'è un minor rischio di fallimento del progetto.

Tra i vantaggi abbiamo anche un testing più esaustivo, infatti, i rilasci iniziali agiscono come prototipi e consentono di individuare i requisiti per i successivi incrementi.

I modelli incrementali sono accomunati ai modelli iterativi perché entrambi prevedono successive versioni del sistema:

- **Sviluppo incrementale**: ogni versione aggiunge nuove funzionalità/sottosistemi;
- **Sviluppo iterativo (evolutivo)**: da subito sono presenti tutte le funzionalità/sottosistemi che vengono successivamente raffinate, migliorate.

Modello a spirale

Il processo è visto come una spirale dove ogni ciclo viene diviso in quattro fasi:

2. Determinazione degli obiettivi della fase;
3. Identificazione e riduzione dei rischi, valutazione delle alternative;
4. Sviluppo e verifica della fase;
1. Pianificazione della fase successiva.

Una caratteristica importante di questo modello è il fatto che i rischi vengono presi seriamente in considerazione e che ogni fine ciclo produce una **deliverables**. In un certo senso può essere visto come un modello a cascata iterato più volte. I rischi sono causati da una scarsa chiarezza sui requisiti, sulle tecnologie e sulla strutturazione del sistema. In alcuni casi, sono state scelte tecnologie innovative, per le quali manca l'esperienza nel gruppo di progetto.

Per risolvere tali problemi, utilizziamo varie iterazioni e la costruzione di prototipi tra cui:

- **Prototipi di interazione** (interfacce utente): per affrontare i rischi legati all'incertezza sui requisiti.
- **Prototipi architetturali** (realizzazione e test di aspetti infrastrutturali): per affrontare i rischi legati alla scelta delle tecnologie ed i dubbi sulla strutturazione del sistema.

Successivamente, quando i rischi principali sono stati messi sotto controllo, ogni iterazione ha lo scopo di costruire, in modo progressivo, nuove porzioni del sistema, via via integrate con le precedenti, e di verificarle con il committente e le altre parti interessate.

Vantaggi

Rende esplicita la gestione dei rischi, focalizza l'attenzione sul riuso, determina errori in fasi iniziali, aiuta a considerare gli aspetti della qualità e integra sviluppo e manutenzione.

Svantaggi

Richiede un aumento nei tempi di sviluppo, delle persone con capacità di identificare i rischi, una gestione maggiore del team di sviluppo e quindi anche un costo maggiore.

Rischio : probabilità di capitare un problema

Modello a spirale e valutazione dei rischi

Il **modello a cascata** genera alti rischi per progetti nuovi e bassi rischi nello sviluppo di applicazioni familiari con tecnologie già note.

Nel **modello a prototipazione** si hanno bassi rischi nelle nuove applicazioni, mentre, alti rischi per la mancanza di un processo definito e visibile.

Nel **modello trasformazionale** si hanno alti rischi a causa delle tecnologie coinvolte e delle professionalità richieste.

Extreme programming

Approccio recente per lo sviluppo del software basato su iterazioni veloci che rilasciano piccoli incrementi delle funzionalità. Abbiamo una partecipazione più attiva del committente al team di sviluppo e un miglioramento costante e continuo del codice.



Capitolo 3 – Project Management e Comunicazione

3.1 Project Management

Il **project management** racchiude le attività necessarie per assicurare che un progetto software venga sviluppato rispettando le scadenze e gli standard. Le entità fisiche che prendono parte al *project management* sono:

1. **Business manager**: definisce i termini economici del progetto;
2. **Project manager**: amministra le risorse, quantifica il lavoro, assicura che gli obiettivi siano raggiunti. Inoltre, pianifica, motiva, organizza e controlla lo sviluppo, seleziona il team di sviluppo, stende i rapporti e le presentazioni.
3. **Practitioners**: hanno competenze tecniche per realizzare il sistema.
4. **Customers** (clienti): specificano i requisiti del software da sviluppare.
5. **End users** (utenti): gli utenti che interagiscono con il sistema.

3.2 Definizione di Progetto

Un **progetto** è un impegno, limitato nel tempo, per raggiungere una serie di obiettivi che richiedono un certo effort (sforzo). Un progetto ha un inizio ed una fine ed è realizzato da un'equipe di persone.

Un progetto include:

- una serie di risultati per un cliente;
- attività tecniche e gestionali necessarie per produrre e consegnare i risultati;
- risorse consumate dalle attività (persone, budget).

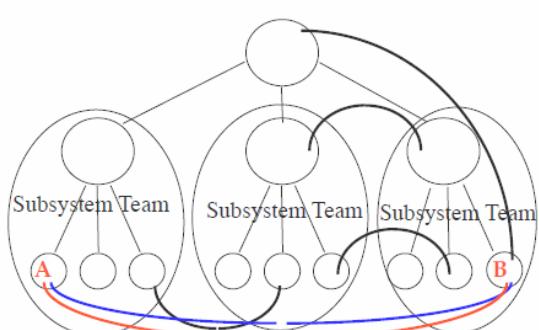
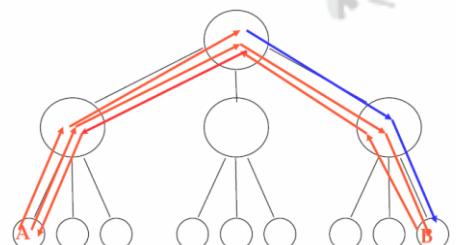
3.3 Organizzazione del Progetto

Un'**organizzazione di progetto** definisce le relazioni tra le risorse, in particolare i partecipanti in un progetto. Un'organizzazione del progetto dovrebbe definire:

- Chi decide (struttura decisionale);
- Chi segnala il proprio stato a chi (struttura di segnalazione);
- Chi comunica con chi (comunicazione struttura).

Inoltre un'organizzazione di progetto può essere strutturata in maniera **gerarchica** in cui il nodo principale (la radice) è il **chief executive** (amministratore). Inoltre non abbiamo comunicazioni dirette tra i figli di un nodo con i figli di un altro nodo.

Con questo tipo di organizzazione la comunicazione è sottocontrollo perché esiste la root che controlla, quantifica e gestisce.



Per comunicare allo stesso livello esiste la comunicazione **peer-to-peer**. In questa organizzazione abbiamo una comunicazione paritaria.

3.4 Attività del project manager

Il **project manager** si occupa della:

- stesura della proposta di progetto;
- stima del costo del progetto;
- pianificazione (planning) e temporizzazione (scheduling);
- monitoraggio e revisioni del progetto;
- selezione e valutazione del personale;
- stesura di rapporti e presentazioni

3.5 Stesura del piano del Progetto

Introduzione

Viene definita una descrizione di massima del progetto, vengono consegnati gli elementi con le rispettive date di consegne e vengono pianificati eventuali cambiamenti.

Organizzazione del progetto

Vengono definite le relazioni tra le varie fasi del progetto, la sua struttura organizzativa, le interazioni con entità esterne, le responsabilità di progetto (le principali funzioni e chi sono i responsabili).

Processi gestionali

Si definiscono gli obiettivi e le priorità, le assunzioni, le dipendenze, i vincoli, i rischi con i relativi meccanismi di monitoraggio, pianificazione dello staff.

Processi tecnici

Vanno specificati i sistemi di calcolo, i metodi di sviluppo, la struttura del team, il piano di documentazione del software e viene pianificata la gestione della qualità.

Pianificazione del lavoro, delle risorse umane e del budget

Il progetto viene diviso in tasks e a ciascuno assegnata una priorità, le dipendenze, le risorse necessarie e i costi. Le attività devono essere organizzate in modo da produrre risultati valutabili dal management. I risultati possono essere **milestone** (rappresenta il punto finale di un'attività di processo) o **deliverables** (risultato fornito al cliente).

3.6 Scheduling di progetto

Divide il progetto in attività e mansioni (tasks) e stima il tempo e le risorse necessarie per completare ogni singola mansione. Vengono organizzate le mansioni in modo concorrente, per ottimizzare la forza lavoro e minimizza la dipendenza tra le singole mansioni per evitare ritardi dovuti all'attesa del completamento di un'altra mansione. Sono necessari intuito ed esperienza.

Problemi dello Scheduling

È difficile stimare la difficoltà dei problemi ed il costo di sviluppo di una soluzione. La produttività non è proporzionale al numero di persone che lavorano su una singola mansione. Infatti, aggiungere personale in un progetto in ritardo può aumentare ancora di più il ritardo.

3.7 Grafico delle attività (PERT)

Abbiamo diversi tipi di rappresentazione grafica dello scheduling del progetto, tra cui il **grafico PERT** che mostra la suddivisione del lavoro in attività evidenziando le dipendenze e il cammino critico. Le mansioni non devono essere troppo piccole (una settimana o due di lavoro).

Il *diagramma di PERT* si divide in:

- **ES** (earliest start time): il minimo giorno di inizio dell'attività, a partire dal minimo tempo necessario per le attività che precedono.
- **EF** (earliest finish time): dato l'ES e la durata dell'attività, l'*EF* è il minimo giorno in cui l'attività può terminare.
- **LF** (latest finish time): il giorno massimo in cui quel job deve finire senza che si crei ritardo per i job che dipendono da lui.
- **LS** (latest start time): dato LF e la durata del job, *LS* è il giorno massimo in cui quel job deve iniziare senza provocare ritardo per i job che dipendono da lui.

Il **grafico a barre** mostra lo scheduling come calendario dei lavori mentre il **diagramma di Gannt** esprime la temporizzazione.

3.8 Risk management

Il **management dei rischi** identifica i rischi possibili e cerca di pianificarli per minimizzare il loro effetto sul progetto.

Un **rischio** è una probabilità che si verificherà una circostanza negativa. I rischi si dividono in:

- **Rischi del progetto**: influenzano la pianificazione o le risorse;
- **Rischi del prodotto**: influiscono sulla qualità o sulle prestazioni del software in fase di sviluppo;
- **Rischi aziendali**: influenzano sullo sviluppo dell'organizzazione.

Identificazione dei rischi

I **rischi da identificare** sono di vari tipi tra cui:

- | | | |
|-------------------------------|-------------------------|---------------------------------|
| - rischi tecnologici; | - rischi organizzativi; | - rischi relativi ai requisiti; |
| - rischi delle risorse umane; | - rischi nei tools; | - rischi di stima/sottostima. |

Le tipologie di rischi sono le seguenti:

- **Tecnologici**: alcune tecnologie di supporto (database, componenti esterne) non sono abbastanza validi come ci aspettavamo.
- **Risorse umane**: non è possibile reclutare staff con la competenza richiesta oppure non è possibile fare formazione allo staff.
- **Organizzativi**: cambiamenti nella struttura.
- **Strumenti**: ad esempio il codice/documentazione prodotto con un determinato strumento non è abbastanza efficiente.
- **Requisiti**: cambiamenti nei requisiti richiedono una revisione del progetto già sviluppato.
- **Stima**: il tempo richiesto, la dimensione del progetto sono stati sottostimati.

Analisi dei rischi

Ad ogni rischio va assegnata una probabilità che esso si verifichi e vanno valutati gli effetti dello stesso che possono essere: **catastrofici, seri, tollerabili, insignificanti**.

Pianificazione dei rischi

Viene considerato ciascun rischio e viene sviluppata una strategia per risolverlo. Le strategie che possiamo prendere possono essere:

- Evitare i rischi con una prevenzione;
- Minimizzare i rischi;
- Gestire i rischi con un piano di contingenza per evitarli.

Monitoraggio dei rischi

Ogni rischio viene regolarmente valutato e viene verificato se è diventato meno o più probabile, inoltre i suoi aspetti vanno discussi con il management per valutare meglio i provvedimenti da adottare.

3.9 Reporting vs Comunicazione

Il **reporting** supporta la gestione del progetto monitorando lo stato del progetto, ovvero:

- Quale lavoro è stato completato?
- Quale lavoro è in ritardo?
- Quali problemi minacciano l'avanzamento del progetto?

Il **reporting** non è sufficiente quando due squadre hanno bisogno di **comunicare**, bensì è necessaria:

- una struttura di comunicazione;
- un partecipante di ogni squadra è responsabile di facilitare la comunicazione tra i due team. Tali partecipanti sono chiamati **collegamento**.

3.10 Ruolo

Un **ruolo** definisce un'insieme di responsabilità ("to-dos"). Esiste un mapping tra i ruoli e i partecipanti:

- **One-to-one** (uno a uno): c'è un ruolo e un partecipante. E' ideale ma raro perché con una sola persona si riesce a ricoprire un intero ruolo ma ciò non è sempre facile.
- **Many-to-Few** (molti verso pochi): ogni project manager assume più ruoli ma ciò è pericoloso perché si deve distribuire in più parti.
- **Many-to-"Too-Many"** (molti a troppi): molti partecipanti che non hanno un ruolo significativo e ciò è controproducente.

3.11 Task

Un **task** descrive una porzione di lavoro tracciata dal management; di solito ha un effort di 3 - 10 giorni lavorativi.

Un **task** è specificato da un **work package** cioè:

- la descrizione del lavoro da svolgere;
- precondizioni, la durata, le risorse necessarie;
- prodotti di lavoro da produrre e criteri di accettazione;
- gestione dei rischi.

Un task deve avere anche **criteri di completamento** cioè include i criteri di accettazione per i *work products* prodotti dal task.

Un **work product** è il risultato di un task (un documento, una presentazione, un pessso di codice, ...).

I *work products* consegnati al cliente sono chiamati **deliverables**.

Task Size

I *task* sono scomposti in varie dimensioni per consentirne il monitoraggio. La **size di un task** dipende dalla natura del lavoro e da quanto bene il compito è compreso. Per trovare la size appropriata di un task è buona norma fare riferimento ai progetti precedenti.

3.12 Attività

L'**attività** è l'unità di lavoro principale e porta ai **milestone** utili per misurare il progresso del progetto software. Le *attività* possono essere raggruppate in step o in fasi.

3.13 Comunicare è fondamentale

In grandi effort si passa più tempo a comunicare che a programmare. Un ingegnere del software deve imparare le cosiddette **competenze trasversali**:

- **Collaborazione**: cioè negoziare i requisiti con il cliente e con membri della sua squadra e di altre squadre;
- **Presentazione**: cioè presentare una parte importante del sistema durante una revisione;
- **Gestione**: facilitare una riunione di squadra;
- **Scrittura tecnica**: scrivere parte della documentazione del progetto.

Comunicazione a eventi vs Comunicazione meccanica

La **comunicazione a eventi** è uno scambio di informazioni con obiettivi e può essere:

- **schedulata**: comunicazione pianificata (Esempi: riunione settimanale del team, revisione). Riguarda:
 - la **definizione del problema** presentando gli obiettivi, i requisiti e i vincoli;
 - la **review del progetto** ponendo l'attenzione sui modelli del sistema e saremo influenzati dai milestone e deliverables;
 - la **review del client** concentrandoci sui requisiti. Questa fase è schedulata dopo la fase di analisi.

Abbiamo diversi modi per gestire questo tipo di comunicazione:

- **Walkthrough** (Informale): verifichiamo quello che abbiamo fatto e incrementiamo la qualità del sottosistema. Questo è schedulato da ogni team.
- **Inspection** (Formale): l'obiettivo è di verificare che tutto sia in linea con i requisiti. Ad esempio, quando facciamo vedere il sistema finale al cliente. Questo è schedulato dal project management.
- **Status Review**: trova i problemi e li corregge.
- **Brainstorming**: genera e valuta un insieme di soluzioni per poi decidere come proseguire.
- **Release**: calcola il risultato di attività di sviluppo del software.
- **Postmortem Review**: descrive le lezioni imparate. Viene effettuata alla fine del progetto.

- **non schedulata**: comunicazione guidata dagli eventi (Esempi: segnalazione del problema, richiesta di modifica, chiarimento). Riguarda:
 - **Richiesta di chiarimenti**: comunicazione che avviene tra sviluppatori, clienti e utenti.
 - **Richiesta di modifica**: un partecipante segnala un problema e propone una soluzione. Le richieste di modifica sono spesso formalizzate quando il progetto ha una dimensione notevole.
 - **Risoluzione dei problemi**: seleziona un'unica soluzione a un problema per il quale sono state proposte più soluzioni.

La **comunicazione meccanica** è uno strumento o una procedura che può essere utilizzata per trasmettere informazione. Può essere:

- **Sincrona**: il mittente e il destinatario stanno comunicando allo stesso tempo. Abbiamo diversi tipi di comunicazione sincrona:
 - **Hallway conversation**:
Supporta conversazioni non pianificate, richiesta di chiarimento, richiesta di modifica.
I **vantaggi** è che tale comunicazione è economica ed efficace per risolvere problemi semplici.
Gli **svantaggi** sono la perdita di informazioni e frequenti malintesi.
 - **Meeting** (faccia a faccia, telefono, videoconferenza):
Supporta conversazioni pianificate, review del cliente, review del progetto, review dello stato, brainstorming, risoluzione dei problemi.
Tra i **vantaggi** abbiamo l'efficacia per la risoluzione dei problemi e la costruzione del consenso.
Per gli **svantaggi** c'è il costo elevato (persone, risorse), larghezza di banda ridotta.
 - **E-mail**:
Supporta la release, richiesta di modifica, brainstorming.
Ideale per comunicazioni pianificate e annunci.
Svantaggiose perché possono essere fraintese, inviate a persone sbagliate, o perse.
 - **Gruppo di notizie**:
Supporta il rilascio, richiesta di modifica, brainstorming.
Vantaggioso per la discussione tra persone che condividono un interesse comune ed è economico.
 - **World Wide Web**:
Supporta il rilascio, richiesta di modifica, ispezioni.
Vantaggioso perché fornisce all'utente documenti ipertestuali cioè documenti collegati ad altri.
Non supporta facilmente documenti in rapida evoluzione.
- **Asincrono**: mittente e destinatario non stanno comunicando allo stesso tempo.

3.14 Problem Statement

Il **problem statement** è sviluppato dal cliente e descrive:

- La situazione attuale;
- Le funzionalità che il nuovo sistema dovrebbe supportare;
- L'ambiente in cui verrà distribuito il sistema;
- Deliverable;
- Date di consegna;
- Criteri di accettazione.

Capitolo 4 – Diagramma UML

4.1 Unified Modeling Language (UML)

Il **modello UML** è un linguaggio (e notazione) univereale, per la creazione di modelli software basato sul paradigma orientato agli oggetti. Nello specifico *UML* è un:

- **linguaggio** per specificare, costruire, visualizzare e documentare gli artefatti di un sistema;
- **universale** perché può rappresentare sistemi molto diversi, da quelli web ai legacy, dalle tradizionali applicazioni Cobol a quelle object oriented.

Quindi *UML* è un **linguaggio di modellazione**, non un metodo, né una metodologia, bensì definisce una notazione standard basata su un **meta-modello** integrato, cioè definisce le relazioni tra i diversi elementi (classi, attributi, moduli, ...). *UML* prevede una serie di **modelli diagrammatici** basati sul *meta-modello*.

Inoltre, *UML* non indica una sequenza di processo, cioè non dice “prima bisogna fare questa attività, poi quest’altra”.

UML è una notazione data dall’unione di vari modelli sviluppati intorno agli anni ‘90:

1. **OMT** (Rumbaugh): si rivelava ottimo in analisi e debole nel disegno;
2. **Bachman**: eccelleva nel disegno e peccava in analisi;
3. **OOSE** (Jacobson): aveva il suo punto di forza nell’analisi dei requisiti e del comportamento di un sistema ma si rivelava debole in altre aree.

4.2 Architettura a 4 livelli

UML è basato su un modello architettonale a 4 livelli:

Livello	Descrizione	Esempio
Meta-meta-modello	Definisce un linguaggio per la specifica dei meta-modelli.	Meta-classi, meta-attributi, meta-operazioni, ...
Meta-modello	Definisce un linguaggio per la specifica dei modelli.	Classi, attributi, operazioni, ...
Modello	Definisce un linguaggio per la specifica di un certo dominio applicativo.	Classi di uno specifico contesto
Oggetti	Istanze del modello.	Oggetti ottenuti istanziando le classi del modello

Nell’architettura a 4 livelli di *UML* ogni livello è un’astrazione di quello sottostante, ed è definito in termini di quello sovrastante.

La **sintassi astratta** di *UML* viene definita utilizzando la notazione dei metamodelli ed è espressa tramite diagrammi e linguaggio naturale.

4.3 Obiettivi di UML

- Fornire all'utente un linguaggio di specifica espressivo, visuale e pronto.
- Offrire meccanismi di estensibilità e specializzazione del linguaggio.
- Essere indipendente dai linguaggi di programmazione e dai processi di sviluppo.
- Incoraggiare la crescita dei tool OO commerciali.
- Supportare concetti di sviluppo ad alto livello come frameworks, pattern ed i componenti.

4.4 Cosa non è UML

- Non è un linguaggio di programmazione visuale (è un linguaggio di specifica visuale).
- UML non è un modello per la definizione di interfacce.
- UML non è dipendente dal paradigma di sviluppo nel quale può essere utilizzato.

4.5 Modelli per descrivere Sistemi Software

La modellazione di un sistema software è data da vari passi: **modellazione funzionale + modellazione ad oggetti + modellazione dinamica**.

- Modellazione funzionale è rappresentata in *UML* con i **case diagrams** e descrive la funzionalità del sistema dal punto di vista dell'utente.
- Modellazione ad oggetti è rappresentata in *UML* con i **class diagrams** e descrive la struttura del sistema in termini di oggetti, attributi, associazioni e operazioni.
- Modellazione dinamica è rappresentata in *UML* con **diagrammi di interazione** e descrive il comportamento interno del sistema, cioè come il sistema reagisce agli eventi esterni.

4.6 Diagrammi

- **Use case diagrams**: descrivono i comportamenti funzionali del sistemi dal punto di vista dell'utente;
- **Class diagrams**: descrivono la struttura statica del sistema (oggetti, attributi, associazioni);
- **Sequence diagrams**: descrivono i comportamenti dinamici tra gli oggetti del sistema;
- **Statechart diagrams**: descrivono il comportamento dinamico di un solo oggetto;
- **Activity diagrams**: descrivono il comportamento dinamico del sistema, in particolare il suo flusso di lavoro.

Use Case Diagrams (Diagrammi dei casi d'uso)

Questo tipo di diagramma mostra come deve essere utilizzato il sistema (casi d'uso) e le relazioni tra attori e casi d'uso. Gli **attori** sono gli utilizzatori del sistema e coloro che interagiscono con esso.

Un **caso d'uso**, quindi, è uno dei modi possibili per usare il sistema, inoltre, descrive l'interazione tra attori e sistema, non la "logica interna" della funzione.

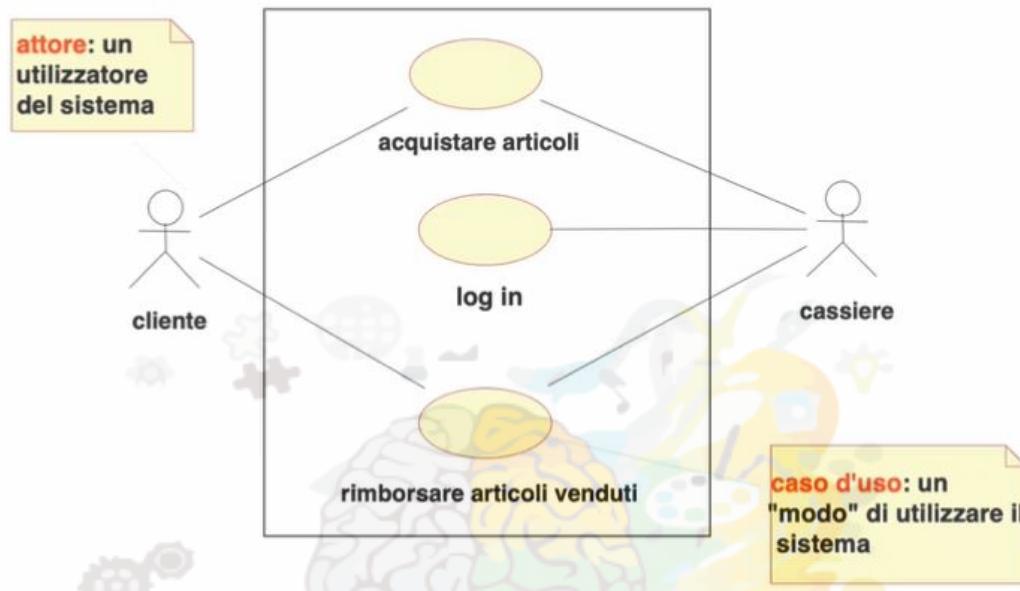
Durante la raccolta dei requisiti l'obiettivo è di realizzare una diagramma dei casi d'uso per capire chi sono gli attori e quali casi d'uso, cioè quali funzionalità, offre il sistema.

Un **caso d'uso** rappresenta un'insieme di funzionalità di un sistema. La descrizione all'interno dovrebbe essere basata su un verbo o su un sostantivo che esprime l'avvenimento. Un caso d'uso è sempre iniziato da un attore detto **primario**, gli altri attori che interagiscono con quel caso d'uso sono detti **secondari**.

Un **attore** rappresenta un ruolo assunto da un utente o altra entità; l'attore non è necessariamente umano.

Esempio:

Abbiamo due attori: il *cliente* e il *cassiere*. I casi d'uso sono 3: *acquistare articoli*, *log in*, *rimborsare articoli venduti*. Il diagramma seguente mostra tutte le funzionalità (o casi d'uso) che può eseguire il *cliente* che non sono altro un sottoinsieme delle funzionalità del *cassiere*, perché in più ha la funzionalità di effettuare il *log in*.



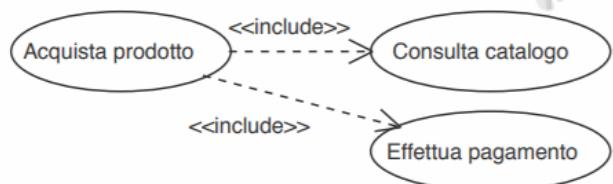
Le **associazioni** collegano gli attori ai casi d'uso mediante un arco. Un attore si può associare solo a casi d'uso, classi e componenti. Un caso d'uso non si può associare ad altri casi d'uso riguardanti lo stesso argomento. Inoltre, abbiamo vari tipi di *associazione*:

- **Generalizzazione:** si ha un associazione tra un attore o caso d'uso ad un altro più generale.

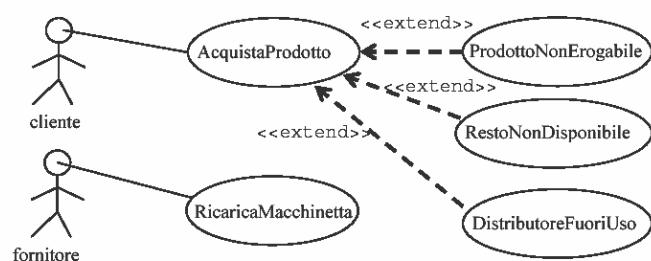


- **<<include>>:** è una dipendenza tra casi d'uso; il caso incluso fa parte del comportamento di quello che lo include. L'inclusione non è opzionale ed avviene in ogni istanza del caso d'uso. Inoltre, l'inclusione viene usata anche per riutilizzare parti comuni a più casi d'uso.

Non si possono formare cicli di include.



- **<<extend>>:** è una dipendenza tra casi d'uso ma a differenza dell'*include* il verso della freccia è opposto, inoltre, rappresenta un caso eccezionale o che si verifica di rado.



Class Diagrams (Diagramma delle classi)

Rappresentano la struttura di un sistema, di conseguenza definisce gli elementi base del sistema. Vengono usati durante la fase di analisi dei requisiti, di system design e object design.

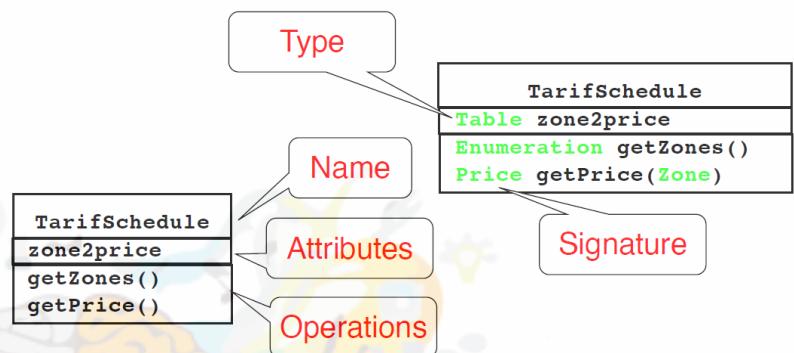
È possibile definire diagrammi contenenti classi di oggetti con i loro attributi e operazioni, mostra le relazioni tra le classi (associazioni, aggregazioni e gerarchie di specializzazione/generalizzazione) e può essere utilizzato a diversi livelli di dettaglio (in analisi e in disegno).

Una **classe** rappresenta un concetto e racchiude in sé tutti gli oggetti che hanno le stesse proprietà (attributi e operazioni).

In UML una *classe* è composta da 3 parti:

- **Nome**: con la quale è identificata;
- **Attributi**: ogni attributo ha un **tipo**;
- **Metodi o Operazioni**: ogni operazione ha una **firma**.

Inoltre, il nome di una classe è unico.

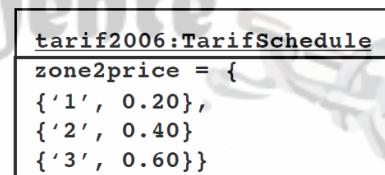


Un **oggetto** è un qualcosa di identificabile che ricorda il proprio stato e che può rispondere a richieste tramite operazioni relative al proprio stato. Gli *oggetti* interagiscono tra di loro richiedendo reciprocamente servizi o informazioni: in risposta ad una richiesta, un oggetto può invocare un'operazione che può cambiare il suo stato.

Un *oggetto* ha due tipi di proprietà:

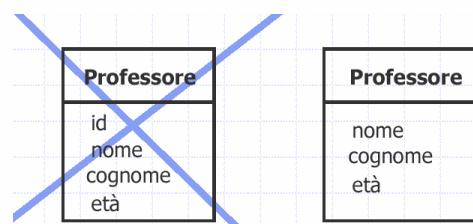
1. **attributi** (o variabili di istanza), che descrivono lo stato;
2. **operazioni** (o metodi), che descrivono il comportamento.

Un'**istanza** rappresenta un fenomeno e il suo nome è sottolineato: il nome deve contenere il nome della classe.



Un **attributo** è una proprietà statica di un oggetto e contiene un valore per ogni istanza. Inoltre, i nomi degli *attributi* devono essere unici all'interno di una classe. Il valore di un attributo non ha identità (non è un oggetto). Per ciascun *attributo* si può specificare il tipo ed un eventuale valore iniziale.

Infine, gli oggetti avranno una loro identità, non bisogna aggiungerla negli *attributi*.

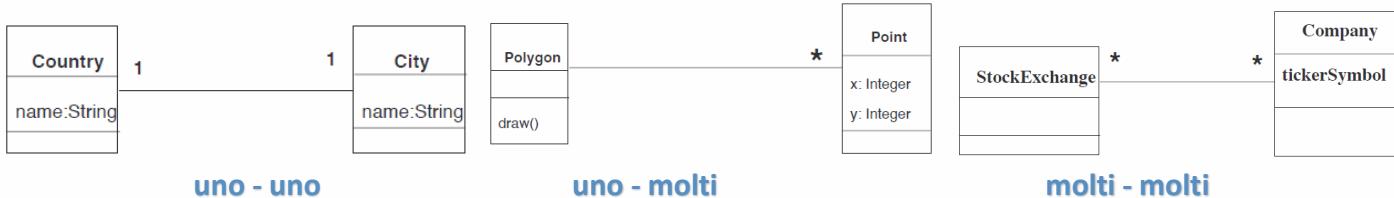


Gli **attributi derivati** sono attributi calcolati e non memorizzati: si usano quando i loro valori variano frequentemente e la correttezza (precisione) del valore è importante.

Un'**operazione** è un'azione che un oggetto esegue su un altro oggetto e che determina una reazione. Le *operazioni* operano sui dati incapsulati dell'oggetto. I tipi di operazione sono:

- **selettore** (query): accedono allo stato dell'oggetto senza alterarlo;
- **modificatore**: alterano lo stato di un oggetto.

Le **associazioni** denotano una relazione tra classi. La **molteplicità** di un'associazione è usata per indicare il numero di istanze di una classe. La molteplicità di un'associazione può essere:



Un **aggregazione** viene definita come una relazione non forte

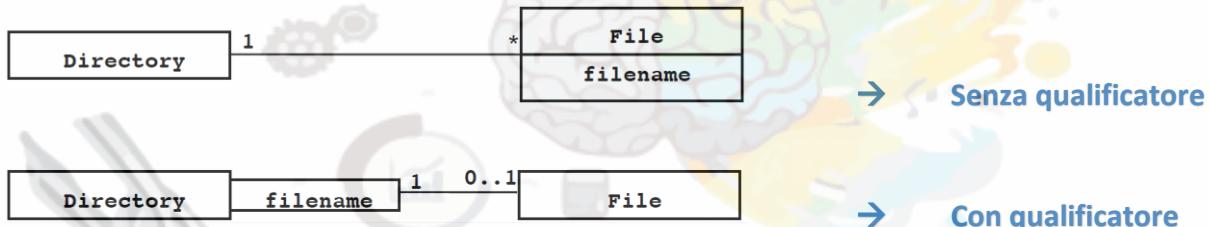
ovvero una relazione che collega la classe padre a più componenti, ovvero le classi figlie.



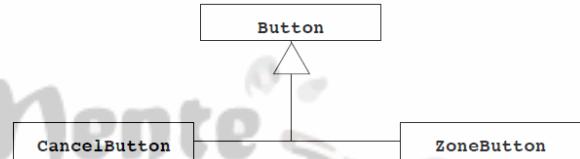
La **composizione** viene definita come una relazione forte ovvero esiste l'aggregato (la classe padre) solo se esistono le componenti.



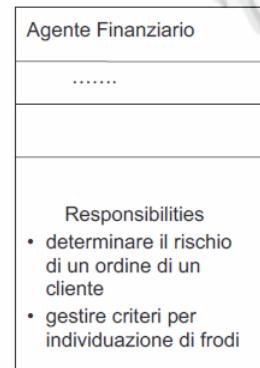
I **qualificatori** sono utilizzati per ridurre la molteplicità (o cardinalità) di un'associazione.



L'**ereditarietà** denota una gerarchia "di tipo". L'**ereditarietà** semplifica il modello di analisi introducendo una tassonomia (o gerarchia). Le classi figlie ereditano gli attributi e operazioni della classe madre.



Una **responsibility** è un contratto o una obbligazione di una classe. Questa è definita dallo stato e comportamento della classe. Una classe può avere un qualsiasi numero di responsabilità, ma una classe ben strutturata ha una o poche responsabilità.
Le **responsabilità** possono essere indicate, in maniera testuale, in una ulteriore sezione, sul fondo della icona della class.



È possibile specificare la **visibilità** di attributi e operazioni attraverso 3 livelli:

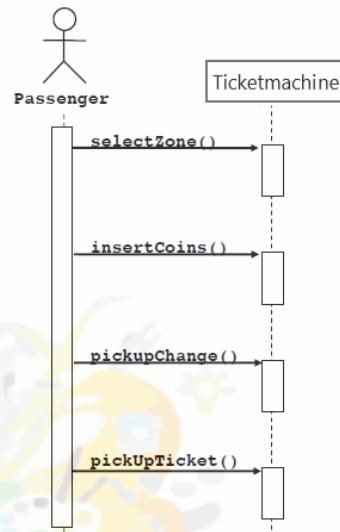
- **+ (public)**: qualsiasi altra classe con visibilità alla classe data può usare l'attributo/operazione (di default se nessun simbolo è indicato);
- **# (protected)**: qualsiasi classe discendente della classe data può usare l'attributo/operazione;
- **- (private)**: solo la classe data può usare l'attributo/operazione.



Sequence Diagrams (Diagramma di sequenza)

Il **Sequence diagram** è utilizzato per definire la sequenza di eventi di un caso d'uso durante la fase di analisi dei requisiti oppure durante il system design. Inoltre, è un **diagramma di interazione**: evidenzia come un caso d'uso è realizzato tramite la collaborazione di un insieme di oggetti.

- le **istanze** sono definite dai rettangoli;
- gli **attori** da omini stilizzati;
- le **lifelines** sono rappresentate da linee tratteggiate;
- i **messaggi** sono rappresentati da frecce;
- le **attivazioni** sono rappresentate da rettangoli stretti.



Il *sequence diagram* di UML è utile per identificare o trovare oggetti mancanti. La costruzione richiede tempo, ma ne vale la pena.

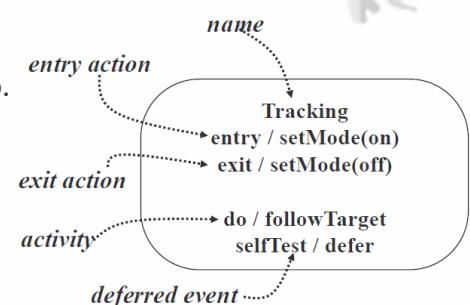
State (o Statechart) Diagrams

Gli **Statechart Diagrams** specificano il ciclo di vita di un oggetto, in particolare, descrivono una sequenza di stati di un oggetto in risposta a determinati eventi. Rappresentano anche le transizioni causate da un evento esterno e l'eventuale stato in cui esso viene riportato.

Ogni **diagramma di stato** inizia con un cerchio scuro che indica lo stato iniziale e termina con un cerchio bordato che denota lo stato finale.

Uno **stato** è una situazione in cui l'oggetto soddisfa qualche condizione, esegue attività o aspetta qualche condizione. Uno stato possiede alcuni attributi che sono opzionali:

- **nome**: una stringa (uno stato può essere anonimo);
- **entry / exit actions**: azioni eseguite all'ingresso / uscita dallo stato.
Non sono interrompibili e hanno una durata istantanea;
- **Transizioni interne**: non causano un cambiamento di stato;



- **Attività** dello stato (interrompibile e ha una durata significativa).
- **Eventi differenti**: non sono gestiti in quello stato ma da altri oggetti in un altro stato;
- **Sottostati**: struttura innestata di stati. Possono essere **disgiunti** (sequenzialmente attivi) o **concorrenti** (concorrentemente attivi).

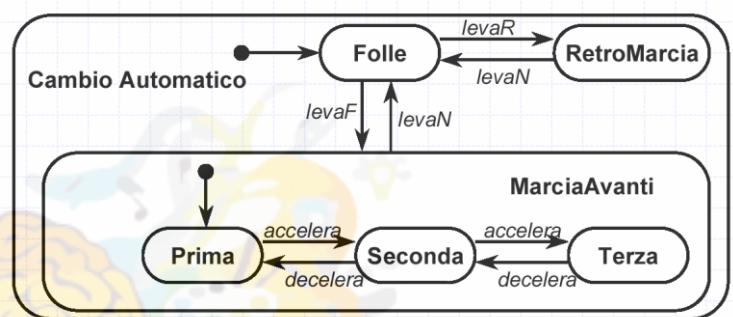
La **transizione** è un cambiamento da uno stato iniziale a uno finale. Possiamo avere una:

- **transizione esterna**: stato finale diverso da stato iniziale;
- **transizione interna**: stato finale uguale a stato iniziale.

Le transizioni possiedono alcuni attributi opzionali:

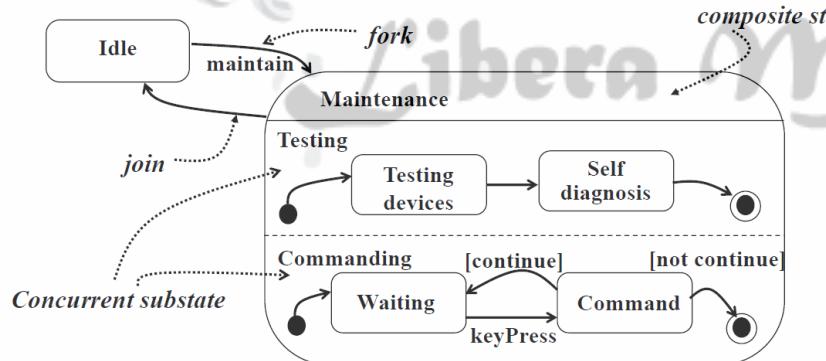
- **evento**: può essere un segnale, un messaggio da altri oggetti o un cambiamento;
- **condizione di guardia**: è una condizione booleana verificata all'avvio di una transizione. La transizione si verifica se l'evento accade e la condizione di guardia è vera.
- **Azione**: è eseguita durante la transizione e non è interrompibile; ha una durata istantanea.
- **Transizione senza eventi** (triggerless) si verificano quando:
 - **con guardia**: se la condizione di guardia diventa vera;
 - **senza guardia**: se l'attività interna allo stato iniziale è completata.

La **decomposizione OR** (o sottostato sequenziale) è un macro stato formato da una scomposizione in *OR* degli stati. I sottostati ereditano le transizioni dei loro superstati.



La **decomposizione AND** (o concurrent substates) riguarda più stati eseguiti in parallelo. Se un sottostato raggiunge lo stato finale prima dell'altro, quest'ultimo dovrà aspettare la fine dell'altro stato che ancora non ha terminato. Quando avviene una transizione in uno stato con *decomposizione AND*, il flusso di controllo subisce una *fork* (il flusso si dirama in due percorsi paralleli) per poi essere ricomposto in un unico flusso tramite una *join*.

Esempio:



All'inizio saremo in **idle**, successivamente tramite una *fork* avremo uno stato con due sottostati: **testing** e **commanding**. Quando i due sottostati saranno entrambi terminati verrà effettuata una *join* e torneremo nuovamente in **idle**.

Rappresentazione grafica dei vari elementi dei *state diagrams*:



Stato



Stato iniziale



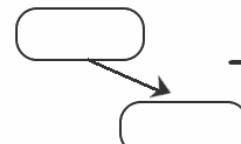
Stato finale



Decomposizione OR

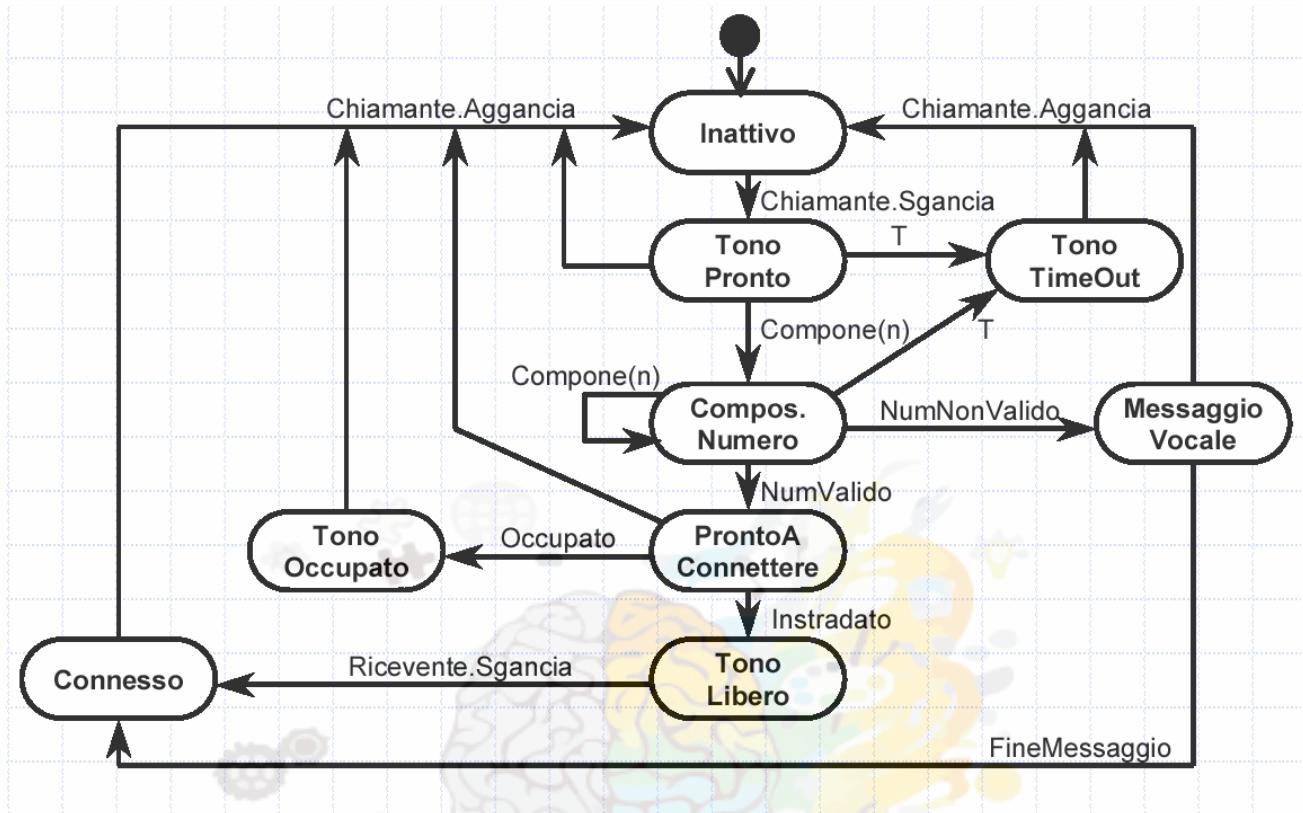


Decomposizione AND



Transizione

Esempio: telefonata



Activity Diagrams

Gli **activity diagrams** forniscono la sequenza di operazioni che formano un'attività più complessa.

Permettono di rappresentare processi paralleli e la loro sincronizzazione. Inoltre, possono essere considerati *State Diagram* particolari perché ogni stato contiene (è) un'azione. Un *Activity Diagram* può essere associato:

- a una classe;
- all'implementazione di un'operazione;
- ad uno Use Case;
- tutto quello che riguarda l'aspetto "funzionale".

Gli *activity diagrams* servono a rappresentare i workflow (flussi di lavoro) oppure la logica interna di un processo di qualunque livello. In altri termini, li utilizziamo per rendere più chiaro qualcosa che non è complesso o articolato bensì difficile da vedere leggendo il tutto.

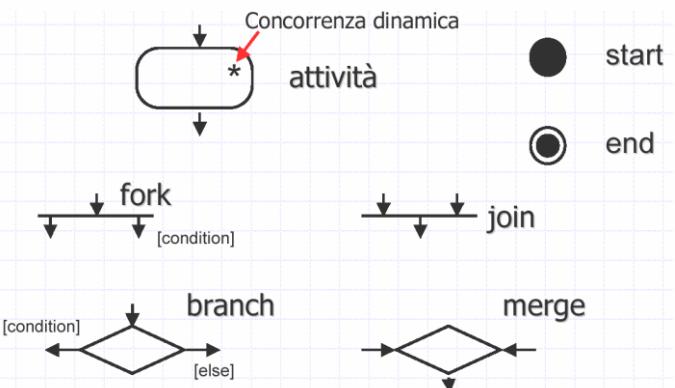
Quindi, sono utili per modellare:

- comportamenti sequenziali;
- non determinismo;
- concorrenza;
- sistemi distribuiti;
- business workflow;
- operazioni.

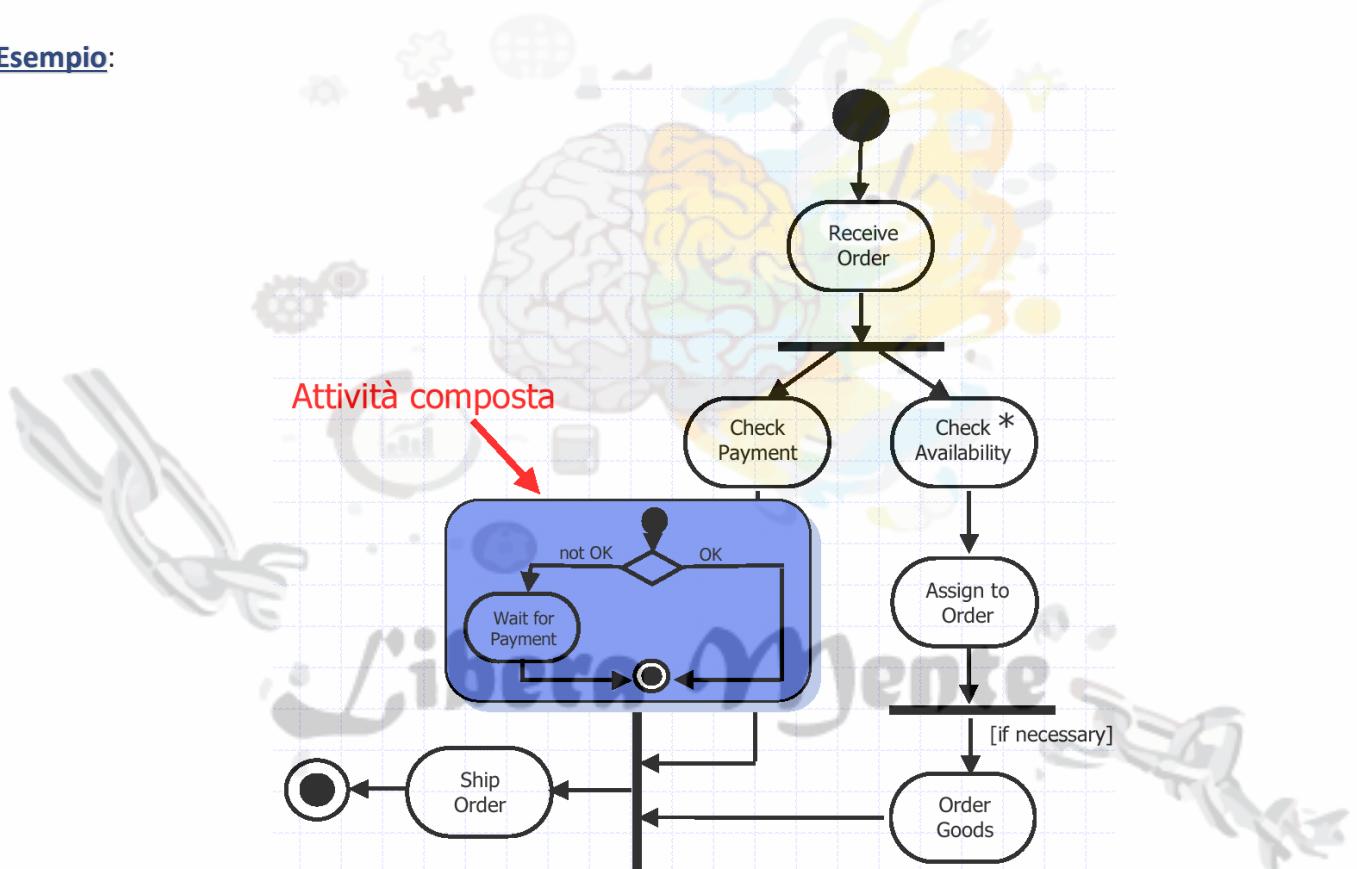
Tra gli elementi dell'*activity diagram* abbiamo:

- **attività**: un'esecuzione non atomica entro uno stato. Le attività possono essere gerarchiche;
- **transizione**: è il flusso di controllo tra due attività successive;
- **guard expression**: espressione booleana (condition) che deve essere verificata per attivare una transition;
- **branch**: specifica percorsi alternativi in base a espressioni booleane; un branch ha una unica transition in ingresso e due o più transition in uscita;
- **synchronization bar**: usata per sincronizzare flussi concorrenti;
- **fork**: per dividere un flusso su più transition verso action state concorrenti;

- **join**: unifica più transition da più action state concorrenti in una sola. E' importante bilanciare il numero di fork e di join.
- **Activity state**: stati non atomici (decomponibili ed interrompibili). Un'activity state può essere a sua volta rappresentato con un activity diagram.
- **Action state**: azioni eseguibili atomiche (non possono essere decomposte né interrotte). Un'action state può essere considerata come un caso particolare di activity state.

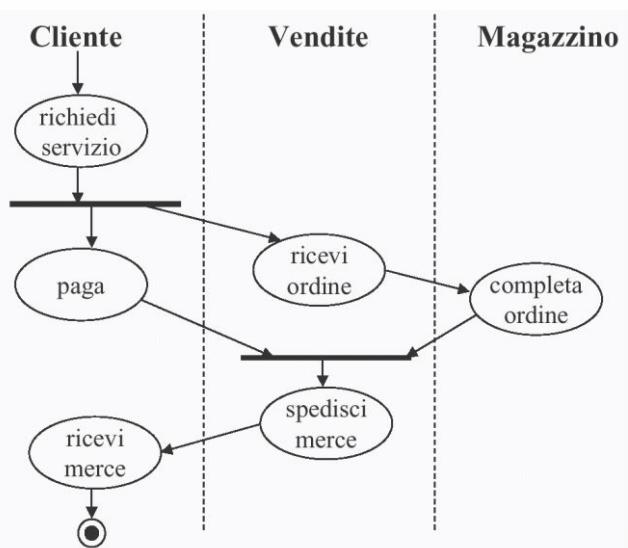


Esempio:



La **Swimlane** è un costrutto grafico che rappresenta un insieme partizionato di *action/activity*. Identificano le responsabilità relative alle diverse operazioni. In un *Business Model*, cioè in un'azienda, identificano le unità organizzative.

Una **swimlane**, è identificata da un nome univoco nel diagramma, mentre le *action/activity state* sono divise in gruppi e ciascun gruppo è assegnato allo *swimlane* dell'oggetto responsabile. L'ordine con cui gli *swimlane* si succedono non ha alcuna importanza. Le *transition* possono attraversare vari *swimlanes* per raggiungere i vari stati.

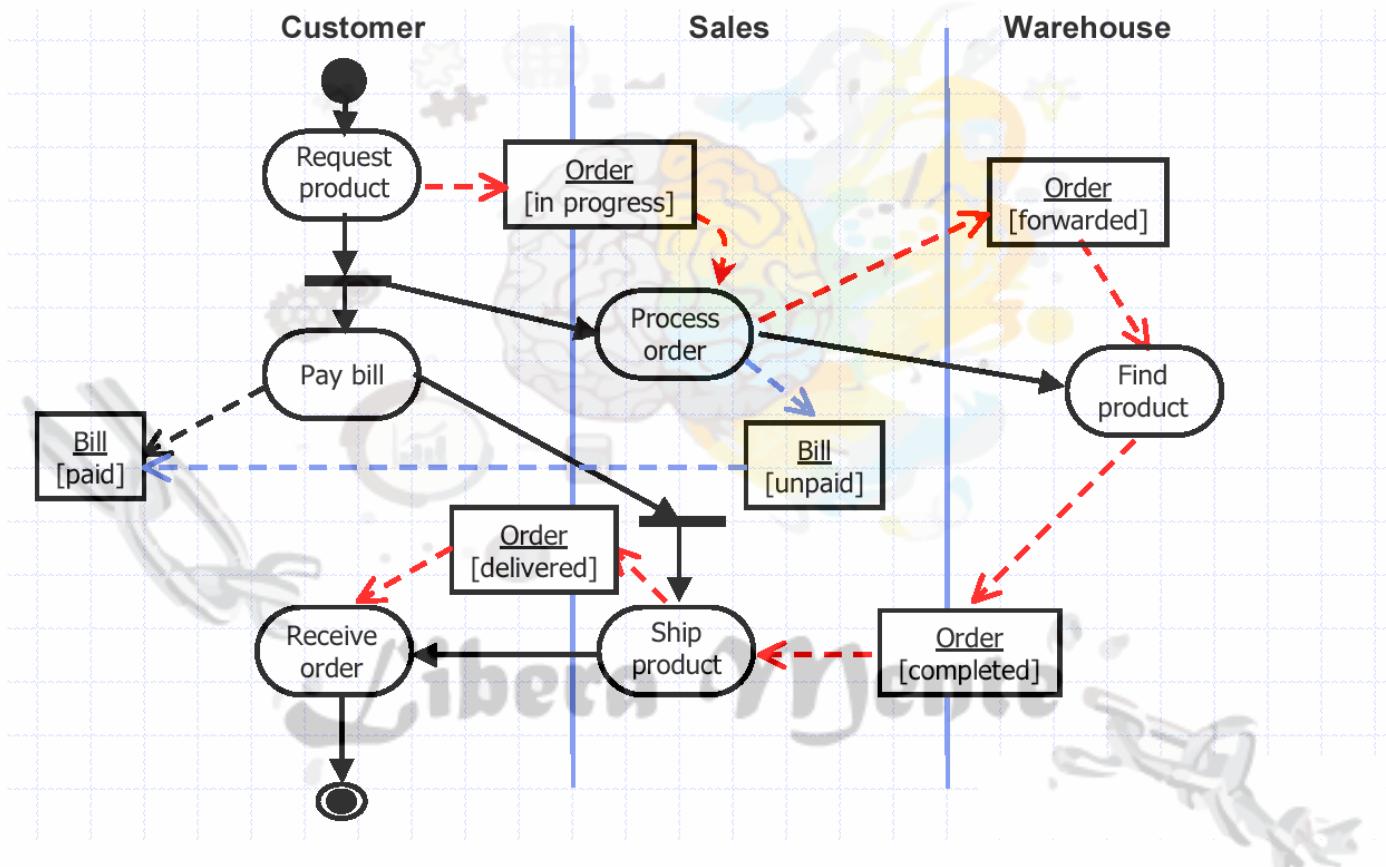


Gli *swimlanes* sono: **Cliente, Vendite, Magazzino**.

Gli **object flow** sono associazioni tra *action/activity state* e *oggetti*. Gli *object* possono essere:

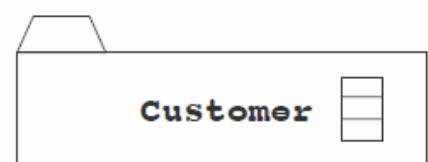
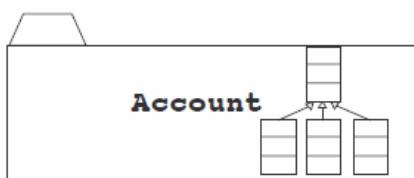
- **L'output di una action:** l'action crea l'object e graficamente abbiamo la freccia della relationship che punta verso l'object (**freccia blu**).
- **L'input di una action:** l'action ha bisogno dell'object e graficamente abbiamo la freccia che parte dall'object e punta alla relationship (**freccia rossa**).
- **manipolati da qualsiasi numero di action:** l'output di una action può essere l'input di un'altra action.
- **presenti più volte nello stesso diagramma:** ogni presenza indica un differente punto della vita dell'object.

Esempio:



Package Diagrams

I **packages** aiutano a organizzare i modelli UML creati nelle fasi precedenti, in particolare il loro obiettivo è di ridurre la complessità del sistema. Ad esempio, possiamo suddividere un sistema in sottosistemi e ognuno di questi sottosistemi è modellato come un *package*.



Un *package* raccoglie un insieme di classi che si occupano della parte funzionale del sistema ma può raccogliere anche altri diagrammi. In fase di Manutenzione e Testing le dipendenze diventano di vitale importanza perché le modifiche su un *package* potrebbero impattare anche su altre parti del sistema e di conseguenza bisogna modificare anche altri *package*.

Component Diagrams

I **component diagrams** rappresentano l'implementazione del sistema. Un **componente** rappresenta un pezzo "fisico" dell'implementazione di un sistema. Questi diagrammi definiscono le relazioni fra i componenti software che realizzano l'applicazione come file sorgenti, binari, eseguibili.

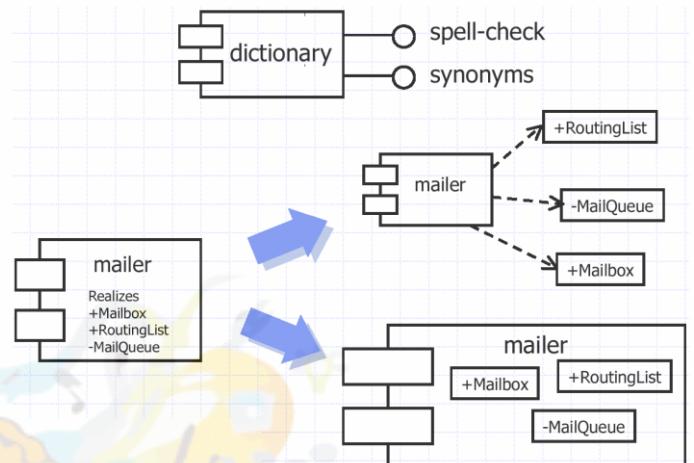
Un **componente** ha un nome e una locazione.

E' mostrato, tipicamente, con il solo nome; per le classi è possibile aggiungere altri dettagli.

I *componenti* possono essere raggruppati in *package*:

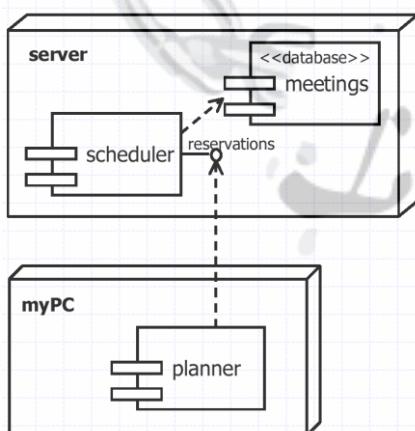
se i componenti sono file, i *package* sono

cartelle/directory. Le componenti collaborano tra di loro tramite interfacce.



Deployment Diagram

Il **deployment diagram**, anche detto *diagramma di allocazione* o di *dislocazione*, permette di rappresentare, a diversi livelli di dettaglio, l'architettura fisica del sistema. Permette anche di evidenziare la configurazione dei nodi run-time e dei componenti situati in questi nodi. Gli elementi del *deployment diagram* sono i **nodi** e le **connessioni** che connettono quest'ultimi.



Ciascun **nodo** è identificato da un nome. Un *nodo* può:

- riportare ulteriori dettagli in compartimenti addizionali o usando *tagged value*.
- può essere in connection con altri nodi, ed avere relationship con componenti e altri nodi.

Capitolo 5 – Raccolta dei requisiti

5.1 Ingegneria dei requisiti

L'ingegneria dei requisiti coinvolge due attività: **raccolta dei requisiti** e **analisi dei requisiti**.

La **raccolta dei requisiti** definisce il sistema in termini compresi dal cliente e richiede la collaborazione tra più gruppi di partecipanti con differenti background. Gli errori commessi durante questa fase sono difficili da correggere e vengono spesso notati nella fase di consegna. Alcuni errori possono essere: funzionalità non specificate o incorrecte o interfacce poco intuitive.

L'**analisi dei requisiti** definisce il sistema in termini compresi dallo sviluppatore.

Quindi, utenti e sviluppatori devono collaborare per scrivere il documento di **specifiche dei requisiti** (prodotto durante la raccolta dei requisiti) in linguaggio naturale per poi essere successivamente formalizzato e strutturato (in UML o altro) durante la fase di analisi per produrre il **modello di analisi**.

Il primo documento (**la specifica dei requisiti**) è utile al fine di favorire la comunicazione con il cliente e gli utenti, mentre il documento prodotto nell'analisi è usato dagli sviluppatori.

La **raccolta dei requisiti** e l'**analisi dei requisiti** si focalizzano sul punto di vista dell'utente e definiscono il confine del sistema da sviluppare, in particolare vengono specificate:

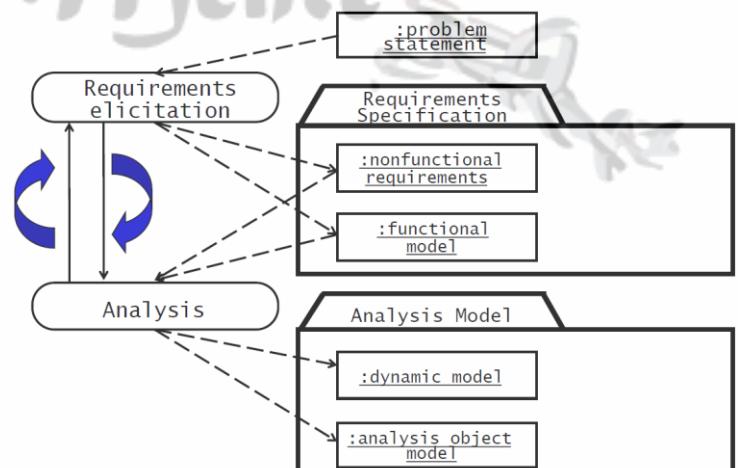
- Funzionalità del sistema;
- Interazione utente-sistema;
- Errori che il sistema deve gestire;
- Vincoli e condizioni di utilizzo.

Non fanno parte dell'attività di raccolta dei requisiti:

- la selezione delle tecnologie da usare per lo sviluppo;
- il progetto del sistema e le metodologie da usare;
- in generale tutto quello che non è direttamente visibile all'utente.

Inoltre, tramite la **raccolta** e l'**analisi dei requisiti** possiamo rispondere principalmente a due domande:

1. Come identifichiamo gli obiettivi del sistema?
2. Cosa c'è all'interno e all'esterno del sistema?



Inizialmente dovremo ridurre la complessità del problema attraverso 3 modalità:

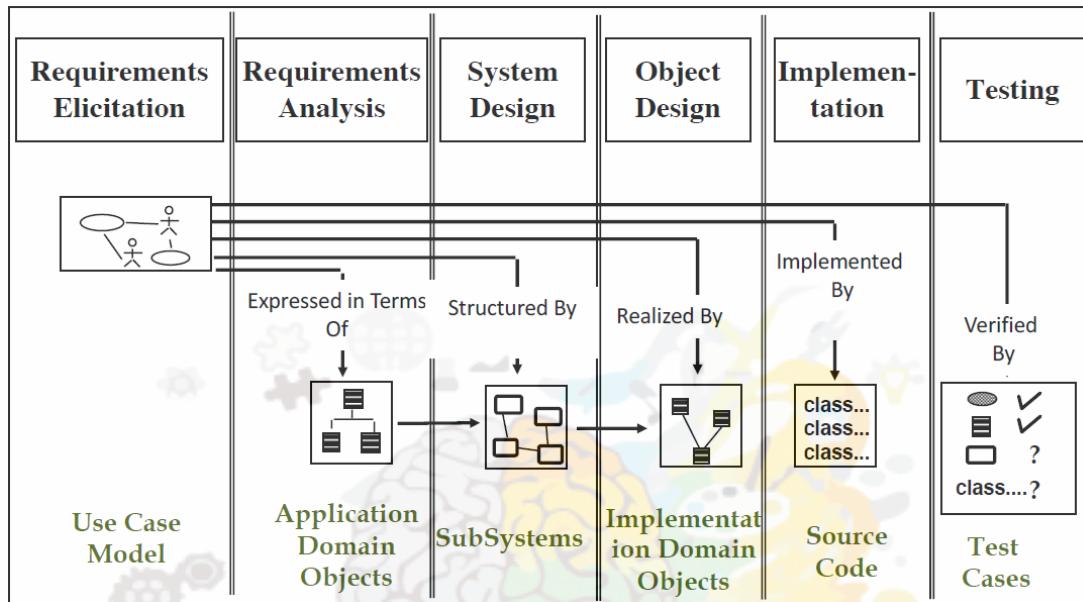
1. **l'astrazione**;
2. **la decomposizione** del problema in sottoproblemi (tecnica *divide et impera*);
3. **la gerarchia** (tecnica a livelli).

Per affrontare la **decomposizione** esistono due modi (che possono essere usati entrambi), ovvero attraverso la decomposizione:

1. **funzionale**;
2. **orientata agli oggetti**.

Quindi all'inizio potremmo procedere con una descrizione delle funzionalità (tramite gli use cases) per poi procedere trovando gli oggetti (modello a oggetti).

Questo ci porta a definire il **ciclo di vita dello sviluppo del software**: cioè un'insieme di attività e delle loro relazioni per supportare il sviluppo di un sistema software.



Documento di specifica dei requisiti

Il **documento di specifica dei requisiti** si focalizza sulla descrizione del sistema da sviluppare e viene stilato tramite il contributo del cliente, degli utenti e degli sviluppatori. Tale documento viene usato come **contratto tra cliente e sviluppatori**.

Il documento di **specifiche dei requisiti** viene poi formalizzato e strutturato durante la fase di analisi per produrre il **modello di analisi**: infatti, gli sviluppatori costruiscono un modello del dominio di applicazione “osservando” gli utenti nel loro ambiente.

Entrambi i documenti rappresentano la stessa informazione ma sono scritti usando linguaggi diversi:

- la **specifiche dei requisiti** è scritta in linguaggio naturale e supporta la comunicazione tra cliente e sviluppatori;
- il **modello di analisi** si basa su notazioni formali o semi-formali e supporta la comunicazione tra gli sviluppatori.

Problem Statement – Statement of Work

Il **problem statement** è sviluppato dal cliente come descrizione del problema che dovrà affrontare il sistema. Un *problem statement* descrive:

- La **situazione attuale** riguarda il problema da risolvere e la descrizione di uno o più scenari;
- Gli **obiettivi**;
- I **requirements** (funzionalità) che il nuovo sistema dovrebbe supportare;
- L'**ambiente** in cui verrà distribuito il sistema;
- **Deliverables** (risultati) attesi dal cliente;
- **Date di consegna**;
- Una serie di **criteri di accettazione** che andranno verificati per poter dire se il contratto, tra cliente e sviluppatore, è stato rispettato.

5.2 Classificazione dei requisiti

Le **specifiche dei requisiti** sono una sorta di contratto tra il cliente e gli sviluppatori e deve essere curata con attenzione in ogni suo dettaglio. Inoltre le parti del sistema che comportano un maggior rischio devono essere prototipate e provate con simulazioni per controllare la loro funzionalità e ed ottenere un riscontro dall'utente. Esistono varie tipologie di requisiti:

- ❖ **Requisiti funzionali:** descrivono le interazioni tra il sistema e l'ambiente esterno (utenti e sistemi esterni) indipendentemente dall'implementazione; in particolare vengono descritti i task dell'utente che il sistema deve supportare.
- ❖ **Requisiti non funzionali:** descrivono aspetti del sistema che non sono legati direttamente alle funzionalità del sistema. Ad esempio, sono *requisiti non funzionali* dettagli implementativi come la riusabilità, le performance o il supporto.
- ❖ **Vincoli (Constraints):** sono “pseudo requirements” e riguardano cose imposte o dal cliente o dall'ambiente.

Tutto ciò che **NON** riguarda i requisiti sono:

- La struttura del sistema e le tecnologie implementate;
- La metodologia di sviluppo;
- I linguaggi implementati;
- L'ambiente usato per sviluppare il software;
- La riusabilità: lo sviluppatore non deve essere vincolato nello scrivere delle funzioni che poi potrebbero essere riutilizzate per altro.

5.2.1 Validazione dei requisiti

I *requisiti* devono essere continuamente validati con il cliente e l'utente, la **validazione** degli stessi è un aspetto molto importante perché ha lo scopo di non tralasciare nessun aspetto.

I requisiti devono rispettare le seguenti caratteristiche:

- **Completezza:** devono essere presi in considerazione tutti i possibili scenari, inclusi i comportamenti eccezionali;
- **Consistenza:** i requisiti non devono contraddirsi;
- **Chiarezza:** deve essere definito un unico sistema e non si devono interpretare i vari requisiti in modi differenti;
- **Correttezza:** deve rappresentare il sistema di cui il cliente ha bisogno;
- **Realismo:** se il sistema può essere implementato in tempi ragionevoli;
- **Verificabilità:** se una volta che il sistema è stato implementato è possibile effettuare dei test;
- **Tracciabilità:** se ogni requisito può essere mappato con una corrispondente funzionalità del sistema;

I problemi che si possono avere con la *validazione dei requisiti* possono essere:

- I requisiti cambiano velocemente durante la loro raccolta;
- Incongruenze ad ogni modifica;
- È necessario il supporto di determinati tool.

5.2.2 Priorità dei requisiti

E' utile classificare i requisiti in base alla loro **priorità** per progettare meglio il sistema software:

- **Alta priorità**: sono i requisiti affrontati durante l'analisi, la progettazione e l'implementazione;
- **Priorità media**: requisiti affrontati durante l'analisi e la progettazione;
- **Bassa priorità**: requisiti affrontati solo durante l'analisi.

5.2.3 Greenfield engineering, re-engineering, interface engineering

Altri requisiti possono essere specificati in base alla sorgente delle informazioni.

Il **greenfield engineering** avviene quando lo sviluppo di un'applicazione parte da zero, senza alcun sistema preesistente.

Il **re-engineering** è un tipo di raccolta dei requisiti dove c'è un sistema preesistente che deve essere riprogettato a causa di nuove esigenze o nuove tecnologie.

L'**interface engineering** avviene quando è necessario riprogettare un sistema per farlo lavorare in un nuovo ambiente. Un esempio possono essere i sistemi legacy che vengono lasciati inalterati nelle interfacce.

5.3 Attività della raccolta dei requisiti

1. Identificare gli attori;
2. Identificare gli scenari;
3. Identificare i casi d'uso;
4. Raffinare i casi d'uso;
5. Identificare le relazioni tra gli attori e i casi d'uso;
6. Identificare gli oggetti partecipanti (verranno ripresi nella fase di analisi);
7. Identificare i requisiti non funzionali.

5.3.1 Identificare gli attori

Un **attore** è un entità esterna che comunica con il sistema e può essere un utente, un sistema esterno o un ambiente fisico.

Ogni attore ha un nome univoco ed una breve descrizione sulle sue funzionalità (es. Teacher: una persona; Satellite GPS: fornisce le coordinate della posizione).

Un modo molto semplice per identificare gli attori di un sistema è porsi le seguenti domande:

- Quali gruppi di utenti sono supportati dal sistema per svolgere il proprio lavoro, quali eseguono le principali funzioni del sistema, quali eseguono le funzioni di amministrazione e mantenimento?
- Con quale sistema hardware o software il sistema interagisce?

5.3.2 Identificare gli scenari

Uno **scenario** è una descrizione testuale, informale, concreta e sintetica di un evento o di una serie di azioni ed eventi. In altri termini vengono elencate una serie di passi che un utente svolge per ottenere quella funzionalità. Infatti, la descrizione è scritta dal punto di vista dell'utente finale. Uno *scenario* può includere testo, video, foto e story telling.

Ogni scenario deve essere caratterizzato da un nome, una lista dei partecipanti e un flusso di eventi.

Esistono vari tipi di scenari:

- **As-is-scenario:** sono usati per descrivere una situazione corrente. Vengono di solito usati nella raccolta dei requisiti di tipo re-engineering.
- **Visionary-Scenario:** utilizzato per descrivere funzionalità future del sistema. Vengono di solito usati nei progetti di tipo greenfield engineering e re-engineering.
- **Evaluation-Scenario:** descrivono le funzioni eseguite dagli utenti sul quale poi il sistema viene testato.
- **Training-Scenario:** sono tutorial per introdurre nuovi utenti al sistema.

L'**identificazione degli scenari** è una fase che avviene in stretta collaborazione con il cliente e l'utente.

Per poter **formulare gli scenari** bisogna porsi e porre all'utente le seguenti domande:

- Quali sono i compiti primari che l'attore vuole che svolga il sistema?
- Quali dati saranno creati/memorizzati/cambiati/cancellati o aggiunti dall'utente nel sistema?
- Di quali cambiamenti esterni l'attore deve informare il sistema?
- Di quali eventi/cambiamenti deve essere informato l'attore?

5.3.3 Identificare gli Use Cases

Un **caso d'uso** descrive una serie di interazioni che avvengono dopo un'inizializzazione da parte di un attore e specifica tutti i possibili scenari per una determinata funzionalità (visto in altri termini uno scenario è un'istanza di un caso d'uso).

Alla fine il risultato della raccolta dei requisiti sarà un **Use Case Model**, cioè l'insieme di tutti gli use cases che specificano tutte le funzionalità del sistema.

Per **trovare gli use cases** possiamo attraversare il sistema in **verticale**, cioè discutiamo in dettaglio con l'utente per capire le sue preferenze; oppure potremmo muoverci in **orizzontale**, cioè quando vogliamo capire lo scope del sistema discutendone con l'utente.

Quindi riassumendo, ci muoveremo:

- **verticalmente**, quando il sistema è chiaro nella sua interezza;
- **orizzontalmente**, quando non sappiamo con precisione cosa fa il sistema.

Se lo scope del sistema non è chiaro potremmo aiutarci con l'uso di **prototipi (mock-ups)** in cui rappresentiamo graficamente l'utilizzo del sistema.

Ogni *caso d'uso* contiene le seguenti informazioni:

1. Un **nome del caso d'uso** che dovrebbe includere dei verbi;
2. I **nomi degli attori** partecipanti che dovrebbero essere sostantivi;
3. Le **condizioni di ingresso/uscita** (entry/exit condition);
4. Un **flusso di eventi** in linguaggio naturale e informale;
5. Le **eccezioni** che possono verificarsi quando qualcosa va male e sono descritte in modo distinto e separato;
6. I **requisiti speciali** che includono i requisiti non funzionali e i vincoli.

Inoltre, molto importante è la **tracciabilità** degli use case.

Nel **flusso di eventi** del *caso d'uso* vengono distinti gli eventi iniziati dagli attori da quelli iniziati dal sistema in quanto quelli del sistema sono più a destra (con un tab) rispetto a quelli dell'attore.

Use Cases Associations

Le **dipendenze** tra gli *use cases* sono rappresentate mediante le **relazioni**. Le relazioni sono usate per ridurre la complessità: vengono dettagliati gli elementi che sono manipolati dal sistema, dettagliate le interazioni a basso livello tra l'attore e il sistema, specificati i dettagli su chi può fare cosa, aggiunte eccezioni non presenti.

I tipi di relazione degli *use cases* sono 3:

1. **<<extend>>**: usata per estendere uno *use case* A in modo da ottenere un nuovo *use case* B con le funzionalità di A più altre specificate solo in B. L'arco è tratteggiato con l'etichetta <<extend>> e con la linea rivolta verso il caso eccezionale. Inoltre:
 - la condizione che attiva lo *use case* B è inserito nella *entry condition* dello *use case* B;
 - non occorre modificare lo/gli *use case* che vengono estesi;
 - l'attivazione può avvenire in un punto qualsiasi del flusso di eventi dello *use case* base;
 - sono spesso situazioni eccezionali (failure, cancel, help...).
2. **<<include>>**: usata per scomporre un caso d'uso in dei casi d'uso più semplici. La freccia dell'arco è tratteggiata, etichettata con <<include>> ed è rivolta verso il caso d'uso che viene usato. Ad esempio, se il caso d'uso A **include** il caso d'uso B, allora possiamo dire che A delega parte delle attività che deve svolgere all'*use case* B.
3. **Generalizzazione**: usata per fattorizzare più *use cases* che hanno dei comportamenti in comune. Gli use cases figli **ereditano** tutti i comportamenti dello use case padre ma aggiungono anche delle nuove funzionalità.

5.3.4 Identificare gli oggetti partecipanti negli Use Cases

Durante la fase di **raccolta dei requisiti** utenti e sviluppatori devono creare un glossario di termini usati nei casi d'uso. Si parte dalla terminologia che gli utenti hanno (quella del dominio dell'applicazione) e successivamente si negoziano cambiamenti. Il glossario creato è lo stesso che viene incluso nel manuale utente finale.

I termini possono rappresentare oggetti, procedure, sorgenti di dati, attori e casi d'uso. Ogni termine ha una piccola descrizione e deve avere un nome univoco e non ambiguo.

5.4 Guida alla scrittura dei Casi d'Uso

- I nomi dei casi d'uso dovrebbero iniziare con i verbi;
- I nomi di Attori dovrebbero essere sostantivi;
- I confini del sistema dovrebbero essere chiari;
- Le relazioni causali tra passi successivi dovrebbero essere chiare;
- Non bisogna usare la forma passiva;
- Un caso d'uso dovrebbe descrivere una transizione utente completa;
- Le eccezioni dovrebbero essere descritte separatamente;
- Un caso d'uso non dovrebbe descrivere un'interfaccia del sistema (meglio attraverso prototipi mock-up);
- Un caso d'uso non dovrebbe superare due o tre pagine; se è troppo lungo si usano le relazioni di **include** o **extend**.

5.5 Requisiti

I **requisiti** esprimono un bisogno, dei vincoli e delle associazioni. Sono scritti in linguaggio naturale e comprendono un soggetto, un verbo e un complemento: bisogna quindi identificare prima l'oggetto del requisito e poi capire che cosa si deve fare.

La forma da rispettare è la seguente: [Condition] [Subject] [Action] [Object] [Constraint].

Devono essere evitate le ambiguità, gli aggettivi superlativi, frasi vaghe o un linguaggio “amichevole”.

Inoltre, vanno evitati gli avverbi e gli aggettivi, le frasi al negativo e le frasi di comparazione (alto come, migliore di, et.).

Alcune qualità dei *requisiti* sono:

- **l'usabilità**: la facilità con cui gli attori possono utilizzare un sistema per eseguire una funzione, inoltre, l'usabilità deve essere **misurabile**;
- **la robustezza**: la capacità di un sistema di funzionare anche se l'utente inserisce un input sbagliato o se ci sono cambiamenti nell'ambiente;
- **la disponibilità**: il rapporto tra il tempo di attività previsto di un sistema e il tempo di down previsto.

Modello FURPS+: Requisiti non Funzionali

Chiamati anche **requisiti di qualità**, sono:

- **Usabilità**: facilità per l'utente di imparare ad usare il sistema, e capire il suo funzionamento.
Includono:
 - convenzioni adottate per le interfacce utenti;
 - portata dell'Help in linea;
 - livello della documentazione utente.
- **Affidabilità**: capacità di un sistema o di una componente di fornire la funzione richiesta sotto certe condizioni e per un periodo di tempo. Includono:
 - un accettabile tempo medio di fallimento;
 - l'abilità di scoprire specificati difetti o di sostenere specificati attacchi alla sicurezza.
- **Performance**: riguardano attributi quantificabili del sistema come:
 - **tempo di risposta**, quanto velocemente il sistema reagisce a input utenti;
 - **throughput**, quanto lavoro il sistema riesce a realizzare entro un tempo specificato;
 - **disponibilità**, il grado di accessibilità di una componente o del sistema quando è richiesta;
 - accuratezza.
- **Supportabilità**: riguardano la semplicità di fare modifiche dopo il deployment. Includono:
 - **adattabilità**, l'abilità di cambiare il sistema per trattare concetti addizionali del dominio di applicazione;
 - **mantenibilità**, l'abilità di cambiare il sistema per trattare nuove tecnologie e per far fronte a difetti.

Modello FURPS+: Pseudo requisiti

Chiamati anche **vincoli**, sono:

- **Implementazione**: vincoli sull'implementazione del sistema, incluso l'uso di tool specifici, linguaggi di programmazione, o piattaforme hardware;
- **Interfacce**: vincoli imposti da sistemi esterni, incluso sistemi legacy e formati di scambio;
- **Operazioni**: vincoli sull'amministrazione e sulla gestione del sistema;
- **Packaging**: vincoli sulla consegna reale del sistema;
- **Legali**: riguardano licenze, regolamentazioni, e certificazioni

5.6 Gestire la raccolta dei requisiti

Uno dei metodi per negoziare le specifiche con il cliente è il **Joint Application Design (JAD)**, una completa specifiche dei requisiti che include definizioni dei dati, flussi di lavoro, e descrizione delle interfacce. E' stato sviluppato da *IBM*, e si compone di 5 attività:

6. **Definizione del progetto**: vengono intervistati il cliente e il project manager e vengono determinati gli obiettivi del progetto (output: Management Definition Guide).
7. **Ricerca**: vengono intervistati gli utenti attuali e quelli futuri e vengono raccolte informazioni sul dominio di applicazione e descritti ad alto livello i casi d'uso (output: Session Agenda e Preliminary Specification).
8. **Preparazione**: si prepara una sessione, un documento di lavoro che è un primo abbozzo del documento finale, un agenda della sessione e ogni altro documenti cartaceo utile che rappresenta informazioni raccolte durante la *Ricerca*.
9. **Sessione**: viene guidato il team nella creazione della specifica dei requisiti e il team si accorda sugli scenari, i casi d'uso e interfaccia utente mock-up.
10. **Documento finale**: viene rivisto il documento finale, rivedendo i documenti di lavoro per includere tutte le decisioni prese durante la sessione. Il documento è rivisto durante la sessione (1-2 ore) e completato.

Tracciabilità

Un altro aspetto importante è la **tracciabilità**, l'abilità di avere una visione chiara del progetto e rendere meno complessa e lunga un'eventuale fase di modifica ad un aspetto del sistema. Per fare ciò è possibile creare dei **collegamenti** tra i documenti per identificare meglio le dipendenze tra le componenti del sistema. La tracciabilità viene implementata mediante una **matrice di tracciabilità**.

RAD

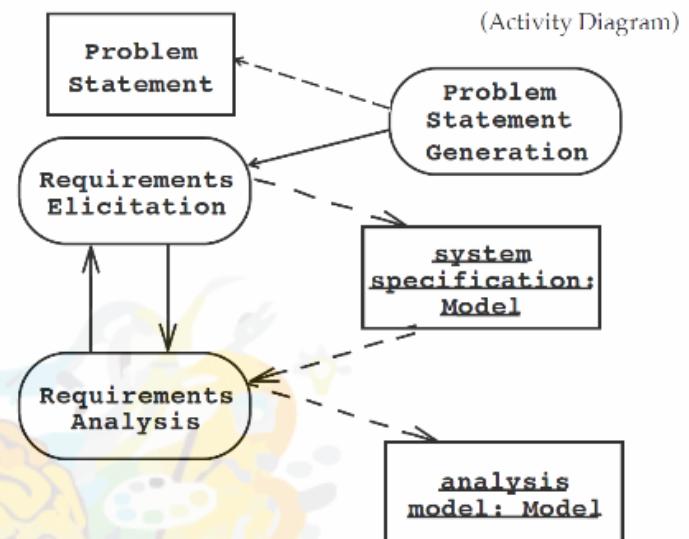
Il documento dell'analisi dei requisiti (RAD) contiene la **raccolta dei requisiti** e l'**analisi dei requisiti** ed è il documento finale del progetto, serve come base contrattuale tra il cliente e gli sviluppatori.

Il **RAD** descrive completamente il sistema in termini di requisiti funzionali e non funzionali; i partecipanti coinvolti sono: cliente, utenti, project manager, analisti del sistema, progettisti.

La prima parte del documento, che include Use Case e requisiti non funzionali, è scritto durante la raccolta dei requisiti, mentre, la formalizzazione della specifica in termini di modelli degli oggetti è scritto durante l'analisi.

Riassunto: Documento di Analisi dei Requisiti (RAD)

1. Introduction
 - 1.1. Purpose of the system
 - 1.2. Scope of the system
 - 1.3. Objectives and success criteria of the project
 - 1.4. Definition, acronyms, and abbreviations
 - 1.5. References
 - 1.6. Overview
2. Current system
3. Proposed system
 - 3.1. Overview
 - 3.2. Functional requirements
 - 3.3. Nonfunctional requirements
 - 3.4. System models
 - 3.4.1. Scenarios
 - 3.4.2. Use case model
 - 3.4.3. Object model **(during analysis)**
 - 3.4.4. Dynamic model **(during analysis)**
 - 3.4.5. User interface – navigational path and screen mock-up
4. Glossary



L'analisi dei requisiti produce in output il **modello dei requisiti** descritto dai seguenti prodotti:

- **Requisiti non funzionali e vincoli**: quali tempo di risposta massimo, minimo throughput, affidabilità, piattaforma per il sistema operativo, etc.
- **Use Case model**: descrive le funzionalità del sistema dal punto di vista degli attori.
- **Object model**: descrive le entità manipolate dal sistema.
- **Un sequence diagram per ogni use case**: mostra la sequenza di interazioni fra gli oggetti che partecipano al caso d'uso.

5.7 User Stories

Una **user story** è una descrizione informale e in linguaggio naturale delle funzionalità di un sistema software evidenziando **chi?**, **cosa?**, **perché?**. Sono scritte dal punto di vista di un utente finale e possono essere registrati su schede, post-it o digitalmente nel software di gestione dei progetti. A seconda del progetto, le *user stories* possono essere scritte da diverse parti interessate come cliente, utente, manager o team di sviluppo.

Capitolo 6 – Analisi dei requisiti

6.1 Analisi dei requisiti

L'**analisi dei requisiti** è finalizzata a produrre un modello del sistema chiamato **modello di analisi** che deve essere corretto, completo, consistente e non ambiguo. Infatti, le **ambiguità** nascono dal modo in cui si comunica oppure dalle assunzioni fatte dagli autori della specifica e non descritte nei dettagli: identificato un problema nella specifica, occorre risolverlo acquisendo maggiore informazioni dal cliente e dall'utente.

L'*analisi dei requisiti* ha come obiettivo quello di tradurre le specifiche dei requisiti in un modello del sistema formale o semiformale:

- formalizzando e strutturando i requisiti si acquisisce maggiore conoscenza ed è possibile scoprire errori nelle richieste;
- la formalizzazione costringe a risolvere subito questioni difficili che altrimenti sarebbero rimandate.

Quindi, viene costruito un modello che descrive il **dominio di applicazione**. Il **modello di analisi** descrive come gli attori e il sistema interagiscono per manipolare il modello del dominio di applicazione.

Poiché il modello di analisi può non essere comprensibile al cliente e all'utente è necessario modificare anche il **documento di specifica delle richieste**.

Il **modello dell'analisi** è composto da 3 modelli individuali:

1. Il **modello funzionale** rappresentato da casi d'uso e scenari;
2. Il **modello ad oggetti** dell'analisi rappresentato da diagrammi di classi e diagrammi ad oggetti;
3. Il **modello dinamico** rappresentato da diagrammi a stati e sequence diagram.

Gli use case e gli scenari prodotti nella fase di raccolta dei requisiti sono raffinati per produrre il modello ad oggetti e il modello dinamico.

Il modello ad oggetti

Il **modello ad oggetti** rappresenta il sistema dal punto di vista dell'utente; si focalizza sui concetti manipolati dal sistema, le loro proprietà e le loro relazioni. Viene rappresentato con un UML class diagram includendo operazioni, attributi e classi.

Il modello dinamico

Il **modello dinamico** si focalizza sul comportamento del sistema utilizzando:

- **sequence diagram**: rappresentano l'interazioni di un insieme di oggetti nell'ambito di un singolo caso d'uso;
- **diagrammi a stati**: rappresentano il comportamento di un singolo oggetto o di alcuni oggetti strettamente accoppiati.

Lo scopo del *modello dinamico* è quello di assegnare le responsabilità ad ogni singola classe e quindi identificare nuove classi, nuove associazioni e nuovi attributi.

Sia il **modello ad oggetti** che il **modello dinamico** rappresentano concetti a livello utente e non componenti sottostare che poi verranno implementate a livello di codice.

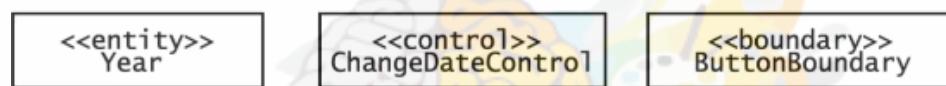
Modello ad oggetti

6.2 Entity, Boundary e Control object

Il **modello ad oggetti** è costituito da oggetti di tipo **entity**, **boundary** e **control**.

- Gli **oggetti entity** rappresentano **informazioni persistenti** tracciate dal sistema (oggetti del dominio di applicazione, “oggetti business”).
- Gli **oggetti boundary** rappresentano **l’interazione tra attore e sistema** (oggetti relativi all’interfaccia utente, oggetti dell’interfaccia dei dispositivi, oggetti di interfaccia del sistema).
- Gli **oggetti di controllo** realizzano i casi d’uso, rappresentano il controllo dei task eseguiti dal sistema, contengono la logica e determinano l’ordine dell’interazione degli oggetti.

UML fornisce il **meccanismo degli stereotipi** per consentire di aggiungere tale meta-informatione agli elementi di modellazione utilizzando le notazioni: **<<entity>>**, **<<control>>**, **<<boundary>>**.



Esistono altre notazioni per rappresentare gli stereotipi:



L’approccio **three-object-type** porta a modelli che sono più flessibili e facili da modificare: l’interfaccia del sistema (rappresentata da oggetti boundary) è più soggetta a cambiamenti rispetto alle funzionalità (rappresentate da oggetti entity e control).

6.3 Attività dell’analisi (trasformare un Use Case in oggetti)

Le attività che consentono di trasformare gli use case e gli scenari della raccolta dei requisiti in un modello di analisi sono:

1. Identificare gli *Oggetti Entity*;
2. Identificare gli *Oggetti Boundary*;
3. Identificare gli *Oggetti Control*;
4. Mappare gli Use Case in oggetti con i sequence diagram;
5. Identificare le associazioni;
6. Identificare le aggregazioni;
7. Identificare gli attributi;
8. Modellare il comportamento e gli stati di ogni oggetto;
9. Rivedere il modello dell’analisi.

6.3.1 Identificare gli Oggetti Entity

Per identificare gli oggetti partecipanti al modello dell’analisi bisogna prendere in considerazione quelli identificati durante la specifica di requisiti.

Visto che il documento della specifica dei requisiti è scritto in linguaggio naturale, è frequente riscontrare imprecisioni nel testo, oppure dei sinonimi sulla notazioni che possono indurre gli sviluppatori a considerare male gli oggetti.

Per limitare gli errori, gli sviluppatori possono usare delle euristiche come quella di **Abbott** che analizza il linguaggio naturale per identificare oggetti, attributi, associazioni dalla specifica dei requisiti.

Il **vantaggio** è che ci si focalizza sui termini dell’utente.

Gli **svantaggi**, invece, sono:

- Il linguaggio naturale è impreciso, anche il modello ad oggetti derivato rischia di essere impreciso;
- La qualità del modello dipende fortemente dallo stile di scrittura dell'analista;
- Ci possono essere molti più sostanzivi delle classi rilevanti, corrispondenti a sinonimi o attributi.

Testo	Modello ad oggetti
Nomi propri	Istanze
Nomi comuni	Classi
Verbi di fare	Operazioni
Verbi essere	Ereditarietà
Verbi avere	Aggregazioni
Verbi modali (es. deve essere)	Costanti
Aggettivi	Attributi

Oltre a quella di *Abbott* è possibile usare questa ulteriore euristica che suggerisce di dare peso alle seguenti caratteristiche dell'analisi dei requisiti:

- **Termini** che gli sviluppatori e gli utenti hanno bisogno di chiarire per comprendere gli use case;
- **Sostanzivi** ricorrenti nei casi d'uso;
- **Entità** del mondo reale che il sistema deve considerare;
- **Attività** del mondo reale che il sistema deve considerare;
- **Sorgenti** o **destinazioni** di dati.

Per ogni *oggetto entity* identificato:

- Si assegna un **nome** (univoco) e una **breve descrizione**. Per gli *oggetti Entity* è opportuno utilizzare gli stessi nomi utilizzati dagli utenti e dagli specialisti del dominio applicativo.
- Si individuano **attributi** e **responsabilità** (non tutti, soprattutto non quelli ovvii).
- Il processo è iterativo e varie revisioni saranno richieste: quando il modello di analisi sarà stabile sarà necessario fornire una descrizione dettagliata di ogni oggetto.

6.3.2 Identificare gli Oggetti Boundary

Gli **oggetti boundary** rappresentano l'interfaccia del sistema con l'attore. In ogni use case, ogni attore dovrebbe interagire con almeno un *oggetto boundary*.

Gli *oggetti boundary* raccolgono informazioni dall'attore e le traducono in un formato che può essere usato dagli *oggetti entity* e *control*.

Inoltre, essi non descrivono in dettaglio gli aspetti visuali dell'interfaccia utente: ad esempio specificare scroll-bar o menu-item (questo può essere descritto nei mockup).

Per scovare gli oggetti boundary è possibile anche in questo caso usare un'euristica:

- Identificare i **controlli della UI** di cui l'utente ha bisogno per iniziare un caso d'uso;
- Identificare i **form** di cui gli utenti hanno bisogno per inserire dati nel sistema;
- Identificare **messaggi e notifiche** che il sistema deve fornire all'utente;
- Non modellare aspetti visuali dell'interfaccia con oggetti boundary;
- Usare sempre il termine utente finale per descrivere le interfacce.

6.3.3 Identificare gli Oggetti Control

Gli **oggetti Control** sono responsabili del coordinamento tra gli *oggetti entity* e *boundary* e lo scopo è quello di prendere informazioni dagli *oggetti boundary* e inviarli agli *oggetti entity*. Un oggetto control viene creato all'inizio di un caso d'uso e termina alla fine di questo.

Anche questo come gli *oggetti entity* e *boundary* si basa su delle euristiche:

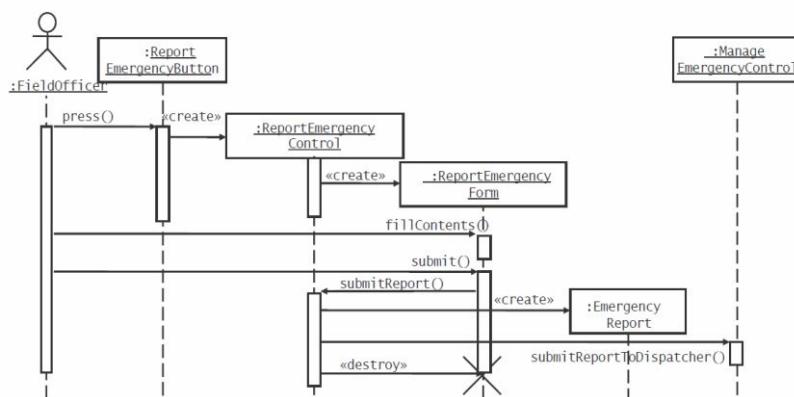
- Identificare un *oggetto control* per ogni caso d'uso;
- Identificare un *oggetto control* per ogni attore nel caso d'uso;
- La vita di un *oggetto control* deve corrispondere alla durata di un caso d'uso o di una sessione utente.
Se è difficile identificare l'inizio e la fine dell'attivazione di un *oggetto Control*, il corrispondente use case probabilmente non ha delle entry ed exit condition ben definite.

6.3.4 Mappare Use Case in Oggetti con Sequence Diagram

Mappare i casi d'uso in *sequence diagram* serve per mostrare il comportamento tra gli oggetti partecipanti e la loro interazione col sistema.

I *sequence diagram* non sono comprensibili all'utente ma sono uno strumento più preciso di supporto agli sviluppatori. In un *sequence diagram*:

- Le **colonne** rappresentano gli oggetti che partecipano al caso d'uso:
 - La prima colonna rappresenta l'attore che inizia il caso d'uso;
 - La seconda colonna è l'*oggetto boundary* con cui l'attore interagisce per iniziare il caso d'uso;
 - La terza colonna è l'*oggetto control* che gestisce il resto del caso d'uso;
 - Gli *oggetti control* creano altri *oggetti boundary* e possono interagire con altri *oggetti Control*.
- Le frecce orizzontali tra le colonne rappresentano messaggi o stimoli inviati da un oggetto ad un altro.
- La ricezione di un messaggio determina l'attivazione di un'operazione. L'attivazione è rappresentata da un rettangolo da cui altri messaggi possono prendere origine. La lunghezza del rettangolo rappresenta il tempo durante il quale l'operazione è attiva. Un'operazione è un servizio fornito ad altri oggetti.
- Per quanto riguarda la vita degli oggetti:
 - Il tempo procede verticalmente dall'alto al basso;
 - Al top del diagramma si trovano gli oggetti che esistono prima del 1° messaggio inviato;
 - Oggetti creati durante l'interazione sono illustrati con il messaggio <<create>>;
 - Oggetti distrutti durante l'interazione sono evidenziati con una croce;
 - La linea tratteggiata indica il tempo in cui l'oggetto può ricevere messaggi.



Mediante i *sequence diagram* è possibile trovare comportamenti o oggetti mancanti. Nel caso manchi qualche entità è necessario ritornare ai casi d'uso, ridefinire le parti mancanti e tornare a questa fase per ricreare il *sequence diagram*.

6.3.5 Identificare le Associazioni (in un Class Diagram)

Il link è un'istanza di un'associazione e stabilisce una relazione tra oggetti. Un'associazione descrive un insieme di link come una classe descrive un'insieme di oggetti. Un associazione tra due classi è di default un mapping bidirezionale.



Ad esempio, la *classe A* accede alla *classe B* e viceversa.

Ogni associazione possiede un:

- **nome**: unico ed identificabile;
- **ruolo**: il quale identifica la funzione di ogni classe;
- **molteplicità**: che indica il numero di istanze possibili (1-to-1, many-to-1, 1-to-many).

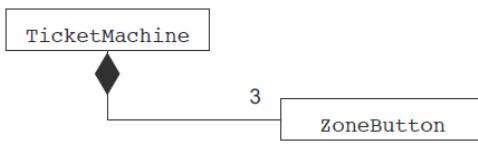
Un'utile **euristica** per trovare le associazioni è la seguente:

- Esaminare i verbi nelle frasi (ha, è parte di, gestisce, riporta a, è iniziata da, è contenuta in, parla di, include...);
- Nominare in modo preciso i nomi delle associazioni e i ruoli;
- Eliminare associazioni che possono essere derivate da altre associazioni;
- Non preoccuparsi della molteplicità delle associazioni fino a quando l'insieme delle associazioni non è stabile;
- Troppe associazioni rendono il modello degli oggetti "illeggibile".

6.3.6 Identificare le Aggregazioni (in un Class Diagram)

L'aggregazione identifica che un oggetto è parte di un altro oggetto o lo contiene. Nella notazione UML, l'aggregazione viene identificata da un rombo.

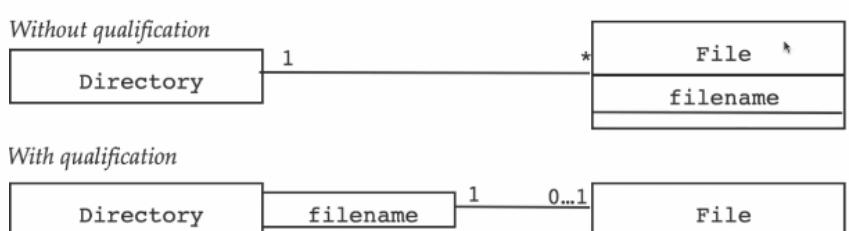
Le associazioni di aggregazione aggiungono informazioni al modello di analisi. Se non si è sicuri che l'associazione che si sta descrivendo sia un'aggregazione o meno è meglio modellarla come un'associazione **1-to-Many** e poi rivederla quando si ha maggiore conoscenza del dominio.



Una forma più forte dell'aggregazione è la **composizione** in cui un aggregato (il padre dell'aggregazione) esiste solo se sono presenti le componenti (i figli di un'aggregazione). E' raffigurato mediante un rombo pieno.

6.3.7 Identificare i Qualificatori (in un Class Diagram)

Gli **qualificato** sono usati per ridurre la molteplicità di un'associazione.



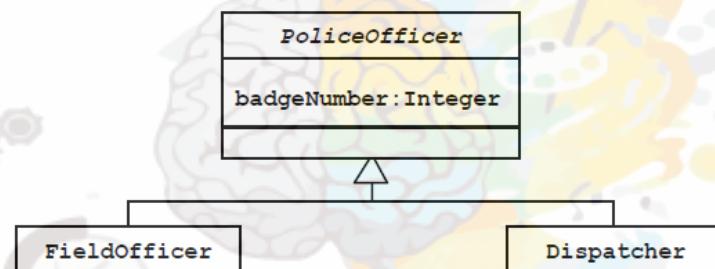
Generalizzazione e specializzazione

- **Ereditarietà:** consente di organizzare concetti in gerarchie:
 - al top della gerarchia concetti più generali;
 - al bottom concetti più specializzati.
- **Generalizzazione:** attività di modellazione che identifica concetti astratti da quelli di più basso livello.
- **Specializzazione:** attività che identifica concetti più specifici da quelli di più alto livello.

Come risultato sia della specializzazione che della generalizzazione abbiamo la **specificità di ereditarietà** tra concetti.

Modellare relazione di ereditarietà tra gli oggetti

Modella una gerarchia: “è uno di”. Tale notazione permette di condividere gli aspetti in comune tra le classi conservando, però, le loro differenze. In UML tale costrutto è rappresentato mediante un **arco con un triangolo**.



6.3.8 Identificare gli attributi (in un Class Diagram)

Gli **attributi** sono proprietà individuali degli oggetti. Devono essere identificati solo gli attributi rilevanti al sistema. Tali attributi sono identificati tramite:

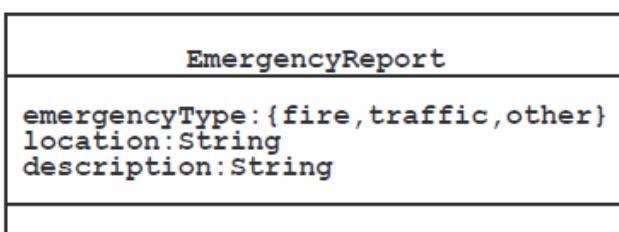
- **Nome**, che lo identifica;
- **Breve descrizione**;
- **Tipo**, che descrive i valori possibili che può assumere.

Gli attributi possono essere aggiunti anche dopo che l’analisi è finita. Infatti, gli attributi non sono direttamente legati alle funzionalità del sistema, per questo gli sviluppatori in questa fase non devono spendere molto del loro tempo per la loro identificazione. Inoltre, le associazioni dovrebbero essere identificate prima degli attributi per evitare confusione.

Abbiamo alcune euristiche:

- Frasi possessive o aggettivi;
- Si deve rappresentare come attributo di un oggetto lo stato memorizzato;
- Descrivere ogni attributo nel dizionario dei dati;
- Non perdere tempo a descrivere i dettagli fino a quando il modello a oggetti non è stabile.

Esempio:



6.3.9 Modellare il comportamento dei singoli oggetti

I **sequence diagram** modellano il comportamento di più oggetti e rappresentano il comportamento del sistema dal punto di vista dell'utente.

Gli **statechart**, invece, rappresentano il comportamento dei singoli oggetti modellando il loro stato. Infatti, è necessario costruire *statechart* per gli oggetti che hanno un ciclo di vita più lungo:

- è il caso degli *oggetti control*;
- meno degli *oggetti entity*;
- no per *oggetti boundary*.

6.3.10 Rivedere il modello di analisi

Una volta che il **modello dell'analisi** non subisce più modifiche o ne subisce raramente è possibile passare alla fase di **revisione del modello**. La revisione del modello deve essere fatta prima dagli sviluppatori e poi insieme dagli sviluppatori e dagli utenti.

L'obiettivo di questa attività di revisione è stabilire che la specifica risulta essere: **corretta, completa, consistente e chiara**.

Domande da porsi per assicurarsi della **correttezza**:

- Il glossario è comprensibile per gli utenti?
- Le classi astratte corrispondono a concetti ad alto livello?
- Tutte le descrizioni concordano con le definizioni degli utenti?
- Oggetti Entity e Boundary hanno nomi significativi?
- Oggetti control e use case sono nominati con verbi significativi del dominio?
- Tutti gli errori/eccezioni sono descritti e trattati?

Domande da porsi per assicurarsi della **completezza**:

- Per ogni **oggetto**: è necessario per uno use case? In quale use case è creato? modificato? distrutto? Può essere acceduto da un oggetto boundary?
- Per ogni **attributo**: quando è inizializzato? Quale è il tipo?
- Per ogni **associazione**: quando è attraversata? Perché ha una data molteplicità?
- Per ogni **oggetto control**: ha le associazioni necessarie per accedere agli oggetti che partecipano nel corrispondente use case?

Domande da porsi per assicurarsi della **consistenza**:

- Ci sono classi o use case con lo stesso nome?
- Ci sono entità con nomi simili e che denotano concetti simili?

Domande da porsi per assicurarsi della **chiarezza**:

- Le richieste di performance specificate sono state assicurate?
- Può essere costruito un prototipo per assicurarsi della fattibilità?

6.4 Assegnare le responsabilità

Ci sono tre tipi di ruoli: **generazione di informazione, integrazione, revisione**:

- L'**utente finale** è esperto del dominio di applicazione e genera informazioni sul sistema corrente.
- Il **cliente** (ruolo di integrazione) definisce lo scopo del sistema sulla base delle richieste dell'utente.
- L'**analista** è lo sviluppatore con la maggiore conoscenza del dominio di applicazione e genera informazioni sul sistema da sviluppare. Ogni analista è responsabile per uno o più use case. Identifica gli oggetti, le associazioni, attributi, ...

- L'architetto (ruolo di integrazione) unifica use case e oggetti dal punto di vista del sistema. Differenti analisti hanno differenti modi di modellare.
- Il manager delle configurazioni è responsabile di mantenere la storia delle revisioni e la tracciabilità del RAD con gli altri documenti.
- Il reviewer valida il RAD per correttezza, completezza, consistenza e chiarezza.

6.5 Comunicazione

E' difficile la **comunicazione** nelle prime fasi. I fattori che contribuiscono ad aumentare tale difficoltà sono:

- Diversi background dei partecipanti;
- Diverse aspettative degli stakeholders;
- Nuovi team che devono imparare a lavorare insieme.

Per gestire la complessità possiamo definire un ambiente chiaro e chiarire i ruoli dei membri del team, adottare forum pubblici e privati, con un database di discussione visibile al cliente e uno no. Inoltre, lo sviluppatore non deve interferire con le politiche interne cliente/utente.

Modello dinamico

6.6 Modelli dinamici

I **modelli dinamici** descrivono il comportamento del sistema in funzione del tempo; sono un tipo di *modello operazionale* (modello che descrive il comportamento desiderato) utili soprattutto per sistemi orientati al controllo (embedded systems, sistemi operativi, software di comunicazione).

I diagrammi utilizzati per i *modelli dinamici* sono:

- gli **interaction diagrams** descrivono il comportamento dinamico tra oggetti. Questo tipo di diagramma si divide in:
 - **sequence diagram**, cioè il comportamento dinamico di un insieme di oggetti disposti in una sequenza temporale.
 - **collaboration diagram**, mostra la relazione tra gli oggetti e non il tempo.
- gli **statecharts diagrams** (diagrammi di stato) descrivono il comportamento dinamico di un singolo oggetto. Questo tipo di diagramma fa uso di:
 - **activity diagram**, uno statechart diagram speciale dove tutti gli stati sono stati di azione.

L'obiettivo del modello dinamico è rilevare e fornire metodi per il modello a oggetti. Per raggiungere l'obiettivo dobbiamo eseguire questi step:

1. iniziare con un caso d'uso o uno scenario;
2. usare il sequence diagram per modellare l'interazione tra gli oggetti;
3. usare lo statechart diagram per modellare il comportamento dinamico di un singolo oggetto.

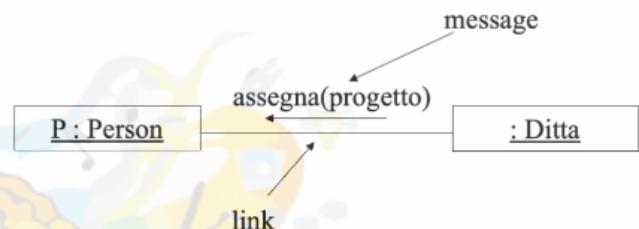
6.7 Interaction Diagrams

Questo tipo di diagramma si utilizza quando non siamo sicuri di aver modellato bene il problema dal punto di vista *object orientation* (non sappiamo con certezza se abbiamo identificato tutti gli oggetti).

Gli **interaction diagrams** descrivono il comportamento dinamico di un gruppo di oggetti che “interagiscono” per risolvere un problema. Un’**interazione** è un insieme di messaggi scambiati tra un gruppo di oggetti per raggiungere uno scopo. Un’interazione avviene tra oggetti tra cui esiste un **link** (istanza di un’associazione).

Un **messaggio** trasmette informazioni con l’aspettativa che verrà svolta un’attività. La ricezione di un messaggio può essere considerata un’istanza di un evento. Per l’invio di un messaggio è necessario specificare:

- *Ricevente*;
- *Messaggio*;
- *Eventuali informazioni aggiuntive*.



UML propone due diversi tipi di *Interaction Diagram* che esprimono informazioni simili, ma le evidenziano in modo diverso:

- **Sequence Diagram**;
- **Collaboration Diagram**.

6.7.1 Sequence Diagram

I **sequence diagram** descrivono le interazioni tra oggetti che collaborano per svolgere un compito. Sono utili per evidenziare il controllo nel sistema (“chi” fa “che cosa” ...).

Gli oggetti collaborano scambiandosi messaggi e lo scambio di un messaggio in OOP equivale all’invocazione di un metodo.

I *sequence diagrams* possono essere utilizzati nei seguenti modi:

- per modellare le interazioni ad alto livello tra oggetti attivi all’interno di un sistema;
- per modellare l’interazione tra istanze di oggetti nel contesto di una collaborazione che realizza un caso d’uso;
- per modellare l’interazione tra oggetti in una collaborazione che realizza un’operazione.

Inoltre, i *sequence diagrams* evidenziano la sequenza temporale delle azioni. Le attività svolte dagli oggetti sono mostrate su linee verticali, la sequenza dei messaggi scambiati tra gli oggetti è mostrata su linee orizzontali.

Questi diagrammi possono corrispondere a uno scenario specifico o a un intero caso d’uso (aggiungendo salti e iterazioni).

Gli oggetti

Sull’**asse delle x** abbiamo gli **oggetti** disposti in maniera orizzontale. La sintassi di un oggetto è nomeOggetto : NomeClasse.

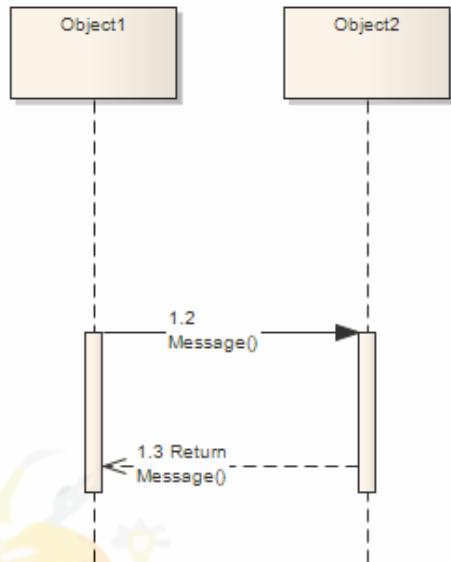
Sull’**asse t** abbiamo il **tempo**, cioè un flusso temporale descritto verticalmente.

Scambio di messaggi sincroni

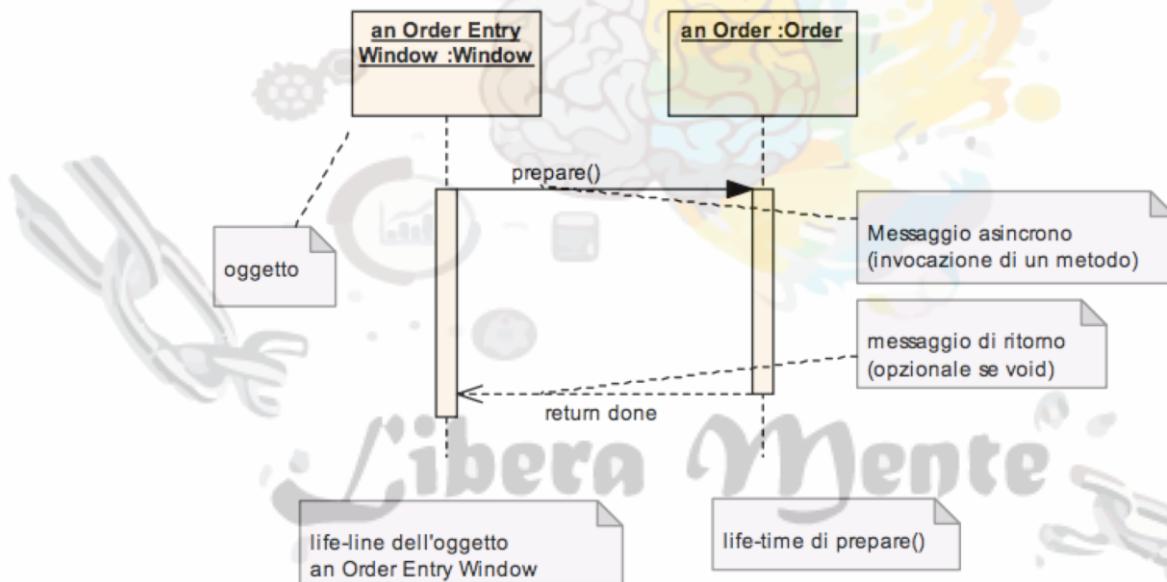
Un **messaggio** è rappresentato mediante una **freccia chiusa** dal chiamante al chiamato. La freccia è etichettata col nome del metodo invocato e, optionalmente, con i suoi parametri e il suo valore di ritorno.

Il valore di ritorno è sempre opzionale (se si omette, la fine del metodo corrisponde alla fine della life-time) ed è rappresentato con una **freccia tratteggiata**. Il chiamante attende la terminazione del metodo chiamato prima di proseguire.

La **life-time** (durata) di un metodo è rappresentato da un rettangolino che collega la freccia di invocazione con la freccia di ritorno. Tale *life-time* corrisponde ad avere un record di attivazione di quel metodo sullo stack di attivazione.



Esempio:



Scambio di messaggi asincroni

I **messaggi asincroni** si usano per descrivere interazioni concorrenti. Si disegnano con una **freccia aperta** da chiamante a chiamato. La freccia è etichettata col nome del metodo invocato e, optionalmente, con i suoi parametri e il suo valore di ritorno.

Il chiamante non attende la terminazione del metodo del chiamato, ma prosegue subito dopo l'invocazione. Il ritorno non segue quasi mai la chiamata.

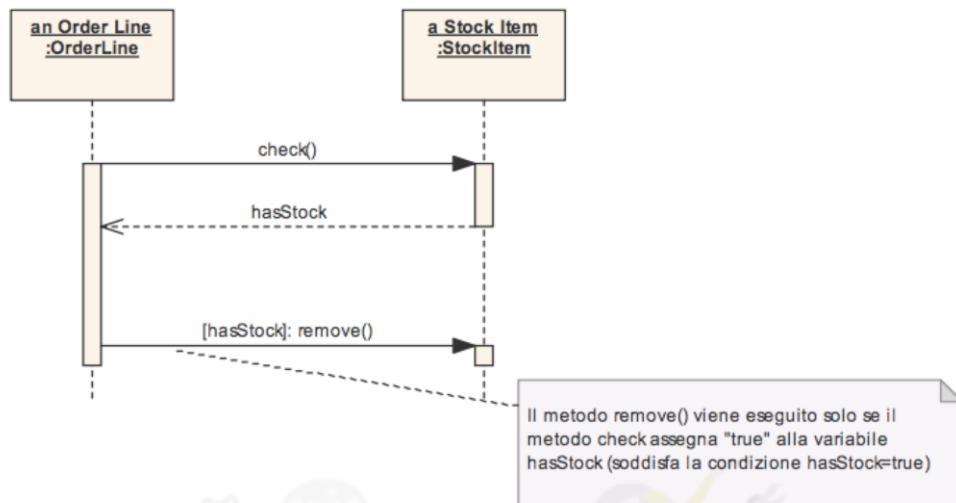
Condizioni sui messaggi

L'esecuzione di un metodo può essere soggetta ad una **condizione**: il metodo viene invocato solo se la condizione risulta verificata run-time.

Se la condizione non è verificata, il diagramma non dice cosa succede (a meno che non venga esplicitamente modellato ciascun caso). La condizione si rappresenta sulla freccia di invocazione del metodo, racchiusa tra parentesi quadre.

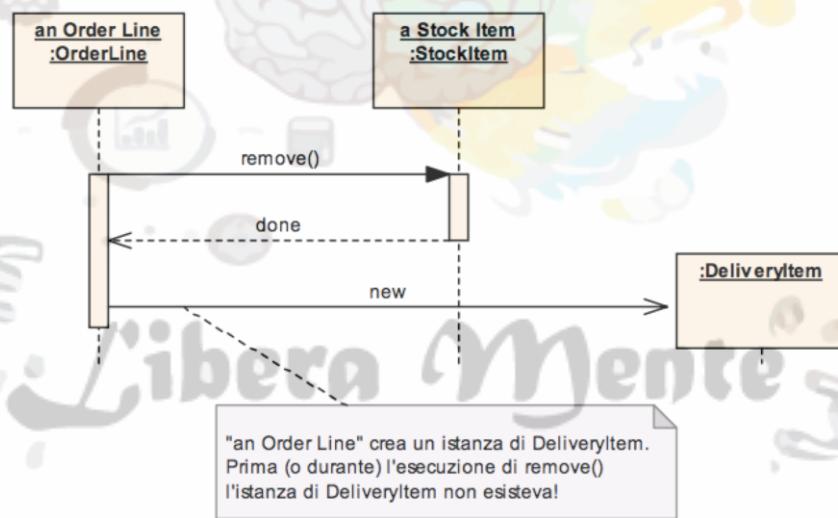
Sintassi: [cond] :nomeMetodo () .

Esempio:



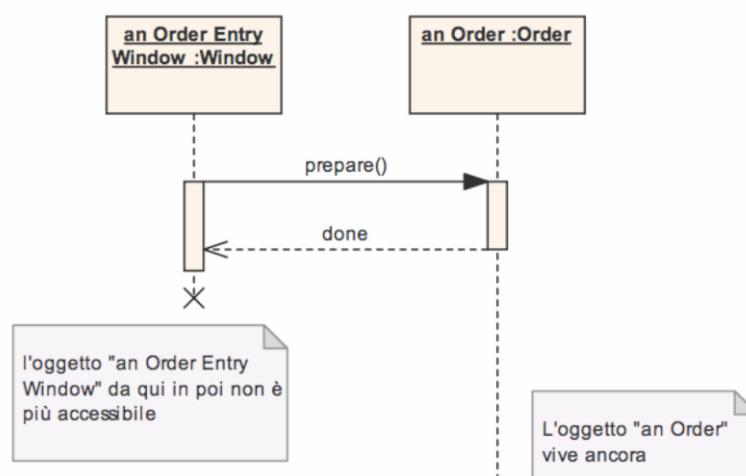
Costruzione di un nuovo oggetto

La costruzione di un nuovo oggetto corrisponde all'**allocazione dinamica** (allocazione nello heap di sistema, istruzione new o create, dopodiché l'oggetto viene collocato nell'asse temporale in corrispondenza del metodo new (o create)).



Distruzione di un oggetto (preesistente)

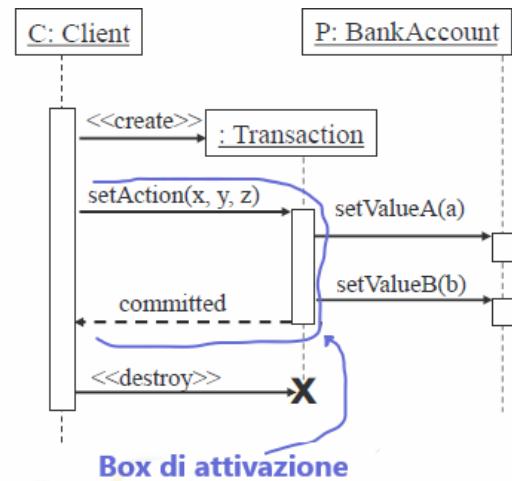
La distruzione di un oggetto corrisponde alla **deallocazione dinamica** (deallocazione dallo heap di sistema, istruzione delete/dispose). La deallocazione si rappresenta con una X posta in corrispondenza della life-line dell'oggetto: da quel momento in poi non è "legale" invocare alcun metodo dell'oggetto distrutto.



Box di attivazione

Il **box di attivazione** indica la durata di una **action** eseguita da un oggetto, direttamente o indirettamente.

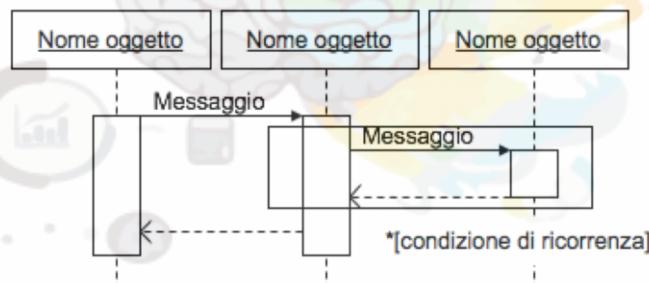
La cima del rettangolo è allineata con lo *start della action* (ricezione del messaggio); il fondo è allineato con il completamento della action, ed eventualmente con un messaggio di ritorno.



Iterazione (ricorrenze)

L'**iterazione** rappresenta l'esecuzione ciclica di più messaggi: si disegna un blocco che raggruppa i messaggi (metodi) su cui si vuole iterare.

Si può aggiungere la **condizione** dell'iterazione sull'angolo in alto a sinistra del blocco. La condizione si rappresenta tra parentesi quadre.



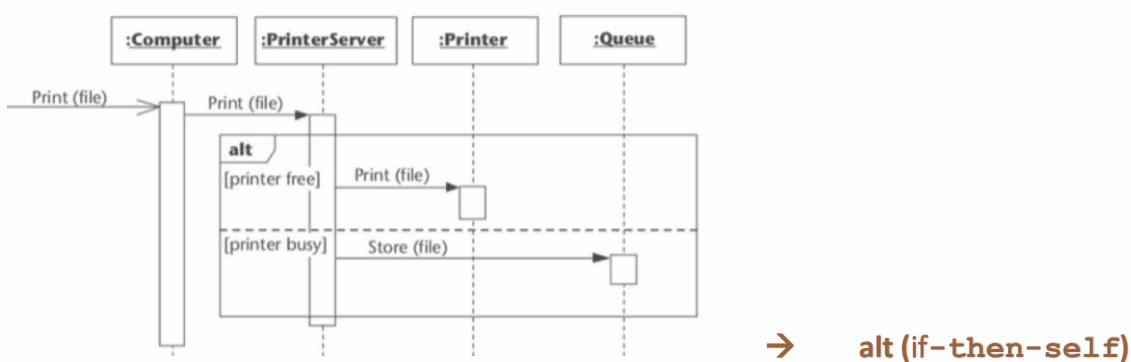
Cicli e condizioni

Cicli e condizioni si indicano con un riquadro (frame) che racchiude una sottosequenza di messaggi.

Nell'angolo in alto è indicato il costrutto. Tra i costrutti possibili abbiamo i:

- **Loop** (ciclo *while-do* o *do-while*): la condizione è indicata tra parentesi quadre all'inizio o alla fine;
- **Alt** (*if-then-else*): la condizione si indica in cima; se ci sono anche dei rami *else* allora si usa una linea tratteggiata per separare la zona *then* dalla zona *else* indicando eventualmente un'altra condizione accanto alla parola *else*;
- **Opt** (*if-then*): racchiude una sottosequenza che viene eseguita solo se la condizione indicata in cima è verificata.

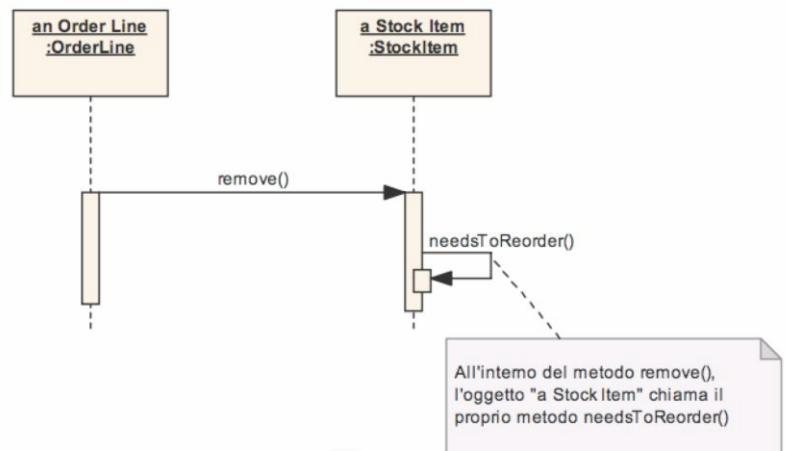
È buona norma utilizzare altri tipi di diagramma quando l'algoritmo da modellare si fa complesso.



Auto-chiamata (ricorsione)

L'auto-chiamata descrive un oggetto che invoca un proprio metodo: chiamante e chiamato in questo caso coincidono. Questo costrutto si rappresenta con una "freccia circolare" che rimane all'interno della life-time di uno stesso metodo.

Viene usata anche per rappresentare la **ricorsione**.



Esprimere vincoli sul tempo di risposta

Si possono esprimere dei vincoli sul **tempo di risposta** della life-time di un metodo.

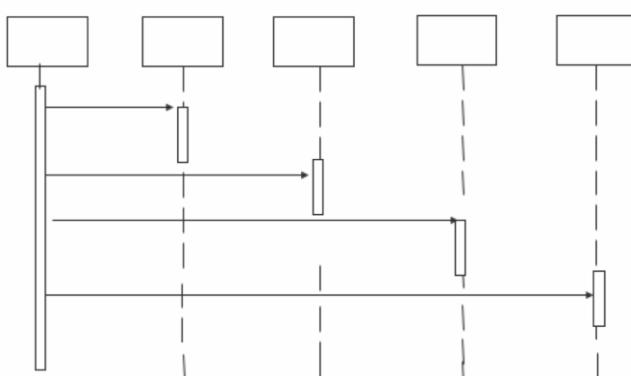
Esempio:



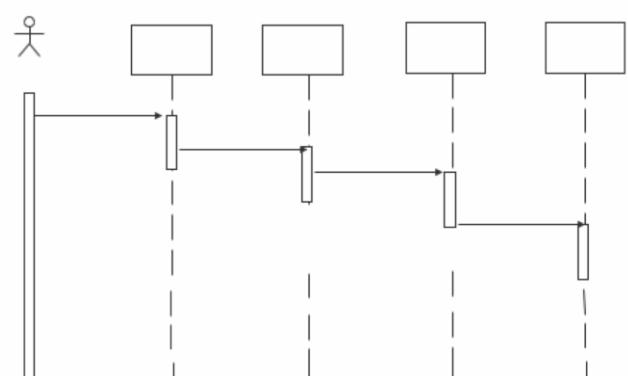
Fork e Stair Diagram

I *sequence diagrams* posseggono due strutture:

1. **Fork Diagram**: un oggetto "principale" divide il lavoro in più parti, ognuna delle quali sarà svolta da altri oggetti.
2. **Stair Diagram**: abbiamo diversi oggetti che svolgono tutti un ruolo cruciale e il lavoro viene **distribuito** tra questi oggetti.



Fork Diagram



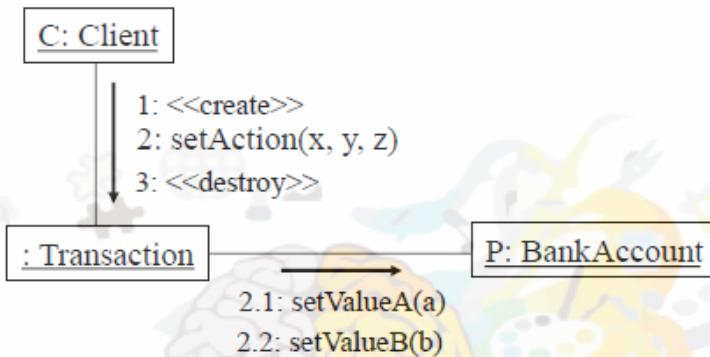
Stair Diagram

6.7.2 Collaboration Diagram

Il **collaboration diagram** specifica gli oggetti che collaborano tra loro in un dato scenario, ed i messaggi che si indirizzano.

La sequenza dei messaggi è meno evidente rispetto al *sequence diagram*, mentre sono più evidenti i legami tra gli oggetti.

Per visualizzare l'ordine sequenziale dello scambio dei messaggi è possibile ‘numerare’ i message. Questo diagramma può essere utilizzato in fasi diverse (analisi, disegno di dettaglio) e rappresentare diverse tipologie di oggetti. È adatto per la concorrenza e i thread.



Capitolo 7 – System Design

7.1 Scopi del System Design

La progettazione di un sistema (*System Design*) è la trasformazione del modello di analisi nel modello di progettazione del sistema.

Gli scopi del *system design* sono di definire gli obiettivi di progettazione del sistema, decomporre il sistema in sottosistemi più piccoli, in modo da poterli assegnare a team individuali, e selezionare alcune strategie quali:

- Scelte hardware e software;
- Gestione dei dati persistenti;
- Il flusso di controllo globale;
- Le politiche di controllo degli accessi;
- La gestione delle condizioni bonari (startup, shutdown, eccezioni).

Quindi, passiamo dal “che cosa” il sistema deve fare al “come” deve farlo. Inoltre, il *system design* si focalizza sul **dominio di implementazione**: prende in input il modello di analisi e dopo averlo trasformato dà in output un modello del sistema che include la decomposizione del sistema in sottosistemi e una descrizione di ognuna delle strategie.

Perché il system design è così difficile?

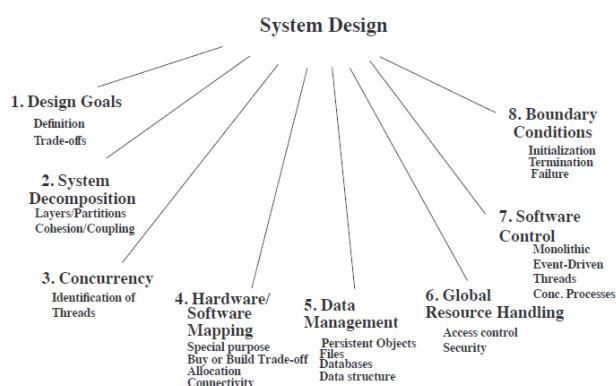
- **Analisi:** si focalizza sul dominio di applicazione;
- **Design:** si focalizza sul dominio di implementazione.

Quindi, gli sviluppatori devono raggiungere dei **compromessi (trade-off)** fra i vari obiettivi di design che sono spesso in conflitto gli uni con gli altri. Non possono anticipare le decisioni relative alla progettazione poiché non hanno non un’idea chiara del dominio della soluzione.

7.2 Output del System Design

L’output del *system design* produrrà:

- **Obiettivi di design:** descrivono la qualità del sistema (vengono derivati dalle richieste non funzionali).
- **Architettura software,** descrive:
 - la decomposizione del sistema in sottosistemi da assegnare ai vari team;
 - le dipendenze fra i sottosistemi;
 - l’hardware associato ai vari sottosistemi;
 - le decisioni (politiche) relative a: *Control flow*, *Controllo degli accessi*, *Memorizzazione dei dati*.
- **Boundary use case:** descrivono la configurazione del sistema, le scelte relative allo startup, allo shutdown ed alla gestione delle eccezioni.



7.3 Attività del System Design

Le attività del *system design* possono essere divise in tre fasi:

1. **Identificare gli obiettivi di design:** gli sviluppatori identificano quali caratteristiche dovrebbero essere ottimizzate e definiscono le priorità di tali caratteristiche.
2. **Decomposizione del sistema in sottosistemi:** basandosi sugli *use case* ed i *modelli di analisi*, gli sviluppatori decompongono il sistema in parti più piccole utilizzando stili architettonici standard.
3. **Raffinare la decomposizione in sottosistemi per rispettare gli obiettivi di design:** la decomposizione iniziale di solito non soddisfa gli obiettivi di design, quindi, gli sviluppatori la raffinano finché gli obiettivi non sono soddisfatti.

7.4 Identificare gli obiettivi di design

È il primo passo del system design: identifica le qualità su cui deve essere focalizzato il sistema. Molti **design goal** possono essere ricavati dai requisiti non funzionali o dal dominio di applicazione, altri sono forniti dal cliente, altri sono derivati da aspetti di management.

È importante formalizzarli esplicitamente poiché ogni decisione di design deve essere fatta seguendo lo stesso insieme di criteri.

Possiamo selezionare gli obiettivi di design da una lista di qualità desiderabili. I criteri sono organizzati in cinque gruppi:

1. **Performance;**
2. **Dependability;**
3. **Cost;**
4. **Maintenance;**
5. **End user criteria.**

1. Criteri di *Performance*

Includono i requisiti imposti sul sistema in termini di **spazio e velocità**.

- **Tempo di risposta:** con quali tempi una richiesta di un utente deve essere soddisfatta dopo che la richiesta è stata immessa.
- **Troughput:** quanti task il sistema deve portare a compimento in un periodo di tempo prefissato.
- **Memoria:** quanto spazio è richiesto al sistema per funzionare.

2. *Dependability* criteria

I **dependability** criteria riguardano la quantità di sforzo che deve essere spesa per minimizzare i crash del sistema e le loro conseguenze. Rispondono alle seguenti domande:

- **Robustness** (robustezza): capacità di sopravvivere ad input non validi immessi dall'utente.
- **Reliability** (affidabilità): differenza fra comportamento specificato e osservato.
- **Availability** (disponibilità): percentuale di tempo in cui il sistema può essere utilizzato per compiere normali attività.
- **Fault tolerance:** capacità di operare sotto condizioni di errore.
- **Security:** capacità di resistere ad attacchi di malintenzionati.
- **Safety:** capacità di evitare di danneggiare vite umane, anche in presenza di errori e di fallimenti.

3. Cost criteria

Includono i costi per sviluppare il sistema, per metterlo in funzione e per amministrarlo.

Quando il sistema sostituisce un sistema vecchio, è necessario considerare il costo per assicurare la compatibilità con il vecchio o per transitare verso il nuovo sistema.

I criteri di costo:

- **Development cost**: costo di sviluppo del sistema iniziale;
- **Deployment cost**: costo relativo all'installazione del sistema e training degli utenti;
- **Upgrade cost**: costo di convertire i dati del sistema precedente. Questo criterio viene applicato quando nei requisiti è richiesta la compatibilità con il sistema precedente (backward compatibility);
- **Maintenance cost** (costo di manutenzione): costo richiesto per correggere errori *sw* o *hw* (bug);
- **Administration cost** (costo di amministrazione): costo richiesto per amministrare il sistema.

4. Maintenance criteria

Determinano quanto deve essere difficile modificare il sistema dopo il suo rilascio.

I criteri di mantenimento sono:

- **Estensibilità**: quanto è facile aggiungere funzionalità o nuove classi al sistema;
- **Modificabilità**: quanto facilmente possono essere cambiate le funzionalità del sistema;
- **Adattabilità**: quanto facilmente può essere portato il sistema su differenti domini di applicazione;
- **Portabilità**: quanto è facile portare il sistema su differenti piattaforme;
- **Leggibilità**: quanto è facile comprendere il sistema dalla lettura del codice;
- **Tracciabilità dei requisiti**: quanto è facile mappare il codice nei requisiti specifici.

5. End User criteria

Includono qualità che sono desiderabili dal punto di vista dell'utente, ma che non sono state coperte dai criteri di performance e dependability.

Questi criteri sono:

- **Utilità**: quanto bene il sistema dovrà supportare il lavoro dell'utente;
- **Usabilità**: quanto dovrà essere facile per l'utente l'utilizzo del sistema.

Design Trade-offs

Quando definiamo gli obiettivi di design, spesso solo un piccolo sottoinsieme di questi criteri può essere tenuto in considerazione.

Gli sviluppatori devono dare delle priorità agli obiettivi di design, tenendo anche conto di aspetti manageriali, quali il rispetto dello schedule e del budget.

Esempi:

Spazio vs. velocità. Se il software non rispetta i requisiti di tempo di risposta e di throughput, è necessario utilizzare più memoria per velocizzare il sistema. Se il software non rispetta i requisiti di memoria, può essere compresso a discapito della velocità.

Tempo di rilascio vs. funzionalità. Se i tempi di rilascio sono brevi, possono essere rilasciate meno funzionalità di quelle richieste, ma nei tempi giusti.

Tempo di rilascio vs. qualità. Se i tempi di rilascio sono stretti, il project manager può rilasciare il software nei tempi prefissati con dei bug e, in tempi successivi, correggerli, o rilasciare il software in ritardo, ma con meno bug.

Tempo di rilascio vs. staffing. Può essere necessario aggiungere delle risorse al progetto per accrescere la produttività.

7.5 Decomposizione del sistema in sottosistemi

Descriviamo in dettaglio:

- Il concetto di **sottosistema** e come è collegato alle classi.
- Le **interfacce** dei sottosistemi poiché i sottosistemi forniscono dei servizi agli altri sottosistemi.
- Il **servizio**, cioè un insieme di operazioni correlate che condividono uno scopo comune.

Due proprietà dei *sottosistemi*:

- **Coupling**: misura la dipendenza fra due sottosistemi.
- **Cohesion**: misura la dipendenza fra classi appartenenti ad un sottosistema.

Per collegare fra loro due sottosistemi utilizzeremo 2 tecniche:

- **Layering**: consente ad un sistema di essere organizzato come una gerarchia di sottosistemi in cui ognuno fornisce servizi al sistema di livello superiore utilizzando i servizi forniti dal sistema al livello inferiore.
- **Partitioning**: organizza i sottosistemi come peer che forniscono differenti servizi agli altri sottosistemi.

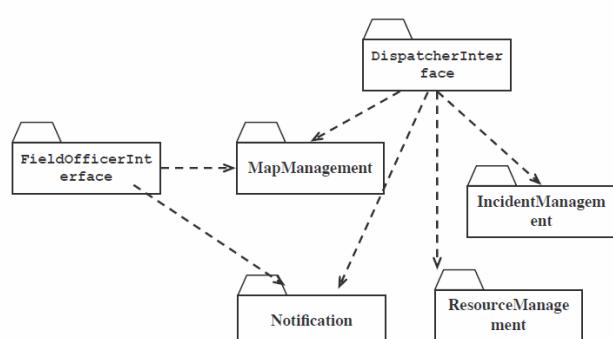
I Sottosistemi

Per ridurre la complessità della soluzione, decomponiamo il sistema in parti più piccole, chiamate **sottosistemi**.

Un **sottosistema** è costituito da un certo numero di **classi** del dominio della soluzione. Un sottosistema corrisponde alla parte di lavoro che può essere svolta da un singolo sviluppatore o da un team di sviluppatori.

Decomponendo il sistema in sottosistemi relativamente indipendenti, i team di progetto possono lavorare sui sottosistemi individuali con un minimo overhead di comunicazione (è importante che siano definiti gli obiettivi di design chiari a tutti i sottoteam).

Nel caso di sottosistemi complessi, applichiamo ulteriormente questo principio e li decomponiamo in sottosistemi più semplici.



Per modellare i sottosistemi possiamo utilizzare come notazione UML i **Package**: collezioni di classi, associazioni, operazioni e vincoli che sono correlati.

Un **sottosistema** è caratterizzato dai **servizi** (insieme di operazioni) che esso offre agli altri sottosistemi.

L'insieme di servizi che un sistema espone viene chiamato **interfaccia** (API: application programming interface) che include, per ogni operazione: i parametri, il tipo e i valori di ritorno. Le operazioni che essi svolgono vengono descritte ad alto livello senza entrare troppo nello specifico.

Coupling

L'**accoppiamento** (coupling) misura quanto un sistema è dipendente da un altro.

Due sistemi si dicono **loosely coupled** (leggermente accoppiati) se una modifica in un sottosistema avrà poco impatto nell'altro sistema, mentre si dicono **strongly coupled** (fortemente accoppiati) se una modifica su uno dei sottosistemi avrà un forte impatto sull'altro.

La condizione ideale di accoppiamento è quella di tipo **loosely** in quanto richiede meno sforzo quando devono essere modificate delle componenti.

Ovviamente dove non sono presenti componenti che si pensa debbano essere modificate spesso, non conviene utilizzare questa strategia in quanto aggiungerebbe complessità di sviluppo e di calcolo al sistema.

Cohesion

La **coesione** (cohesion) misura la dipendenza tra le classi contenute in un sottosistema.

La **coesione è alta** se le classi di un sottosistema realizzano compiti simili e sono collegate le une alle altre attraverso associazioni (es. ereditarietà), è invece **bassa** nel caso contrario.

L'ideale (il giusto trade-off) sarebbe quello di avere sottosistemi con **coesione interna alta**. Però un'alta **coesione** porta ad un numero elevato di sottosistemi aumentando l'**accoppiamento**. In definitiva bisogna trovare un compromesso tra **coesione** e **accoppiamento**.

Divisione del sistema con i Layer

Con la decomposizione in **layer** il sistema viene visto come una gerarchia di sottosistemi. Un **layer** è un raggruppamento di sottosistemi che forniscono servizi correlati. I **layer** per implementare un servizio potrebbero usare a loro volta servizi offerti dai **layer** sottostanti ma non possono usare servizi dei **layer** più alti.

Con i **layer** si possono avere due tipi di architettura:

- **Architettura chiusa**: un **layer** può accedere solo alle funzionalità del **layer** immediatamente sotto di esso. In questo caso si ottiene un'alta **manutenibilità** e **portabilità**.
- **Architettura aperta**: un **layer** può accedere alle funzionalità di tutti i **layer** sottostanti. In questo caso avremo una maggiore **efficienza** in quanto si risparmia l'overhead delle chiamate in cascata.

Basic Layer pattern

Un sistema di Base contiene tipicamente i seguenti sottosistemi:

- **Interface**: contiene gli *oggetti boundary* ed è ulteriormente decomposto in: **user interface** e **system interface**.
- **Function**: contiene gli *oggetti control* e la logica dell'applicazione.
- **Model**: contiene gli *oggetti entity* del dominio di applicazione.

Divisione del sistema con le Partizioni

Il sistema viene diviso in sottosistemi **pari** (peer), ognuno dei quali è responsabile di diverse classi di servizi. In generale una decomposizione di un sistema avviene utilizzando ambedue le tecniche.

Infatti, il sistema viene prima diviso in sottosistemi tramite le **partizioni** e successivamente ogni partizione viene organizzata in **layer** finché i sottosistemi non sono abbastanza semplici da essere sviluppati da un singolo sviluppatore o team.

7.6 Architetture software

Un **architettura software** include le scelte relative alla decomposizione in sottosistemi, flusso di controllo globale, gestione delle condizioni limite e i protocolli di comunicazione tra i sottosistemi.

È da notare che la decomposizione dei sottosistemi è una fase molto critica in quanto una volta iniziato lo sviluppo con una determinata decomposizione è complesso ed oneroso dover tornare indietro in quanto molte interfacce dei sottosistemi dovrebbero essere modificate.

Alcuni stili architetturali che possono essere usati sono:

- **Architettura Client/Server;**
- **Architettura Peer-To-Peer;**
- **Architettura a Repository;**
- **MVC** (Model, View, Controller).

Architettura a “Repository”

Con questo stile tutti i sottosistemi accedono e modificano i dati tramite un una struttura dati chiamata **repository**. Il flusso di controllo viene gestito dal *repository* tramite un cambiamento dei dati oppure dai sottosistemi tramite meccanismi di sincronizzazione o lock.

I vantaggi di questo stile si vedono quando si implementano applicazioni in cui i dati cambiano di frequente poiché si evitano incoerenze.

Tra gli svantaggi si può notare che il *repository* può facilmente diventare un collo di bottiglia in termini di prestazioni e inoltre è altissimo il coupling tra i sottosistemi e il *repository*: una modifica all’*API* del *repository* comporta la modifica di tutti sottosistemi che lo utilizzano.

Architettura “MVC”

Con questo tipo di architettura, il sistema viene diviso in tre sottosistemi: **Model, View e Control**.

- Il sottosistema Model: fornisce i metodi per accedere ai dati utili dall’applicazione;
- Il sottosistema View: visualizza all’utente gli oggetti del dominio dell’applicazione; è la parte di interazione con l’utente.
- Il sottosistema Controller: gestisce il flusso di controllo; si occupa di prendere l’input dall’utente (tramite la *View*) e di inviarlo al *Model*.

MVC è un caso particolare di architettura di tipo *repository*:

- il sottosistema Model implementa la struttura dati centrale;
- il sottosistema Controller gestisce il control flow: ottiene gli input dall’utente e manda messaggi al *Model*;
- i sottosistemi View visualizzano il *Model* e sono notificati (attraverso un protocollo subscribe/notify) ogni volta che il *Model* è modificato.

Il motivo per cui si separano *Model*, *View* e *Controller* è che le interfacce utenti sono soggette a cambiamenti più spesso di quanto avviene per la conoscenza del dominio (*il Model*).

Per questo motivo *MVC* è l'ideale per sistemi interattivi e quando il sistema deve avere viste multiple dello stesso *Model*. D'altro canto, questo sistema architettonale introduce lo stesso collo di bottiglia visto per lo stile architettonico a *Repository*.

Architettura “Client – Server”

Il sottosistema **server** fornisce servizi ad una serie di istanze di altri sottosistemi detti **client** i quali si occupano dell'interazione con l'utente. La maggior parte della computazione viene svolta lato **server**.

Inoltre, il flusso di controllo nei *client* e nei *server* è indipendente: i *client* conoscono l'interfaccia del *server* ma i *server* non conoscono le interfacce dei *client*.

Questo stile è spesso usato in sistemi basati su database in quanto è più facile gestire l'integrità e la consistenza dei dati.

Architettura “Peer – to – Peer”

È una generalizzazione dello stile **client-server** in cui però *client* e *server* possono essere scambiati di ruolo ed ognuno dei due può fornire e richiedere servizi. Inoltre, questa architettura introduce la possibilità di deadlock e complica il flusso di controllo.

Architettura “Three – tier”

I sottosistemi vengono organizzati in **tre livelli hardware**:

- **Interface layer**: include tutti gli *oggetti boundary* di interazione con l'utente;
- **Application logic layer**: include gli *oggetti control* ed *entity*;
- **Storage layer**: effettua la memorizzazione, il recupero e l'interrogazione di oggetti persistenti.

La separazione dell'interfaccia dalla logica applicativa consente di modificare e/o sviluppare diverse interfacce utente per la stessa logica applicativa.

Architettura a “pipeline” (flusso di dati)

Questo stile architettonico si rivela efficace nel caso in cui si ha un insieme di dati in input da trasformare attraverso una catena di componenti e di filtri software al fine di ottenere una serie di dati in output.

Ogni componente della catena lavora in modo indipendente rispetto agli altri, trasforma i dati in input in dati in output e delega ai componenti successivi le ulteriori trasformazioni.

Ogni parte è inconsapevole delle modalità di funzionamento dei componenti adiacenti. Inoltre, non è adatto per sistemi interattivi o che richiedono maggiore interazione tra le componenti.

7.6.1 Scelta dei Sottosistemi

Quando si decidono le componenti di un sottosistema bisognerebbe tenere presente che la maggior parte dell'interazione tra le componenti dovrebbe avvenire all'interno di un sottosistema allo scopo di ottenere un'alta coesione.

Le **euristiche** per scegliere le componenti dei sottosistemi sono le seguenti:

- Gli oggetti identificati in un caso d'uso dovrebbero appartenere ad uno stesso sottosistema.
- Bisogna creare dei sottosistemi che si occupano di trasferire i dati tra i sottosistemi.
- Minimizzare il numero di associazioni tra i sottosistemi (devono essere *loosely coupled*).
- Tutti gli oggetti di un sottosistema dovrebbero essere funzionalmente correlati.

7.7 Descrizione delle attività del System Design

Le attività del *system design* sono le seguenti:

- **Mapping Hardware/Software;**
- **Gestione dei Dati Persistenti;**
- **Controllo di Accesso;**
- **Controllo del Flusso;**
- **Condizioni limite.**

Mapping Hardware/Software

Molti sistemi complessi necessitano di lavorare su più di un computer interconnessi da rete. L'uso di più computer può ottimizzare le performance e permettere l'utilizzo del sistema a più utenti distribuiti sulla rete.

In questa fase vanno prese alcune decisioni per quanto riguarda le piattaforme hardware e software su cui il sistema dovrà girare (es. Unix vs Windows, Intel vs Sparc, etc).

Una volta decise le piattaforme è necessario mappare le **componenti** su di esse poiché tali componenti consentono di "muovere" le informazioni da un **nodo** ad un altro e di gestire problemi di concorrenza.

Sfortunatamente, da una parte, l'introduzione di nuovi nodi hardware distribuisce la computazione, dall'altro introduce alcune problematiche, tra cui la sincronizzazione, la memorizzazione, il trasferimento e la replicazione di informazioni tra sottosistemi.

Gestione dei Dati Persistenti

Il modo in cui i dati vengono memorizzati può influenzare l'architettura del sistema e la scelta di uno specifico database. In questa fase vanno identificati gli **oggetti persistenti** e il tipo di infrastruttura da usare per memorizzarli (DBMS, file o altro).

Gli **oggetti entity**, identificati durante *l'analisi dei requisiti*, possono diventare persistenti ma non è detto che lo siano tutti. In generale, i dati sono persistenti se sopravvivono ad una singola esecuzione del sistema: il sistema dovrà memorizzare i dati persistenti quando questi non servono più e ricaricarli quando necessario.

Una volta decisi gli oggetti dobbiamo decidere come questi saranno essere memorizzati. Principalmente potremmo avere a disposizione tre mezzi: **file**, **DBMS relazionale** e **DBMS ad oggetti**.

La scelta tra una tecnologia o un'altra per la memorizzazione dei dati può essere influenzata da vari fattori. In particolare, conviene usare un **file** in questi casi:

- Dimensione elevata dei dati (es. immagini, video ecc.);
- Dati temporanei e logging.

Conviene invece usare un **DBMS** (relazionale e ad oggetti) **in casi di:**

- Accessi concorrenti (i DBMS effettuano controlli di consistenza e concorrenza bloccando i dati quando necessario);
- Uso dei dati da parte di più piattaforme;
- Particolari politiche di accesso a dati.

1. File

Il **file** richiede una logica più complessa per la scrittura e lettura, dall'altra permette un accesso ai dati più efficiente.

2. DBMS relazionale

Un **DMBS relazione** fornisce un' interfaccia di più alto livello rispetto al file: i dati vengono memorizzati in tabelle ed è possibile utilizzare un linguaggio standard per le operazioni (**SQL**). Gli oggetti devono essere mappati sulle tabelle per poter essere memorizzati.

3. DBMS ad oggetti

Un **database orientato ad oggetti** è simile ad un **DBMS relazionale** con la differenza che non è necessario mappare gli oggetti in tabelle in quanto questi vengono memorizzati così come sono. Questo tipo di database riduce il tempo di setup iniziale (si risparmia sulle decisioni di mapping) ma sono più lenti e le query sono di più difficile comprensione.

Controllo di Accesso

In un sistema multiutenza è necessario fornire delle politiche di accesso alle informazioni. In questa fase vanno definite, in modo più preciso, le operazioni e le informazioni effettuabili da ogni singolo attore e come questi si autenticano al sistema.

È possibile rappresentare queste politiche tramite una **matrice di accesso** in tre modi:

1. **Tabella di accesso globale**: ogni riga della matrice contiene una *tupla* (attore, classe, operazione). Se la *tupla* è presente per una determinata classe e operazioni, l'accesso è consentito altrimenti no.
2. **Access control list (ACL)**: ogni classe ha una lista che contiene una tupla (attore, operazione) che specifica se l'attore può accedere a quella determinata operazione della classe a cui la *ACL* appartiene.
3. **Capability**: una *capability* è associata ad un attore ed ogni riga della matrice contiene una tupla (classe, operazione che l'attore a cui è associata può eseguire).

Controllo del Flusso

Un **flusso di controllo** è una sequenza di azioni di un sistema. In un sistema *Object Oriented* una sequenza di azioni include di prendere decisioni su quali operazioni eseguire e in che ordine. Queste decisioni sono basate su eventi esterni generati da attori o causati dal trascorrere del tempo.

È un problema che riguarda la fase di **system design** perché bisogna tener conto che non tutti gli oggetti hanno un proprio processore a disposizione. Invece, durante *l'analisi*, assumiamo che tutti gli oggetti siano eseguiti simultaneamente, eseguendo le operazioni ogni volta che necessitano di eseguirle.

Esistono 3 tipi di controlli di flusso:

1. **Procedure-driven control**: le operazioni rimangono in attesa di un input dell'utente ogni volta che hanno bisogno di elaborare dati. Questo tipo di controllo di flusso è particolarmente usato in sistemi legacy di tipo procedurale.
2. **Event-driven control**: in questo controllo di flusso un ciclo principale aspetta il verificarsi di un evento esterno. Non appena l'evento diventa disponibile la richiesta viene direzionata all'opportuno oggetto. Questo tipo di controllo ha il vantaggio di centralizzare tutti gli input in un ciclo principale ma ha lo svantaggio di rendere complessa l'implementazione di sequenze di operazioni composte di più passi.
3. **Threads**: questo controllo di flusso è una modifica del *procedure-driven control* che aggiunge la gestione della concorrenza. Il sistema può creare un arbitrario numero di threads, ognuno assegnato ad un determinato evento. Se si sceglie di usare un control-flow di tipo threads bisogna stare attenti a gestire situazioni di concorrenza in quanto più thread possono accedere contemporaneamente alle stesse risorse e creare situazioni non previste.

Quando è stato scelto un meccanismo per il flusso di controllo, esso viene realizzato con un insieme di uno o più **oggetti control**.

Il ruolo degli *oggetti control* è quello di memorizzare gli eventi esterni e il loro stato temporaneo, gestire la sequenza di chiamate di operazioni sugli *oggetti boundary* ed *entity* associati con gli eventi esterni.

Inoltre, localizzare le decisioni sul flusso di controllo per uno use case in un singolo oggetto non solo consente di avere un codice più comprensibile, ma rende anche il sistema più flessibile ai cambiamenti nell'implementazione del flusso di controllo.

Condizioni limite

Nella fase di **system design** bisogna determinare le **condizioni limite** per il sistema che si sta sviluppando:

- **inizializzazione**: descrive come il sistema è portato da uno stato non inizializzato ad uno stato stabile ("startup use cases");
- **terminazione**: descrive quali risorse sono rilasciate e quali sistemi sono notificati della terminazione ("termination use cases");
- **fallimento**: causato da vari problemi come errori o problemi esterni (es. alimentazione elettrica). Buoni sistemi progettano i fallimenti fatali ("failure use cases").

Le condizioni limite non vengono trattati nella fase di analisi poiché sono determinate da decisioni di design.

Quindi, le **condizioni limite** del sistema includono lo startup, lo shutdown, l'inizializzazione e la gestione dei fallimenti come corruzione di dati, caduta di connessione e caduta di componenti.

A tale scopo vanno elaborati dei casi d'uso che specificano la sequenza di operazioni da svolgere in ciascuno dei casi sopra elencati. In generale, gli use case per le condizioni limite vengono identificati esaminando ogni sottosistema e ogni oggetto persistente:

- **Configurazione**. Per ogni oggetto persistente, si esamina in quale use case è creato o distrutto. Per ogni oggetto non creato o non distrutto in uno degli use case, si aggiunge uno use case invocato dall'amministratore di sistema.

- **Avvio e terminazione.** Per ogni componente si aggiungono tre use case: *start*, *shutdown*, *configure*.
- **Gestione eccezioni.** Per ogni tipo di fallimento di componente, si decide come il sistema debba reagire. Documentiamo ognuna di queste decisioni con uno use case eccezionale che estende lo use case di base.

Un'eccezione è un evento o errore che si verifica durante l'esecuzione del sistema. Una situazione del genere può verificarsi in tre casi:

- un **fallimento hardware** (dovuto all'invecchiamento dell'hardware);
- un **cambiamento nell'ambiente** (interruzione di corrente);
- un **fallimento del software** (causato da un errore di progettazione).

Nel caso in cui un errore dipenda da un input errato dell'utente, tale situazione deve essere comunicata all'utente tramite un messaggio così che lo stesso possa correggere l'input e riprovare.

Nel caso di caduta di un collegamento il sistema dovrebbe salvare lo stato del sistema in modo da poter riprendere l'esecuzione non appena il collegamento ritorna.

Rivedere il modello di system design

Come l'analisi, il *system design* è un'**attività iterativa**. A differenza dell'analisi, non ci sono agenti esterni (come il cliente) per revisionare il lavoro e le scelte fatte. Quindi, il manager del progetto e gli sviluppatori devono organizzare un processo di revisione per sostituirsi al cliente. Per raggiungere gli obiettivi del progetto bisogna rispettare i seguenti criteri:

- **Correttezza:** il system design è corretto se il modello di analisi può essere mappato su di esso.
- **Completezza:** la progettazione di un sistema è completa se ogni requisito e ogni caratteristica è stata portata a compimento.
- **Consistenza:** il system design è consistente se non contiene contraddizioni.
- **Realismo:** un progetto è realistico se il sistema può essere realizzato ed è possibile rispettare problemi di concorrenza e tecnologie.
- **Leggibilità:** un system design è leggibile se anche sviluppatori non coinvolti nella progettazione possono comprendere il modello realizzato.

7.8 UML Components

Durante il *system design* dobbiamo modellare la **struttura statica** e quella **dinamica**:

- il **diagramma delle componenti** per la struttura statica mostra la struttura al "design time", o al "compile time";
- il **diagramma di deployment** per la struttura dinamica mostra la struttura al "run-time".

UML Component Diagram

In UML le **componenti** possono essere dei file eseguibili, delle librerie, delle pagine web, etc.

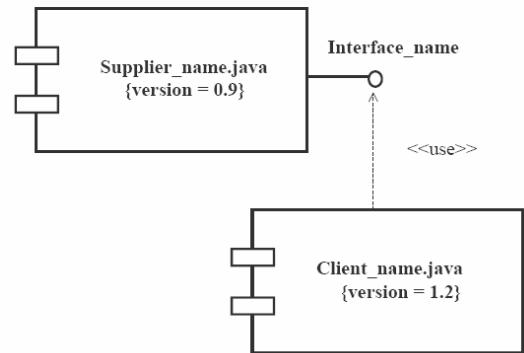
Un **nodo**, invece, è un processore disponibile che può eseguire dei processi (es. la rete).



Il diagramma che esprime le componenti non è altro che un **grafo delle componenti** connesse attraverso relazioni di **dipendenza**.

Le **dipendenze** sono mostrate con *archi* dalla componente cliente alla componente fornitrice; i tipi di dipendenza sono specifici dei linguaggi di implementazione.

L'**interfaccia** indica un insieme di operazioni da mettere a disposizione per una *componente*. Inoltre, può aiutare a specificare quale parte di una classe è attualmente usata dalle classi client.



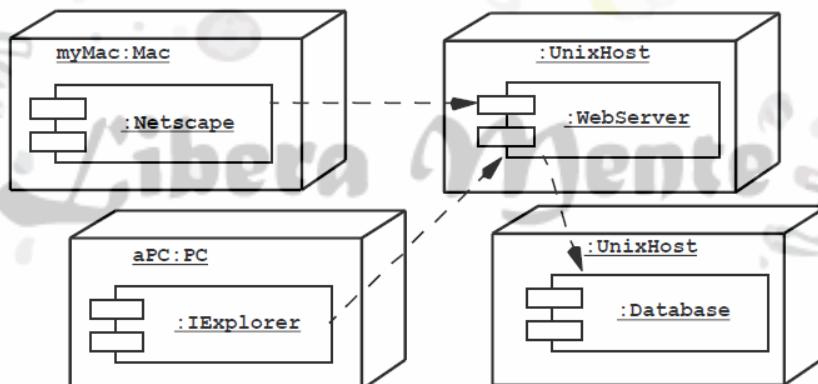
UML Deployment Diagram

I **deployment diagrams** sono utili per mostrare il progetto del sistema dopo che le seguenti decisioni sono state prese:

- decomposizione in sottosistemi;
- concorrenza;
- mapping Hardware/Software.

Il **deployment diagram** è un *grafo di nodi* connessi attraverso associazioni di comunicazione:

- i nodi sono mostrati come box 3D;
- i nodi possono contenere istanze di componenti;
- le componenti possono contenere oggetti.



7.9 Documentazione del System Design

Il **System Design** è documentato nel **System Design Document (SDD)**, il quale descrive:

- l'insieme dei **design goals** del progetto;
- la decomposizione del sistema in sottosistemi (con gli *UML class diagram*);
- mapping Hardware/Software (con *UML deployment diagrams*);
- gestione dei dati persistenti;
- controllo di accesso;
- controllo del flusso globale;
- condizioni limite.

L'**SDD** è un'interfaccia tra gli sviluppatori e coloro che hanno svolto la progettazione ad alto livello del sistema, cioè coloro che hanno deciso l'architettura di tale sistema. In altri termini, l'**SDD** è una forma di comunicazione (avviene uno scambio di informazioni) tra sviluppatori e progettisti ad alto livello.

Riassunto: System Design Document (SDD)

1. Introduction
 - 1.1. Purpose of the system
 - 1.2. Design Goals
 - 1.3. Definition, acronyms, and abbreviations
 - 1.4. References
 - 1.5. Overview
2. Current software architecture
3. Proposed software architecture
 - 3.1. Overview
 - 3.2. Subsystem decomposition
 - 3.3. Hardware/software mapping
 - 3.4. Persistent data management
 - 3.5. Access control and security
 - 3.6. Global software control
 - 3.7. Boundary conditions
4. Subsystems services
5. Glossary



Capitolo 8 – Object Design

8.1 Object Design

Durante l'**analisi** si descrive lo scopo del sistema, e si identificano gli **oggetti di applicazione**.

Gli **oggetti di applicazione** (chiamati anche oggetti del dominio) rappresentano concetti del dominio dell'applicazione che sono rilevanti nel sistema (la maggior parte degli oggetti entity sono di questo tipo).

Durante il **system design** viene descritta l'architettura del sistema, la piattaforma *HW/SW* che permette di selezionare le componenti già pronte, etc.

Durante l'**object design** chiudiamo il gap tra oggetti di applicazione e componenti già pronte (*off-the-shelf*) identificando gli **oggetti di soluzione** e raffinando gli oggetti esistenti.

Gli **oggetti di soluzione** sono oggetti che non sono mappabili su concetti relativi al dominio dell'applicazione e servono a rendere funzionale il sistema. Un esempio di oggetti di questo tipo, identificati durante l'analisi, sono gli *oggetti boundary* e *control*.

L'**object design** è il processo che si occupa di aggiungere dettagli all'analisi dei requisiti e prendere decisioni di implementazione. Questa fase deve implementare il modello di analisi con l'obiettivo di minimizzare il tempo di esecuzione, la memoria ed altri costi. Quindi, l'*object design* serve come base dell'implementazione.

Le fasi dell'*object design* sono le seguenti:

- ❖ **Riuso**
- ❖ **Specifiche dei servizi**
- ❖ **Ristrutturazione** (del modello ad oggetti):
- ❖ **Ottimizzazione** (del modello ad oggetti):

Riuso

Il **riuso** include le **componenti off-the-shelf** (riutilizzate), identificate durante il **system design**, utilizzate nella realizzazione di ogni sottosistema. Vengono selezionate librerie di classi ed altre componenti utili per strutture dati e servizi di base. Vengono selezionati dei *Design Pattern* per risolvere problemi comuni e per proteggere classi da futuri cambiamenti.

Molte volte le componenti devono essere adattate prima di poterle utilizzare. Può essere fatto attraverso:

- **oggetti wrapper**;
- **raffinandoli utilizzando l'ereditarietà**.

Durante tutte queste attività gli sviluppatori devono prendere decisioni di *trade off* tra **costi** e **costruzione**.

Specifiche delle interfacce

I servizi forniti dai sottosistemi (identificati nel system design) vengono dettagliati in termini di interfacce di classe, includendo operazioni, argomenti, firme di metodi ed eccezioni. Vengono anche aggiunte eventuali operazioni o oggetti necessari a trasferire i dati tra i sottosistemi.

Il risultato di questa attività è una specifica completa delle interfacce per ogni sottosistema. La specifica delle interfacce dei sottosistemi è spesso chiamata “**API** (Application Programmer Interface) **del sottosistema**”.

Ristrutturazione

La **ristrutturazione** manipola il modello del sistema per incrementare il riuso di codice o per soddisfare altri design goal. Durante la ristrutturazione ci si occupa anche di come soddisfare design goal come mantenimento, leggibilità, e comprensione del modello di sistema.

Attività tipiche sono:

- Trasformare associazioni N-arie in binarie;
- Implementare associazioni binarie attraverso riferimenti;
- Fondere classi simili in differenti sottosistemi in un'unica classe;
- Trasformare classi con nessun comportamento in attributi;
- Decomporre classi complesse in classi più semplici;
- Aumentare l'ereditarietà ed il packaging modificando classi ed operazioni.

Ottimizzazione

L'**ottimizzazione** ha lo scopo di migliorare le performance del sistema, aumentando la velocità, riducendo l'uso di memoria e diminuendo la molteplicità delle associazioni.

Questa attività include:

- Cambiare gli algoritmi per rispondere ai requisiti di memoria e velocità;
- Ridurre le molteplicità nelle associazioni per velocizzare le query;
- Aggiungere associazioni ridondanti per aumentare l'efficienza;
- Modificare l'ordine di esecuzione;
- Aggiungere attributi derivati per migliorare il tempo di accesso agli oggetti;
- Aprire l'architettura (aggiungere la possibilità di accedere a strati di basso livello).

8.2 Attività di Object Design

L'attività di *object design* non è sequenziale, di solito viene realizzata in maniera **concorrente**. Ci potrebbero però essere delle dipendenze:

- una componente off-the-shelf può vincolare il tipo di eccezioni di un'operazione e quindi può influenzare l'interfaccia di un sottosistema;
- la ristrutturazione e l'ottimizzazione possono ridurre il numero di oggetti da implementare e quindi aumentare il riuso.

Di solito vengono realizzate prima le attività di riuso e di specifica delle interfacce, ottenendo un **modello a oggetti di design** che viene verificato rispetto ai corrispondenti casi d'uso. Una volta che il modello si è stabilizzato vengono svolte le attività di ristrutturazione ed ottimizzazione.

8.3 Object Design: Fase di Riuso

Gli obiettivi dell'**attività di riuso** sono di utilizzare funzionalità già disponibili per realizzare delle nuove funzionalità: viene usata la conoscenza acquisita durante le esperienze precedenti di progettazione per risolvere il problema corrente.

Gli strumenti impiegati sono:

- **Ereditarietà**: le nuove funzionalità sono ottenute per ereditarietà. Questo strumento è anche detto **riuso White-Box** perché la struttura interna delle classi antenate è spesso visibile alle sottoclassi.
- **Composizione**: le nuove funzionalità sono ottenute per aggregazione. Questo strumento è anche detto **riuso Black-Box** perché i dettagli implementativi interni agli oggetti non sono visibili all'esterno.
- **Design Pattern**: template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità.

8.3.1 Ereditarietà

Durante l'**analisi** abbiamo usato l'ereditarietà per organizzare gli oggetti in una gerarchia comprensibile (descrizione di tassonomie), infatti, partendo dagli oggetti astratti i lettori del modello di analisi comprendono le funzionalità chiave del sistema.

L'utilizzo dell'ereditarietà durante l'**object design** permette di ridurre le ridondanze, migliorare l'estendibilità, la modificabilità e il riuso del codice. I comportamenti ridondanti sono fattorizzati in una singola superclasse, riducendo il rischio di introdurre inconsistenze in seguito a futuri cambiamenti.

L'**ereditarietà di implementazione** è usata al solo scopo di riusare codice. Con questo tipo di ereditarietà gli sviluppatori possono riusare codice in modo veloce estendendo una classe esistente e riferendo il suo comportamento.

Un problema con l'*ereditarietà di implementazione* è che le operazioni della superclasse sono esposte all'utilizzatore della sottoclasse e quindi alcune operazioni ereditate possono essere utilizzate in modo inaspettato.

L'**ereditarietà di specifica** (o *interfaccia* o *sottotipo*) definisce la possibilità di utilizzare un oggetto al posto di un altro: si eredita da una classe astratta che possiede operazioni già specificate ma non implementate.

Principio di sostituzione di Liskov

Una definizione più precisa dell'*ereditarietà di specifica* viene data dal **principio di Liskov** il quale dice che: "se un oggetto di tipo S può sostituire un oggetto di tipo T in qualunque posto in cui ci si aspetta di trovare T, allora S può essere definito un sottotipo di T".

In altre parole, l'oggetto di tipo S è **sottotipo** di T se esso non sovrascrive metodi di T cambiandone il comportamento atteso.

Inoltre, gli oggetti appartenenti a una sottoclasse devono essere in grado di esibire tutti i comportamenti e le proprietà esibiti da quelli appartenenti alla superclasse in modo tale da poter essere "sostituiti" senza intaccare la funzionalità del programma.

Una relazione di ereditarietà che soddisfa tale principio è chiamata **Ereditarietà stretta**.

8.3.2 Composizione: la delega

La **delega** è un'alternativa all'*ereditarietà di implementazione* applicabile quando si vuole riusare codice.

Abbiamo due oggetti che collaborano nel gestire una richiesta:

- un **Client** richiede l'esecuzione di un'operazione all'oggetto **Receiver**;
- **Receiver delega** ad un altro oggetto (**Delegate**) l'esecuzione dell'operazione.

In questo modo l'oggetto delegato non può essere utilizzato in modo scorretto dall'utilizzatore dell'oggetto ricevente. L'uso della *delegazione* porta alla scrittura di codice più robusto e non interferisce con le componenti già esistenti.



Esempio:

Ereditarietà di implementazione

```
/* Implementation of Stack using inheritance*/
class Stack extends List {
    //Constructor omitted
    Object top() {
        return get(this.size()-1);
    }
    void push (Object elem) {
        add(elem);
    }
    void pop() {
        remove(this.size(-1));
    }
}
```

Delega

```
/* Implementation of Stack using delegation*/
class Stack {
    private List aList;
    Stack() {
        aList = new List();
    }
    Object top() {
        return aList.get(aList.size()-1);
    }
    void push (Object elem) {
        aList.add(elem);
    }
    void pop () {
        aList.remove(aList.size () -1);
    }
}
```

Usando la delega la classe **Stack** non include nella sua interfaccia i metodi di **List** (quindi, l'utilizzatore (client) della classe Stack non può utilizzare i metodi di List) ed il nuovo campo **aList** è **privato**.

Delega

✓ Pro:

- L'incapsulamento non è violato: si accede agli oggetti solo attraverso la loro interfaccia.
- Flessibilità: consente di comporre facilmente comportamenti in fase di esecuzione e di cambiare il modo in cui questi comportamenti sono composti.

❖ Contro:

- È definita dinamicamente durante l'esecuzione attraverso oggetti che acquisiscono riferimenti ad altri oggetti: più inefficiente in esecuzione rispetto a software statico.

Ereditarietà

✓ Pro:

- Definita staticamente al momento della compilazione.
- Immediata da utilizzare (è supportata direttamente dal linguaggio di programmazione).

❖ Contro:

- È impossibile cambiare l'implementazione ereditata durante l'esecuzione.
- L'implementazione di una sottoclasse diventa strettamente dipendente dalla classe padre infatti qualsiasi modifica nell'implementazione della classe padre induce modifiche nella sottoclasse (devono essere ricompilate entrambe).

8.3.3 Design Pattern

Oltre al riuso nella scrittura del software esiste anche un riuso nella progettazione. Infatti, l'esperienza è un fattore fondamentale per una buona progettazione: un progettista esperto non parte mai da zero, riutilizza soluzioni che si sono dimostrate valide in passato.

La “comunità dei pattern” si propone come obiettivo la catalogazione di questi schemi ricorrenti in modo da costituire un dizionario a disposizione dei progettisti.

Alcuni cambiamenti possibili nel software includono:

- **Nuovo produttore o nuova tecnologia:** spesso le componenti usate per costruire il sistema vengono sostituite da altre di diversi vendori o da componenti che rispecchiano i cambiamenti del trend del mercato.
- **Nuove implementazioni:** quando i sistemi vengono integrati è possibile che ci si renda conto che il sistema non è abbastanza performante.
- **Nuove interfacce grafiche:** la poca usabilità del software rende necessario riprogettare l'intera interfaccia grafica.
- **Nuova complessità nel dominio di applicazione:** nel dominio di applicazione è necessario aggiungere alcune caratteristiche che rendono il sistema più complesso (es. passare da un sistema a singolo utente ad un sistema multi utente).
- **Errori:** molti errori vengono trovati solo quando gli utenti iniziano ad usare il prodotto software.

Nello sviluppo *OO*, i **design pattern** sono template di soluzioni (che gli sviluppatori hanno raffinato nel tempo) utilizzabili per risolvere un insieme di problemi ricorrenti.

Un *design pattern* è solitamente composto da poche classi correlate tramite delegazione ed ereditarietà per fornire una soluzione robusta e modificabile. Tali classi possono essere **adattate e ridefinite** per lo specifico sistema che si vuole realizzare (personalizzazione del sistema; riuso di soluzioni esistenti).

Ricorda: i *design pattern* non spiegano come modellare liste a puntatori o hashmap, né trattano progetti complessi relativi ad intere applicazioni o sottosistemi, ma descrivono oggetti comunicanti e classi, adattate in maniera tale da risolvere specifici problemi di progettazione.

Com'è fatto un design pattern?

Un design pattern ha 4 elementi:

1. un **nome** che lo identifica;
2. una **descrizione del problema** che risolve;
3. una **soluzione** che descrive gli elementi che costituiscono il progetto, le loro relazioni, responsabilità e collaborazioni;
4. un insieme di **conseguenze** che descrivono i risultati e i vincoli che si ottengono applicando il pattern.
Sono utili per valutare i trade-off e le alternative che devono essere considerate rispetto ai design goal affrontati.

A questo schema di base possono corrispondere schemi più dettagliati, per esempio quello **GoF**.

Struttura di un pattern: schema di descrizione Gof

- **Nome e classificazione:** importante per il vocabolario.
- **Scopo:** cosa fa il pattern e suo fondamento logico.
- **Nomi alternativi:** molti pattern sono conosciuti con più nomi.
- **Motivazione:** scenario che illustra un problema di progettazione e la soluzione offerta.
- **Applicabilità:** quando può essere applicato il pattern.
- **Struttura** (o modello): rappresentazione *UML* del pattern.
- **Partecipanti:** le classi/oggetti con le proprie responsabilità.
- **Collaborazioni:** come collaborano i partecipanti.
- **Conseguenze:** pro e contro dell'applicazione del pattern.
- **Implementazione:** come si può implementare il pattern.
- **Codice d'esempio:** frammenti di codice.
- **Pattern correlati:** relazioni con altri pattern.
- **Utilizzi noti:** esempi di utilizzo reale del pattern in sistemi esistenti.

Classificazione dei design pattern (GoF)

Design of Four

- **Pattern strutturali (structural patterns)**

Si occupano delle modalità di composizione di classi e oggetti per formare strutture complesse.

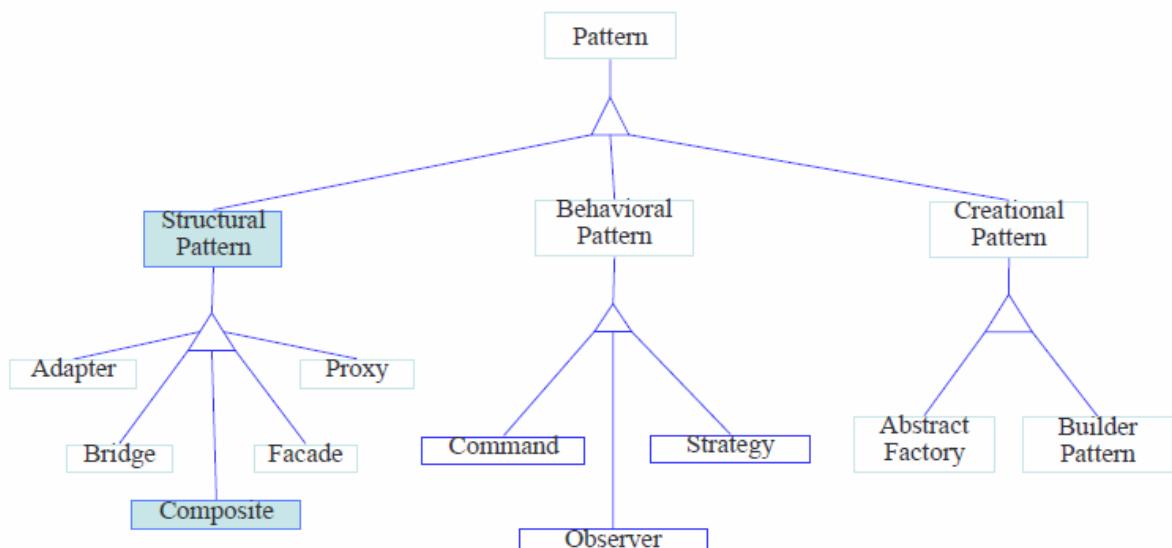
Riducono l'accoppiamento tra due o più classi, encapsulano strutture complesse, introducono una classe astratta per permettere estensioni future.

- **Pattern comportamentali (behavioral patterns)**

Si occupano di algoritmi e dell'assegnazione delle responsabilità tra oggetti che collaborano tra loro (chi fa che cosa?). Caratterizzano i flussi di controllo complessi difficili da seguire a run time.

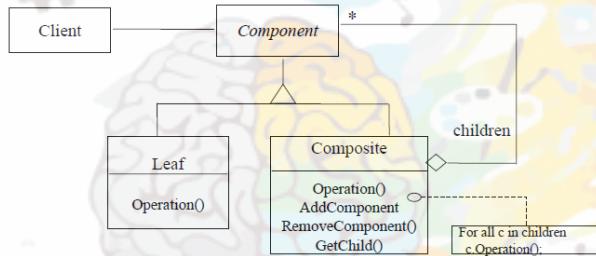
- **Pattern creazionali (creational patterns)**

Forniscono un'astrazione del processo di creazione degli oggetti. Aiutano a rendere un sistema indipendente dalle modalità di creazione, composizione e rappresentazione degli oggetti utilizzati.



Pattern strutturali: Composite pattern

- **Nome:** *Composite*.
- **Descrizione del problema:** compone gli oggetti in strutture ad albero di ampiezza e profondità variabile per rappresentare gerarchie *tutto-parte*. Inoltre, permette al client di trattare gli oggetti individuali (foglie) e gli oggetti composti (nodi interni) in maniera uniforme attraverso un'interfaccia comune.
- **Soluzione:** l'interfaccia *Component* specifica i servizi che sono condivisi tra *Leaf* e *Composite*. Un *Composite* ha un'associazione di aggregazione con *Component* e implementa ogni servizio iterando su ogni *Component* che contiene. I servizi *Leaf* fanno il lavoro effettivo.
- **Conseguenze:** il Client usa lo stesso codice per gestire i *Leaf* o i *Composite*. Inoltre, possono essere cambiati i comportamenti specifici di un *Leaf* e aggiungere nuove classi di un *Leaf* senza cambiare la gerarchia.

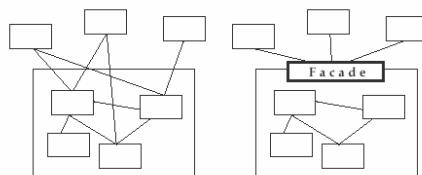


Ambienti per lo sviluppo di interfacce come *Swing* di Java utilizzano il **composite design pattern**. Infatti, in queste situazioni si ha a che fare con interfacce utente in cui si devono includere molte componenti grafiche all'interno di una finestra.

Nelle *Swing* alla radice della gerarchia di classi c'è un'interfaccia **Component** che fornisce un comportamento generico per tutte le componenti grafiche (es. spostamento o ridimensionamento). Al di sotto di questa classe troviamo componenti come bottoni o label di testo e una particolare componente grafica **Composite** che rappresenta un contenitore di *Component* (es. può contenere bottoni, label, checkbox, ecc..).

Pattern strutturali: Facade pattern

- **Nome:** *Facade*.
- **Descrizione del problema:** fornisce un'unica interfaccia per accedere ad un insieme di oggetti che compongono un sottosistema.
- **Possibili applicazioni:** un *facade pattern* dovrebbe essere usato da tutti i sottosistemi di un sistema software. Infatti, *facade* definisce tutti i servizi di un sottosistema, quindi, ci consente di fornire un'**architettura chiusa**.
- **Conseguenze:** fornisce un'interfaccia unificata alle interfacce all'interno di un sottosistema. Definisce un'interfaccia di livello superiore che renda il sottosistema più facile da usare (cioè estrae i dettagli del sottosistema); protegge i client dai componenti del sottosistema e promuove un accoppiamento debole tra client e componenti del sottosistema.

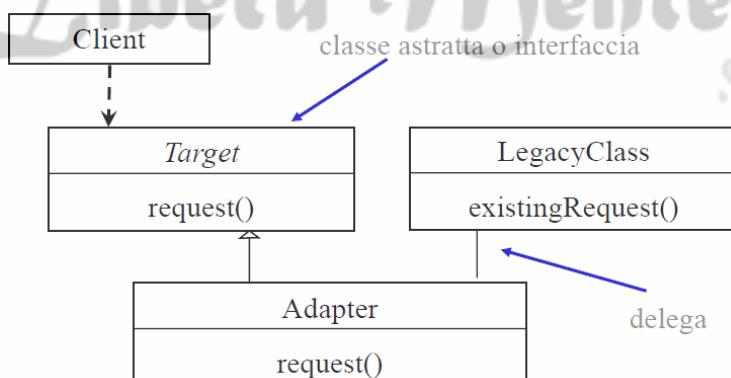


Pattern strutturali: Adapter pattern

Il **pattern adapter** viene usato quando è necessario inglobare componenti esistenti nel sistema. Questo tipo di soluzione viene adottata spesso nei sistemi interattivi, dove si usano finestre, dialog e bottoni. Tipicamente quando si riusano componenti legacy o off-the-shelf gli sviluppatori devono lavorare con codice che non possono modificare e che solitamente non è stato progettato per il loro sistema. Il *pattern adapter* converte l'interfaccia esistente di un sistema nell'interfaccia che gli sviluppatori si aspettano di trovare.

Una classe *Adapter* fa da connettore tra la classe che userà l'applicazione (Client Interface) e la classe legacy.

- **Nome:** *Adapter* (Wrapper).
- **Descrizione del problema:** converte l'interfaccia di una classe in un'interfaccia diversa che il cliente si aspetta, in maniera tale che classi diverse possano operare insieme nonostante abbiano interfacce incompatibili.
- **Possibili applicazioni:** encapsulare componenti esistenti per riutilizzare componenti da progetti precedenti o usare componenti off-the-shelf (COTS). Fornire una nuova interfaccia a componenti legacy esistenti (interface engineering, reengineering).
- **Soluzione:** ogni metodo dell'interfaccia Target è implementato in termini di richieste alla classe LegacyClass. Ogni conversione tra strutture dati o variazioni del comportamento sono realizzate dalla classe Adapter.
- **Conseguenze:** se Client utilizza Target allora può utilizzare qualsiasi istanza dell'Adapter in maniera trasparente senza dover essere modificato. Adapter lavora con la classe LegacyClass e con tutte le sue sottoclassi. L'*Adapter pattern* viene utilizzato quando l'interfaccia (es: Target) e la sua implementazione (es: LegcyClass) esistono già e non possono essere modificate.

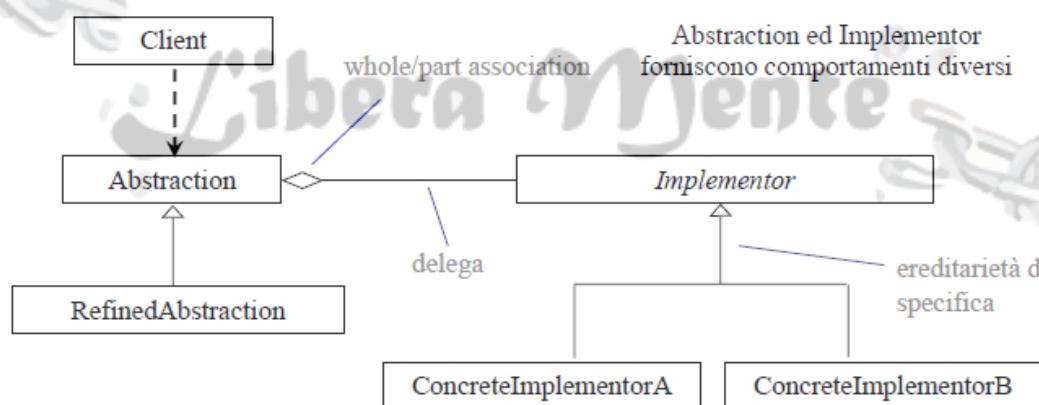


Pattern strutturali: Bridge pattern

Quando si usa uno sviluppo di tipo incrementale con testing e integrazione di sottosistemi creati da diversi sviluppatori, spesso si hanno dei ritardi nell'integrazione dei sottosistemi. Un problema simile si ha quando si vogliono avere diverse implementazioni dello stesso sottosistema (ad esempio una ottimizza la memoria e l'altra la complessità).

Per evitare ciò è possibile usare il **bridge pattern** che fa uso di un'**interfaccia comune** (**Implementor**) ereditata da tutti i sottosistemi che implementano la stessa funzionalità.

- **Nome:** *Bridge* (Handle/Body).
- **Descrizione del problema:** separa un'astrazione da una implementazione così che una diversa implementazione possa essere sostituita, eventualmente a runtime (es. testare differenti implementazioni della stessa interfaccia).
- **Possibili applicazioni:** utile per interfacciare un insieme di oggetti
 - quando l'insieme non è ancora completamente noto (ad es. in fase di analisi, design, testing);
 - quando c'è necessità di estendere un sottosistema dopo che il sistema è stato consegnato ed è in esecuzione (estensione dinamica).
- **Soluzione:** una classe *Abstraction* definisce l'interfaccia visibile al codice Client. *Implementor* è un'interfaccia astratta che definisce i metodi di basso livello disponibili ad *Abstraction*. Un'istanza di *Abstraction* mantiene un riferimento alla corrispondente istanza di *Implementor*. *Abstraction* e *Implementor* possono essere raffinate indipendentemente.
- **Conseguenze:**
 - **Disaccoppiamento tra interfaccia ed implementazione:** un'implementazione non è più legata in modo permanente ad un'interfaccia. L'implementazione di un'astrazione può essere configurata durante l'esecuzione. La parte di alto livello di un sistema dovrà conoscere soltanto le classi *Abstraction* e *Implementor*.
 - **Maggiore estendibilità:** le gerarchie *Abstraction* e *Implementor* possono essere estese indipendentemente.
 - **Mascheramento dei dettagli dell'implementazione ai client:** i client non devono preoccuparsi dei dettagli implementativi.



Adapter pattern VS Bridge pattern

Somiglianze: entrambi nascondono i dettagli lavorando sulle interfacce.

Differenze:

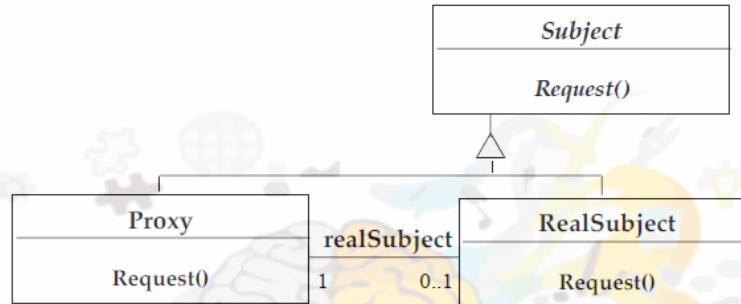
- L'adapter pattern è orientato alla realizzazione di componenti non correlati che lavorare insieme ed è applicato ai sistemi dopo che sono stati progettati.
- Il bridge pattern viene utilizzato nel momento in cui la parte di astrazione e implementazione devono lavorare in maniera indipendentemente.

Pattern strutturali: Proxy pattern

Il **proxy pattern** riduce i costi relativi all'accesso degli oggetti usando un oggetto fittizio (proxy) meno costoso che funge appunto da oggetto originale. Il *proxy* crea l'oggetto reale solo se l'utente lo richiede.

L'*ereditarietà dell'interfaccia* viene utilizzata per specificare l'interfaccia condivisa da Proxy e da RealSubject. La *delega* viene utilizzata per catturare e inoltrare qualsiasi accesso al RealSubject (se desiderato).

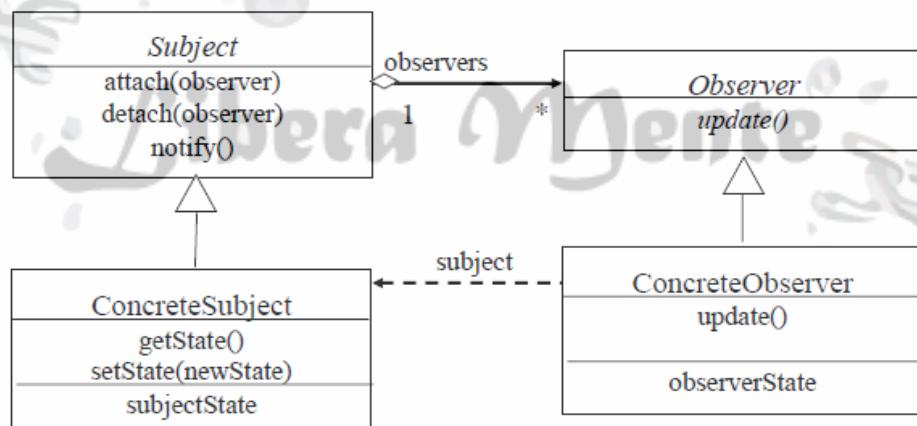
I **proxy pattern** possono essere utilizzati per l'invocazione remota e possono essere implementati con un'interfaccia Java.



Pattern comportamentali: Observer pattern

Definisce una dipendenza *uno-a-molti* tra gli oggetti in modo che quando un oggetto cambia stato, tutti gli oggetti dipendenti vengono avvisati e aggiornati automaticamente.

Tale pattern viene usato per mantenere la coerenza nello stato ridondante e per ottimizzare le modifiche batch (sempre per mantenere la coerenza).



Il **Subject** rappresenta lo stato attuale, gli **Observers** rappresenta le varie viste dello stato.

L'**Observer** può essere implementato come un'interfaccia Java, mentre, il **Subject** è una super classe

Pattern creazionali: AbstractFactory pattern

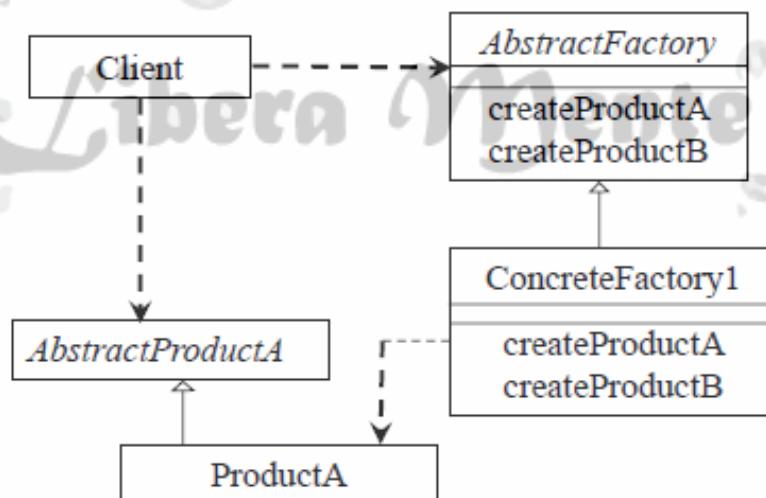
In una situazione in cui un unico sistema deve interagire con delle componenti esterne sviluppate da diversi produttori (es condizionatore Daikin e condizionatore Samsung) non è facile ottenere l'interoperabilità delle componenti con il sistema in quanto ogni componente/produttore offre le stesse funzionalità ma con delle interfacce differenti.

In una situazione simile è possibile usare l'**abstract factory pattern** in cui sono presenti più *factory* (una per ogni casa produttrice), una classe astratta per ogni tipo di prodotto e un'implementazione di tale classe per ogni prodotto concreto.

Tutte le *factory* presenti implementano un'interfaccia comune *AbstractFactory* che permette loro un comportamento standard.

L'applicazione (Client) in questo modo può usare, oltre ad un'interfaccia standard per accedere alle *factory*, anche un'interfaccia generica per usare i prodotti.

- **Nome:** *Abstract Factory*.
- **Descrizione del problema:** fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le loro classi concrete.
- **Soluzione:** una piattaforma è rappresentato con un insieme di *AbstractProducts*, ciascuno dei quali rappresenta un concetto. Una classe *AbstractFactory* dichiara le operazioni per creare ogni singolo prodotto. Una piattaforma specifica è poi realizzata da un *ConcreteFactory* ed un insieme di *ConcreteProducts*.
- **Conseguenze:**
 - **Isola le classi concrete:** *Abstract Factory* incapsula la responsabilità e il processo di creazione di oggetti prodotto e rende il client indipendente dalle classi utilizzate per l'implementazione degli oggetti. Permette di sostituire facilmente famiglie di prodotti a runtime (una factory concreta compare solo quando deve essere istanziata).
 - **Promuove la coerenza nell'utilizzo dei prodotti:** famiglia di prodotti correlati è progettata per essere utilizzata insieme. Il Client può creare prodotti solo utilizzando la *AbstractFactory*.
 - **Aggiungere nuove tipologie di prodotti è difficile:** devono essere create nuove realizzazioni per ogni *factory*.



8.4 Object Design: Fase di Specifica delle interfacce

Durante l'*object design* identifichiamo e raffiniamo gli **oggetti soluzione** per realizzare i sottosistemi definiti durante il *system design*.

Con il **system design**, il focus è l'identificazione di grandi parti di lavoro da assegnare ai vari team o sviluppatori.

Con l'**object design**, il focus è la specifica dei confini tra i vari oggetti usando la **specifiche delle interfacce**, in cui il focus è comunicare chiaramente e precisamente i dettagli di basso livello degli oggetti del sistema e descrivere l'interfaccia di ogni oggetto così che non ci sia la necessità di integrare oggetti realizzati da diversi sviluppatori.

Attività della specifica delle interfacce

Le attività della specifica delle interfacce sono:

1. Identificare gli **attributi**;
2. Specificare le **signature** e la **visibilità** di ogni operazione;
3. Specificare le **precondizioni**;
4. Specificare le **postcondizioni**;
5. Specificare le **invarianti**.

Object Constraint Language (OCL) consente di specificare *precondizioni*, *postcondizioni* e *invarianti*.

Euristiche e linee guida ci consentono di scrivere *vincoli leggibili*.

Overview della specifica delle interfacce

Tutti i modelli sin qui costruiti forniscono una visione parziale del sistema, molti pezzi mancano e altri sono da raffinare:

- Il **modello a oggetti di analisi**: descrive gli *oggetti entity, boundary e control* che sono visibili all'utente.
- La **decomposizione in sottosistemi**: descrive come questi oggetti sono partizionati in pezzi coesi realizzati da diversi team. Ogni sottosistema fornisce un insieme di servizi (ad alto livello) ad altri sottosistemi.
- Il **mapping Hardware/software**: identifica le componenti che costituiscono la macchina virtuale su cui costruiamo gli oggetti soluzione (es. classi e API definite da componenti esistenti).
- **Use case Boundary**: descrivono dal punto di vista dell'utente, casi amministrativi e eccezionali gestiti dal sistema.
- **Design pattern** (selezionati durante l'*object design reuse*): descrivono *object design* parziali che risolvono problemi specifici.

L'obiettivo dell'**object design** è di produrre un modello che integri tutte le informazioni in modo coerente e preciso.

Tale documento è chiamato **Object Design Document (ODD)**, il quale contiene la specifica di ogni classe per supportare lo scambio di informazioni consentendo di prendere decisioni sia tra i vari sviluppatori che con gli obiettivi di design.

Tipologie di sviluppatori

Precedentemente abbiamo parlato di sviluppatori in modo generico. E' possibile dividere gli sviluppatori in base al loro punto di vista:

- **Class implementor:** scrivono delle classi del sistema.
- **Class user:** usano classi già create da altri sviluppatori.
- **Class extender:** estendono classi già create da altri sviluppatori.

Attività dell'Object Design

Le attività dell'analisi dei requisiti è l'identificazione di attributi e operazioni senza specificare il loro tipo e i loro parametri.

L'**object design**, invece, si divide in tre attività:

1. **Aggiungere informazione sulla visibilità:** diversi sviluppatori hanno diverse necessità e non tutti accedono alle operazioni e agli attributi di una classe.
2. **Aggiungere informazione sui tipi e sulle signature:** il **tipo** di un attributo fornisce informazione sul range dei valori consentiti e le possibili operazioni. La **signature** fornisce informazioni sui parametri delle operazioni ed eventuali valori di ritorno.
3. **Aggiungere contratti:** consentono ai vari sviluppatori di condividere le stesse informazioni sulle classi.

1. Aggiungere informazione sulla visibilità

UML definisce 3 livelli di visibilità:

- : Privato (solo per *Class implementor*): ad un *attributo privato* può accedere solo la classe in cui è definito. Un'operazione privata può essere invocata solo dalla classe in cui è definita. Ad attributi e operazioni private non possono accedere sottoclassi o altre classi.

#: Protetto (*Class extender*): ad un *attributo o operazione protetto* può accedere solo la classe in cui è definito e ogni discendente della classe.

+: Pubblico (*Class user*): ad un *attributo o operazione pubblica* possono accedere tutte le classi (interfaccia pubblica).

Euristiche per l'Information Hiding

Si deve definire attentamente l'interfaccia pubblica per le classi applicando il principio **"Need to know"**: come regola bisognerebbe esporre in modo pubblico solo le informazioni strettamente necessarie.

Infatti, una operazione meno conosce e più bassa sarà la probabilità che sarà influenzata da qualche cambiamento e più facilmente la classe potrà essere cambiata.

Inoltre, bisogna trovare un *trade-off*: **Information hiding Vs efficienza**. Accedere a un *attributo privato* potrebbe essere troppo lento (per esempio per sistemi in real-time o giochi).

2. Tipi e signature

Il **tipo** di un attributo specifica il range dei valori che può avere quell'attributo e le operazioni che possono essere applicate. La **signature** dell'operazione è una tupla che fornisce informazioni sui tipi dei parametri e del valore di ritorno.

3. Aggiungere Contratti (precondizioni, postcondizioni, invarianti)

Spesso il tipo non è sufficiente a specificare il range dei valori consentiti di un attributo, per questo motivo si possono aggiungere i **contratti**. I *contratti* su una classe consentono a *class users, implementors* ed *extenders* di condividere le stesse assunzioni sulla classe.

I *contratti* includono 3 tipi di **vincoli**:

1. **Invariante**: un predicato che è sempre vero per tutte le istanze di una classe. *Invarianti* sono vincoli associati a classi o interfacce.
2. **Precondizioni**: sono predici associati ad una specifica operazione e deve essere vera prima che l'operazione sia invocata. *Precondizioni* sono usate per specificare vincoli che un chiamante deve soddisfare prima di chiamare un'operazione.
3. **Postcondizioni**: sono predici associati ad una specifica operazione e devono essere soddisfatti dopo che l'operazione è stata invocata. *Postcondizioni* sono usate per specificare vincoli che l'oggetto deve assicurare dopo l'invocazione dell'operazione.

Invarianti, precondizioni e postcondizioni possono essere usati per specificare senza ambiguità casi speciali o eccezionali.

8.4.1 Object Constraint Language (OCL)

Per esprimere **contratti** in modo più formale è possibile usare l'**OCL**. In OCL un **contratto** è una espressione che ritorna un valore booleano vero quando il contratto è soddisfatto, altrimenti falso.

Le espressioni hanno tutte questo template:

`context NomeClasse::firmaMetodo() tipoContratto: espressione.`

Per esprimere un **contratto di classe** e non di *metodo* si può omettere la firma del metodo e il doppio "due punti" dal template citato prima (es. `context NomeClasse pre: nomeAttributo = 0`).

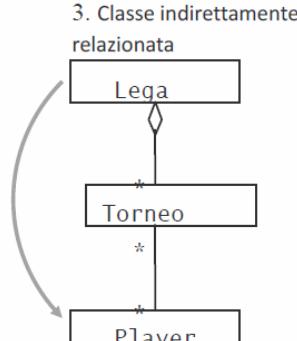
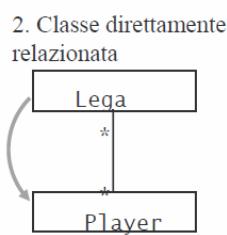
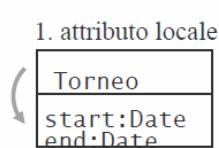
Nell'espressione è possibile usare **nomi di metodi**

(es. `context Hashtable::put(key,value) pre: !containsKey(key)`).

Nelle espressioni di **postcondizione** è possibile indicare il valore restituito da un'operazione o il valore di un attributo prima della chiamata del metodo usando `@pre.nomeMetodo()` oppure `@pre.nomeAttributo`.

(es. `context Hashtable::put(key,value) post: getSize() = @pre.getSize() + 1`).

È possibile esprimere vincoli che coinvolgono più di una classe mettendo una freccia verso la classe che fa parte dell'espressione. Infatti, un *class diagram* può essere costruito usando solo una combinazione di questi 3 tipi di navigazione.



Quando abbiamo a che fare con **associazioni molti e molti**, *OCL* fornisce le **collezioni**. Abbiamo 3 tipi di *collezioni*:

1. **OCL sets** (insiemi): **insieme non ordinato** di oggetti esprimibile con la forma `{elemento1, elemento2, elemento3}`. Sono usati quando si naviga una singola associazione. Inoltre, se l'associazione ha molteplicità 1 abbiamo un elemento e non un insieme e per riferirci ad una associazione usiamo il nome del ruolo (della classe) presente sull'associazione: se non esiste nessun nome utilizziamo il nome della classe relazionata denotata con la lettera minuscola.
2. **OCL sequences** (sequenze): **insieme ordinato** di oggetti esprimibile con la forma `[elemento1, elemento2, elemento3]`. Sono usati quando si naviga una singola associazione ordinata.
3. **OCL bags** (multinsiemi): insiemi usati per accumulare oggetti quando si accede a oggetti correlati in modo indiretto. La differenza con *Sets* è che gli oggetti possono essere presenti più volte o l'insieme può essere vuoto. (es. `{elemento1, elemento2, elemento2, elemento3}`). Inoltre, se non siamo interessati al numero di occorrenze di ogni oggetto nel *bag*, allora il *bag* può essere convertito in un insieme usando l'operatore `asSet(collection)`.

Per accedere alle collezioni, *OCL* fornisce delle operazioni standard usabili con la sintassi `collezione->operazione()`. Le più usate sono:

- `size()`: restituisce la dimensione della collezione;
- `includes(oggetto)`: restituisce `true` se l'oggetto appartiene all'insieme;
- `select(expression)`: restituisce una collezione che contiene gli oggetti su cui l'espressione è vera.
- `union(collection)`: restituisce una collezione che è l'unione della collezione in input e quella su cui viene eseguita l'operazione.
- `intersection(collection)`: restituisce una collezione che contiene gli elementi in comune tra i due insiemi.
- `asSet(collection)`: restituisce un insieme contenente gli elementi della collezione.

È possibile, inoltre, usare dei **quantificatori** come l'espressione:

- `collezione->forAll(condizione)`: restituisce `true` se la condizione è vera per tutti gli elementi della collezione;
- `collezione->exists(condizione)`: restituisce `true` se esiste almeno un elemento nella collezione che rispetta quella condizione.

8.4.2 Documentare l'*Object Design (ODD)*

Abbiamo due principali problemi di gestione durante *Object Design*:

1. **Aumento della complessità di comunicazione**: il numero di persone che prendono parte all'*OD* aumenta notevolmente. È necessario assicurare che le decisioni prese siano in accordo con gli obiettivi del progetto.
2. **Consistenza con le precedenti decisioni e documenti**: dettagliando e raffinando il modello a oggetti, gli sviluppatori possono rivedere alcune decisioni prese durante le fasi precedenti. Occorre tener traccia di questi cambiamenti e assicurarsi che tutti i documenti li riflettano in modo consistente

ODD serve per scambiare informazione sulle interfacce tra i team e come riferimento per il testing.

È rivolto a:

- Architetti che partecipano al system design;
- Sviluppatori che realizzano ogni sottosistema;
- Tester.

8.4.3 Riassunto: Object Design Document (**ODD**)

1. Introduzione

- 1.1 Object Design Trade-off
- 1.2 Linee Guida per la Documentazione delle Interfacce
- 1.3 Definizioni, acronimi e abbreviazioni
- 1.4 Riferimenti

2. Packages

3. Class interfaces

4. Class Diagram

5. Glossario

Introduzione

L'**introduzione** è una descrizione dell'analisi dei *trade-off* realizzati dagli sviluppatori (comprare vs. costruire, spazio di memoria vs. tempo di risposta, ecc) e vengono usate convenzioni e linee guida che servono a migliorare la comunicazione. Queste linee guida devono essere definite prima dell'inizio dell'*OD* e non devono variare:

- alle classi sono assegnati nomi singolari;
- i metodi nominati con verbi, i campi e i parametri con i sostantivi;
- lo status di un errore è restituito via un'eccezione, non con un valore di ritorno.

Packages

Questa sezione descrive la decomposizione di sottosistemi in **package**, l'organizzazione in file del codice e le dipendenze tra i **package** e il loro uso.

Class interfaces (Interfacce delle Classi)

Describe le classi e le loro **interfacce pubbliche**: overview di ogni classe, le sue dipendenze con altre classi e package, i suoi attributi e operazioni pubblici, casi eccezionali.

8.4.4 Javadoc

Le sezioni **Package** e **Class Interfaces** dell'*ODD* possono essere generati da un tool utilizzando i commenti del codice sorgente. Tale tool è **Javadoc**, il quale genera pagine web dai commenti del codice.

Gli sviluppatori annotano le dichiarazioni di interfacce e classi con commenti **tagged**; usando i vincoli è anche possibile includere *pre* e *post* condizioni nell'header dei metodi.

Con l'utilizzo di **Javadoc** teniamo insieme materiale per l'*ODD* e per il codice sorgente consentendo di mantenere la consistenza più facilmente.

Capitolo 9 – Mappare il modello nel codice

9.1 Object Design: Fase di Ristrutturazione del modello a oggetti

Con la **ristrutturazione del modello a oggetti** il *modello di object design* viene trasformato per migliorare la sua comprensibilità ed estensibilità.

Infatti, gli sviluppatori possono incontrare diversi **problemi di integrazione**:

- parametri non documentati possono essere stati aggiunti alle *API* in seguito a modifiche nei requisiti;
- attributi addizionali possono essere stati aggiunti all'*object model* ma non nello schema di memorizzazione persistente a causa di una comunicazione inefficiente.

Di conseguenza, il codice che si va ad implementare potrebbe essere lontano dal progetto originario e difficile da comprendere. Per questo motivo gli sviluppatori eseguono 4 tipi di trasformazioni sul modello a oggetti:

1. trasformare localmente il modello degli oggetti per migliorarne modularità e prestazioni;
2. trasformare le associazioni del modello a oggetti in collezioni di riferimenti a oggetti, in quanto i linguaggi di programmazione non supportano il concetto di associazione;
3. scrivere codice per identificare e gestire le violazioni dei contratti se il linguaggio di programmazione non supporta i contratti;
4. rivedere la specifica dell'interfaccia per soddisfare nuovi requisiti richiesti dal cliente.

Tutte queste attività non sono particolarmente complesse ma presentano aspetti ripetitivi e meccanici che possono indurre lo sviluppatore ad introdurre errori.

Una soluzione è di utilizzare un **approccio disciplinato** per evitare la degradazione del sistema quando lo sviluppatore deve:

- ottimizzare il modello delle classi;
- mappare le associazioni in collezioni;
- mappare i contratti delle operazioni in eccezioni;
- mappare il modello delle classi in uno schema di memorizzazione persistente.

9.2 Trasformazioni

Una **trasformazione** ha lo scopo di migliorare un aspetto del modello (ad es. la sua modularità) e preservare allo stesso tempo tutte le altre proprietà (ad es. le funzionalità). Generalmente una *trasformazione* è:

- localizzata;
- impatta su un numero ristretto di classi, attributi e operazioni;
- viene eseguita in una serie di semplici passi.

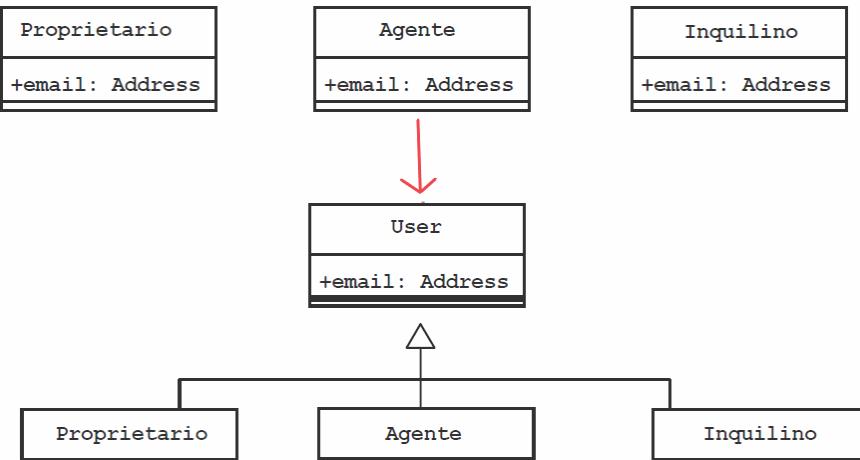
9.2.1 Tipi di Trasformazioni

Trasformazioni di modello

Le **trasformazioni di modello** operano sul modello a oggetti del sistema e non sul codice (es. convertire una stringa che rappresenta un indirizzo in una classe contenente i campi città, via, cap ecc.).

L'input e l'output di questa trasformazione è il **modello**. Lo scopo di questa trasformazione è quello di ottimizzare o semplificare il modello originale. Una trasformazione di questo tipo potrebbe aggiungere, rimuovere o rinominare classi, attributi, associazioni e operazioni.

Esempio:



L'attributo ridondante `email` viene eliminato creando una superclasse.

Refactoring

I **refactoring** sono trasformazioni che operano sul codice sorgente. Effettuano un miglioramento del codice senza intaccare le funzionalità del sistema.

Lo scopo di questa trasformazione è quello di aumentare la **leggibilità** e la **modificabilità**. Questa trasformazione si focalizza sulla trasformazione di un singolo metodo o campo di una classe.

Al fine di non cambiare il comportamento del sistema, il *refactoring* deve essere attuato mediante piccoli passi incrementali intervallati da **test**: l'utilizzo di **test driver** per ogni classe incoraggia lo sviluppatore a modificare il codice per migliorarlo.

Esempio:

Passo 1: spostare il campo `email` dalle sottoclassi alla superclasse.

Passo 2: spostare il codice di inizializzazione del campo `email` dalle sottoclassi alla superclasse.

Passo 3: spostare i metodi che manipolano il campo `email` dalle sottoclassi alla superclasse.

Forward Engineering

Il **forward engeneering** produce un template del codice sorgente corrispondente al modello a oggetti.

Quindi, come **input** possiede un **insieme di elementi del modello a oggetti**, mentre come **output** un **insieme di istruzioni di codice sorgente**.

Gli **obiettivi** sono:

- mantenere una corrispondenza fra il modello di design ad oggetti ed il codice;
- ridurre il numero di errori introdotti durante l'implementazione;
- ridurre gli sforzi di implementazione.

Molti costrutti (associazioni, attributi, ...) possono essere meccanicamente mappati nei costrutti del codice sorgente (es. classi e dichiarazioni di campi in Java), mentre il corpo dei metodi ed altri metodi privati vanno aggiunti dagli sviluppatori. Ad esempio:

- Ogni classe del *diagramma UML* è mappata in una *classe JAVA*.
- La relazione di *generalizzazione UML* è mappata in una istruzione `extends` (della classe Proprietario).
- Ogni attributo del *modello UML* è mappato in un campo privato della *classe Java* e due metodi pubblici per settare e visualizzare i valori del campo.

- Gli sviluppatori possono raffinare il risultato della trasformazione con comportamenti aggiuntivi (es: controllare se un campo è positivo).
- Il codice risultante da una trasformazione di questo tipo è sempre lo stesso (eccetto i nomi degli attributi).
- Se le classi sono progettate in modo adeguato si introducono meno errori.

Reverse Engineering

Il reverse engineering produce un modello a oggetti che corrisponde al codice sorgente.

Quindi, come **input** possiede un **insieme di elementi di codice sorgente**, mentre come **output** un **insieme di elementi del modello a oggetti**.

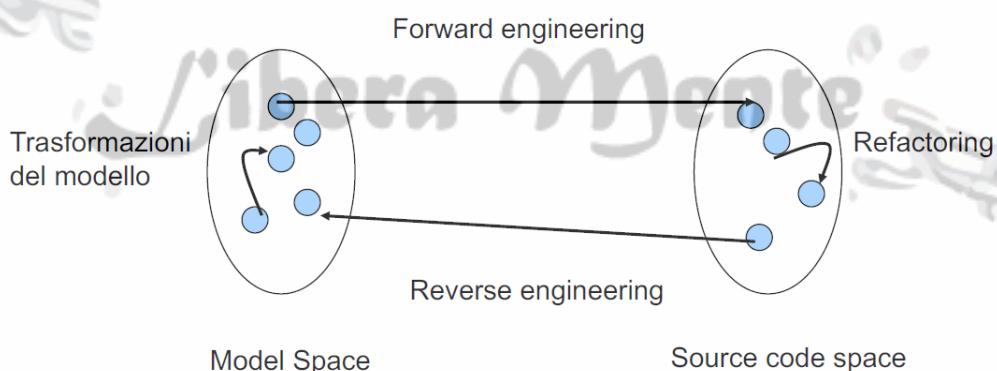
Questa trasformazione viene fatta per ricreare il modello per un sistema esistente quando il design del sistema viene perduto o non è mai stato creato.

Il *Reverse Engineering* è la trasformazione inversa del *Forward Engineering*:

- crea una classe UML per ciascuna classe;
- aggiunge un attributo per ciascun campo;
- aggiunge un'operazione per ciascun metodo.

Inoltre, il *reverse engineering* non crea necessariamente il modello originario (forward engineering può far perdere informazioni come le associazioni).

È supportata da **CASE tool** ma richiede comunque l'intervento dello sviluppatore per ricostruire un modello il più possibile vicino a quello originario.



9.2.2 Principi di Trasformazioni

Per evitare che le trasformazioni inducano in errori difficili da trovare e riparare, le trasformazioni dovrebbero seguire questi semplici principi:

- ogni trasformazione deve soddisfare un solo **design goal** per volta.
- ogni trasformazione deve essere locale e dovrebbe cambiare solo pochi metodi e poche classi.
- ogni trasformazione deve essere applicata singolarmente e non devono essere effettuate trasformazioni simultaneamente.
- ogni trasformazione deve essere seguita da una fase di **validazione/testing**.

9.3 Attività del mapping

Le attività riguardanti il mappaggio del modello a oggetti sul codice coinvolgono tutte una serie di **trasformazioni**. Le trasformazioni che verranno trattate sono:

- **Ottimizzare il modello di Object Design:** quest'attività ha lo scopo di migliorare le performance del sistema. Questo può essere ottenuto:
 - riducendo la molteplicità delle associazioni per velocizzare le query;
 - l'aggiunta di associazioni ridondanti per migliorare l'efficienza;
 - l'aggiunta di attributi derivati per migliorare i tempi di accesso agli oggetti.
- **Mappare le associazioni:** durante quest'attività mappiamo le associazioni in riferimenti o collezioni di riferimenti.
- **Mappare i contratti in eccezioni:** in questa fase descriviamo le operazioni che il sistema deve effettuare quando vengono violati i contratti.
- **Mappare l'Object Model in uno schema di memorizzazione:** in questa attività mappiamo il modello delle classi in uno specifico schema di memorizzazione (es. definire le tabelle).

9.3.1 Ottimizzare il modello di Object Design

Le attività della fase di **ottimizzazione** sono le seguenti:

Ottimizzare i cammini di accesso alle informazioni

L'ottimizzazione dei cammini si ottiene eliminando il ritardo causato da:

1. **attraversamento ripetuto di associazioni multiple:** in questo caso le operazioni più frequenti non dovrebbero richiedere molti traversamenti di associazioni (es. `metodo().metodo2().metodo3().metodo4()`), ma dovrebbero avere un'**associazione diretta** tra i due oggetti richiedente e fornitore.
La frequenza dei *path di accesso* è facile da determinare nell'*interface engineering* e *re-engineering*. Mentre può essere determinata durante il testing di sistema nel caso di *greenfield engineering*.
2. **attraversamento di associazioni di tipo “molti”:** in questo caso si riducono le associazioni di tipo “**molti**” al tipo “**uno**”. Se ciò non è possibile, si può provare a ordinare o indicizzare gli oggetti sul lato “**molti**” per migliorare il tempo di accesso.
3. **presenza di attributi mal collocati:** gli attributi, nella fase di analisi, potrebbero essere stati collocati male nelle classi se vengono acceduti solo tramite i metodi `get()` e `set()`. La soluzione è di spostare gli attributi nella classe che usa i metodi `get` e `set` per velocizzarne l'accesso.

Collassare gli oggetti in attributi

Dopo che il modello è stato ottimizzato alcune classi potrebbero contenere pochi attributi o operazioni. Queste classi, se sono associate ad una sola altra classe, possono essere trasformate in **attributi semplici**.

Ritardare le elaborazioni costose

Alcuni oggetti sono **costosi da creare** quindi la loro creazione può essere ritardata finché il loro contenuto non diventi necessario. Ad esempio, possiamo utilizzare il **design pattern Proxy** per caricare tutti i pixel di un'immagine nel momento in cui ne abbiamo bisogno.

Memorizzare il risultato di elaborazioni costose

A volte caricare grosse quantità di dati che non vengono usati può essere inutile. Conviene, quindi, caricare in memoria solo la parte di informazione che è strettamente necessaria e mettere il resto in una struttura temporanea, ad esempio, in un **attributo privato**.

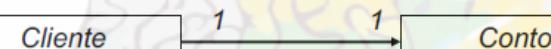
9.3.2 Mappare le associazioni in collezioni

Le associazioni viste nei diagrammi sono concetti astratti *UML* che nei linguaggi di programmazione (come Java) vengono mappate in **riferimenti** (nel caso di associazioni uno-a-uno) o **collezioni** (nel caso di associazioni uno-a-molti).

Durante l'**object design** trasformiamo le associazioni in riferimenti considerando la **molteplicità** e la **direzione** delle *associazioni*.

Associazioni uno-a-uno unidirezionali e bidirezionali

Se l'associazione tra due classi è **uno-a-uno unidirezionale**, solo la classe che utilizza le operazioni dell'altra avrà il riferimento all'altra classe.



```
public class Cliente {  
    private Conto conto;  
  
    public Cliente() {  
        conto = new Conto();  
    }  
  
    public Conto getConto() {  
        return conto;  
    }  
}
```

L'associazione si traduce inserendo un campo `conto` nella classe `Cliente` che referenzia l'oggetto `Conto`.

Un valore nullo nel campo `conto` può essere presente solo quando l'oggetto `Cliente` è creato.

Se invece l'associazione è **uno-ad-uno bidirezionale** ambedue le classi avranno un riferimento all'altra.



```
public class Cliente {  
    private Conto conto;  
  
    public Cliente() {  
        conto = new Conto(this);  
    }  
  
    public Conto getConto() {  
        return conto;  
    }  
}  
  
public class Conto {  
    private Cliente owner;  
  
    public Conto(Cliente owner) {  
        this.owner = owner;  
    }  
  
    public Cliente getOwner() {  
        return owner;  
    }  
}
```

I valori iniziali di `conto` e `owner` sono inizializzati e non vengono più modificati.

Associazioni uno-a-molti e molti-a-molti

Le associazioni **uno-a-molti** non possono essere realizzate usando un singolo riferimento.

Esempio:

supponiamo che ad un cliente possano corrispondere più conti.



```
public class Cliente {  
    private Set conti;  
  
    public Cliente() {  
        conti = new HashSet();  
    }  
  
    public void addConto (Conto c) {  
        conti.add(c);  
        c.setOwner(this);  
    }  
  
    public void removeConto (Conto c) {  
        conti.remove(c);  
        c.setOwner(null);  
    }  
}
```

```
public class Conto {  
    private Cliente owner;  
  
    public void setOwner (Cliente newOwner) {  
        if (owner != newOwner) {  
            Cliente old = owner;  
            owner = newOwner;  
  
            if (newOwner != null)  
                newOwner.addConto(this);  
  
            if (old != null)  
                old.removeConto(this);  
        }  
    }  
}
```

Nel caso in cui l'associazione sia **molti-a-molti** si usano delle collezioni come liste, insiemi o tabelle hash ed entrambe le classi hanno campi che sono collezioni di riferimenti ed operazioni per mantenere queste collezioni consistenti.

Esempio:

supponiamo che un conto possa essere intestato a più clienti.



```
public class Cliente {  
    private List conti;  
  
    public Cliente() {  
        conti = new ArrayList();  
    }  
  
    public void addConto (Conto c) {  
        if (!conti.contains(c)) {  
            conti.add(c);  
            c.addCliente(this);  
        }  
    }  
}
```

```
public class Conto {  
    private List clienti;  
  
    public Conto() {  
        clienti = new ArrayList();  
    }  
  
    public void addCliente (Cliente c) {  
        if (!clienti.contains(c)) {  
            clienti.add(c);  
            c.addConto(this);  
        }  
    }  
}
```

Classi associative

Le **classi associative** sono utilizzate in *UML* per contenere gli **attributi** e le **operazioni di una associazione**.

Queste possono essere mappate nel codice convertendole in normali classi e inserendo le classi dell'associazione come riferimenti.

9.3.3 Mappare i contratti in eccezioni

Un **contratto** è un vincolo su di una classe che deve essere soddisfatto prima di utilizzare tale classe. Molti linguaggi di programmazione *object-oriented* non forniscono supporto per i contratti per questo si utilizza il meccanismo delle **eccezioni** per segnalare e gestire le violazioni dei contratti.

In Java si solleva una eccezione con la parola chiave `throw` seguita da un *oggetto eccezione*.

L'*oggetto eccezione* fornisce un posto dove memorizzare le informazioni sull'eccezione, di solito un messaggio di errore. L'effetto di lanciare un'eccezione è duplice:

- interrompere il flusso di controllo;
- svuotare lo stack delle chiamate finché non si trova un blocco `catch` che gestisce l'eccezione.

Per ogni operazione nel contratto si deve:

- **controllare le precondizioni** prima dell'inizio del metodo con un test che lancia una eccezione se una precondizione non è verificata.
- **controllare la postcondizione** alla fine di ciascun metodo e lanciare una eccezione se il contratto è violato. Se più di una *postcondizione* non è soddisfatta, lanciare una eccezione solo per la prima violazione.
- **controllare le invarianti** allo stesso modo delle *postcondizioni*.
- **gestire l'ereditarietà** incapsulando il codice di controllo per *precondizioni* e *postcondizioni* in metodi separati che possono essere richiamati dalle sottoclassi.

Alcune euristiche per mappare i contratti in eccezioni sono le seguenti:

- si può omettere il codice di controllo per *postcondizioni* e *invarianti*: è ridondante inserirlo insieme al codice che realizza la funzionalità della classe, inoltre, non individua molti bug a meno che non venga scritto da un altro sviluppatore.
- si può omettere il codice di controllo per metodi privati e protetti se è ben definita l'interfaccia del sottosistema.
- concentrarsi sulle componenti che hanno una lunga durata: *oggetti Entity*, non *oggetti boundary* associati all'interfaccia utente.
- riusare il codice per il controllo dei vincoli: molte operazioni hanno precondizioni simili e si incapsula il codice per il controllo degli stessi vincoli in metodi così possono condividere le stesse classi di eccezioni.

9.3.4 Mappare l'Object Model in uno schema di memorizzazione persistente

I linguaggi di programmazione *object-oriented* di solito non forniscono un modo efficiente per memorizzare gli *oggetti persistenti*. È necessario mappare gli **oggetti persistenti** in strutture dati che possono essere memorizzate nei sistemi di gestione dei dati selezionati durante il **system design** (database o file).

Se usiamo **database object-oriented** non devono essere effettuate trasformazioni.

Se usiamo **database relazionali o file** è necessario:

- mappare il modello degli oggetti in uno schema di memorizzazione;
- fornire una infrastruttura per convertire gli oggetti in schemi di memorizzazione persistente e viceversa.

- ✓ Le **classi** vengono mappate in **tabelle** che hanno lo stesso nome della classe.
- ✓ Gli **attributi** vengono mappati in una **colonna** della tabella con lo stesso nome dell'attributo.
- ✓ Ogni **riga** della tabella corrisponde ad un'**istanza** della classe.

Mantenendo gli stessi nomi nel modello a oggetti e nelle tabelle garantiamo la tracciabilità fra le due rappresentazioni.

- ✓ Nelle *tabelle* bisogna scegliere una **chiave primaria** che identifichi univocamente un'istanza della classe.
- ✓ Per quanto riguarda le **associazioni uno-ad-uno** o **uno-a-molti** è necessario usare una **chiave esterna** che collega le due tabelle. Se l'associazione è **uno-a-molti** la **chiave esterna** è nella classe del lato "molti", se è **uno-ad-uno** vengono mappate usando la chiave esterna di una qualsiasi delle due tabelle. Le associazioni **molti-a-molti** sono implementate usando una tabella aggiuntiva che ha due colonne contenenti le chiavi esterne delle tabelle relative alle classi in relazione: tale tabella è detta **tabella associativa**.

Mappare le relazioni di ereditarietà

I *database relazionali* non supportano l'**ereditarietà**. Esistono due opzioni per mappare l'*ereditarietà* in uno schema di un database:

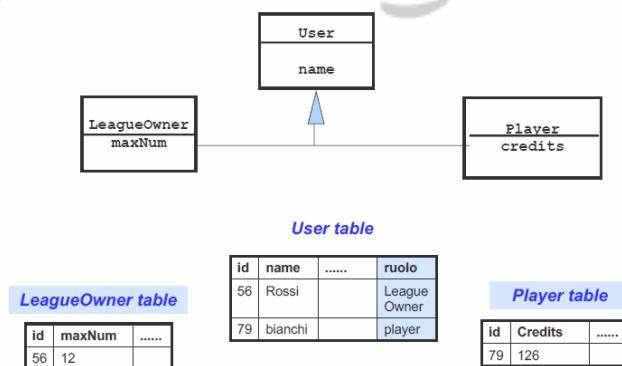
1. **Mapping verticale**: ogni classe è rappresentata da una tabella e utilizza una chiave esterna per collegare la tabella corrispondente ad una **sottoclasse** con quella corrispondente alla **superclasse**.

La tabella relativa alla **superclasse** include:

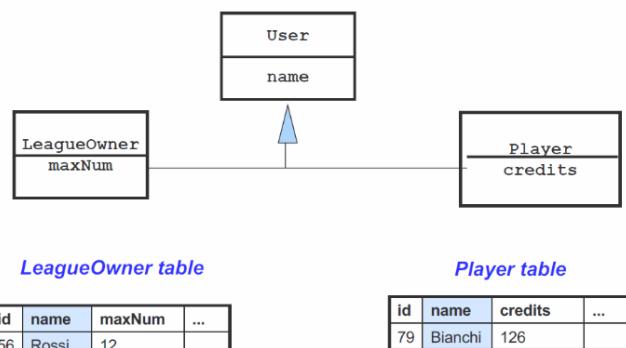
- una colonna per ogni attributo definito nella superclasse;
- una colonna addizionale che indica quale tipo di sottoclasse rappresenta quell'istanza.

La tabella relativa alla **sottoclasse** include una colonna per ogni attributo della sottoclasse.

Entrambe le tabelle hanno la stessa chiave primaria.



2. **Mapping orizzontale**: gli attributi della superclasse sono ricopiatati in tutte le sottoclassi e la superclasse viene eliminata.



Mapping Verticale

Utilizzando una tabella separata:

- ✓ possiamo facilmente aggiungere un attributo alla superclasse aggiungendo una colonna alla tabella superclasse;
- ✓ aggiungere una sottoclasse significa aggiungere una tabella per la sottoclasse con un attributo per ogni colonna della sottoclasse;
- ❖ ricercare tutti gli attributi di un oggetto richiede una operazione di Join.

Mapping Orizzontale

Duplicando le colonne:

- ✓ gli oggetti non sono frammentati fra più tabelle e le query sono più veloci;
- ❖ modificare lo schema è più complesso.

Trade-off tra mapping orizzontale e verticale

I trade-off tra i due meccanismi riguardano la **modificabilità** e il **tempo di accesso**.

In un **mapping orizzontale** la modificabilità è più semplice in quanto aggiungere un attributo alla superclasse richiede l'aggiunta di una colonna in una tabella e aggiungere una sottoclasse richiede di aggiungere una tabella che contiene solo gli attributi della sottoclasse. La soluzione di **mapping orizzontale** consente un più rapido accesso alle informazioni ma una più bassa modificabilità ed estensibilità.

D'altra parte, scegliere un **mapping verticale** porta ad una bassa efficienza in quanto caricare una sottoclasse dal database richiede l'accesso a due tabelle anziché una.

Per scegliere una o l'altra soluzione bisogna trovare dei compromessi tra **efficienza** e **modificabilità**.

9.4 Euristiche per effettuare trasformazioni

Utilizzare sempre il medesimo strumento per effettuare le trasformazioni:

ad esempio, se è stato utilizzato un particolare strumento CASE per trasformare le associazioni in codice, bisogna utilizzare lo stesso tool per modificare la molteplicità delle associazioni.

Mantenere traccia dei contratti nel codice sorgente e non solo nel modello di object design:

se si utilizza la specifica dei contratti come commento del codice sorgente è più probabile che tale specifica venga modificata se viene modificato il codice sorgente.

Utilizzare gli stessi nomi per gli stessi oggetti:

se un nome viene modificato nel modello bisogna modificarlo anche nel codice sorgente e/o nello schema della base di dati, ciò permette di mantenere la tracciabilità tra i diversi modelli.

Utilizzare delle linee guida per le trasformazioni:

se si riportano in un manuale le convenzioni che si vogliono utilizzare per le trasformazioni tutti gli sviluppatori potranno applicarle allo stesso modo.

9.5 Responsabilità

Nelle fasi di *trasformazione* ci sono vari ruoli che cooperano.

- Il **core architect** seleziona le trasformazioni che devono essere applicate in maniera sistematica.
- L'**architecture liason** è responsabile di documentare i contratti associati alle interfacce dei sottosistemi. Quando questi contratti cambiano è responsabile di comunicare i cambiamenti ai *class user*.
- Lo **sviluppatore** deve seguire le convenzioni dettate dal *core architect* e convertire il modello in codice sorgente.



Capitolo 10 – Testing

10.1 Verifica e convalida

Un software con zero difetti è impossibile da ottenere e garantire, per questo motivo è necessaria una attenta e continua **verifica e convalida**.

La **verifica** è l'insieme delle attività volte a stabilire se il software costruito soddisfa le specifiche (non solo funzionali). Quindi le specifiche devono esprimere in modo esauriente tutto ciò che il committente desiderava.

Alla fine, tutto deve essere **verificato**: documenti di specifica, di progetto, dati di collaudo, ..., programmi. Quindi, questa verifica si fa lungo tutto il processo di sviluppo, non solo alla fine.

La **convalida** stabilisce che il sistema soddisfa le esigenze vere dell'utente. Può essere svolta sulla specifica e/o sul sistema.

Per eseguire quindi queste operazioni *verifica e convalida* possiamo procedere in vari modi:

- **Prova di correttezza**: dobbiamo argomentare sistematicamente (in modo formale o informale) che il programma funziona correttamente per tutti i possibili dati di ingresso.
- **Testing**: particolare tipo di attività sperimentale fatta mediante l'esecuzione del programma, selezionando alcuni dati di ingresso e valutando i risultati. Il *testing* dà un riscontro parziale poiché il programma è provato solo per quei dati. Inoltre, è una tecnica dinamica rispetto alle verifiche statiche fatte dal compilatore. Quindi, l'obiettivo del testing è di trovare “**controesempi**” trovando *dati di test* che massimizzano la probabilità di scoprire errori durante l'esecuzione.

10.2 Testing

Il **testing** è l'attività che cerca le differenze tra il comportamento atteso dalle specifiche (o requisiti) del modello del sistema e il comportamento osservato dal sistema implementato.

Esistono diversi tipi di testing:

- ✓ **Testing di Unità**: trovare differenze tra *object design model* e corrispondente componente.
- ✓ **Testing Strutturale**: trovare differenze tra *system design model* e un sottoinsieme integrato di sottosistemi.
- ✓ **Testing Funzionale**: trovare differenze tra *use case model* e il sistema.
- ✓ **Testing di Performance**: trovare differenze tra requisiti non funzionali e le performance del sistema reale.

L'obiettivo è di progettare test per provare il sistema e rivelare problemi, quindi, massimizzare il numero di errori scoperti che consentirà agli sviluppatori di correggerli.

Questa attività va in contrasto con le altre attività svolte prima (analisi, design, implementazione) poiché queste sono attività “**costruttive**”, il testing invece tenta di “**rompere**” il sistema.

Inoltre, il testing dovrebbe essere realizzato da sviluppatori che non sono stati coinvolti nelle attività di costruzione del sistema.

Terminologia

Diamo ora alcune definizioni che verranno usate in seguito:

- **Affidabilità**: la misura di successo con cui il comportamento osservato di un sistema è conforme ad una certa specifica del relativo comportamento.
- **Fallimento** (*failure*): una deviazione del comportamento osservato rispetto a quello atteso.
- **Errore**: il sistema è in uno stato in cui qualsiasi operazione porta ad un fallimento.
- **Difetto** (*Bug/fault*): causa algoritmica o meccanica che ha portato a un comportamento errato. La causa algoritmica avviene a causa di una comunicazione cattiva tra i team oppure un'implementazione sbagliata della specifica da parte di un team.

Definizioni

Non tutti i ***fault*** generano ***failure***, una *failure* può essere generata da più *fault*, un *fault* può generare diverse *failure*.

Il termine ***defect*** (difetto) viene usato quando non è importante distinguere fra *fault* e *failure*.

Un programma è esercitato da un ***test case*** (insieme di dati di input). Un ***test*** è formato da un insieme di *test case*. L'esecuzione del *test* consiste nell'esecuzione del programma per tutti i *casi di test*.

Un *test* ha ***successo*** se rileva uno o più malfunzionamenti del programma.

Oracolo

Per effettuare un test confronteremo l'***oracolo***, il quale conosce il comportamento atteso, con il comportamento osservato. Abbiamo 2 tipi di *oracolo*:

- **Oracolo umano** si basa sulle specifiche o sul giudizio;
- **Oracolo automatico** è generato dalle specifiche (formali) oppure dallo stesso software però sviluppato da altri. Un altro tipo di *oracolo automatico* consiste nel mettere a confronto una versione con quella precedente (test di regressione) ed esaminare la differenza.

Come trattare gli *errori* e i *difetti*

Ci sono vari metodi per trattare gli errori, ad esempio:

- **Verifica**;
- **Introdurre ridondanza**;
- **Trasformare un bug in una feature**;
- **Patching**;
- **Testing**.

10.3 Tecniche per aumentare l'**affidabilità** di un sistema software

Ci sono varie tecniche per aumentare l'**affidabilità** di un sistema software:

Fault avoidance (prevenzione degli errori). Vengono usate tecniche per rilevare difetti senza eseguire il sistema. Questa tecnica prova a prevenire l'inserimento di errori nel sistema prima che lo stesso venga rilasciato.

Fault detection. Fanno parte di questa tecnica il **debugging**, il **testing** e la **review** svolti durante la fase di sviluppo del software allo scopo di trovare errori. Questa tecnica non cerca di risolvere i difetti ma ha il solo scopo di identificarli.

Fault tolerance (tolleranza agli errori). Questa tecnica assume che un sistema può essere rilasciato con bug e che i fallimenti del sistema possono essere gestiti a runtime.

Review

La **review** (revisione) è un controllo manuale di alcuni o di tutti gli aspetti del sistema senza mandarlo in esecuzione e ne esistono di due tipi:

1. **Walkthrough** (attraversamento): gli sviluppatori presentano il codice corredata di API e documentazione di una componente da testare al team di revisione. Il team di revisione commenta il codice aggiungendo dettagli riguardanti il mappaggio dell'analisi e del object design usando il *RAD*.
2. **Inspection** (ispezione): un'ispezione è simile al *walkthrough* ma la presentazione del codice non viene fatta dagli sviluppatori. Il team di revisione controlla le interfacce e il codice delle componenti rispetto ai requisiti. Il team è anche responsabile di controllare l'efficienza degli algoritmi rispetto ai requisiti non funzionali e di controllare se i commenti inseriti sono coerenti rispetto al comportamento del codice.

Debugging

Il **debugging** assume che un bug possa essere trovato partendo da un failure che non era stato previsto. Lo sviluppatore attraversa una sequenza di stati senza errore fino ad arrivare ad identificare uno stato di errore. A questo punto bisogna determinare il bug algoritmico o meccanico che ha causato questo stato.

Esistono due tipi di debugging:

1. **Debugging per correttezza**: cerca di trovare gli errori
2. **Debugging per performance**: cerca di valutare l'efficienza.

Testing

Il **testing** cerca di creare fallimenti o stati di errore in modo pianificato. Quindi, il **successo del testing** indica la situazione in cui il fault è trovato.

La caratteristica di un buon modello per il testing è che contiene *test case* che identificano fault. I *test case* dovrebbero includere un range ampio di valori di input, incluso input invalidi, e condizioni limite: questo approccio richiede molto tempo per il testing anche per sistemi di piccole dimensioni.

Il settore del testing è tormentato da problemi **indecidibili**. Un problema è detto **indecidibile** (irrisolubile) se è possibile dimostrare che non esistono algoritmi che lo risolvono.

In fine, il **testing esaustivo** (esecuzione per tutti i possibili ingressi) è impossibile da realizzare in generale poiché i tempi di esecuzioni sono troppo elevati. Per capire quando il testing termina possiamo utilizzare vari criteri:

- **Criterio temporale**: periodo di tempo predefinito;
- **Criterio di costo**: sforzo allocato predefinito;
- **Criterio di copertura**: percentuale predefinita degli elementi di un modello di programma; legato ad un criterio di selezione dei casi di test;
- **Criterio statistico**: MTBF (mean time between failures) predefinito e confronto con un modello di affidabilità esistente.

10.4 Concetti di testing

Una **componente** è una parte del sistema che può essere isolata per essere testata (un oggetto, un gruppo di oggetti, uno o più sottosistemi).

Un **test case** è un insieme di input e di output attesi che esercitano una componente con lo scopo di causare **fallimenti** e rilevare **faults**. Un *test case* ha 5 attributi:

1. **Name**. Per distinguere da altri *test case* (euristica: determinare il name a partire dal nome della componente o il requisito che si sta testando).
2. **Location**. Descrive dove il *test case* può essere trovato (un path name oppure un URL al programma da eseguire e il suo input).
3. **Input**. Descrive l'insieme di dati in input o comandi che l'attore del *test case* deve inserire.
4. **Oracle**. Il comportamento atteso dal *test case*, ovvero la sequenza di dati in output o comandi che la corretta esecuzione del test dovrebbe far avere.
5. **Log**. Memorizza varie esecuzioni del test e determina le differenze tra il comportamento atteso e quello osservato.

Una volta che i test sono identificati e descritti si determinano le relazioni tra questi. I *test case* possono avere le associazioni di:

- **Aggregazione**: un test case può essere composto di più sotto test;
- **Precedenza**: un test case deve essere eseguito prima di un altro.

Inoltre, i *test case* possono essere classificati in:

- **Whitebox**: si concentra sulla struttura interna della componente.
- **Blackbox**: si focalizza solo sull'input e l'output del programma senza andare ad indagare nel codice.

Test stub e test driver

I **test stub e driver** sono usati per sostituire e simulare parti mancanti del sistema.

Un **test stub** è un'implementazione parziale di componenti da cui dipende la componente testata.

Un **test driver** è una particolare implementazione di una componente che fa uso della componente sotto testing.

Un *test stub* deve fornire la stessa API del metodo della componente e ritornare un valore il cui tipo è conforme con il tipo del valore di ritorno specificato nella signature.

Se l'interfaccia di una componente cambia anche il corrispondente *test driver* e *test stub* devono cambiare.

Spesso *test stub* e *test driver* sono scritti dopo che la componente è stata realizzata (purtroppo se si è in ritardo non vengono scritti...).

Correzioni

Quando i test sono stati eseguiti e i fallimenti sono stati rilevati, gli sviluppatori cambiano la componente per eliminare il fault effettuando una **correzione**.

Una **correzione** è un cambiamento di una componente effettuato allo scopo di risolvere un errore. Una **correzione** potrebbe, in alcuni casi, introdurre nuovi errori.

Per evitare questo possiamo utilizzare alcune tecniche:

- **Problem tracking:** mantiene traccia degli errori riscontrati e delle relative soluzioni tramite una documentazione.
- **Regression testing:** vengono rieseguiti i test precedenti non appena viene corretta una componente.
- **Rational maintenance:** include una documentazione che specifica i motivi di un cambiamento o le motivazioni dello sviluppo di una componente.

10.5 Managing testing

Alla fine, gli sviluppatori dovrebbero rilevare e quindi riparare un numero sufficiente di bug tale che il sistema soddisfi i *requisiti funzionali* e *non funzionali* entro un limite accettabile da parte del cliente. Per fare ciò bisogna pianificazione il **testing activity** (diagramma di PERT che mostra le dipendenze).

Le attività sono documentate in 4 tipi di documenti:

1. **Test Plan** che si focalizza sugli aspetti manageriali;
2. ogni *test case* è documentato attraverso un **Test Case Specification**;
3. ogni esecuzione di un *test case* è documentata attraverso il **Test Incident Report**;
4. il **Test Report Summary** elenca tutti i fallimenti rilevati durante i test che devono essere investigati.

Documentazione Testing: Planning

❖ **Test Plan:** si concentra sugli aspetti gestionali del testing. Documenta lo scope, l'approccio, le risorse, programma le attività di test. I **requisiti** e i **componenti** da testare sono identificati in questo documento.

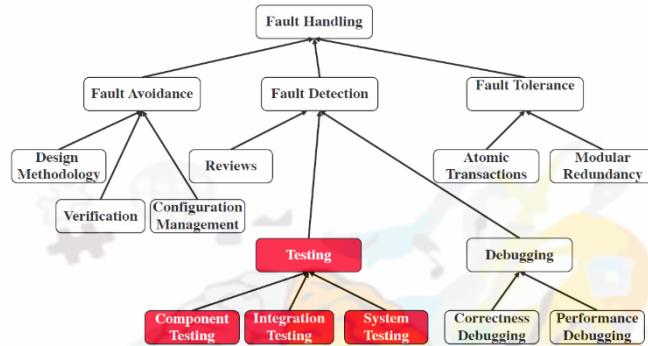
- | | |
|---|----------------------|
| 1. Introduction | 6. Approach |
| 2. Relationship to other documents | 7. Testing materials |
| 3. System overview | 8. Test Cases |
| 4. Features to be tested/not to be tested | 9. Testing Schedule |
| 5. Pass/Fail criteria | |

❖ **Test Case Specification:** documentiamo ogni test. Questo documento contiene gli input, i driver, gli stub e gli output previsti dei test, nonché le attività da svolgere.

1. Test case specification identifier
2. Test items
3. Input Specifications
4. Output Specifications
5. Environmental needs
6. Special procedural requirements
7. Intercase dependencies

Documentazione Testing: Execution Documents

- ❖ **Test Incident Report:** per qualsiasi test fallito, questo documento descrive il risultato effettivo rispetto a quello previsto e altre informazioni che spiegano il motivo per cui un test ha fallito. Questo documento è chiamato *incident report* perché si può verificare una differenza tra i risultati attesi e quelli effettivi a causa di una serie di motivi diversi.
- ❖ **Test Summary Report:** fornisce uno stato generale del test. In questo documento gli sviluppatori analizzano e danno una priorità a ogni errore e pianificano i cambiamenti nel sistema e nei modelli.



10.6 Tipi di testing

- ✓ **Unit Testing:** è eseguito dagli sviluppatori con l'obiettivo di testare ogni sottosistema in modo che sia codificato correttamente ed esegua la funzionalità prevista.
- ✓ **Integration Testing:** vengono testati gruppi di sottosistemi (insieme di classi) ed eventualmente l'intero sistema. Questo testing è eseguito dagli sviluppatori con l'obiettivo di testare l'interfaccia tra i sottosistemi.
- ✓ **System Testing:** testiamo il sistema nella sua interezza. È realizzato da sviluppatori con l'obiettivo di confermare che il sottosistema, nella sua interezza, sia codificato correttamente ed esegua le funzionalità previste.
- ✓ **Acceptance Testing:** valuta il sistema fornito dagli sviluppatori ed è eseguito dal cliente. Come obiettivo bisogna dimostrare che il sistema soddisfa i requisiti del cliente e che sia pronto per l'uso.

10.7 Tecniche di verifica

Abbiamo diverse tecniche di **verifica**:

Informali: il sistema viene incrementato e non è una buona pratica perché non abbiamo sotto controllo ciò che facciamo;

Analisi statica: sono tecniche a livello di unità e non sono scalabili. Alcuni esempi sono *Hand execution*, *Walk-Through* e *Code Inspection*, i quali ci forniscono dei tool per automatizzare il checking (errori sintattici e semantici) o degli standards.

Analisi dinamica (testing): si divide in *Black-box testing* (funzionale) e *White-box testing* (strutturale).

Random (uniforme): abbiamo uno spazio di input distribuito in maniera uniforme, quindi, tutti gli input hanno la stessa importanza. Questo metodo è svantaggioso quando la distribuzione dei faults non è uniforme.

Sistematico (non uniforme): seleziona gli input più importanti prendendo i rappresentanti delle classi di input. Lo spazio degli input è partizionato in classi.

10.8 Attività di testing

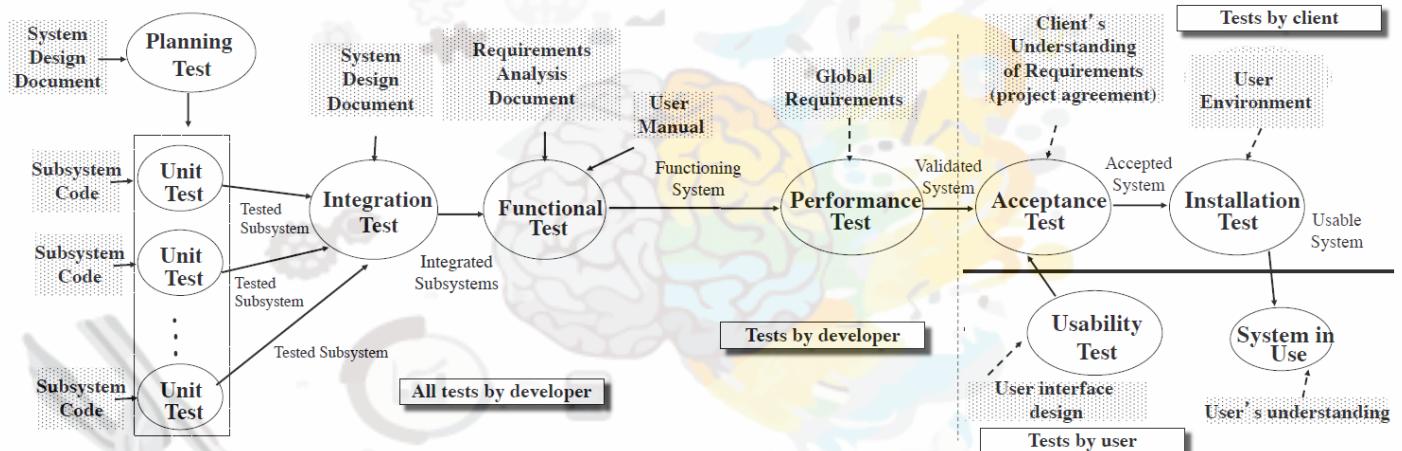
Component inspection. Trova i fault in una componente individuale attraverso l'ispezione manuale del codice sorgente.

Usability testing. Trova le differenze tra il sistema e l'attesa dell'utente per quanto riguarda l'uso del sistema.

Unit testing. Trova fault isolando una componente individuale usando *test stub* e *test driver* ed esercitando la componente tramite un test case.

Integration testing. Trova fault integrando insieme diverse componenti.

System testing. Si focalizza sul sistema completo, i suoi requisiti funzionali e non funzionali, e il suo ambiente.



Component inspection

Trova gli errori in singole componenti attraverso l'**ispezione** del codice sorgente. L'**ispezione** è condotta da un team di sviluppatori incluso l'autore della componente attraverso un meeting formale.

Un metodo proposto da *Fagan* è quello che divide l'ispezione in varie fasi:

- **Overview:** l'autore della componente presenta l'obiettivo e lo scope della componente e i goal dell'ispezione.
- **Preparazione:** i revisori analizzano l'implementazione della componente.
- **Meeting di ispezione:** una persona legge il codice della componente e il team di ispezione elabora proposte. Un moderatore tiene traccia delle cose dette. La maggior parte del tempo è passata a discutere, ma le soluzioni per riparare il bug non sono esplorate in questo punto.
- **Rework:** l'autore rivede e modifica la componente.
- **Follow-up:** il moderatore controlla la qualità della revisione e determina la componente che necessita di essere ispezionata di nuovo.

Il metodo proposto da *Fagan* è percepito come **time-consuming**, cioè eseguire tutte queste fasi richiede molto tempo.

Un'altra soluzione riguarda il processo di ispezione proposta da *Parnas*, chiamato **Active Design Review**, il quale si divide in:

- **Preparation:** i revisori analizzano l'implementazione della componente e individuano i fault. Compilano un questionario che testa la loro comprensione della componente.
- **Non è presente un *meeting di ispezione*:** l'autore incontra separatamente ogni revisionatore per collezionare feedback sulla componente.

I metodi di *ispezione* sono molto efficaci, infatti vengono individuati circa 85% dei fault.

Usability testing

L'**usability testing** viene effettuato poiché spesso le interfacce utente potrebbero risultare poco intuitive.

Gli sviluppatori selezionano prima un insieme di obiettivi, successivamente osservano gli utenti e raccolgono una serie di informazioni "misurando" le performance degli utenti (per esempio, il tempo per realizzare le attività) e le loro opinioni per identificare problemi specifici del sistema o collezionare idee per migliorarlo.

Ci sono 3 tipi di *usability testing*:

1. **Scenario test:** viene presentato uno **scenario** ad uno o più utenti e viene valutato in quanto tempo gli utenti lo comprendono. Lo *scenario* selezionato dovrebbe essere il più realistico possibile. Inoltre è possibile usare anche dei mock-up o dei storyboard.

Vantaggi: sono economici da realizzare e da ripetere.

Svantaggi: gli utenti non possono interagire direttamente con il sistema, i dati sono fissi.

2. **Prototype testing:** agli utenti finali viene presentato una parte del software che implementa gli aspetti chiave del sistema. Il *prototype testing* è di tipo:

- **vertical prototype**, se implementa solo un caso d'uso;
- **horizontal prototype**, se implementa un solo layer nel sistema che coinvolge più casi d'uso.

Vantaggi: forniscono una vista realistica del sistema all'utente e il prototipo può essere concepito per collezionare informazioni dettagliate.

Svantaggi: richiede un impegno maggiore nella costruzione rispetto agli scenari cartacei.

3. **Product test:** viene usata una versione funzionale del sistema e fatta visionare all'utente. Il test può essere affrontato solo dopo che una buona parte del sistema è stata sviluppata e richiede che il sistema sia facilmente modificabile.

Tutti e tre le tecniche hanno delle cose in comune poiché prevedono:

- Sviluppo degli obiettivi del test (confronto tra due stili di interazione, qualche help necessario, quale tipo di training è richiesto, ...);
- Selezionare un campione rappresentativo degli utenti finali;
- Una simulazione (o il reale) dell'ambiente di lavoro;
- Interrogazioni controllate ed estensive degli utenti alle prese con l'utilizzo del sistema attraverso i test;
- Collezione e analisi dei risultati qualitativi e quantitativi;
- Raccomandazioni su come migliorare il sistema.

Unit testing

Nell'**unit testing** i singoli sottosistemi o oggetti vengono testati separatamente. Questo comporta il vantaggio di ridurre il tempo di testing poiché vengono testate in parallelo piccole unità di sistema. I candidati da sottoporre al testing vengono presi dal modello a oggetti e dalla decomposizione in sottosistemi.

Gli *unit testing* possono essere divisi principalmente in 2 tipologie:

1. **White-box testing** cura solo il codice e la struttura senza guardare l'input e output.
2. **Black-box testing** guarda solo l'input e l'output senza curare il codice. Se per ogni input dato l'output corrisponde a quello predetto allora il modulo supera il test, inoltre, è quasi sempre impossibile generare tutti i possibili input ("test case").

Il *black-box testing* è possibile effettuarlo in due modi:

- 2.1 **Equivalence testing**: l'obiettivo è di ridurre il numero di *test case* dividendo gli input in classi di equivalenza (ad esempio, tutti gli input di numeri negativi e tutti gli input di numeri positivi).
- 2.2 **Boundary testing**: si selezionano degli input che sono "ai confini" per le classi di equivalenza (es. per i numeri positivi si testa il numero 1 o il più grande numero positivo rappresentabile).

Lo **svantaggio** di **Equivalence** e **Boundary test** è che non vengono testate combinazioni miste di input ma solo quelle interne alle classi di equivalenza.

- **Black-box è funzionale**: i *casi di test* sono determinati in base a ciò che il componente deve fare, la sua specifica. Non è necessario che esista il codice per determinare i dati di test ma basta la specifica che può essere formale o informale. Questi *casi di test* possono essere determinati in fase di progettazione. Esistono delle euristiche per scegliere le classi di equivalenza degli input del test.

Se l'**input valido è un range**, creiamo tre classi di equivalenza corrispondenti ai valori sotto, dentro e sopra il range di valori.

Se invece l'**input valido è un insieme discreto** vengono create due classi di equivalenza corrispondenti ai valori dentro e fuori l'insieme.

- **White-box è strutturale**: i casi di test sono determinati in base a come il componente è implementato, il codice.

Il **whitebox testing** è diviso in 4 sottotipi:

1. **Statement testing**: vengono testati i singoli statement del codice.
2. **Loop testing**: vengono dati vari input in modo da effettuare ciascuna delle seguenti operazioni: saltare un ciclo, entrare in un ciclo una sola volta e ripetere un ciclo più volte.
3. **Path testing**: viene costruito un diagramma di flusso del programma e si controlla se tutti i blocchi del diagramma vengono attraversati. Vengono, quindi, individuati dei test case che possano essere attraversati da tutti i blocchi del programma.
4. **Branch testing**: ci si assicura che ogni uscita da un'istruzione di condizione sia testata almeno una volta (es. il blocco `if` o il blocco `else` devono essere attraversati almeno una volta).

White-box Testing	Black-box Testing
<ul style="list-style-type: none"> • Possiamo avere un numero infinito di path da testare; • Questo tipo di test spesso controlla quello che è già stato fatto; • Impossibile rilevare i casi d'uso mancanti. 	<ul style="list-style-type: none"> • Possiamo avere un numero infinito di use case. • Spesso non è chiaro se i casi di test selezionati rivelano un errore particolare. • Non scopre casi d'uso nuovi ("features")

Integration testing

L'**integration testing** rileva bug che non sono stati rilevati durante l'**unit testing** focalizzandosi su un piccolo gruppo di componenti che sono integrate.

Non appena il piccolo sottoinsieme è perfettamente funzionante e non vengono evidenziati errori è possibile aggiungere componenti all'insieme.

Inoltre, sviluppare **test stub** e **test driver** per un **test di integrazione sistematico** è **time-consuming**.

Esistono varie strategie che decidono in che modo vengono scelti i sottoinsiemi di unità. La **strategia di testing** è scelta in base all'ordine in cui i sottosistemi sono selezionati per il testing e l'integrazione.

Ognuna di queste strategie è stata concepita per una decomposizione gerarchica del sistema, infatti, ogni componente appartiene ad una gerarchia di vari *layer*, ordinati in base all'associazione "Call".

Big bang integration (Non-incremental)

Le componenti sono testate prima singolarmente e poi messe insieme in un unico sistema. Il vantaggio è che non è necessario sviluppare **stub** o **driver** per rendere funzionale un sottoinsieme. È però difficile, in sistemi complessi, individuare la componente responsabile di un errore.

Bottom up integration

Vengono testate le componenti del livello più basso, vengono integrate e successivamente unite con le componenti del livello superiore. In questo caso non è necessario avere dei **test stub** in quanto si inizia ad integrare dal livello più basso a salire. I **test driver** sono usati per simulare le componenti dei layer più "in alto" che non sono stati ancora integrati.

Un **vantaggio** di questa tecnica è che gli errori nelle interfacce grafiche vengono trovati subito poiché, quando si testa l'interfaccia, si ha già un sistema sottostante funzionante e ben definito.

Quindi è utile per i:

- Sistemi Object-oriented;
- Sistemi real-time;
- Sistemi con rigide richieste sulle performance.

È **svantaggioso** per sistemi decomposti funzionalmente perché testa i sottosistemi più importanti alla fine.

Top down integration

Contrariamente al testing *Bottom-up*, il **top down testing** inizia ad integrare le componenti prima del livello più alto e successivamente si integrano quelle del livello inferiore.

Non sono necessari **test driver** ma solo **test stub** per simulare le componenti inferiori.

Vantaggi

I test cases possono essere definiti in termine delle funzionalità del sistema e si possono riutilizzare nelle varie iterazioni.

Svantaggi

Scrivere gli *stub* può essere difficile perché devono consentire tutte le possibili condizioni da testare. Uno dei problemi di questo tipo di testing è che è necessario scrivere molti *stub*, specialmente se il livello più in basso del sistema contiene molti metodi.

Una soluzione per evitare molti *stub* è il **Modified top-down testing strategy**, il quale testa individualmente ogni layer della decomposizione prima di fare il merge dei layer. Lo svantaggio di questa modifica è che sono necessari sia **test stub** che **test driver**.

Sandwich testing

Questa strategia combina *bottom-up* e *top-down* cercando di usare il meglio di queste. Il sistema viene diviso in 3 livelli:

1. il **livello target**;
2. un **livello sopra il target**;
3. un **livello sotto il target**.

In questo modo possiamo effettuare il *testing top-down* e *bottom-up* in parallelo con lo scopo di arrivare ad integrare il **target level**. Un problema di questo testing è che le componenti non vengono testate separatamente prima di integrarle.

La soluzione è di utilizzare una modifica: **Modified sandwich testing strategy**. Questa modifica testa i 3 layer individualmente prima di combinarli in test incrementali con gli altri. I test individuati dei layer, consistono di un gruppo di 3 test (in parallelo):

1. **Test del layer al top** (con stub per il layer target);
2. **Test del layer nel mezzo** (con driver and stub per i layer al top e al bottom rispettivamente);
3. **Test del layer al bottom** (con driver per il layer target).

I test dei layer combinati consistono in 2 test:

1. Il **layer al top** accede al **layer target**. Può riusare i test del *layer target* dai test individuali, sostituendo i driver con le componenti del *layer al top*.
2. Il **layer al bottom** è acceduto dal **layer target**. Può riusare i test del *layer target* dai test individuali, sostituendo gli stub con le componenti del *layer al bottom*.

System testing

Una volta che le componenti sono state integrate è testate prima con lo **Unit testing** e poi con l'**Integration testing**, è necessaria una fase di **testing globale**, chiamata System Testing, la quale assicura che il sistema completo è conforme ai requisiti funzionali e non funzionali. Le attività di questa fase sono le seguenti.

Test funzionale (o test dei requisiti)

Vengono controllate le differenze tra i requisiti funzionali e il sistema; il sistema è trattato come un **black box**. I test case vengono presi dai casi d'uso più importanti per l'utente finale e che hanno una buona probabilità di riscontrare un fallimento.

La differenza con *usability testing* è che mentre negli *usability testing* vengono trovate differenze tra il modello dei casi d'uso e le aspettative dell'utente, qui vengono trovate le differenze tra il modello dei casi d'uso e il comportamento osservato.

Performance Testing

Questo test trova le differenze tra gli obiettivi di design specificati e il sistema. L'obiettivo dei *test cases* è di rompere il sistema in modo da capire come questo si comporta quando è sovraccarico. Vengono effettuati i seguenti test:

- **Stress testing**: controlla se il sistema può rispondere a molte richieste simultanee.
- **Volume testing**: cerca errori quando il sistema elabora una grossa quantità di dati.
- **Security testing**: cerca di trovare falliche nella sicurezza del sistema usando tipici errori di sicurezza.
- **Timing testing**: prova a valutare il tempo di risposta del sistema.
- **Recovery test**: valuta l'abilità del sistema di ripristinarsi dopo una condizione di errore.

Pilot Testing

Durante questo testing il sistema viene installato e fatto usare da un insieme di utenti selezionati. Successivamente gli utenti vengono invitati a dare il loro feedback agli sviluppatori.

Un **alpha test** è un test effettuato nell'ambiente di sviluppo.

Un **beta test** è un test effettuato nell'ambiente di utilizzo.

Acceptance Testing

L'utente può valutare in tre modi un **acceptance testing**:

- **Benchmark test**: il cliente prepara una serie di test case su cui il sistema deve operare.
- **Competitor testing**: il sistema viene confrontato e testato rispetto ad un altro sistema concorrente.
- **Shadow testing**: viene testato il sistema legacy e il sistema attuale e gli output vengono messi a confronto.

Se il cliente è soddisfatto, il sistema è accettato, eventualmente con una lista di cambiamenti da effettuare.

Installation Testing

Dopo che il sistema è stato accettato, esso viene installato nel suo ambiente. In molti casi il **test di installazione** ripete i test case eseguiti durante il *function testing* e il *performance testing*. Quando il cliente è soddisfatto, il sistema viene formalmente rilasciato, ed è pronto per l'uso.

10.9 Partizionamento sistematico

Si cerca di partizionare il dominio di input in modo tale che da tutti i punti del dominio ci si attende lo stesso comportamento (e quindi si possa prendere come rappresentativo in un punto qualunque di esso). L'esperienza dimostra poi che è anche opportuno prendere punti sui confini delle regioni. Talvolta non è una partizione in senso proprio (le classi di valori hanno intersezione non vuota).

10.10 Testing classe di equivalenza

L'obiettivo è di ridurre ed eliminare le ridondanze nei partizionamenti delle classi di equivalenza. Quindi:

- l'intero set di input è coperto: **completezza**;
- le classi devono essere **disgiunte** per evitare la ridondanza;
- per ogni classe di equivalenza dobbiamo avere dei casi di test.

Weak Equivalence Class Testing: viene preso il valore di una variabile per ogni classe di equivalenza;

Strong Equivalence Class Testing: si basa sul **prodotto cartesiano** (quindi vengono trovate tutte le combinazioni).

10.11 Boundary value testing

Alcuni tipici errori di programmazione si verificano al confine di classi diverse. Questo è ciò su cui si concentra il **test del valore limite**. È più semplice ma complementare alle tecniche precedenti.

L'idea è di prendere il valore **minimo** e provo il **minimo + 1**, successivamente prendo il valore **massimo** e provo il **massimo - 1** e infine prendo uno **nominale**.

Per irrobustire il **testing** il numero di casi utilizzeremo un prodotto cartesiano formato da
 $\{min, min+, nom, max-, max\}$.

10.12 Category Partition

L'obiettivo del **Testing funzionale** è di trovare discrepanze tra il **comportamento attuale** delle funzioni del Sistema e il **comportamento desiderato**.

Per raggiungere questi obiettivi, i test devono essere eseguiti per tutte le funzioni del sistema. Inoltre, tali test devono essere progettati in modo da massimizzare la probabilità di trovare gli errori nel software.

Il **test funzionale** deriva da:

- la specifica del sistema;
- progettazione delle informazioni;
- codice.

L'idea è di partizionare il dominio dell'input e selezionare dei **test data** per ogni classe della partizione. Il problema di tutte queste tecniche è la **sistematicità**.

Innanzitutto, partiamo da una **specificia** la quale:

- è una rappresentazione uniforme delle informazioni di test di una funzione;
- può essere facilmente modificata;
- offre al tester un modo logico di controllare il volume dei test.

Alla fine, viene generato un **tool** che ci fornisce un modo automatizzato per produrre test. Il metodo si focalizza sia sulla copertura che sugli aspetti di rilevamento degli errori di test.

La strategia adottata per la **generazione dei test case** consiste nel dividere il sistema in funzioni che possono essere testate indipendentemente. Il metodo individua:

- i **parametri** (input) di ogni “funzione”;
- per ogni parametro individua **categorie** distinte, cioè le principali proprietà o caratteristiche.
- per ogni categoria, si devono individuare le **scelte** alle quali vengono assegnati dei valori: *normal values, boundary values, special values, error values*. Vengono individuati i **vincoli** che esistono tra le scelte, ossia in che modo l’occorrenza di una scelta può influenzare l’esistenza di un’altra scelta.
- vengono, poi, generati **Test frames** che sono delle combinazioni valide di scelte nelle categorie.
- per ogni *Test Frame*, si deve indicare l’**esito**: **errato**, se non consente il raggiungimento dell’obiettivo del requisito, **Corretto**, altrimenti.

