INGEGNERIA DEL SOFTWARE

Sommario

M1 – CONCETTI GENERALI DELL'INGEGNERIA DEL SOFTWARE

1.1 INTRODUZIONE

La produzione del software ha avuto diverse evoluzioni quali:

- ARTE, applicazioni sviluppate da singole persone e utilizzate dagli stessi sviluppatori;
- ARTIGIANATO, applicazioni sviluppate da piccoli gruppi specializzati per un cliente;
- INDUSTRIA, diffusione del software in diversi settori, crescita di dimensioni, mercato e concorrenza, necessità di migliorare la produttività e la qualità, gestione dei progetti ed evoluzione del software.

Fra un programma e un prodotto software vi sono varie differenze, quali:

- PROGRAMMA, l'autore è anche l'utente, molto spesso non è documentato e testato e non c'è un progetto alla base che precede l'implementazione. Non serve un approccio "formale";
- PRODOTTO SOFTWARE, usato da persone diverse dallo sviluppare, in genere è un software industriale il cui costo è circa 10 volte il costo del corrispondente programma e per essere sviluppato necessita di un approccio ingegneristico.

I prodotti software di dividono in:

- Prodotti generici, che sono sostanzialmente sistemi stand-alone prodotti da una organizzazione e venduti a un mercato di massa;
- Prodotti specifici, che sono sistemi commissionati da uno specifico utente e sviluppati specificatamente per questo da un qualche contraente.

Si hanno spese maggiori per la realizzazione di prodotti generici ma uno sforzo maggiore per lo sviluppo di prodotti specifici. Un prodotto software dunque non è solo codice ma è l'insieme di tutti 'gli artefatti' (codice, documentazione, cadi di test, specifiche di progetto, procedure di gestione, manuali utente, …) che lo accompagnano e che sono prodotti durante l'intero sviluppo.

Durante il processo di produzione di un prodotto software si riscontrano vari problemi, quali:

- COSTI, cioè il software stesso ha costi elevati che sono rappresentati dalle risorse utilizzate (ore di lavoro o "manpower" che risulta essere dominante, hardware, software e risorse di supporto), il testing impiega fino al 50% dei costi di sviluppo e la manutenzione costa più dello sviluppo stesso, ciò si evince dai costi delle varie manutenzioni attuate su un sistema rimasto a lungo in esercizio;
- RITARDI, molti progetti sono "runaway" cioè in ritardo o fuori dal budget stimato;
- ABBANDONI, molti progetti vengono abbandonati per via di stime errate sul tempo e sui costi di produzioni necessari;
- AFFIDABILITÀ, spesso il software è inaffidabile e molti malfunzionamenti sono rilevati solo durante l'operatività del sistema;

Per via di tutte queste problematiche si ha la necessità di un approccio ingegneristico, cioè si ha la necessità di applicare principi ingegneristici alla produzione software per sviluppare il *giusto* prodotto al *giusto* costo nel tempo *giusto* e con la *giusta* qualità.

L'ingegneria del software ha come scopo la costruzione di software di grandi dimensioni e di notevole complessità sviluppati tramite lavoro di gruppo, progetti software di questo tipo hanno tipicamente versioni multiple, lunga durata e frequenti cambiamenti (eliminazione di difetti, adattamento a nuovi ambienti, miglioramenti e nuove funzionalità). Molto importante risulta essere il contesto, la maggior parte dei software è collocata all'interno di un "sistema" misto di hardware/software, l'obiettivo finale di chi produce il software è di creare un sistema che soddisfi in modo globale i requisiti degli utenti che vengono coinvolti nella definizione di questi ultimi, dunque risulta necessaria una conoscenza del dominio applicativo che è essenziale per un efficace sviluppo del software che risulta essere utile solo quando riesce ad addensare nei suoi algoritmi la conoscenza del dominio applicativo in caso contrario risulta essere inutile e dannoso.

L'ingegneria del software si basa su alcuni principi, quali:

- RIGORE, che è un concetto primitivo molto relativo alla precisione e all'accuratezza;
- FORMALITÀ, che è un concetto che va oltre il rigore molto vicino al fondamento matematico
- SEPRAZIONE DI ASPETTI DIVERSI, affrontare separatamente i vari aspetti di un problema complesso;
- MODULARITÀ, suddividere un sistema complesso in parti più semplici
- ASTRAZIONE, si identificano gli aspetti cruciali in un certo istante ignorando gli altri aspetti;
- ANTICIPIAZIONE DEL CAMBIAMENTO, la progettazione deve favorire l'evoluzione del software
- GENERALITÀ, tentare di risolvere il problema nella sua accezione più generale
- INCREMENTALITÀ, lavorare per passi successivi.

E si occupa dei metodi, delle metodologie, dei processi e degli strumenti per la gestione professionale del software

- METODO (O TECNICA), è un procedimento generale per risolvere classi di problemi specificati di volta in volta (metodo di Newton, metodo di Montecarlo)
- METODOLOGIA, insieme di principi, di metodi, degli elementi di cui una o più discipline si servono per garantire la correttezza e l'efficacia del proprio procedere (cicli di vita del software).
- STRUMENTO (TOOL), ci si riferisce ad un artefatto, un sistema per fare qualcosa in modo migliore (ad esempio un cavatappi per stappare una bottiglia, tutti i supporti software pratici all'applicazione)
- PROCEDURA, una combinazione di strumenti e metodi che assieme permettono di produrre un certo prodotto;
- PARADIGMA, un particolare approccio o filosofia per fare qualcosa.

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

Un processo software è un metodo per sviluppare del software, è un insieme organizzato di attività che sovrintendono alla 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) ed è descritto da un modello che può essere informale, semi-formale o formale (maturità del processo). Secondo la definizione dello standard IEEE 610.12-1990 un processo di sviluppo software è un processo con il quale le esigenze degli utenti si trasformano in un prodotto software, questo processo prevede la traduzione delle esigenze degli utenti in requisiti software, la trasformazione dei requisiti software in progettazione, l'implementazione del design in codice, il testing del codice e, a volte, l'installazione e la verifica del software per l'utilizzo.

I progetti software vengono classificati per classi:

- SMALL, costituito da 2000 linee di codice con un team generalmente composto da 1-2 persone con un tempo di produzione di 4-5 mesi;
- INTERMEDIATE, costituito da 8000 linee di codice con un team composto da 2-6 persone con una durata di produzione di 8-9 mesi;
- MEDIUM, costituito da 32000 linee di codice con un team composto da 6-16 persone con una durata di produzione di 14 mesi;
- LARGE, costituito da 128000 linee di codice con un team composto da 16-51 persone con una durata di produzione di 24 mesi;
- VERY LARGE, costituito da 512000 linee di codice con un team composto da 60-157 persone con una durata di produzione di 41-42 mesi.

I progetti software necessitano dunque di un insieme di attività di back office (attività di gestione operativa di un'azienda, tutto ciò che il cliente non vede ma che consente di realizzare i prodotti e i servizi a lui destinati) e front office (attività che gestiscono l'interazione con il cliente) aziendale svolte tipicamente da figure professionali quali i project volte all'analisi (project monitoring), progettazione, pianificazione (project planning) e realizzazione degli obiettivi che ci si propone di realizzare con il progetto software, tali attività nel loro insieme prendono il nome di project management.

Per supportare le attività di un processo software ci sono opportuni sistemi software che si compongono di strumenti appositi che si distinguono in due tipologie:

- UPPER-CASE, strumenti che supportano le attività delle fasi di analisi e specifica dei requisiti e progettazione di un processo software. Includono editor grafici per sviluppare modelli di sistema, dizionari dei dati per gestire entità del progetto etc;
- LOWER-CASE, strumenti che supportano le attività delle fasi finali del processo, come programming, testing e debugging. Includono generatori di graphical UI per la costruzione di interfacce utente, debuggers per supportare la ricerca di program fault, traduttori automatici per generare nuove versioni di un programma etc..

1.2 CICLI DI VITA DEL SOFTWARE

Il ciclo di vita del software è quel periodo di tempo che inizia quando un prodotto software viene concepito e termina quando il prodotto non è più disponibile per l'utilizzo, esso si compone tipicamente di una fase concettuale (concept phase), una fase dei requisiti (requirement phase),

una fase di progettazione (design phase), una fase di implementazione (implementation phase), una fase di test (test phase), una fase di installazione e verifica (operation and maintenace phase) e, a volte, una fase di ritiro (retirement phase), queste fasi possono sovrapporsi tra loro o essere eseguite in modo iterativo a seconda dell'approccio di sviluppo del software utilizzato.

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 di processo software sono precise, formalizzate descrizioni di dettaglio delle attività, degli oggetti, delle trasformazioni e degli eventi che includono strategie per realizzare ed ottenere l'evoluzione del software.

Esistono vari modelli di CVS nati negli ultimi 40 anni, quali: Cascata (Waterfall), prototipazione (prototyping), Approcci evolutici (Incremental delivery), Modello a spirale (Spiral model). Sono molti gli aspetti che influenzano la definizione del modello che un prodotto software dovrà adottare, quali: specificità dell'organizzazione produttrice, competenza (know-how), area applicativa del progetto, strumenti di supporto e ruoli produttore/committente.

Un CVS si articola in varie fasi e adottando una vista di alto livello tali fasi possono essere suddivise generalmente in:

- DEFINIZIONE (si occupa del COSA), fase che comprende la determinazione dei requisiti, informazioni da elaborare, funzioni e prestazioni attese, comportamento del sistema, interfacce, vincoli progettuali, criteri di validazione;
- SVILUPPO (si occupa del COME), fase che comprende definizione del progetto, dell'architettura software, della strutturazione dei dati e delle interfacce e dei dettagli procedurali, traduzione del progetto nel linguaggio di programmazione e collaudi;
- MANUTENZIONE (si occupa delle MODIFICHE), fase che comprende correzioni, adattamenti, miglioramenti, prevenzione etc.

Il **MODELLO A CASCATA (WATERFALL),** molto popolare negli anni 70 fu creato prendendo ispirazione dall'industria manifatturiera, è caratterizzato da una progressione sequenziale (in cascata) di fasi, senza ricicli, al fine di controllare al meglio tempi e costi, definisce e separa le varie fasi e attività del processo producendo un overlap minimo o nullo fra le fasi e consente un controllo dell'evoluzione del processo attraverso l'utilizzo di attività trasversali alle diverse fasi. Per ogni fase raccoglie un insieme di attività omogenee per metodi, tecnologie, skill del personale, e così via, ogni fase è caratterizzata dalle attività (tasks), dai prodotti di tali attività (deliverables), dai controlli relativi (quality control measures), la fine di ogni fase è un punto rilevante del processo (milestone), i semilavorati output di una fase sono input alla fase successiva e risultano non modificabili se non innescando un processo formale e sistematico di modifica. Le fasi del modello a cascata possono essere suddivise in:

- FASI ALTE DEL PROCESSO, che comprendono le fasi di:
    1. STUDIO DI FATTIBILITÀ, che consiste di effettuare una valutazione preliminare di costi e benefici e varia a seconda della relazione committente/produttore. Ha come **obiettivo** quello di stabilire se avviare il progetto, individuare le possibili opzioni e le scelte più adeguate, valutare le risorse umane e finanziare necessarie, e produce come **output** un documento di fattibilità che fornisce una definizione preliminare del problema, scenari – strategie alternative, costi, tempi e modalità di sviluppo per ogni alternativa;

2. ANALISI DEI REQUISITI, che consiste di effettuare una analisi completa dei bisogni dell'utente e del dominio del problema coinvolgendo sia il committente che gli ingegneri del software. Ha come **obiettivo** quello di descrivere le funzionalità e le caratteristiche di qualità che l'applicazione deve soddisfare, e produce come **output** un documento di specifica dei requisiti;
3. PROGETTAZIONE, che consiste di definire una struttura opportuna per il software scomponendo il sistema in componenti e moduli ai quali vengono allocate delle funzionalità e tra i quali vengono definite le relazioni. Ha come **obiettivo** quello di stabilire il come il software soddisferà i requisiti dell'utente, e produce come **output** il documento di specifica di progetto.

- FASI BASSE DEL PROCESSO, che comprendono le fasi di:
    4. SVILUPPO E TEST DI UNITÀ, che consiste nella codifica nel linguaggio scelto di ogni modulo testandolo successivamente in isolamento;
    5. INTEGRAZIONE E TEST DI SISTEMA, che consiste nella composizione dei moduli nel sistema globale e verificarne il corretto funzionamento e successivamente procedere ad effettuare un α-test rilasciando il sistema internamente al produttore e in seguito un β-test rilasciando il sistema a pochi utenti selezionati;
    6. DEPLOYMENT, che consiste nella distribuzione e gestione del software presso l'utenza;
    7. MANUTENZIONE, che consiste nell'evoluzione del software seguendo l'esigenza dell'utenza, ciò comporta ulteriore sviluppo racchiudendo in sé nuove iterazioni di tutte le fasi precedenti.

Tale modello presenta vantaggi e svantaggi, fra i vantaggi ci sono la definizione di molti concetti utili come semilavorati, fasi, ecc., vi sono bassi rischi nello sviluppo di applicazioni familiari con tecnologie note, ha rappresentato un punto di partenza per lo studio dei processi software e risulta essere facilmente comprensibile e applicabile, fra gli svantaggi ci sono sicuramente gli alti rischi per sistemi nuovi non familiari per problemi di specifica e progetto, l'aspetto che l'interazione con il committente avviene solo all'inizio e alla fine, ciò comporta il congelamento dei requisiti nella fase di analisi e molto spesso l'utente si accorge di quello che vuole a software rilasciato, e anche l'installazione del sistema software diventa possibile solo quando è totalmente terminato e ciò significa che né il management né l'utente possono giudicare il sistema rispetto alle loro aspettative prima del completamento del software.

Nella realtà le specifiche del prodotto sono incomplete o inconsistenti e l'applicazione evolve durante tutte le fasi, non esiste una netta separazione tra le fasi di specifica progettazione e produzione e vi sono overlap e ricicli.

Una variante del Waterfall model è **V&V E RETROAZIONE (FEEDBACK),** tale modello aggiunge al classico modello a cascata un'attività di V&V ad ogni fase, cioè Verification, che risponde alla domanda "stiamo costruendo il prodotto corretto?" cioè verifica se il sistema è conforme a ciò che si è specificato e Validation, che risponde alla domanda "stiamo costruendo il prodotto giusto?" cioè se il prodotto che si sta costruendo è quello giusto per l'utente che dovrà utilizzarlo. Ogni fase dunque produce un feedback, che può essere inviato ad una qualsiasi fase precedente, che indica sostanzialmente l'esito della fase e se questo risulta essere negativo si può ritornare nella fase precedente (retroazione).

IL MODELLO A V, dispone logicamente le varie fasi del processo formando una vera e propria v da cui deriva proprio il nome del modello, a sinistra sono situate le fasi di analisi e specifica dei requisiti, progetto di sistema e progetto di dettaglio, al centro è situata la fase di codifica e a destra ci sono le fasi di testing di unità e integrazione, testing di sistema, testing di accettazione e deployment e manutenzione, fra le fasi situate nelle due parti della v sinistra e destra vi sono delle attività collegate in modo bidirezionale che sono validazione dei requisiti connesso alla fase di analisi dei requisiti e alla fase di testing di accettazione e altre due attività di verifica design connesse una alla fase di progetto sistema e alla fase di testing di sistema e l'altra attività è connessa alla fase di progetto di dettaglio alla fase di testing di unità e integrazione. In tale modello se si trova un errore in una fase di destra si può rieseguire il pezzo della v collegato ed è possibile iterare le fasi per migliorarne i derivelables.

MODELLI BASATI SU PROTOTIPO, tali modelli basano il loro funzionamento sui prototipi, un prototipo è un mezzo con il quale si interagisce con il committente, sostanzialmente è una prima implementazione più o meno incompleta, per accertarsi di aver compreso le sue richieste, per poterle specificare meglio tali richieste e per valutare la fattibilità del prodotto. Vi sono varie tipologie di prototipazione:

- MOCK-UPS, che consiste nel produrre l'interfaccia utente in modo completo senza implementare nessun requisito, ciò consente di definire con completezza e senza ambiguità i requisiti;
- BREADBOARDS, che consiste nell'implementare le funzionalità senza alcun accenno alle interfacce utente producendo feedback su come implementare la funzionalità.
- PROTOTIPAZIONE "ESPLORATIVA", che consiste nel giungere ad un prodotto finale partendo dall'implementazione dei requisiti più chiari lavorando a stretto contatto con il committente.
- PROTOTIPAZIONE "THROW-AWAY" (USA E GETTA), che consiste nel giungere ad una migliore comprensione dei requisiti del prodotto software, infatti si parte dall'implementazione dei requisiti meno chiari producendo dei prototipi che non vengono rilasciati agli utenti che vengono poi gettati dopo il loro utilizzo.

Tale modello presenta bassi rischi per le nuove applicazioni, specifica e sviluppo vanno di pari passo e alti rischi per la mancanza di un processo definito e visibile

MODELLO MISTO CASCATA/PROTOTIPI, che consiste di aggiungere alle prime due fasi del modello a cascata l'utilizzo dei prototipi avendo come obiettivo quello di lavorare con i clienti ed evolvere i prototipi verso il sistema finale avendo però un'idea ben precisa e aver compreso bene i requisiti.

MODELLI EVOLUTIVI, utili a soddisfare le esigenze del software che evolve continuamente anche una volta rilasciato al committente, l'evoluzione costante del software comporta aggiunta di funzionalità, adeguamento del software per via di cambiamenti aziendali ecc. In tali modelli vi è la presenza di un'ulteriore fase chiamata "Evolution" che consiste nell'analizzare l'esperienza di utilizzo del software per stabilire nuove esigenze e funzionalità non previste precedentemente dal sistema, dopo tale fase si riprende il ciclo di vita del software dalla fase inziale alla finale per rilasciare un software arricchito di funzionalità. Nell'utilizzo di tale modello vi sono vari problemi quali perdita di visibilità del processo anche da parte del management, prodotto finito scarsamente strutturato, richiesta di specifiche nell'uso di linguaggi di prototipazione rapida e

proprio per via di questi problemi tale modello è applicabile per sistemi interattivi di taglia medio-piccola con ciclo di vita breve.

MODELLO TRASFORMAZIONALE, dove lo sviluppo viene visto come una sequenza di passi graduali che trasformano formalmente una specifica in un'implementazione. I problemi che si riscontrano nell'utilizzo di questo modello sono: competenze e skill specifici per l'applicazione delle tecniche, difficoltà nella specifica formale di parti del sistema, costi di sviluppo in genere più elevati, non comprensione delle specifiche formali da parte del committente. Tale modello trova applicazione nello sviluppo di sistemi critici e presenta alti rischi per le tecnologie coinvolte e le professionalità rischieste.

MODELLI DI SVILUPPO A COMPONENTI, basati sul riuso sistematico dei componenti, i sistemi software sono integrati da componenti esistenti e sono caratterizzati dallo sviluppo rapido di applicazioni. Un modello di riuso completo prevede repository di componenti riusabili a diversi livelli di astrazione prodotti durante le diverse fasi del ciclo di vita e durante lo sviluppo di un nuovo sistema si possono riutilizzare le componenti esistenti e popolare la repository con delle nuove componenti. Tale modello risulta particolarmente adatto per lo sviluppo di software object-oriented.

MODELLO INCREMENTALE, dove le fasi alte del processo sono realizzate copletamente e il sitema cosi progettato viene decomposto in sottositemi che vengono implementati, testati, rilasciati, installati e messi in manutenzione secondo uno schema di priorità in tempi differenti. Tale modelo dunque adotta uno sviluppo incrementale che risolve le difficoltà di produzione di un sitema in una sola volta nel caso di grandi progetti software consegnando il prodotto con più rilasci. I vantaggi dell'adottare uno sviluppo di questo tipo sono: possibilità di anticipare da subito delle funzionalità al committente che sono ovviamente quelle con priorità più alta con un minore rischio di un completo fallimento del progetto e testing più esaustivo cioè i rilasci iniziali, che sono quelli con maggiore priorità che vengono testati maggiormente, agiscono come prototipi e consentono di individuare i requisiti per i successivi incrementi.

MODELLO A SPIRALE, che è un meta-modello cioè si ha la possibilità di utilizzare uno o più modelli e si basa sul riciclo, il processo viene rappresentato come una spirale piuttosto che come sequenza di attività, ogni giro della spirale rappresenta una fase del processo che non è predefinita ma viene scelta in accordo al tipo di prodotto e prevede la scoperta, la valutazione e il trattamento esplicito dei "rischi". La figura che ha il compito di minimizzare i rischi è il manager, il rischio è una misura dell'incertezza del risultato dell'attività ed è presente in ogni attività umana, risulta essere collegato alla quantità e alla qualità delle informazioni disponibili dunque meno informazione si ha più alti sono i rischi. L'articolazione di un progetto che adotta tale modello di sviluppo è guidata dunque non da una rigida sequenza di fasi predefinite ma da una gestione sistematica dei rischi di progetto per arrivare alla loro progressiva diminuzione, all'inizio di un progetto i rischi sono tipicamente molto elevati, ogni iterazione ha lo scopo di ridurre tali rischi inizialmente tramite la costruzione di prototipi di interazione per affrontare i rischi sull'incertezza dei requisiti e architetturali per affrontare i rischi sulla scelta delle tecnologie e della struttura del sistema, successivamente ogni iterazione ha lo scopo di costruire progressivamente nuove porzioni del sistema via via integrate con le precedenti e di verificare con il committente le altre parti interessare. Tale modello come gli altri presenta vantaggi e svantaggi, fra i vantaggi ci sono sicuramente rendere che esplicita la gestione dei rischi, sostiene la determinazione di errori nelle

fasi iniziali, obbliga a considerare gli aspetti di qualità e integra sviluppo e manutenzione, fra gli svantaggi invece abbiamo che la pianificazione risulta essere più complessa ed è richiesto un controllo sistematico degli avanzamenti, richiede un'elevata collaborazione fra committenti e gruppi di progetto, richiede persone capaci di rilevare i rischi e per essere usato deve essere adottato alla realtà aziendale.

LE ITERAZIONI DEL PROCESSO possono essere applicate a qualsiasi modello di sviluppo del software, i requisiti sono sempre soggetti a modifiche nel corso dello sviluppo. Ciò comporta iterazioni di "rework" soprattutto nelle fasi iniziali. Vi sono due approcci correlati: sviluppo incrementale e sviluppo a spirale.

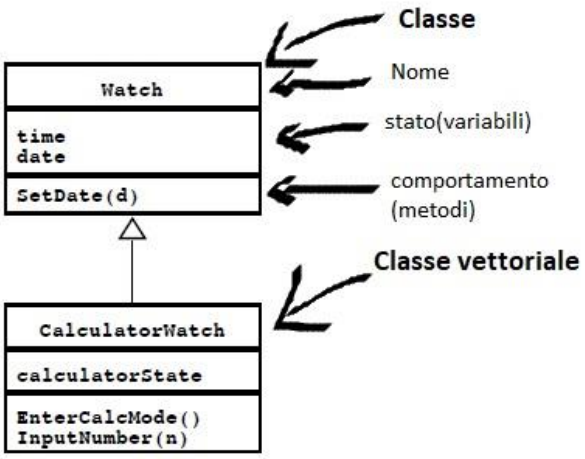
L'EXTREME PROGRAMMING, è un approccio recente allo sviluppo del software basato su iterazioni veloci che rilasciano piccoli incrementi delle funzionalità, si ha una partecipazione più attiva del committente al team di sviluppo e un miglioramento costante e continuo del codice. Per attuare tale approccio al meglio vi sono dodici regole, quali:

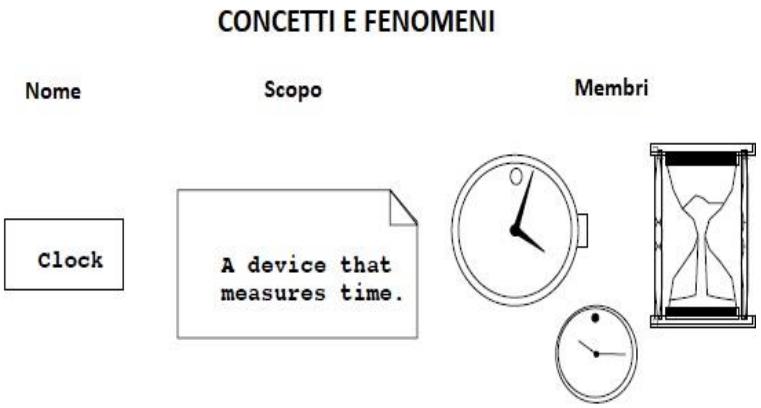
1. Progettare con il cliente;
2. Test funzionali e unitari;
3. Refactoring (riscrivere codice senza alterarne le funzionalità esterne);
4. Progettare al minimo;
5. Descrivere il sistema con una metafora anche per la descrizione formale;
6. Proprietà del codice collettiva;
7. Scegliere ed utilizzare un preciso standard di scrittura del codice;
8. Integrare continuamente i cambiamenti al codice;
9. Il cliente deve essere presente e disponibile a verificare;
10. Open workspace;
11. 40 ore di lavoro settimanali;
12. Pair programming (due programmatori lavorano insieme su un solo computer).

1.3 MODELLAZIONE E UML

La modellazione consiste nel costruire un'astrazione della realtà, le astrazioni sono semplificazioni perché ignorano dettagli irrilevanti rappresentando solo quelli rilevanti, ciò che è rilevante o irrilevante dipende poi ovviamente dallo scopo del modello. La modellazione è dunque un mezzo per gestire la complessità, essa si concentra sulla creazione di un modello abbastanza semplice da consentire a una persona di comprenderne a pieno il significato, ciò risulta essere molto utile per i prodotti software questo perché con il passare degli anni i software diventano sempre più complessi. Un sistema è un insieme organizzato di parti comunicanti tra loro che possono essere considerate come sottosistemi più semplici. Alcune keyword per la modellazione sono **modello** che è un'astrazione che descrive un sottoinsieme di un sistema, **vista** che descrive gli aspetti selezionati di un modello, **notazione** che è un insieme di regole grafiche o testuali per la rappresentazione di viste (Le viste e i modelli di un singolo sistema possono sovrapporsi l'un l'altro), **fenomeno** che è il modo di percepire un oggetto del dominio e **concetto** che descrive le proprietà dei fenomeni comuni, risulta essere costituito da **nome**, utile a distinguere un concetto da un altro, **scopo**, che determina se un fenomeno è membro di un concetto, e **members** che è l'insieme dei fenomeni che fanno parte del concetto. Alcuni concetti importanti dal punto di vista software sono **tipo** che è un'astrazione nel contesto dei linguaggi di programmazione composto da **nome** ad esempio int, **scopo** numeri interi e **membri** che sono i veri e propri valori che quel tipo

può assumere, **istanza** che è un membro di un tipo specifico, cioè il tipo di una variabile rappresenta tutte le possibili istanze che la variabile può assumere, **tipi di dati astratti**, che è un tipo speciale la cui implementazione è nascosta dal resto del sistema, **classe** che è un'astrazione tipica nel contesto dei linguaggi orientati agli oggetti che incapsula sia lo stato (variabili) che il comportamento (metodi) e **classe vettoriale**, che sono classi definite in termini di altre classi utilizzando l'ereditarietà





Molto importanti sono **l'Application Domain** che è l'ambiente in cui il sistema è in funzione e il **Domain Solution** che sono le tecnologie per costruire il sistema.

L'UML è uno standard per la modellazione del software orientato agli oggetti derivato dalla convergenza delle notazioni di tre metodi principali orientati agli oggetti: OMT, OOSE e Booch.

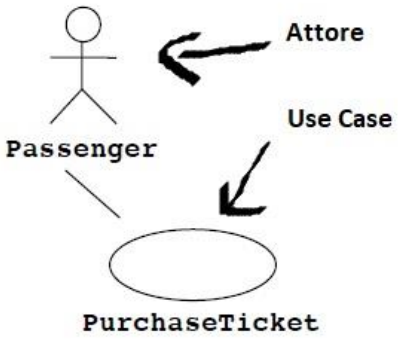
Le convenzioni di base dell'UML prevedono che ogni diagramma rappresenti un grafico di nodi e collegamenti, ogni nodo rappresenta un'entità e sono rappresentati tramite figure rettangolari o ovali, i rettangoli indicano classi i cui nomi non vengono sottolineati o istanze di classi i cui nomi vengono sottolineati mentre gli ovali denotano funzioni, un collegamento tra due nodi denota una relazione tra le entità collegate.

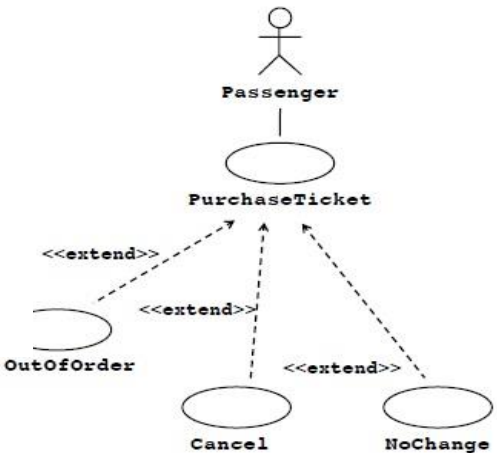
Si dovrebbe prima modellare o codificare o viceversa? Tutto dipende dalla tipologia in cui rientra il progetto software:

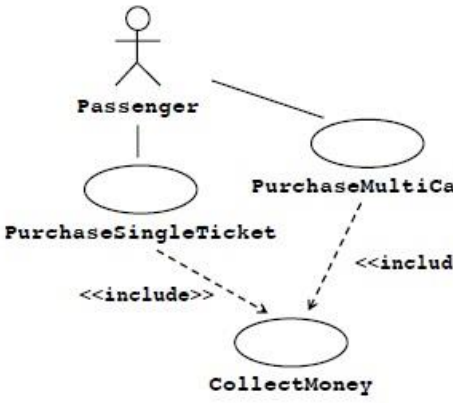
- **FORWARD ENGINEERING**, che parte dalla modellazione per poter creare codice da un modello, approccio adottato da progetti Greenfield;
- **REVERSE ENGINEERING,** che prevedere la creazione di un modello dal codice esistente, approccio adatto a progetti d'interfaccia o di riprogettazione;
- **ROUNDTRIP ENGINEERING,** che prevede la reingegnerizzazione di progetti quando requisiti, tecnologie e programma cambiano frequentemente, approccio che prevede lo sposarsi costantemente dalla modellazione alla codifica.

L'UML viene utilizzato per:

- **USE CASE DIAGRAMS**, usati durante le fasi di requirements elicitation ed analysis per descrivere il comportamento funzionale del sistema dal punto di vista dell'utente, gli elementi raffigurati in questi diagrammi sono **attore**, che è un modello per un'entità esterna che interagisce con il sistema avviando un caso d'uso e può essere un utente, un sistema esterno o un ambiente fisico, esso ha un nome univoco e una descrizione opzionale , **use case**, che rappresenta una classe di funzionalità fornita dal sistema e viene descritto come un flusso di eventi tra attore e sistema che produce un risultato visibile per gli attori, la descrizione si compone di un nome univoco, attori partecipanti, condizione d'ingresso condizione d'uscita, flusso di eventi e requisiti speciali, uno use case può essere correlato con **extend relationship,** che rappresentano use case invocati raramente o funzionalità eccezionali, i flussi di eventi eccezionali sono presi in considerazione dal flusso principale degli eventi per maggiore chiarezza e gli use case che rappresentano tali flussi eccezionali possono estendere più di un solo use case dal punto di vista grafico tale tipologia di relazione viene rappresentata da una freccia tratteggiata con la dicitura <<extend>> con una direzione che va dallo use case esteso a quello che lo estende, e **include relationship,** che rappresentano funzionalità comuni a più use case, il flusso di eventi incluso viene considerato al di fuori di quello dello use case che lo include e la relazione è rappresentata da una freccia tratteggiata con la dicitura <<include>> con una direzione che va dallo use case che include a quello incluso originale uno **use case model** è invece l'insieme di tutti i casi d'uso che descrivono completamente la funzionalità del sistema;

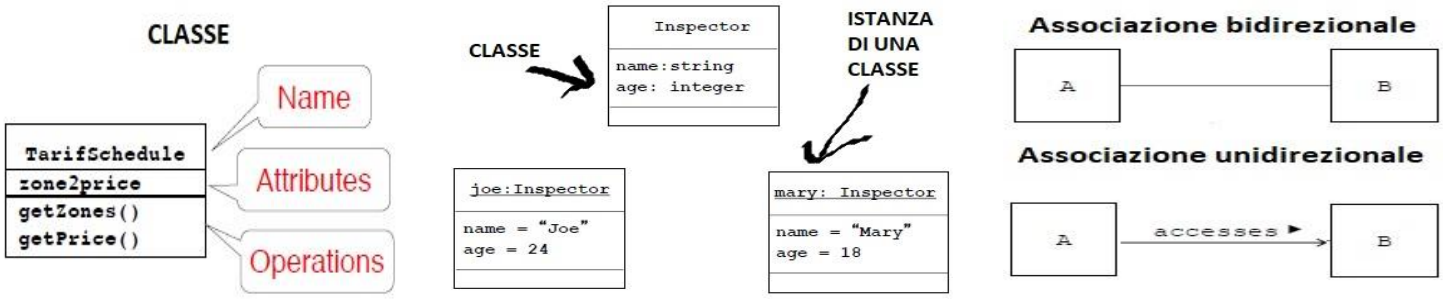


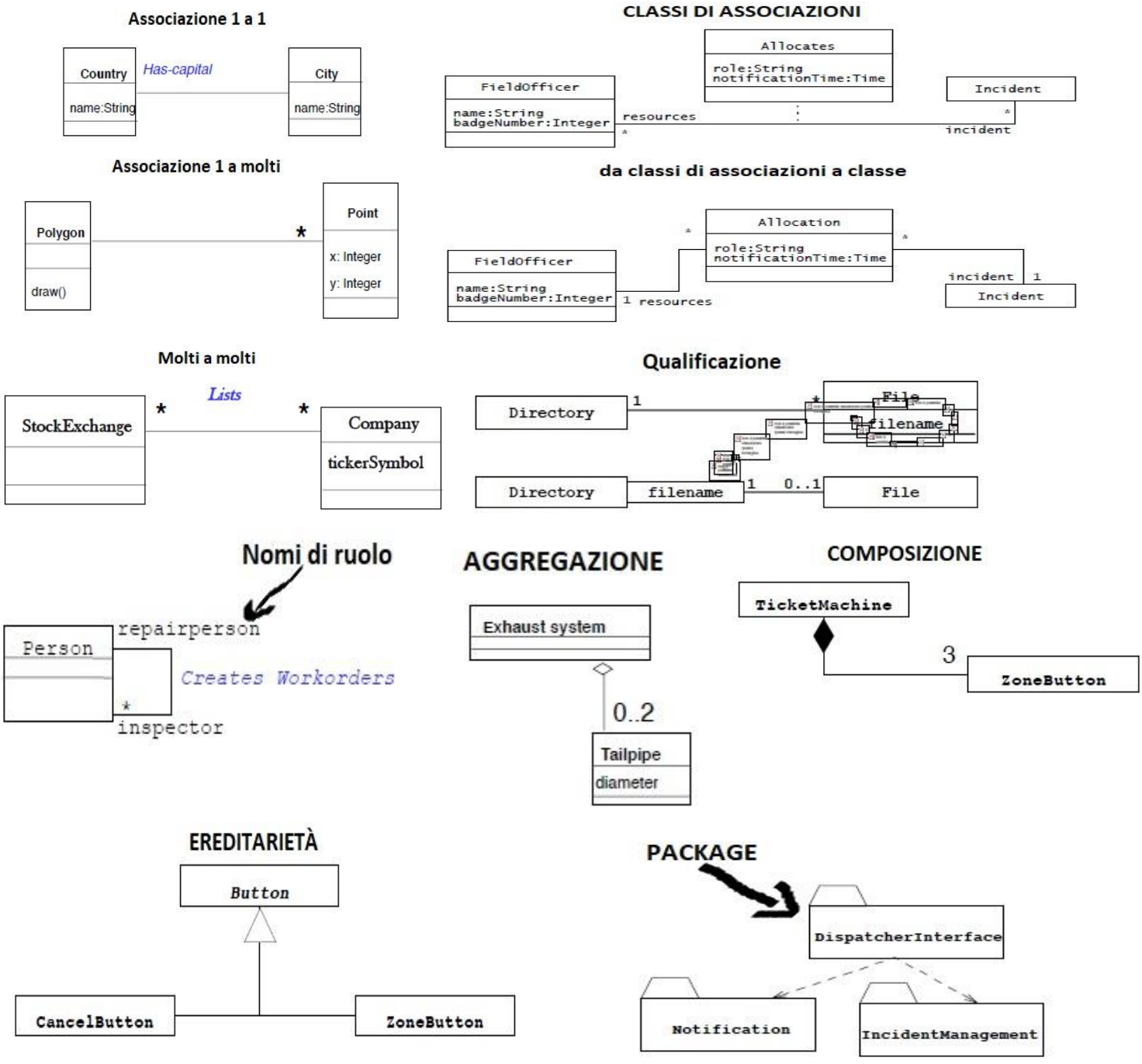




- **CLASS DIAGRAMS**, utilizzati per descrivere la struttura statica del sistema durante le fasi di analisi dei requisiti per modellare concetti del dominio problematico, progettazione del sistema per modellare sottosistemi e interfacce e progettazione di oggetti per modellare classi. Una **classe** rappresenta un concetto e incapsula **stato** (attributi) e un **comportamento** (operazioni), ogni attributo ha un tipo e ogni operazione ha una firma queste informazioni sono tutte opzionali ad eccezione del nome della classe che è obbligatorio. **Un'istanza** rappresenta un fenomeno, essa è caratterizzata dal nome sottolineato e dal fatto che gli attributi vengono rappresentati con i loro valori. La differenza sostanziale tra attore, classe e istanza è che un attore è un'entità esterna al

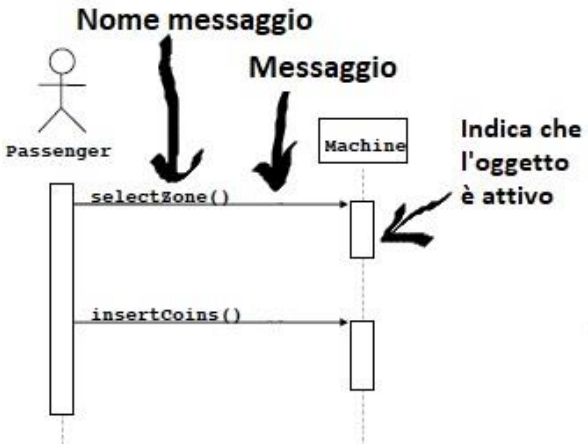
sistema da modellare che interagisce con il sistema, la classe invece è un'astrazione che modella un'entità all'interno del sistema nel dominio del problema mentre un'istanza rappresenta un oggetto di una classe avente il proprio stato. Tra le varie classi di un class diagram vi possono essere delle **associazioni** che denotano le relazioni fra le classi, alle estremità dell'associazione viene indicata la **molteplicità** che indica a quanti oggetti può fare riferimento l'oggetto di origine, le associazioni possono essere **1 a 1, 1 a molti e molti a molti**, molte volte per ridurre la molteplicità da molti a molti ad una molteplicità uno a molti si utilizza un **qualificatore**. Un'associazione tra due classi è di default una mappatura **bidirezionale**, cioè ambedue le classi possono accedere l'una all'altra svolgendo il ruolo di agente, se si vuole realizzare una relazione **unidirezionale** bisogna inserire una freccia la cui direzione della punta indica quale classe accede all'altra. Un **nome di ruolo** è il nome che identifica univocamente un'estremità di un'associazione, viene scritto accanto alla linea di associazione vicino alla classe che interpreta il ruolo, risultano essere necessari per associazioni tra due oggetti della stessa classe e sono utili per distinguere tra due associazioni tra le stesse classi. Quando esiste una sola associazione tra una coppia di classi distinte i nomi stessi delle classi fungono da nome di ruolo. Un particolare tipo di associazioni sono le **classi di associazioni** che sono associazioni alle quali possono essere associati attributi e operazioni proprie e possono essere trasformate in una semplice associazione. Un altro particolare tipo di associazioni sono le aggregazioni che denotano una gerarchia fra classi del tipo "costituita da", cioè l'aggregato è la classe genitore costituita da una classe figlia che ne rappresenta le componenti, **l'aggregazione** viene rappresentata graficamente da una linea con diamante "vuoto" posto all'estremità collegata alla classe genitore, quando il diamante diventa "pieno" si indica una **composizione** cioè una forma più forte dell'aggregazione dove i componenti non possono esistere senza l'aggregato. **L'ereditarietà** è un'altra forma di associazione rappresentata da una freccia "vuota" collegata alla classe padre e all'altra estremità vi sono collegate le classi figlie che ereditano attributi e operazioni dalla classe padre, questa associazione è molto utile per eliminare la ridondanza. Un meccanismo molto utile e spesso utilizzato per scomporre un sistema complesso in sottosistemi sono i **package**, che consentono di organizzare gli elementi in gruppi per aumentarne le leggibilità.

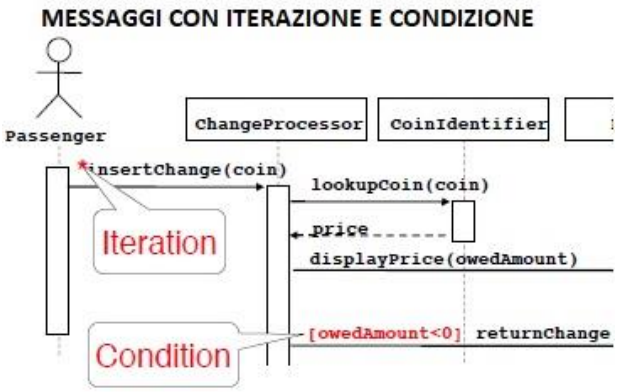


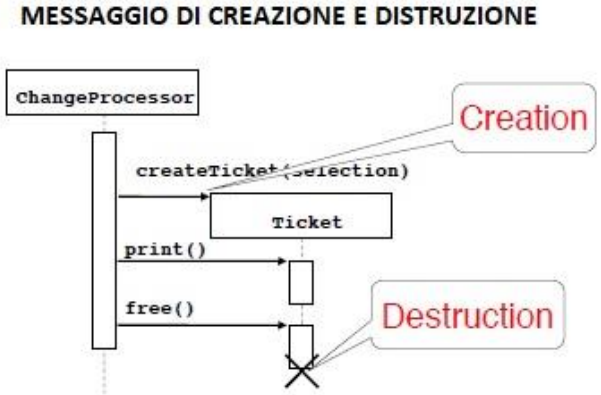


- **SEQUENCE DIAGRAMS**, utilizzato durante l'analisi dei requisiti per perfezionare le descrizioni degli use case e per trovare oggetti partecipanti aggiuntivi e nella fase di progettazione del sistema per perfezionare le interfacce del sottosistema, le **classi** sono rappresentate da colonne i **messaggi** da frecce e le attivazioni sono rappresentate da rettangoli stretti e le **life-line** sono rappresentate da linee verticali tratteggiate. I messaggi partono da un oggetto mittente e arrivano ad un altro oggetto destinatario **frecce tratteggiate** indicano il transito di dati, prima del nome del messaggio possono essere specificate: **l'iterazione** denotata con "*" che indica l'iterazione di quel messaggio e una **condizione** denotata da un'espressione booleana racchiusa tra parentesi quadre [] che indica che il messaggio si invia solo se si verifica la condizione specificata. I messaggi

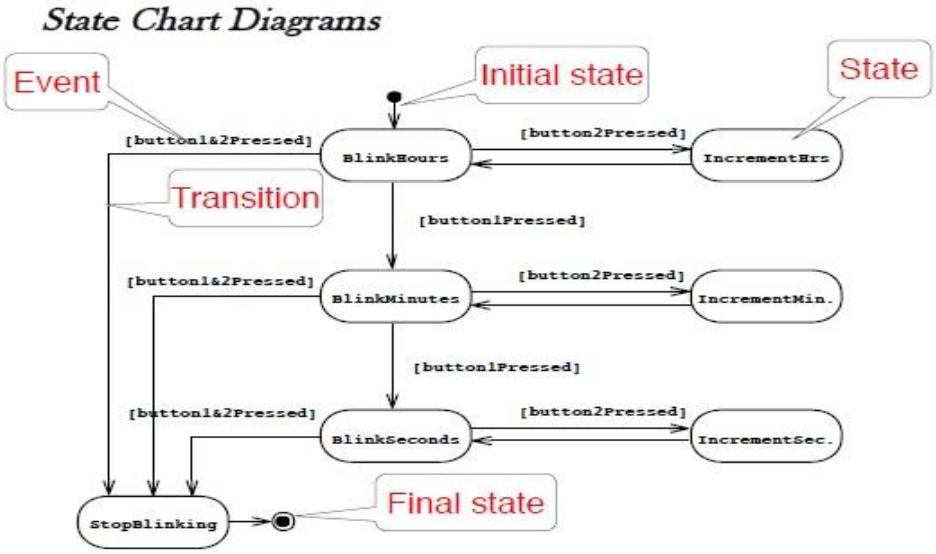
possono anche indicare la **creazione di oggetti** quando puntano all'oggetto creato mentre la **distruzione** di un oggetto è indicata da una X posta sopra la life-line dell'oggetto;





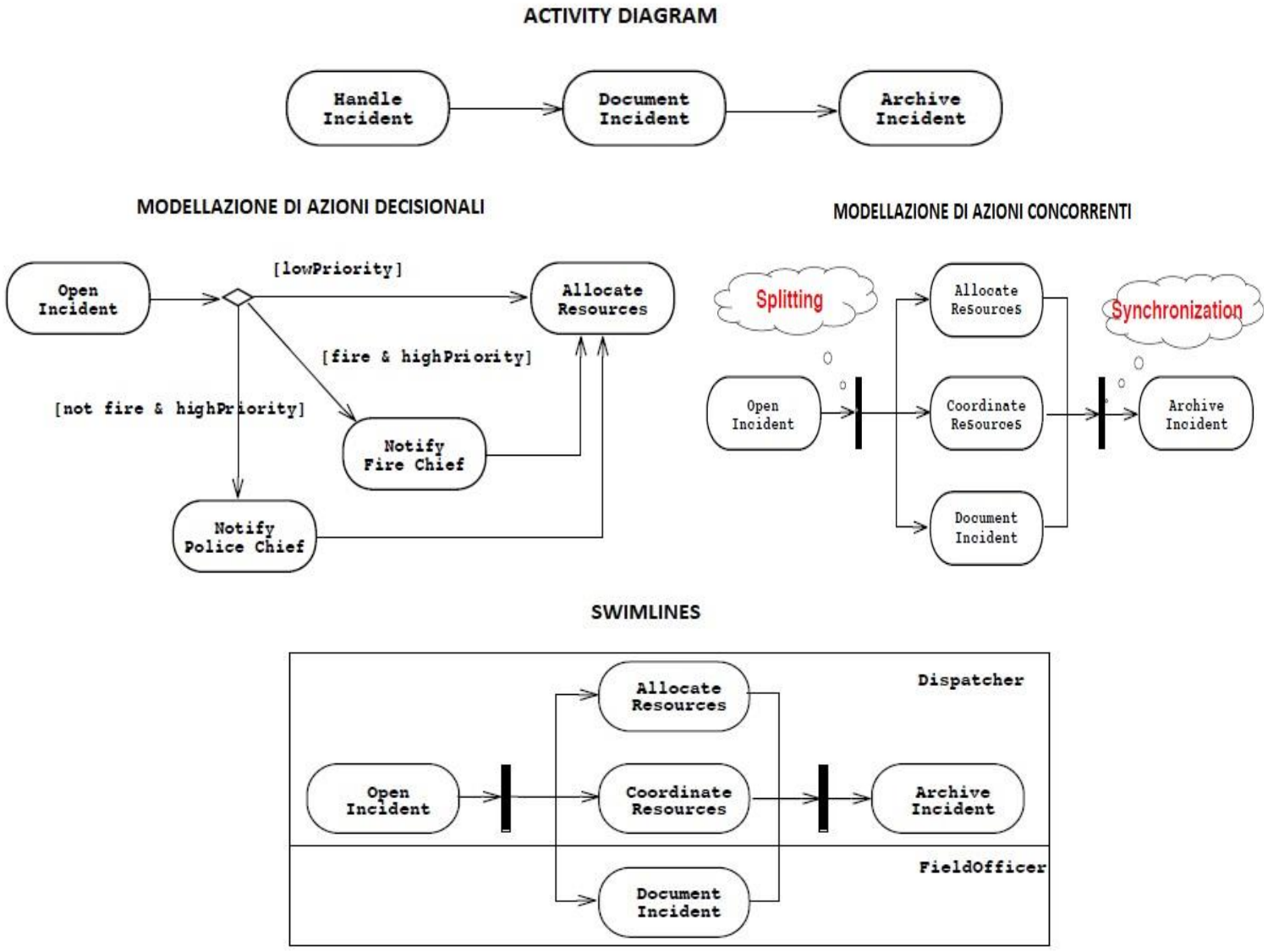


- **STATE DIAGRMAS**, per descrivere il comportamento dinamico di un singolo oggetto;



- **ACTIVITY DIAGRAMS,** per descrivere il comportamento dinamico di un sistema, in particolare il flusso di lavoro. Un activity diagram è un caso speciale di state diagram in cui

gli stati sono attività, gli stati possono essere di due tipi: Stato di azione che non può essere scomposto ulteriormente e Stato dell'attività che può essere ulteriormente scomposto facendo modellare 'attività da un altro activity diagram. In tale diagramma le azioni decisionali vengono rappresentate tramite un rombo da cui partono varie line ognuna delle quali porta ad un'attività se si verifica una condizione espressa con una dicitura accanto alla linea collegata a quell'attività, azioni concorrenti invece vengono rappresentate dapprima splittando un'attività in n attività posizionate in verticale l'una sopra l'altra che poi vengono sincronizzate in un'inca attività, invece per indicare l'oggetto o il sottosistema che implementa alcune attività si usano le swimline.



1.4 SOFTWARE PROJECT MANAGEMENT

Con software project management si indicano le attività necessarie per assicurare lo sviluppo di un prodotto software, ad esempio rispettare le **scadenze** fissate, rispondere a determinati **standard,** ecc., durate queste attività si ha molta interazione fra gli aspetti economici e tecnici del progetto e ad incidere è anche l'esperienza. Un progetto è un insieme ben definito di attività ed ha un inizio,

una fine, è realizzato da un'equipe di persone per perseguire un obiettivo, non è riconducibile a routine ed utilizza un certo insieme di risorse. I problemi che si riscontrano sono relativi al fatto che un prodotto software è intangibile e per valutarne i progressi ci si deve basare sulla documentazione, al fatto che l'ingegneria del software non è ancora riconosciuta come disciplina "solida" al pari delle altre materie ingegneristiche ciò comporta l'assenza di standard per il processo di produzione software. Le figure coinvolte nel processo di sviluppo di un prodotto software sono:

- **Business managers**, che definiscono i termini economici del progetto;
- **Project managers**, che pianificano, motivano, organizzano e controllano lo sviluppo del progetto;
- **Practitioners**, che hanno le competenze tecniche per realizzare il sistema;
- **Costumers**, che specificano i requisiti del software da sviluppare;
- **End users**, che interagiscono con il sistema una volta realizzato.

La maggior parte dei progetti software sono troppo impegnativi per essere realizzati da una sola persona e dunque vi è la necessità di un team il cui numero è ricavabile da una formula:

$$Numero\ persone\ necessarie = \frac{persone}{mese} * Tempo\ allocato$$

Un team deve innanzitutto decidere gli strumenti per la cooperazione e deve condividere i risultati relativi alla pianificazione, a chi fa cosa, le scelte fatte e cosa è stato fatto. Vi sono varie tipologie di team:

- **Democratico Decentralizzato,** dove non è presente un leader permanente ma è il gruppo intero a rilasciare consensi sulle soluzioni e sull'organizzazione del lavoro favorendo la comunicazione orizzontale, i vantaggi sono un'attitudine positiva a ricercare presto gli errori e funziona bene per problemi difficili, mentre gli svantaggi sono che è difficile da imporre e non è scalabile;
- **Controllato Decentralizzato,** dove vi è un leader riconosciuto che coordina il lavoro, la risoluzione dei problemi è sempre di gruppo ma l'implementazione delle soluzioni è assegnata a sottogruppi, scelti dal leader. Tale tipologia presenta una comunicazione orizzontale nei sottogruppi e verticale con il leader, i ruoli sono Project Manager che pianifica coordina e supervisiona le attività del team, Technical staff, che conduce l'analisi e lo sviluppo e varia da 2 a 5 persone, Backup engineer, supporta il project manager ed è responsabile della validazione, e Software librarian che mantiene e controlla la documentazione, i listati del codice, i dati ecc.;
- **Controllato Centralizzato,** dove il team leader decide sulle soluzioni e sull'organizzazione e vi è una comunicazione verticale tra team leader e gli altri membri.

Le attività di un project manager sono: **Stesura della proposta di progetto, stima del costo del progetto, pianificazione e temporizzazione (planning and scheduling), monitoraggio e revisioni del progetto, selezione e valutazione del personale e stesura di rapporti e presentazioni;**

Per l'attività di stima dei costi di un progetto il project manager può utilizzare varie tecniche: dilazionare la stima fino a quando il progetto non è in stato di sviluppo, ma ciò non e praticabile poiché la stima deve essere fatta all'inizio, basare la stima su progetti simili già sviluppati, usare tecniche di decomposizione per generare stime di costo e risorse necessarie e usare uno o più metodi empitici basati su dati storici. Una tipologia di stima molto importante sono le stime quantitative **LOC** che misura la vastità di un progetto dal numero di linee di codice, **KLOC** indica migliaia di linee di codice e le metriche sono € per KLOC, errori o difetti per KLOC, LOC per mese/persona, pagina di documentazione per KLOC, errori/mese-persona, €/pagina di documentazione. Dunque LOC dipende dal linguaggio di programmazione e penalizza programmi scritti in modo chiaro e conciso. Risulta però difficile stimare la dimensione in LOC poiché tale stima non tiene conto della diversa complessità e potenza delle varie istruzioni, risulta anche difficile definire in modo preciso il criterio di conteggio delle linee di codice. Un'altra tipologia di stime sono le stime funzionali **FP (Function Points)** di prima generazione, che misurano le funzionalità offerte dall'applicazione a partire dal dominio informativo e da un giudizio sulla complessità del software, i parametri di un FP sono:

- **Numero di input**, che sono informazioni distinte fornite dall'utente e utilizzate dal programma come dati di ingresso;
- **Numero di output**, che sono informazioni distinte ritornate all'utente come risultato delle proprie elaborazioni;
- **Numero di richieste**, che è il numero di interrogazioni in linea che producono una risposta immediata del sistema;
- **Numero di files**, che è il numero di file creati ed utilizzati internamente dal programma;
- **Numero di interfacce esterne**, che è il numero di files o di altri insiemi di dati scambiati con altri programmi.

Altre stime funzionali sono COSMIC di seconda generazione.

L'attività di **strutturazione del paino di progetto** si compone di vari passi:

1) **INTRODUZIONE**, che si compone di varie sottosezioni:
   i) OVERVIEW DEL PROGETTO, dove si fornisce una descrizione di massima del progetto e del prodotto;
   ii) DELIVERABLES DEL PROGETTO, dove si inseriscono tutti gli items che saranno consegnati, con data e luogo di consegna;
   iii) EVOLUZIONE DEL PROGETTO, dove si inseriscono piani per cambiamenti ipotizzabili e non;
   iv) MATERIALE DI RIFERIMENTO, dove si inserisce una lista dei documenti cui ci si riferisce nel paino di progetto;
   v) DEFINIZIONI E ABBREVIAZIONI.
2) **ORGANIZZAZIONE DEL PROGETTO,** divisa nelle sottosezioni;
   i) MODELLO DEL PROCESSO, contenete le relazioni tra le varie fasi del processo;
   ii) STRUTTURA ORGANIZZATIVA, contenente la gestione interna, chart dell'organizzazione;
   iii) INTERFACCE ORGANIZZATIVE, contenente le relazioni con altre entità;
   iv) RESPONSABILE DI PROGETTO, contenente le principali funzioni e attività la loro natura e il loro responsabile.
3) **DESCRIZIONE DEI PROCESSI GESTIONALI**, divisa nelle sottosezioni;

i) OBIETTIVI E PRIORITÀ;
ii) ASSUNZIONI, DIPENDENZE, VINCOLI, dove si specificano tutti i fattori esterni;
iii) GESTIONE DEI RISCHI, contenente l'identificazione, valutazione e monitoraggio dei rischi. L'identificazione dei rischi consiste nell'identificare i rischi legati alla taglia del prodotto da realizzare o modificare, legati dai vincoli imposti da mercato o dal contratto, legati alle caratteristiche del cliente, legati all'ambiente di sviluppo, alla buona definizione del processo, alla complessità del sistema e alla dimensione ed esperienza del team di sviluppo, dopo di che si passa allo sviluppo di una tabella che indica probabilità e impatto di ogni rischio e infine si individuano le strategie di gestione di tali rischi;
iv) MECCANISMI DI MONITORAGGIO E DI CONTROLLO, contenete i meccanismi di reporting, format, flussi di informazione e revisioni;
v) PIANIFICAZIONE DELLO STAFF, contenente gli skill necessari

4) **DESCRIZIONE DEI PROCESSI TECNICI**, divisa nelle sottosezioni:
   i) METODI, STRUMENTI E TECNICHE, dove sono specificati gli strumenti di calcolo, metodi di sviluppo, struttura del team, standards, linee guida e politiche;
   ii) DOCUMENTAZIONE DEL SOFTWARE, contenente il piano di documentazione che deve includere milestones e revisioni varie;
   iii) FUNZIONALITÀ DI SUPPORTO AL PROGETTO, contenente la pianificazione della qualità e della gestione delle configurazioni.
5) **PIANIFICAZIONE DEL LAVORO, DELLE RISORSE UMANE E DEL BUDGET**, divisa nelle sottosezioni:
   i) WORK PACKAGES, contenente la descrizione di ogni task di cui si compone il progetto;
   ii) DIPENDENZE, contenente le relazioni di precedenza tra funzioni, attività e task;
   iii) RISORSE NECAESSARIE, contenente una stima delle risorse necessarie in termini di personale, di tempo di computazione, di hardware particolare, di supporto software ecc.
   iv) ALLOCAZIONE DEL BUDGET E DELLE RISORSE, contenente le associazioni tra funzioni, attività e task con il loro costo relativo;
   v) PIANIFICAZION, contenente deadlines e milestones.

Le attività sono le principali unità di lavoro con precise date di consegna e sono scomponibili in una serie di task, che sono unità di lavoro atomiche con una durata stimabile che necessitano di alcune risorse e producono risultati tangibili e nel work package sono specificati da nome e descrizione del lavoro da svolgere, precondizioni per poter avviare il loro lavoro, durata, e risorse necessarie, risultato del lavoro e criteri di accettabilità e rischi, e culminano in una milestone, che è il punto finale di ogni attività, in un progetto devono essere organizzate in modo da produrre risultati valutabili dal management che sono consegnati al committente e prendono il nome di deliverables. L'insieme delle attività o le attività che coprono tutta la durata del progetto prendono il nome di funzioni e in tale categoria vi rientrano Training, Quality Control, Documentation, Configuration Management e Project management.

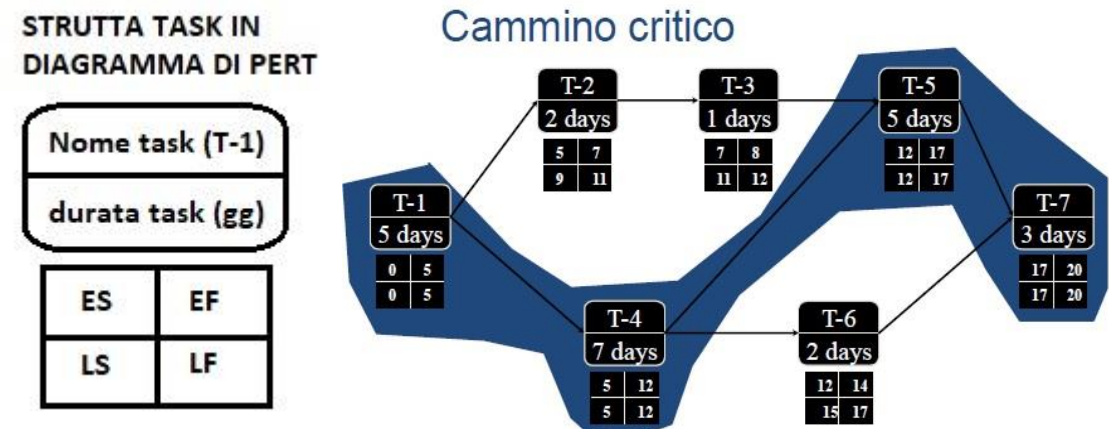
Lo scheduling di progetto dunque consiste nel dividere il progetto in attività e tasks e stimare il tempo e le risorse necessarie per completare ogni singolo tasks, i tasks devono essere organizzati in modo concorrente per ottimizzare la forza lavoro. Tale attività minimizza la dipendenza tra

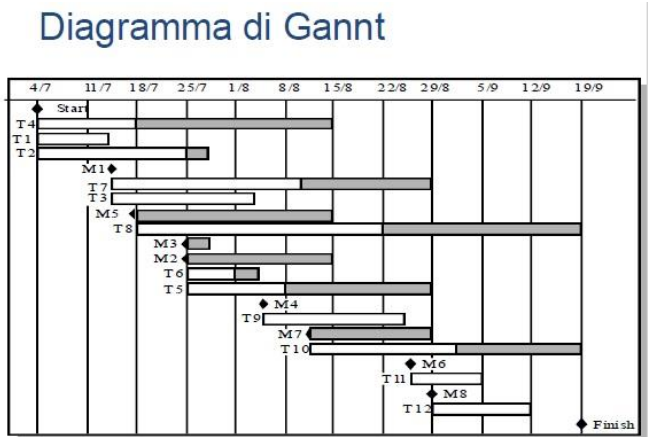
singole mansioni per evitare ritardi dovuti all'attesa del completamento di un'altra mansione e sono necessari intuito ed esperienza per poterla adoperare poiché si riscontrano molti problemi quali: difficoltà a stimare la difficoltà dei problemi ed il costo di sviluppo di una soluzione, la produttività non e proporzionale al numero di persone che lavorano su una singola mansione, aggiungere personale in un progetto in ritardo può aumentare ancora di più il ritardo.

Per tale attività ci sono diversi tipi di rappresentazione grafica, utili a mostrare la suddivisione del lavoro in mansioni che non devono essere troppo piccole, e sono grafico delle attività di **PERT** che evidenzia le dipendenze e il cammino critico, il grafico a barre che mostra lo scheduling come calendario lavori e il diagramma di **GANNT** che esprime la temporizzazione.

Nel diagramma di PERT vengono utilizzati alcuni acronimi, quali:

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





M2 – ANALISI E SPECIFICA DEI REQUISITI

REQUIREMENTS ELICITATION

Per sviluppare un sistema è necessario rispondere a due domande, come identificare lo scopo del sistema e qual è il confine del sistema cioè cosa comprende e cosa non comprende il sistema, queste due domande trovano risposta nella requirements engineering che si compone di due attività principali: Requirements Elicitation (raccolta dei requisiti), che consiste nel definire il sistema in termini comprensibili dal cliente (descrizione del problema) e Requirements Analysis (analisi dei requisiti) che consiste nel fornire le specifiche tecniche del sistema in termini comprensibili allo sviluppatore (specifica del problema). Dunque dal processo dei requisiti vengono prodotti: **problem statement**, che viene sviluppata dal cliente come descrizione del problema che il sistema dovrà risolvere e rappresenta il punto di partenza, una buona descrizione del problema descrive: la situazione attuale, le funzionalità che il nuovo sistema dovrebbe supportare, l'ambiente in cui verrà distribuito il sistema, i derivelables attesi, date di consegna e una serie di criteri di accettazione, **Requirements Specification** e **Analysis Model**, entrambi i modelli si concentrano sui requisiti del sistema dal punto di vista dell'utente ma il primo modello utilizza un linguaggio naturale derivato dal problem statement, mentre il modello di analisi utilizza una notazione formare (UML). Dunque le componenti del problem statement sono: situazione attuale cioè il problema da risolvere, descrizione di uno o più scenari, requisiti, requisiti funzionali e non funzionali, vincoli ("pseudo requisiti"), programma del progetto, principale milestones che implicano l'interazione con il cliente inclusa la scadenza per la consegna del sistema , ambiente di destinazione, l'ambiente in cui il sistema consegnato deve eseguire una serie di test specifici, criteri di accettazione del cliente e criteri per i test di sistema.

I requisiti possono essere di vario tipo:

- **REQUISITI FUNZIONALI,** che descrivono le interazioni tra il sistema e il suo ambiente indipendentemente dall'implementazione;
- **REQUISITI NON FUNZIONALI,** che descrivono gli aspetti visibili all'utente che non sono direttamente correlati ad un suo comportamento funzionale;
- **VINCOLI (PSEUDO REQUISITI),** che sono vincoli imposti dal cliente o dall'ambiente in cui il sistema opera.

Solitamente a non rientrare nei requisiti sono: la struttura del sistema, la tecnologia di implementazione, metodologia di sviluppo, linguaggio d'implementazione, riusabilità ecc. cioè tutti elementi che è auspicabile che non siano vincolati dal cliente. Solitamente dopo le fasi requirements engineering o requirements analysis si passa alla fase di **Requirements validation (convalida dei requisiti)** che risulta essere un passaggio fondamentale nel processo di sviluppo e prende in considerazione alcuni criteri di valutazione quali:

- **CORRECTNESS (CORRETTEZZA),** cioè i requisiti devono rappresentano i bisogni reali del cliente;
- **COMPLETENESS (COMPLETEZZA),** cioè vengono descritti tutti gli scenari possibili in cui il sistema può essere utilizzato incluso un comportamento eccezionale da parte dell'utente o del sistema

- **CONSISTENCY (CONSISTENZA),** non devono esistere requisiti funzionali o non funzionali che si contraddicono a vicenda;
- **REALISM (REALISMO),** i requisiti devono poter realmente essere implementati e consegnati;
- **TRACCIABILITÀ (TRACEABILITY),** ogni funzione del sistema deve poter essere ricondotta a un corrispondente insieme di requisiti funzionali.

Durante questa fase ci si può scontrare con un problema molto frequente che è la **Requirements Evolution (evoluzione dei requisiti)** infatti i requisiti durante la fase di Requirements Elicitation tendono a cambiare molto frequentemente, e dunque e buona prassi considerare la memorizzazione dei requisiti in un repository condiviso, forinire l'accesso multiutente, creare automaticamente un documento delle specifiche di sistema dal repository, consentire la gestione delle modifiche e fornire la tracciabilità lungo tutto il ciclo di vita del progetto.

L'attività di Requirements Elicitation varia a seconda della categoria in cui rientra il progetto software che si vuole sviluppare:

- **GREENFIELD ENGINEERING,** dove lo sviluppo del progetto software viene attivato dalle esigenze dell'utente e inizia da zero senza l'esistenza di un sistema precedente e ciò implica che i requisiti vengono estrapolati dagli utenti finali e dai clienti;
- **RE-ENGINEERING,** che consiste nel re-design o re-implementation di un sistema esistente usando una nuova tecnologia che attiva essa stessa l'intero processo;
- **INTERFACE ENGINEERING,** che consiste nel fornire i servizi di un sistema esistente in un nuovo ambiente e viene innescato dall'abilitazione della tecnologia o dalle nuove esigenze di mercato.

Dunque tale attività risulta essere molto impegnativa poiché richiede la collaborazione di persone con differenti background, utenti con una piena conoscenza del dominio dell'applicazione e sviluppatori, con una piena conoscenza del dominio delle soluzioni, per colmare il divario tra questi ultimi vengono utilizzati: **SCENARI,** che sono esempi di utilizzo del sistema in termini di una serie di interazioni tra l'utente e il sistema, e **USE CASES**, che sono un'astrazione che descrive una classe di scenari, questo perché risultano facilmente comprensibili dall'utente. Gli use cases modellano un sistema dal punto di vista degli utenti, definiscono ogni possibile flusso di eventi attraverso il sistema e descrivono l'interazione tra gli oggetti, essi possono costituire la base per l'intero processo di sviluppo incrementale e iterativo. Gli scenari sono una descrizione narrativa di ciò che le persone fanno e sperimentano mentre cercano di utilizzare il sistema, rappresentano una descrizione concreta, mirata ed informale di una singola funzionalità del sistema utilizzato e possono avere diversi utilizzi durante il ciclo di vita del software, esistono varie tipologie di scenari:

- **AS-IS SCENARIO (COSÍ COM'È),** utilizzato solitamente nei progetti di reingegnerizzazione dove l'utente descrive il sistema e servono appunto a descrivere un sistema esistente così com'è;
- **VISIONARY SCENARIO (VISIONARIO),** utilizzato solitamente in progetti di greenfield engineering ma anche in progetti di reingegnerizzazione per descrivere un sistema futuro;
- **EVALUATION SCENARIO (VALUTATIVO),** utilizzato per descrivere attività degli utenti in un sistema che poi dovrà essere valutato;

- **TRAINING SCENARIO (D'ALLENAMENTO),** utilizzato per introdurre nuovi utenti al sistema fornendo istruzioni passo per passo.

Per trovare gli scenari non bisogna aspettarsi che sia il cliente a darci informazioni dettagliate di sua iniziativa ma bisogna impegnarsi a creare e a mantenere un approccio verbale evolutivo e incrementale in modo tale da aiutare il cliente a formulare i requisiti e farsi aiutare dai clienti nella comprensione dei requisiti che evolveranno durante lo sviluppo degli scenari, molto utile e di supporto a questa fase è seguire delle euristiche che prevedono di chiedersi o di chiedere al cliente: quali sono le attività principali che il sistema deve eseguire?, quali dati creerà, memorizzerà, modificherà, rimuoverà o aggiungerà l'attore nel sistema?, di quali cambiamenti esterni dovrà essere informato il sistema?, di quali cambiamenti del sistema dovranno essere informati gli attori?. Tuttavia non bisogna fare affidamento soltanto sui questionari ma bisogna insistere sull'osservazione delle attività, se il sistema già esiste, e chiedere di parlare con l'utente finale.

Dopo aver formulato gli scenari il prossimo obiettivo consiste nel trovare uno use case per ogni scenario che specifichi tutte le possibili istanze su come svolgere l'attività prevista, bisogna dunque descrivere in maggior dettaglio questo caso d'uso descrivendo le condizioni d'ingresso, il flusso degli eventi, la condizione di uscita, le eccezioni e i requisiti speciali (vincoli e requisiti non funzionali). Uno use case dunque è un flusso di eventi tra sistema e attore iniziato sempre dall'attore, ha sempre un nome che richiama brevemente la funzionalità che specifica e una condizione di terminazione, viene rappresentato graficamente con un ovale che ne racchiude il nome. Anche per trovare i casi d'uso risulta utile seguire un'euristica che consiste nell'effettuare dapprima una selezione verticale cioè selezionare uno scenario del sistema, dialogare dettagliatamente con l'utente per capire lo stile d'interazione che preferisce, e successivamente effettuare una selezione orizzontale, cioè molti scenari, per definire l'ambito del sistema. Dunque per specificare uno use case bisogna specificare un nome, gli attori coinvolti, condizione d'ingresso con una frase sintattica del tipo "questo use cae inizia quando…", flusso di eventi in forma libera attraverso un linguaggio naturale, condizione d'uscita che inizia con "questo use case termina quando…", eccezioni descrivendo cosa succede se le cose vanno male, requisiti speciali elencando requisiti non funzionali e vincoli.

Uno use case model è l'insieme di tutti gli use case che specificano la completa funzionalità del sistema, è dunque composto da use case e **relazioni** fra use case. Le relazioni si dividono in due grandi categorie:

- **DEPENDECIES (DIPENDENZE),** che di divide in due tipologie:
  a) **<<INCLUDE>>,** quando uno use case utilizza un altro use case ottenendo una decomposizione funzionale, essa va utilizzata quando una funzione nella dichiarazione del problema originale (**SUPPLIER USE CASE**) risulta essere troppo complessa per poter essere risolta immediatamente e quindi si procede a descriverla attraverso l'aggregazione di funzioni più semplici (**BASE USE CASE**) (use case principale scomposto in use case più semplici), e anche quando ci sono già funzioni esistenti e si ha il bisogno di riutilizzarle. Tale relazione <<include>> da uno use case A a uno use case B indica che un'istanza di use case A esegue tutto il flusso di eventi dello use case B (A delega a B), una cosa molto importante da ricordare è che il base use case non può esistere da solo ma viene sempre richiamato dal supplier use case;

b) **<<EXTENDS>>,** quando uno use case estende un altro use case, essa va utilizzata quando la funzionalità nella dichiarazione del problema originale deve essere estesa, una relazione <<extend>> da uno use case A ad uno use case B indica che lo use case A è un'estensione dello use case B, molto importante da ricordare è che in una relazione <<extend>> il base use case può essere eseguito senza lo use case che lo estende;

- **GENERALIZATION (GENERALIZZAZIONE),** quando uno use case astratto ha diverse specializzazioni, essa viene utilizzata quando si vuole tenere presente un comportamento comune tra use case, la relazione di generalizzazione determina dunque un comportamento comune tra use case, gli use case figli ereditano il comportamento e il significato dello use case genitore e sovrascrivono alcuni comportamenti.

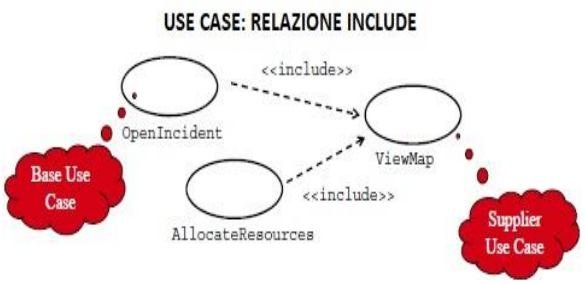
I requisiti non funzionali vengono categorizzati secondo lo standard **FURPS+** che prevede le seguenti categorie:

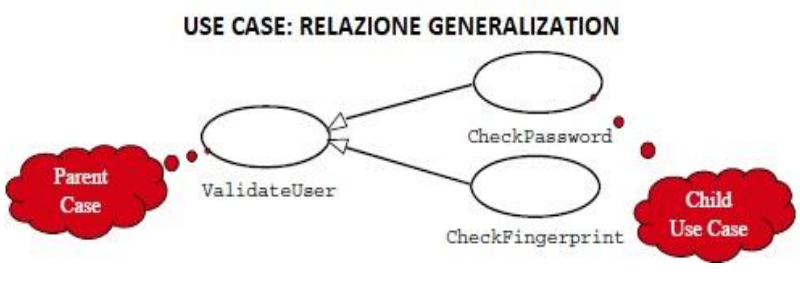
- **USABILITY (USABILITÀ),** cioè la facilità con cui un utente può imparare a operare con il sistema, quali input vanno forniti, come interpretare gli output, ecc., rientrano nelle convenzioni adottate dall'interfaccia utente;
- **RELIABILITY (AFFIDABILITÀ),** cioè la capacità di un sistema di eseguire le funzioni richieste in un periodo di tempo specificato garantendo robustezza e sicurezza;
- **PERFORMANCE,** riguardano gli attributi quantificabili del sistema come tempo di risposta, throughput e disponibilità;
- **SUPPORTABILITY (SUPPORTABILITÀ),** che riguarda la facilità con cui si attuano modifiche al sistema dopo la distribuzione, dunque manutenibilità e portabilità.

Pseudorequisiti (vincoli)

- **IMPLEMENTATION (IMPLEMENTAZIONE),** riguardano vincoli d'implementazione del sistema come linguaggio di programmazione, strumenti specifici, piattaforme hardware ecc.;
- **INTERFACE (INTERFACCIA),** riguardano vincoli imposti da sistemi esterni tra cui formato dei dati ecc.;
- **PACKAGING,** riguardano vincoli sul supporto d'installazione, sulla consegna effettiva del sistema ecc.;
- **LEGAL (LEGALI),** riguardano vincoli sulla concessione di licenze, problemi di certificazione ecc.;

La gestione dell'attività di requirements elicitation pevede che le specifiche vadano sempre negoziate con il cliente, diversi stackholder presentano i loro punti di vista che vanno ascoltati per poter negoziare con i propri trovando una soluzione reciprocamente accettabile e mantenimento della tracciabilità.





REQUIREMENTS SPECIFICATION

La tecnica di specifica di base dei requisiti più utilizzata in pratica è quella fornita dallo standard ISO/IEC/IEEE standard 29148:2011 (E). Si dice che un requisito è **ben formato** quando: può essere verificato, deve essere supportato da un sistema per risolvere e perseguire l'obiettivo di uno stakeholder, è qualificato da condizioni misurabili e limitato da vincoli, definisce le prestazioni del sistema quando utilizzato da uno specifico stakeholder ma non una capacità corrispondente dell'utente, operatore o altro stakeholder. Un requisito in realtà è una dichiarazione che esprime un'esigenza e i suoi vincoli associati e condizioni, viene scritto in linguaggio naturale e comprende un soggetto del requisito e l'azione che deve essere compiuta. La sintassi di un requisito è dunque:

[CONDIZIONE][SOGGETTO][AZIONE][OGGETTO][VINCOLO]

Un esempio: Quando il segnale x viene ricevuto **[CONDIZIONE],** il sistema **[OGGETTO],** imposta **[AZIONE],** il segnale x ricevuto **[OGGETTO],** entro 2 secondi **[VINCOLO]**

I punti chiave per scrivere de buoni requisiti sono:

- Per esprimere disposizioni vincolanti obbligatorie bisogna utilizzare "deve";
- Per esprimere preferenze e obiettivi desiderati non obbligatori o non vincolanti bisogna utilizzare "dovrebbe";
- Per esprimere suggerimenti non obbligatori e non vincolanti bisogna utilizzare "Può";
- Per esprimere testo descrittivo bisogna utilizzare "Sono", "È", "Era";
- Bisogna utilizzare affermazioni positive evitando dichiarazioni negative come "Non deve";
- Bisogna utilizzare la forma "ATTIVA" e non quella passiva, dunque non scrivere "deve essere in grado di rilevare" bensì "rileverà";
- Bisogna definire e applicare chiaramente e in modo corretto a tutti i requisiti di sistema tutti i termini specifici.

I requisiti hanno dunque delle **caratteristiche singole** che sono:

- **NECESSARIO,** cioè il requisito definisce una funzionalità necessaria applicabile ora e non obsoleta che se rimossa crea un deficit al sistema;
- **IMPLEMENTAZIONE GRATUITA,** evitare vincoli non necessari sui requisiti;
- **NON AMBIGUO,** vi è una sola interpretazione di quel requisito e risulta facile da capire;
- **COERENTE,** privo di conflitti con altri requisiti;
- **COMPLETO,** cioè nessuna aggiunta lo descrive in modo migliore ed è misurabile;
- **SINGOLO,** un solo requisito senza nessuna congiuntura;
- **FATTIBILE,** tecnicamente realizzabile;
- **TRACCIABILE,** interiormente ed esteriormente al sistema;

REQUIREMENTS TRACEABILITY

Esistono molte relazioni tra requisiti e tra questi ultimi e la progettazione del sistema, dunque quando vengono proposte modifiche è necessario tracciare l'impatto di queste modifiche su altri requisiti e sulla progettazione del sistema, la tracciabilità è la proprietà di una specifica dei requisiti che riflette la facilità di trovare i requisiti correlati, essa richiede che ogni requisito o prodotto possegga un identificativo univoco. Le **source traceability information** sono utilizzate per collegare i requisiti alle parti interessate che hanno proposto i requisiti, quando viene proposto un cambiamento sono usate per scoprire gli stakeholder in modo da consultarli in merito al cambiamento, le **requirements traceability information** collegano i requisiti dipendenti all'interino del documento dei requisiti, e sono utilizzate per valutare il numero di requisiti che possono essere interessati da una modifica proposta e l'entità di tale modifica, le **design traceability information** collegano i requisiti ai moduli di progettazione in cui vengono implementati, e sono utilizzate per valutare l'impatto delle modifiche ai requisiti proposti sulla progettazione e implementazione del sistema, le **testing traceability information** collegano i requisiti ai casi di test in cui vengono testati. Le informazioni sulla tracciabilità dei requisiti sono spesso rappresentate mediante matrici di tracciabilità che mettono in relazione i requisiti con le parti interessare, tra loro, moduli di progettazione o casi di test, in tale matrice ogni requisito è inserito in una riga e in una colonna e dove esistono dipendenze tra requisiti diversi questi sono registrati nella cella d'intersezione.

REQUIREMENTS ANALYSIS

Gli obiettivi della requirements analysis sono quelli di fornire un modello del sistema che sia corretto, completo, consistente e non ambiguo, gli sviluppatori dunque formalizzano la specifica delle richieste prodotta durante la fase di requirements elicitation e validano, correggono e chiariscono eventuali errori presenti nella specifica delle richieste ed esaminano più in dettaglio le condizioni limite e i casi eccezionali, il cliente e l'utente sono coinvolti se devono essere cambiate delle richieste o se c'è bisogno di ulteriori informazioni. Viene dunque costruito un modello che descrive il **dominio dell'applicazione.**

Le specifiche contengono ambiguità date dall'inesattezza del linguaggio naturale e da assunzioni fatte dagli autori della specifica e non esplicitate, per aiutare ad individuare aree di ambiguità, inconsistenza e omissioni si attua un processo di formalizzazione.

Il modello di analisi prodotto è composto di tre modelli: **Modello funzionale,** rappresentato da use case e scenari, **Modello degli oggetti di analisi**, rappresentato dal diagramma delle classi e dal diagramma degli oggetti, rappresenta il sistema dal punto di vista dell'utente e si focalizza sui concetti che sono manipolati dal sistema, le loro proprietà e relazioni, è un dizionario visuale dei concetti, attributi e operazioni (class diagram), e **Modello dinamico**, rappresentato da statechart e sequence diagram, che si focalizza sul comportamento del sistema, i sequence diagram rappresentano le interazioni tra un insieme di oggetti durante un singolo use case, gli statechart rappresentano il comportamento di un singolo oggetto o di alcuni oggetti strettamente accoppiati, tale modello consente di assegnare le responsabilità alle classi e quindi individuare nuove classi che sono aggiunte al modello degli oggetti dell'analisi.

Il modello degli oggetti di analisi consiste di oggetti:

- **ENTITY,** che rappresentano l'informazione persistente

- **BOUNDARY,** che rappresentano le interazioni tra gli attori e il sistema come oggetti relativi all'interfaccia ecc.;
- **CONTORL,** che si occupano di realizzare gli use case, rappresentano il controllo dei task eseguiti dal sistema contengono la logica e determinano l'ordine dell'interazione degli oggetti.

Avere questi tipi di oggetti porta a modelli che sono più resistenti al cambiamento, aiuta ad identificare gli oggetti e le responsabilità a loro associate. In UML per facilitare la lettura e la comprensione di questi diagrammi è opportuno usare delle convenzioni sui nomi, gli oggetti control possono avere il suffisso Control e gli oggetti boundary dovrebbero avere nomi che ricordano aspetti dell'interfaccia tipo button, form, ecc.

Le attività che consentono di trasformare gli use case e gli scenari della requirements elicitation in un modello di analisi sono: Indentificare gli oggetti Entity, identificare gli oggetti boundary, identificare gli oggetti control, mappare gli use case in oggetti con sequence diagram, identificare le associazioni, identificare gli aggregati, identificare gli attributi, modellare il comportamento dipendente dallo stato degli oggetti individuali, modellare le relazioni di ereditarietà e rivedere il modello di analisi, tutte queste attività sono guidate da euristiche e la qualità dei risultati dipende dall'esperienza degli sviluppatori nell'applicare le euristiche e i metodi.

Identificare gli oggetti Entity, gli oggetti partecipanti formano la base del modello di analisi, per individuare gli oggetti partecipanti si esaminano gli use case e si individuano i candidati, l'euristica di Abbot si basa sull'analisi del linguaggio naturale per identificare oggetti, attributi, associazioni dalla specifica dei requisiti.

Euristica di Abbott

Parti del parlato	Componente del modello	esempio
Nome proprio	Istanza	Alice
Nome comune	Class	Funzionario(FieldOfficer)
Verbo fare/azione	Operazione	Crea, Sottoponi, Seleziona
Verbo essere	gerarchia	È un tipo di, è uno di
Verbo avere	aggregazione	Ha, consiste di, include
aggettivo	attributo	Descrizione dell'incidente

In congiunzione all'euristica di Abbott è possibile usare la seguente euristica: termini che gli sviluppatori e gli utenti hanno bisogno di chiarire per comprendere gli use case, sostantivi ricorrenti negli use case, entità del mondo reale, per ogni oggetto Entity identificato si assegna un nome univoco che deve essere lo stesso utilizzato dagli utenti e dagli specialisti del dominio applicativo e una breve descrizione, si individuano gli attributi e responsabilità.

Identificare gli oggetti Boundary, gli oggetti boundary rappresentano l'interfaccia del sistema con gli attori, in ogni use case un attore interagisce almeno con un oggetto boundary che colleziona informazioni dall'attore e le traduce in una forma che può essere usata sia dal control che dalle Entity, essi modellano l'interfaccia senza descriverne gli aspetti visuali. L'euristica da seguire per individuare gli oggetti boundary è: identificare i controlli della UI di cui l'utente ha bisogno per iniziare lo use case, identificare form di cui l'utente ha bisogno per inserire dati nel sistema, identificare avvisi e messaggi che il sistema usa per rispondere all'utente, non modellare aspetti visuali della UI con gli oggetti boundary, utilizzare sempre i termini dell'utente finale per descrivere l'interfaccia mai termini del dominio di implementazione.

Identificare gli oggetti Control, gli oggetti control sono responsabili del coordinamento degli oggetti boundary ed Entity, si preoccupano di collezionare informazione dagli oggetti boundary e inviarla agli Entity, solitamente hanno una controparte nel mondo reale e spesso esiste una stretta relazione tra Control e use case esso viene creato all'inizio dello use case e cessa di esistere alla fine. L'euristica da adottare per l'identificazione di oggetti control è: identificare un oggetto control per ogni use case, identificare un oggetto control per ogni attore in uno use case, la vita di un control deve corrispondere alla durata di uno use case o di una sessione utente.

Un **Sequence Diagram** mostra come il comportamento di uno use case è distribuito tra i suoi oggetti partecipanti, infatti vengono assegnate responsabilità ad ogni oggetto in termini di un insieme di operazioni, illustra la sequenza i interazioni tra gli oggetti necessaria per realizzare uno use case, non è adatto alla comunicazione con il cliente ma è più preciso ed intuitivo per gli esperti rispetto agli use case e fornisce una prospettiva diversa che consente di individuare oggetti mancanti e aree non chiare nelle specifiche. Le colonne rappresentano gli oggetti che partecipano nello use case, la colonna più a sinistra rappresenta l'attore che inizia lo use case, la seconda oggetto boundary con cui l'attore interagisce per iniziare lo use case e la terza l'oggetto control che gestisce il resto dello use case, gli oggetti control creano altri oggetti boundary e possono interagire con altri 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 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. Il tempo procede verticalmente dal top al bottom al top del diagramma si trovano gli oggetti che esistono prima, oggetti creati durante l'interazione sono illustrati con il messaggio <<create>> oggetti distrutti sono evidenziati con una croce, la linea tratteggiata indica il tempo in cui l'oggetto può ricevere messaggi.

M3 – SYSTEM DESIGN

SYSTEM DESIGN

Il System Design è il processo di trasformazione del modello di analisi nel modello di progettazione del sistema ed ha come scopo quello di:

- Definire gli obiettivi di design del progetto;
- Decomporre il sistema in sottosistemi più piccoli che possono essere realizzati da team individuali;
- Selezionare le strategie per costruire il sistema quali: strategie hardware e software, strategie relative alla gestione dei dati persistenti, il flusso di controllo globale, le politiche di controllo degli accessi e la gestione delle condizioni limite.

Tale processo produce come output un **modello del sistema** che include la decomposizione del sistema in sottosistemi e una chiara **descrizione di ognuna delle strategie**, e risulta essere un'attività molto complessa poiché l'analisi si focalizza sul dominio di applicazione mentre il design si focalizza sul dominio di implementazione, dunque il compito degli sviluppatori è trovare i giusti compromessi fra i vari obiettivi di design che spesso sono in conflitto tra loro non potendo però anticipare tutte le decisioni relative alla progettazione poiché non si ha un'idea chiara del dominio della soluzione.

Il risultato dell'analisi dei requisiti è dunque un modello di analisi che descrive il sistema dal punto di vista degli attori che funge da base fra cliente e sviluppatori, non contiene informazioni sulla struttura interna del sistema, sulla sua configurazione hardware e in generale su come il sistema dovrebbe essere realizzato, tale modello di analisi è descritto da alcuni deliverables, quali; requisiti non funzionali e vincoli, che sono tempo di risposta massimo, minimo throughput, affidabilità, piattaforma per il sistema operativo, etc.., use case model, che descrive le funzionalità del sistema dal punto di vista degli attori, object model, che descrive le entità manipolate dal sistema, e da un sequence diagram per ogni use case, che mostra la sequenza di interazioni fra gli oggetti che partecipano al caso d'uso.

I prodotti del System Design sono:

- **OBIETTIVI DI DESIGN**, che descrivono la qualità del sistema e vengono derivati dai requisiti non funzionali;
- **ARCHITETTURA SOFTWARE**, che descrive:
  - La decomposizione del sistema in termini delle responsabilità dei sottosistemi così che ogni sottosistema possa essere assegnato ad un team e realizzato indipendentemente;
  - Le dipendenze fra sottosistemi;
  - L'hardware associato ai vari sottosistemi e le decisioni (politiche) relative a control flow, controllo degli accessi e memorizzazione dei dati;
- **BOUNDARY USE CASE**, che descrivono la configurazione del sistema, che comprendono le scelte relative allo startup, allo shutdown ed alla gestione delle eccezioni.

I risultati della Requirements Analysis vengono utilizzati per il System Design nel seguente modo:

- **REQUISITI NON FUNZIONALI**, vengono utilizzati per definire i Design Goals;

- **MODELLO FUNZIONALE,** viene utilizzato come base di partenza per attuare la suddivisione in sottosistemi;
- **MODELLO AD OGGETTI,** viene utilizzato per il mapping l'hardware/software e per la gestione dei dati persistenti;
- **MODELLO DINAMICO,** viene utilizzato per la concorrenza, per la gestione globale delle risorse e per il controllo del software;
- **SCOMPOSIZIONE IN SOTTOSISTEMI,** viene utilizzato per le boundary conditions.

IDENTIFICARE GLI OBIETTIVI DI DESIGN

L'identificazione degli obiettivi di design è il primo passo del System Design e consiste nell'identificare le qualità su cui il sistema dovrà focalizzarsi.
Molti design goal possono essere ricavati dai requisiti non funzionali o dal dominio dell'applicazione, altri invece vengono forniti dal cliente o vengono ricavate dalle attività di management.
Formalizzare esplicitamente i design goal è un procedimento di notevole importanza poiché ogni importante decisione di design deve essere fatta seguendo lo stesso insieme di criteri.
I criteri di design sono organizzati in cinque gruppi:

- **PERFORMANCE,** che include tutti i requisiti imposti sul sistema in termini di spazio e velocità, e sono:
  - **TEMPO DI RISPOSTA,** che indica con quali tempi una richiesta di un utente deve essere soddisfatta dopo che è stata emessa;
  - **TROUGHPUT,** che indica quanti task il sistema deve portare a termine in un lasso di tempo prefissato;
  - **MEMORIA,** che indica quanto spazio è richiesto al sistema per funzionare.
- **DEPENDABILITY,** che include tutti quei requisiti legati agli sforzi che bisogna fare per evitare i crash del sistema e le loro conseguenze, e sono:
  - **ROBUSTNESS (ROBUSTEZZA),** che indica la capacità di sopravvivere ad input errati immessi dagli utenti;
  - **RELIABILITY (AFFIDABILITÀ),** che indica la differenza fra comportamento specificato e osservato;
  - **AVAILABILITY (DISPONIBILITÀ),** che indica la percentuale di tempo in cui il sistema può essere utilizzato per compiere normali attività;
  - **FAULT TOLLERANCE,** che indica la capacità di operare sotto condizioni di errore;
  - **SECURITY,** che indica la capacità di resistere ad attacchi di malintenzionati;
  - **SAFETY,** che indica la capacità di evitare di danneggiare vite umane, anche in presenza di errori o fallimenti.
- **COST,** che include tutti i costi per sviluppare il sistema, per metterlo in funzione e per amministrarlo, nel caso in cui il sistema che si sta progettando debba andare a sostituire un sistema preesistente è necessario considerare il costo per assicurare compatibilità con il vecchio sistema o per transitare al nuovo sistema, e sono:
  - **DEVELOPMENT COST,** che indica il costo di sviluppo del sistema iniziale;
  - **DEPLOYMENT COST,** che indica il costo relativo all'installazione del sistema e training degli utenti;
  - **UPGRADE COST,** che indica il costo di conversione dei dati del sistema precedente.

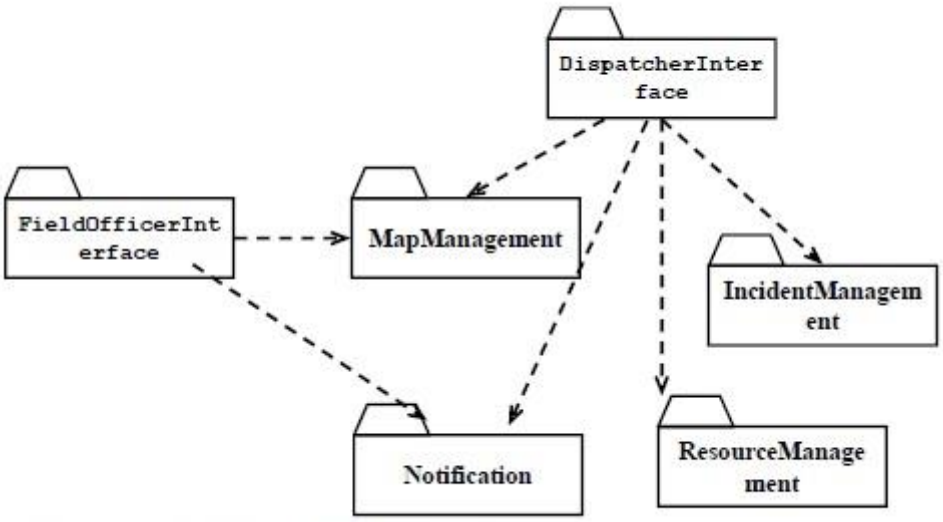
Tale costo viene applicato quando nei requisiti è richiesta la compatibilità con il sistema precedente (backward compatibility);
  - **MAINTENANCE COST,** che indica il costo richiesto per correggere errori sw o hw;
  - **ADMINISTRATOR COST,** che indica il costo richiesto per amministrare il sistema.
- **MAINTENANCE,** che include i requisiti che determinano quanto deve essere difficile modificare il sistema dopo il suo rilascio, e sono:
  - **ESTENSIBILITÀ,** che indica quanto è facile aggiungere funzionalità o nuove classi al sistema;
  - **MODIFICABILITÀ,** che indica quanto facilmente possono essere cambiate le funzionalità del sistema;
  - **ADATTABILITÀ,** che indica quanto facilmente può essere portato il sistema su differenti domini di applicazione;
  - **PORTABILITÀ,** che indica quanto è facile portare il sistema su differenti piattaforme;
  - **LEGGIBILITÀ,** che indica quanto è facile comprendere il sistema dalla lettura del codice;
  - **TRACCIABILITÀ DEI REQUISITI,** che indica quanto è facile mappare il codice nei requisiti specifici.
- **END USER CRITERIA,** costituito dai requisiti che includono qualità che sono desiderabili dal punto di vista dell'utente, ma che sono state coperte da criteri di performance e dependability, e sono:
  - **UTILITÀ,** che indica quanto bene il sistema dovrà supportare il lavoro dell'utente;
  - **USABILITÀ,** che indica quanto dovrà essere facile per l'utente utilizzare il sistema.

Quando definiamo gli obiettivi di design spesso solo un piccolo sottoinsieme di questi criteri può essere tenuto in considerazione, lo scopo degli sviluppatori è quello di dare priorità agli obiettivi di design tenendo anche conto degli aspetti manageriali, quali il rispetto dello schedule e del budget.
I Design Trade-offs tipici sono **Functionality vs Usability, Cost vs Portablity, Efficiency vs Portability, Rapid development vs Functionality, Cost vs Reusability e Backward Compatibility vs Readability.**

CONCETTI DI SYSTEM DESIGN

La **decomposizione in sottosistemi** ci è utile a ridurre la complessità della soluzione e consiste molto semplicemente nel decomporre il sistema in parti più piccole chiamate appunto sottosistemi.
Un **sottosistema** è costituito da un certo numero di classi del dominio della soluzione e tipicamente corrisponde a una parte di lavoro che può essere svolta da un singolo sviluppatore o da un team di sviluppatori, la suddivisione in sottosistemi consente ai team di progetto di lavorare ai sottosistemi individuali con un minimo overhead di comunicazione, nel caso in cui si hanno sottosistemi complessi si applica lo stesso principio suddividendoli ulteriormente in altri sottosistemi più semplici.
In UML un sottosistema viene rappresentato mediante package che rappresentano collezioni di classi, associazioni, operazioni e vincoli che sono correlati, JAVA fornisce i package che sono costrutti per modellare i sottosistemi.

Esempio: decomposizione in sottosistemi



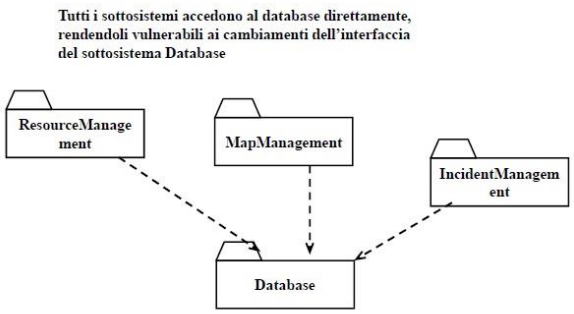
Si notino le dipendenze di UML package

Un sottosistema è dunque caratterizzato dai servizi che fornisce agli altri sottosistemi, un **servizio** è un insieme di **operazioni** correlate fornite dal sottosistema per uno specifico scopo.
L'insieme di operazioni di un sottosistema che sono disponibili agli altri sottosistemi forma **l'interfaccia** del sottosistema, essa include il **nome** delle operazioni, i loro **parametri**, il loro **tipo** ed i loro **valori di ritorno.**
Il System Design si focalizza sulla definizione dei servizi forniti da ogni sottosistema in termini di **operazioni**, loro **parametri** e loro **comportamento ad alto livello**, l'Object Design invece si focalizza sulle **operazioni**, l'API (Application Programmer Interface) che raffina ed estende le interfacce definendo i tipi dei parametri ed i valori di ritorno di ogni operazione.
Definire i sottosistemi in termini dei servizi aiuta a concentrarci sull'interfaccia piuttosto che sulla implementazione, quando si descrive un'interfaccia si dovrebbero cercare di ridurre la quantità di informazioni sull'implementazione, ciò consente di ridurre l'impatto dei cambiamenti di un sottosistema sugli altri.

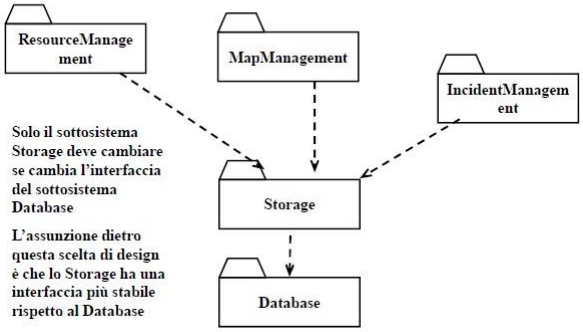
I sottosistemi godono di due proprietà:

- **COUPLING,** che ha come obiettivo quello di ridurre la complessità e rappresenta il numero di dipendenze fra due sottosistemi, intercettabili attraverso aggregazioni, specializzazioni e chiamate di operazioni.
  Se due sistemi sono **loosely coupled** (scarsamente accoppiati) sono relativamente indipendenti ciò significa che se si effettuano modifiche ad uno dei sottosistemi queste avranno un impatto insignificante sull'altro sottosistema, se invece due sistemi sono **strongly coupled** (fortemente accoppiati) ciò significa che se si effettuano modifiche ad uno dei sottosistemi queste avranno impatto sull'altro sottosistema, ovviamente si desidera che i sottosistemi siamo loosely coupled.
  Con l'obiettivo di ridurre il coupling si rischia molto spesso di aggiungere livelli di astrazione che consumano tempo di sviluppo e di elaborazione, infatti un coupling estremamente basso è auspicabile solo se ci sono elevate probabilità che qualche sottosistema cambi.

Esempio: accesso diretto al database



Esempio: accesso al database attraverso un sottosistema di Storage



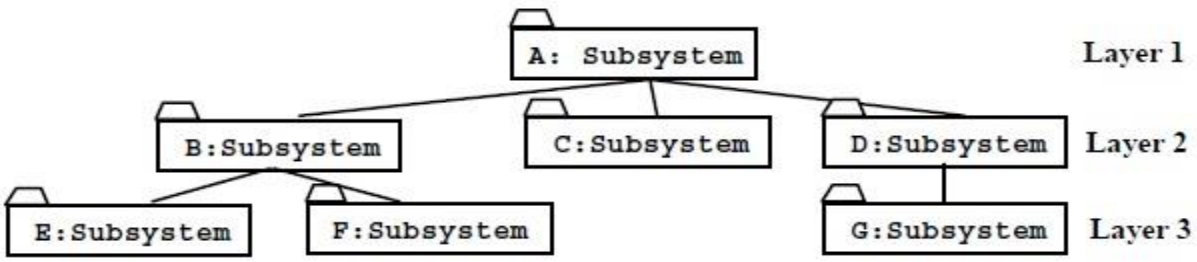
- **COHESION,** che può essere intesa a diversi livelli:
  - **CLASSES,** le operazioni costituiscono un "intero" funzionale, gli attributi e le strutture dati descrivono gli oggetti in stati ben definiti che sono modificati dalle operazioni, le operazioni si usano a vicenda;
  - **SOTTOSISTEMI,** le classi dei sottosistemi sono concettualmente correlate, le relazioni strutturali tra le classi sono fondamentalmente generalizzazioni e aggregazioni (formano cluster), le operazioni chiave sono eseguite all'interno dei sottosistemi.

  Tale proprietà misura le dipendenze delle classi in un sottosistema e può essere **high cohesion**, cioè le classi di un sottosistema realizzano compiti simili e sono collegate le une dalle altre attraverso associazioni, o **low cohesion**, cioè il sottosistema contiene un certo numero di oggetti non correlati.

Aumentando la coesione si aumenta il numero di sottosistemi ciò implica l'aumento del numero delle interfacce con un conseguente aumento di coupling.
L'euristica suggerisce che gli sviluppatori possono trattare ad ogni livello di astrazione un numero di concetti pari a 7±2, qualcosa non va se ci sono più di 9 sottosistemi ad un livello di astrazione e un sottosistema fornisce più di 9 servizi.

Un sistema di grandi dimensioni è solitamente scomposto in sottosistemi usando **layer** e **partizioni**, una decomposizione gerarchica di un sottosistema consiste di un **insieme ordinato di layer (strati)**, un layer è un raggruppamento di sottosistemi che forniscono servizi correlati, eventualmente realizzati utilizzando servizi di altri layer, esso può dipendere solo da layer di livello più basso e non ha conoscenza dei layer dei livelli più alti.
Le euristiche per la decomposizione di sottosistemi indicano che ogni layer deve contenere non più di 7±2 sottosistemi (più sottosistemi ci sono più aumenta la cohesion e la complessità) e che si devono avere non più di 5±2 layer.

Decomposizione di sottosistemi in Layer

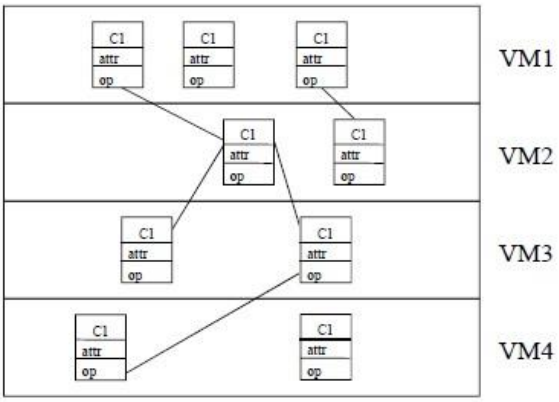


I layer possono implementare due tipi di architetture:

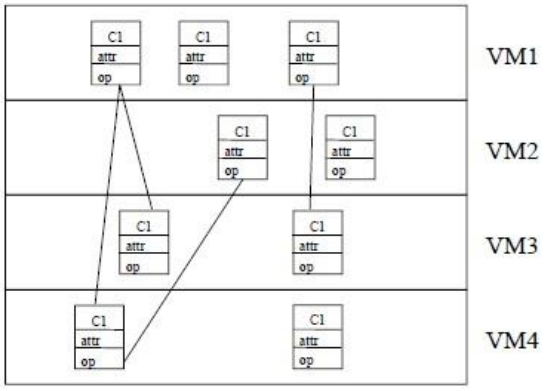
- **ARCHITETTURA CHIUSA,** dove ogni layer può accedere solo al layer immediatamente sotto di esso;
- **ARCHITETTURA APERTA,** dove un layer può anche accedere ai layer di livello più basso.

I layer godono di alcune proprietà, innanzitutto sono gerarchici e ciò riduce la complessità, le architetture chiuse sono portabili mentre quelle aperte sono più efficienti e se un sottosistema è un layer spesso viene chiamato **macchina virtuale**.
Dijkstra nel 1965 affermò che un sistema dovrebbe essere sviluppato da un insieme di macchine virtuali ognuna costruita in termini di quelle al di sotto di essa.
Una macchina virtuale è un'astrazione che fornisce un insieme di attributi e operazioni, è un sottosistema connesso a macchine virtuali di livello più alto e più basso attraverso associazioni del tipo "fornisce servizi per", e possono implementare due tipi di architetture software:

- **ARCHITETTURA CHIUSA (OPAQUE LAYERING),** dove una macchina virtuale può solo chiamare le operazioni dello strato sottostante, e il design goal è l'alta manutenibilità;



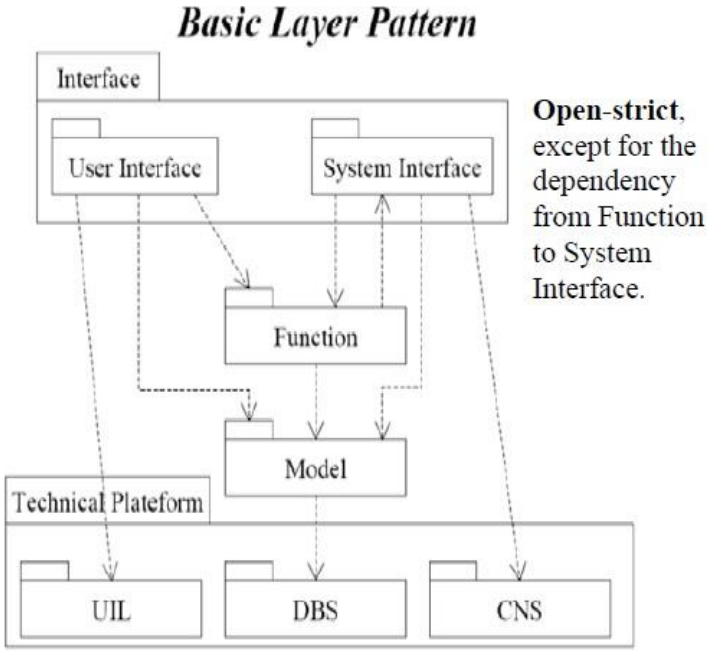
- **ARCHITETTURA APERTA (TRANSPARENT LAYERING)**, dove una macchia virtuale può utilizzare i servizi delle macchine virtuali dei layer sottostanti, e il design goal è l'efficienza relativa al tempo di esecuzione (runtime).



L'architettura chiusa presenta i vantaggi di basso accoppiamento tra le componenti e di integrazione e testing incrementale e gli svantaggi che ogni livello aggiunge overhead in termini di tempo e memoria, diventa difficile soddisfare alcuni obiettivi di design e aggiungere funzionalità al sistema può essere difficile.

Un sistema di base (basic layer pattern) contiene tipicamente i sottosistemi:

- **Interface**, che contiene gli oggetti **boundary/interface** e viene ulteriormente decomposto in **user interface** e **system interface**;
- **Function**, che contiene gli oggetti **control e la logica dell'applicazione**;
- **Model**, che contiene gli oggetti **entity** del dominio di applicazione.



UIL, è l'acronimo di qualsiasi libreria che fa parte della piattaforma tecnica e che offre funzionalità GUI, come ad esempio Awt, Swing in Java.
DBS, è l'acronimo di sottosistema di database e rappresenta qualsiasi funzione relativa ai dati persistenti, ad esempio DBMS relazionali.
CNS, è l'acronimo di sottosistema di rete contenente le funzionalità per comunicare con altri host, macchine su una rete, etc.

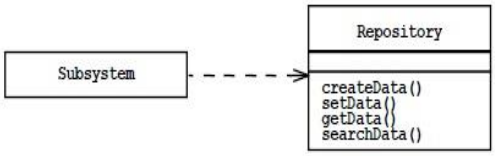
Un altro approccio consiste nel partizionare (**partition**) il sistema in sottosistemi **pari (peer)** fra loro, ognuno responsabile di differenti classi di servizi, in generale la decomposizione in sottistemi è il risultato di entrambi gli approcci partition e layering, ogni sottosistema aggiunge overhead di elaborazione a causa della sua interfaccia verso gli altri sottosistemi, eccessiva frammentazione accresce la complessità.

L'architettura software è dunque composta da:

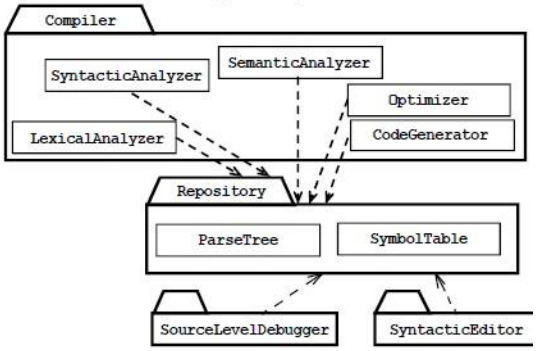
- **decomposizione del sistema in sottosistemi**, che serve ad identificare i sottosistemi, i servizi e le relazioni fra di essi, e la cui specifica rappresenta un punto critico poiché modificarla in corso di sviluppo risulta essere molto difficile perché comporta la modifica delle interfacce dei sottosistemi;
- **flusso di controllo globale;**
- **gestione delle condizioni limite;**
- **protocollo di comunicazione tra sottosistemi;**
- **relativi pattern applicati.**

Gli stili architetturali che possono essere usati come base per le architetture software sono:

- Architettura **A REPOSITORY**, dove i sottosistemi sono loosely coupled poiché accedono e modificano una singola struttura dati chiamata appunto **repository** e dunque interagiscono solo tramite quest'ultima che non ha conoscenza degli altri sottosistemi.
  Chiari esempi di sistemi basati su questa architettura sono i compilatori, DBMS, etc., dove il control flow è determinato dai sottosistemi (locks, primitive di sincronizzazione), nel compilatore ad esempio ogni tool è invocato dall'utente il repository assicura solo che gli accessi concorrenti siano serializzati, e dal repository (triggers sui dati invocano i vari sottosistemi), i sottosistemi possono essere invocati sulla base dello stato della struttura dati centrale.
  L'applicazione di tale architettura presenta il vantaggio che i repository essendo adatti per applicazioni con task di **elaborazione dati che cambiano frequentemente**, una volta che un repository centrale è stato definito possono essere definiti facilmente nuovi servizi sotto forma di **sottosistemi addizionali**, e gli svantaggi che il repository centrale può facilmente diventare un collo di bottiglia per aspetti sia di presentazione sia di modificabilità e che il coupling fra ogni sottosistema ed il repository è alto e ciò rende difficile cambiare repository senza avere impatti su tutti i sottosistemi;



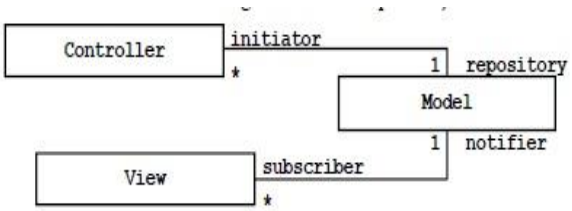
Esempio di Architettura a Repository

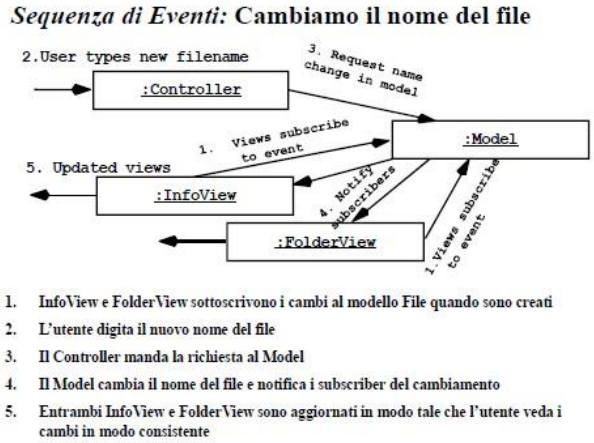


- Architettura **Model/View/Controller**, in tale stile architetturale i sottosistemi sono classificati in:
  - **sottosistema MODEL**, che mantiene la conoscenza del dominio di applicazione (fornisce le operazioni per accedere ai dati utili all'applicazione);
  - **sottosistema VIEW**, visualizza all'utente gli oggetti del dominio dell'applicazione (visualizza i dati del model e gestisce l'interazione con l'utente);
  - **sottosistema CONTROLLER**, che è responsabile della sequenza di interazione con l'utente (riceve i comandi dell'utente, generalmente attraverso view, e li attua modificando lo stato degli altri due componenti).

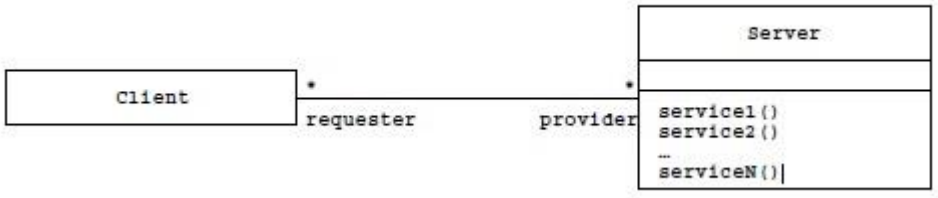
  Sostanzialmente MVC è un caso particolare di architettura di tipo repository poiché il model implementa la struttura dati centrale, il controller gestisce il control flow, cioè ottiene gli input dell'utente e manda messaggi al Model, i sottosistemi View visualizzano il Model e sono notificati (attraverso 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 (Model), MVC è appropriato per i sistemi interattivi specialmente quando si utilizzano viste multiple dello stesso Model, e soffre dello stesso problema del collo di bottiglia dei repository essendo appunto un particolare tipo di architettura a repository.

MVC è un pattern architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi software object-oriented, originariamente impiegato dal linguaggio Smaltalk è stato adottato ad oggi da numero tecnologie moderne come framework basati su Java, Ruby, PHP, .NET, etc.





- Architettura **Client/Server**, originariamente sviluppata per gestire la distribuzione tra alcuni processori geograficamente separati, oggigiorno viene spesso utilizzata nei sistemi di database e si compone di un sottosistema detto **Server**, che fornisce servizi specifici e che non conoscono le interfacce dei Client e che svolge le funzioni di gestione centralizzata dei dati, garantire integrità dei dati e consistenza del database, garantire la sicurezza del database, gestire la concorrenza delle operazioni e elaborazioni centralizzate, un insieme di **Client**, che richiedono questi servizi e dunque conoscono l'interfaccia del server e che svolgono le funzioni di fornire un'interfaccia utente personalizzata, elaborazione front-end dei data per verificare i vincoli e iniziare la transazione quando i dati sono stati collezionati, e una rete che consente ai client di accedere ai server. Gli utenti interagiscono solo con il Client ed il flusso di controllo nei client e nei server è indipendente.
  Il problema che si ha principalmente utilizzando questa architettura è che la comunicazione Peer-to-peer è necessaria.

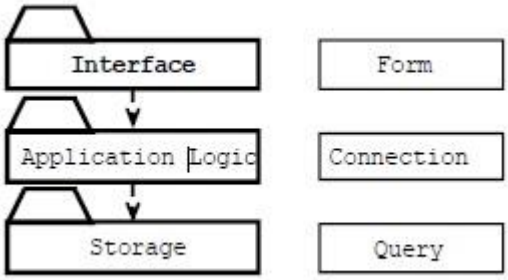


- Architettura **Peer-To-Peer**, è una generalizzazione dell'architettura client/server dove ogni sottosistema può agire sia come Client o come Server dunque può richiedere e fornire

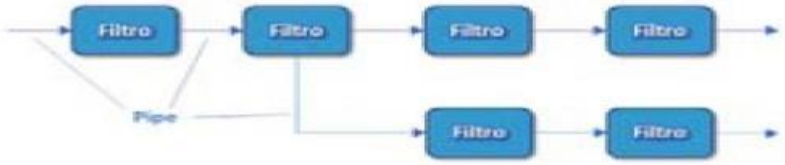
servizi, il control flow di ogni sottosistema è indipendente dagli altri, eccetto per la sincronizzazione sulle richieste.

- Architettura **Three-tier**, dove i sottosistemi sono organizzati in tre strati:
  - **INTERFACE LAYER**, che include tutti i boundary object che interfacciano con l'utente;
  - **L'APPLICATION LOGIC LAYER**, che include tutti gli oggetti relativi al controllo e alle entità che realizzano l'elaborazione, le regole di verifica e la notifica della richiesta dell'applicazione;
  - **STORAGE LAYER**, effettua la memorizzazione, il recupero e l'interrogazione di oggetti persistenti.

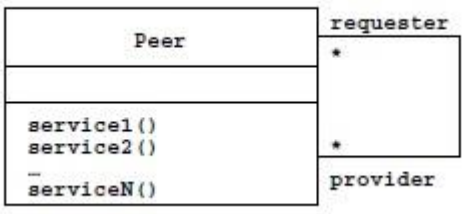
  Sperare l'interfaccia dalla logica applicativa consente di modificare e/o sviluppare diverse interfacce utente per la stessa logica applicativa.



- Architettura **a flusso di dati (PIPELINE)**, tale stile architetturale si rileva efficace nel caso in cui si abbia 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.
  Tale architettura non è adatta per sistemi interattivi o per sistemi che richiedono maggiore interazione tra componenti.



Trovare i sottosistemi è simile ad individuare oggetti nell'analisi, i criteri per la selezione dei sottosistemi sono che la maggior parte delle interazioni dovrebbe essere entro i sottosistemi piuttosto che attraverso i limiti dei sottosistemi (alta coesione) e ciò si ottiene rispondendo alla domanda di quale sottosistema chiama un altro sottosistema per ottenere i servizi, poi bisogna



porsi le domande, quale tipo di servizio è fornito dal sottosistema e quali possono essere i sottosistemi ordinati gerarchicamente (layers).
Le euristiche da seguire sono che tutti gli oggetti nello stesso sottosistema dovrebbero essere funzionalmente correlati, assegnare gli oggetti identificati in un caso d'uso allo stesso sottosistema, creare un sottosistema dedicato per gli oggetti usati per muovere i dati fra i sottosistemi e minimizzare il numero di associazioni che attraversano i limiti dei sottosistemi.

SYSTEM DESIGN PT.2

Gli obiettivi (design goal) guidano le decisioni che gli sviluppatori devono prendere specialmente quando sono necessari dei compromessi, essi dividono il sistema in sottosistemi per gestire la complessità in modo da poter assegnare lo sviluppo di un sottosistema ad un team portando a termine questa attività in modo indipendente, per far ciò si deve far fronte a determinate "scelte" quando viene decomposto il sistema:

- **Mapping Hardware/Software**, attività in cui ci si concentra su quale sia la configurazione hardware del sistema, le responsabilità di ogni nodo, come viene gestita la comunicazione tra i nodi e quali servizi sono realizzati utilizzando componenti software esistenti e come queste componenti sono incapsulate.
  Molte volte il mapping porta alla definizione di componenti addizionali che consentono di "muovere" le informazioni da un nodo ad un altro, di gestire problemi di concorrenza, ad esempio le componenti legacy o Off-the-shelf consentono agli sviluppatori di realizzare servizi complessi in modo più economico;
- **Gestione dei Dati Persistenti**, attività in cui ci si concentra su quale dovrebbe essere l'informazione persistente, dove dovrebbe essere memorizzata e come vi si dovrebbe accedere.
  Per la corretta gestione di tali dati spesso si ricorre ai DBMS e a sottosistemi addizionali, molto importante e da evitare sono i colli di bottiglia che si possono creare per via delle molte richieste d'accesso ai dati persistenti, quindi l'accesso dovrebbe essere veloce e affidabile;
- **Controllo di Accesso**, attività in cui ci concentra su chi può accedere alle informazioni, se il controllo di accesso può cambiare dinamicamente e come viene specificato e realizzato il controllo di accesso, molto importante è la consistenza, cioè la politica utilizzata per specificare chi può e chi non può accedere a certe informazioni dovrebbe essere la stessa per tutti i sottosistemi;
- **Flusso di controllo globale**, attività in cui ci si concentra su come è gestita la sequenza delle operazioni, se il sistema è guidato o meno da eventi e se il sistema può gestire più di un'interazione utente alla volta.
  La scelta del controllo del flusso ricade sulle interfacce delle componenti, nel caso di un controllo guidato da eventi i sottosistemi forniranno un gestore degli eventi, nel caso di un controllo basato su threads i sottosistemi devono garantire la mutua esclusione nelle sezioni critiche;
- **Condizioni limite**, attività in cui ci si concentra su come è avviato il sistema, come è interrotto e come sono individuati e gestiti i casi eccezionali.
  L'avvio e l'interruzione del sistema spesso rappresentano molta della complessità di un sistema, specialmente nei sistemi distribuiti.

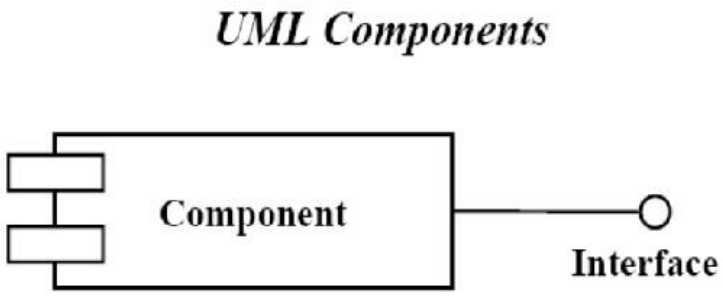
La definizione di tali condizioni incidono sulle interfacce di tutti i sottosistemi.

La struttura del System Design Document (SDD) si divide in 4 sezioni principali:

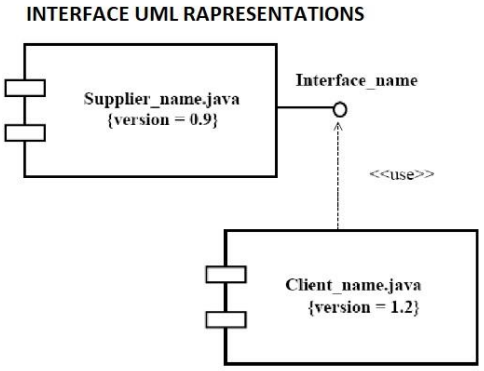
1. **INTRODUCTION**, che contiene una breve panoramica dell'architettura del software e degli obiettivi di design e si divide appunto in **purpose of the system**, **design goals**, **definition acronymous and abbreviations**, **reference**, contente i riferimenti ad altri documenti per garantire tracciabilità e **overview**;
2. **CURRENT SYSTEM ARCHITECTURE**, che ha come scopo quello di esplicitare le informazioni le informazioni precedentemente raccolte, le assunzioni e i problemi comuni che il nuovo sistema indirizzerà;
3. **PROPOSED SOFTWARE ARCHITECTURE**, che ha come scopo quello di documentare il modello di design del nuovo sistema ed è suddiviso in **overview**, che fornisce una rapida panoramica dell'architettura software descrivendo brevemente le funzionalità di ciascun sottosistema, **subsystem decomposition**, che contiene la descrizione della scomposizione in sottosistemi e la responsabilità di ognuno (prodotto principale del system design), **hardware/software mapping**, che descrive come vengono assegnati i sottosistemi all'hardware e al software ed elenca inoltre i problemi introdotti da più nodi e riutilizzo del software, **Persistent data management**, che descrive i dati persistenti memorizzati dal sistema e l'infrastruttura necessaria per gestirli, include inoltre la descrizione degli schemi di dati, la selezione di un database e la descrizione dell'incapsulamento di quest'ultimo, **Access control and security**, che descrive lo user model attraverso una matrice degli accessi, descrive inoltre problemi di sicurezza come la selezione di un meccanismo di autenticazione, l'uso della crittografia e la gestione delle chiavi, **Global software control**, che descrive come viene implementato il controllo globale del software, in particolar modo come vengono avviate le richieste, come si sincronizzano i sottosistemi e indirizzare i problemi di sincronizzazione e concorrenza, e **Boundary conditions**, che descrive l'avvio, l'arresto e il comportamento in caso di errore del sistema;
4. **SUBSYSTEM SERVICES**, che descrive i servizi forniti da ciascun sottosistema in termini di operazioni, sebbene tale sezione sia usualmente vuota o incompleta nelle prime versioni dell'SDD, serve come riferimento per il team per carpire meglio i confini di ciascun sottosistema.
   L'interfaccia di ciascun sottosistema è derivata da questa sezione e dettagliata nell'Object Design Document.

Durante il System Design dobbiamo modellare la struttura statica e quella dinamica, producendo sostanzialmente il **diagramma delle componenti per la struttura statica**, che mostra la struttura al **"design time"** o al **"compile time"**, e il **diagramma di deployment per la struttura dinamica**, che mostra la struttura al **"run-time"**.

Nel contesto di UML diciamo che distribuiamo **componenti** ai nodi, la definizione di componente dipende dall'ambiente del sistema, può comprendere file di codice sorgente ma anche eseguibili, sistemi di database, librerie, etc., un nodo dunque è un processo e nel caso più semplice ogni sottosistema viene eseguito come una (o più) componente che vengono distribuite

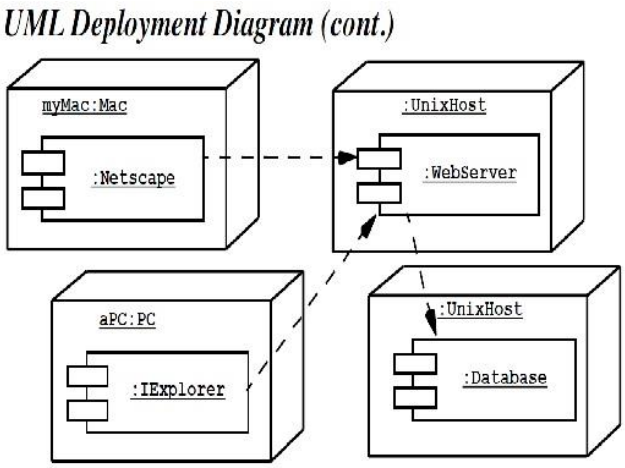


Fra vari componenti possono esistere delle **dipendenze** che indicano che un componente si riferisce ai sevizi offerti da altri componenti, un **grafo delle componenti** connesse attraverso archi dalla componente cliente alla componente fornitore, la tipologia di dipendenza poi varia da linguaggio a linguaggio.
Quando una componente viene modificata possono essere soggette a modificate anche le altre componenti che dipendono da essa, le **dipendenze dunque sono transitive** e derivano dalle proprietà rilevate nella progettazione logica, come generalizzazione, aggregazione, interfaccia.
Le interfacce, sostanzialmente generalizzano il concetto d'interfaccia di JAVA, sono caratterizzate da un nome e rappresentano un insieme di operazioni che caratterizzano il comportamento di una componente, possono aiutare a specificare quale parte di una classe viene effettivamente utilizzata dalle classi client.
Le componenti sono conformi alle interfacce e supportano tutte le operazioni nell'interfaccia.



Gli **UML Deployment Diagram**, sono utili per mostrare il progetto del sistema dopo che sono state prese le decisioni relative alla decomposizione in sottosistemi, concorrenza e mapping hardware/software, sostanzialmente tale diagramma è un grafo di nodi connessi attraverso associazioni di comunicazione, i nodi vengono mostrati come box 3D e possono contenere istanze di componenti, le componenti possono contenere oggetti per indicare che un oggetto fa parte di quella componente.

Tali diagrammi sono spesso utilizzati per descrivere le relazioni tra le componenti run-time e i nodi hardware, le componenti sono entità autonome che forniscono servizi ad altre componenti o attori.



Le attività del System Design dunque si dividono in:

- **MAPPARE LE COMPONENTI SU PIATTAFORME E PROCESSORI**, per selezionare una configurazione hardware e una piattaforma bisogna porsi questioni relative ai processori, cioè chiedersi se la computazione richiede più processori e se le mansioni possono essere distribuite, per velocizzarle, su tali processori.
  Molte delle difficoltà della progettazione del sistema software sono legate alle scelte relative all'hardware e al software imposte, ciò significa selezionare una macchina virtuale che riduce il gap tra hardware e software da realizzare, può essere vincolata dal cliente, da ciò che già possiede o da questioni di costi trade-off tra costruire o comprare.
  Per mappare gli oggetti sull'hardware, scelto, e sul software occorre mappare gli oggetti sui processori e mappare le associazioni: connettività.
  Se l'architettura è distribuita abbiamo bisogno di descrivere la rete (dunque il sistema di comunicazione) rispondendo sostanzialmente alle domande relative a quali media sono utilizzati (Ethernet, Wireless), qual è il tasso medio di trasmissione, quale tipo di protocollo di comunicazione può essere utilizzato, se utilizzare interazioni sincrone o asincrone e qual è la richiesta di larghezza di banda tra i sottosistemi.
  Altre importanti domande a cui rispondere si riferiscono a quale sia la connessione tra le unità fisiche (albero o matrice, protocollo di comunicazione tra sottosistemi, se è richiesta affidabilità), se alcune funzionalità non sono già disponibili nell'hardware scelto, se alcune mansioni richiedono specifiche locazioni per controllare l'hardware o per permettere operazioni concorrenti e qual è il tempo di risposta desiderato.
  Allocare gli oggetti e sistemi a nodi consente di distribuire le funzionalità e l'elaborazione dove è necessario, di identificare nuovi oggetti e sottosistemi per spostare informazioni tra i nodi e di contro implica il bisogno di fare delle scelte di memorizzazione, trasferimento, sincronizzazione dell'informazione fra sottosistemi
- **IDENTIFICARE E MEMORIZZARE INFORMAZIONI PERSISTENTI**, per identificare e memorizzare le informazioni persistenti possiamo rifarci a selezionare le informazioni che sopravvivono ad una singola esecuzione del sistema.
  Gli oggetti entity identificati durante la fase di analisi sono candidati ad essere dati persistenti, non tutti necessariamente.

Una volta identificati gli oggetti persistenti bisogna decidere come devono essere memorizzati, gli approcci sostanzialmente sono due:

- **FILE**, che risultano essere più economici e semplici fornendo una memorizzazione permanente, operazioni di basso livello (read e write) e comportano l'aggiunta di codice nell'applicazione per fornire un opportuno livello di astrazione.
  Questa tipologia viene preferita quando i dati sono voluminosi (immagini), quando si ha necessità di tenere traccia delle informazioni solo per un tempo breve e la densità delle informazioni è bassa;
- **DATABASE**, che sono senza dubbio più potenti e portabili e supportano letture e scritture multiple.
  Questa tipologia viene preferita quando le informazioni richiedono un accesso ad un livello raffinato di dettaglio attraverso utenti multipli, quando le informazioni devono essere portate attraverso piattaforme multiple e quando più programmatori hanno accesso alle informazioni.
  Le tipologie di database più usate sono **Database Relazionali**, basati sull'algebra relazionale, dove le informazioni sono presentate come tabelle 2D aventi un dato numero di colonne e un arbitrario numero di righe, ogni colonna rappresenta un attributo, gli attributi più importanti sono **chiave primaria**, combinazione di attributi che identifica univocamente una riga nella tabella, e **chiave esterna**, che è un riferimento alla chiave primaria di un'altra tabella, SQL è il linguaggio standard per definire e manipolare le tabelle e i database commerciali supportano i vincoli di **integrità referenziale**, e **Database Object-Oriented**, che forniscono servizi simili ad un database relazionale con la differenza di supportare tutti i concetti fondamentali della modellazione object-oriented (classi, attributi, metodi, associazioni, ereditarietà), ma che soffrono di efficienza.

- **STABILIRE CONTROLLO DI ACCESSO**, in sistemi multi-utente, differenti attori hanno accesso a diverse funzionalità e informazione, ad esempio, un attore qualsiasi può accedere solo ai dati che crea e un amministratore ha accesso a tutti i dati del sistema, durante l'analisi si è modellato questa distinzione associando differenti use case a differenti attori, durante il system design invece di modellano gli accessi determinando quali oggetti sono condivisi tra gli attori, ciò vuol dire descrivere i diritti di accesso per classi differenti di attori, descrivere come gli oggetti "vigilano" contro gli accessi non autorizzati e meccanismi di autenticazione, meccanismi di crittografia dei dati.
  Le domande a cui rispondere dunque sono relative al verificare che il sistema richieda un meccanismo di autenticazione, quale schema applicare e quale interfaccia utente utilizzare.
  La matrice di accesso modella i diritti di accesso su una classe, le righe rappresentano attori e le colonne le classi, ogni entry (attore, classe) contiene le operazioni consentite da quell'attore sulle istanze di quella classe, ci sono diversi modi di rappresentarli in base ai design goal:
  - **GLOBAL ACCESS TABLE**, che richiede molto spazio per essere memorizzata, rappresenta esplicitamente ogni cella nella matrice come una tupla (actor, class, operator), se una tale tupla non c'è allora l'accesso è negato;
  - **ACCESS CONTROL LIST**, che associa una lista di coppie (actor, operator) per ogni classe a cui si può fare accesso, ogni volta che si accede ad un oggetto la sua lista

degli accessi è controllata per il corrispondete attore e operazione, consente di rispondere facilmente a domande del tipo “Chi accede a questo oggetto?”;
  - **CAPABILITY**, che associa una coppia (class, operator) con un attore, una capability consente ad un attore di accedere ad un oggetto della classe descritta nella capability, e di rispondere facilmente a domande tipo “a quali oggetti accede questo attore?”.
- **PROGETTARE IL FLUSSO DI CONTROLLO GLOBALE**, il flusso di controllo globale è la sequenza di azioni del sistema, in un sistema Object-Oriented la sequenza delle azioni include le decisioni su quali operazioni dovrebbero essere eseguite e in quale ordine, queste decisioni si basano su eventi esterni generati da attori o dal trascorrere del tempo.
  Durante l’analisi assumiamo che tutti gli oggetti siano eseguiti simultaneamente, eseguendo le operazioni ogni volta che necessitano di eseguirle, nel System Design invece bisogna tener conto che non tutti gli oggetti hanno un proprio processore a disposizione.
  I meccanismi per il controllo del flusso globale sono:
  - **PROCEDURE-DRIVEN CONTROL**, dove i controlli risiedono nel codice del programma e le operazioni aspettano che un attore fornisca l’input, tale meccanismo risulta semplice, facile da costruire e molto utilizzato per sistemi procedurali e legacy;
  - **EVENT-DRIVEN CONTROL**, dove il ciclo principale attende un evento esterno che quando si verifica è spedito all’oggetto appropriato sulla base dell’informazione associata all’evento, il controllo risiede in un dispatcher che chiama le funzioni del sottosistema. Tale meccanismo risulta essere flessibile e ottimo per le interfacce utenti;
  - **THREADS**, che rappresentano la versione concorrente del Procedure-driven control, il sistema può creare un numero arbitrario di threads ed ognuno di essi risponde ad un differente evento, se un threads necessita di informazioni addizionali aspetta un input da uno specifico attore.

  Scelto il meccanismo per il flusso di controllo si passa alla sua realizzazione attraverso un insieme di uno o più oggetti control, il ruolo di tali oggetti è quello di memorizzare gli eventi esterni, il loro stato temporaneo, gestire la giusta sequenza di chiamate di operazioni sugli oggetti boundary e entity associati con gli eventi esterni.
  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.
- **IDENTIFICARE LE CONDIZIONI LIMITE**, nella fase di design bisogna determinare le condizioni limite per il sistema che si sta sviluppando:
  - **INIZIALIZZAZIONE**, che descrive come il sistema è portato da uno stato non inizializzato ad uno stato stabile, le domande a cui rispondere per tale condizione sono relative a come parte il sistema, quali informazioni sono necessarie per l’avvio, quali servizi devono essere registrati, cosa fanno le interfacce utenti all’avvio del sistema e come si presentano all’utente;
  - **TERMINAZIONE**, che descrive quali risorse sono rilasciate e quali sistemi sono notificati della terminazione, le domande a cui rispondere per tale condizione sono relative a come possono terminare i singoli sistemi, se gli altri sistemi sono notificati alla terminazione di un sottosistema e se gli aggiornamenti locali sono comunicati al database;

- **FALLIMENTO**, molte possibili cause, tra cui, errore, problemi esterni, etc., buoni sistemi progettano i fallimenti fatali, le domande a cui rispondere per tale condizione sono relative a come il sistema si comporta quando un nodo o link di comunicazione fallisce, se ci sono link di comunicazione di backup, come il sistema recupera da un fallimento e se è differente dall'inizializzazione;

Tali condizioni non vengono trattate nell'analisi perché parecchie condizione limite sono determinate da decisioni di design.
In generale, gli use case per le condizioni limite vengono identificati esaminando ogni sottosistema e ogni oggetti 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 e configure;
- **GESTIONE ECCEZZIONI**, 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, sono causate da tre differenti risorse **fallimento hardware**, quando l'hardware invecchia e fallisce (esempio crash hdd), **cambiamento nel sistema operativo**, l'ambiente influenza il lavoro del sistema (pausa di erogazione elettrica), **fallimento del software**, un errore si verifica perché il sistema o una delle sue componenti contiene un errore commesso durante la fase di progetto.
  Per gestire le eccezioni dunque si ha la necessità di utilizzare un meccanismo attraverso cui il sistema tratta le eccezioni, nel caso di errore utente, il sistema mostra all'utente un messaggio così da poter far fronte all'errore, nel caso di fallimento di un link di comunicazione, il sistema dovrebbe salvare lo stato temporaneo cosi che possa essere recuperato quando la rete ritorna ad essere funzionante, e si raffinano gli use case in modo da descrivere le situazioni in cui si possono verificare le eccezioni.
  Durante il System Design si gestiscono le eccezioni a livello di componente, mentre durante l'Object design a livello degli oggetti.

- **RIVEDERE IL MODELLO DEL SYSTEM DESIGN**, come l'analisi il design è un'attività iterativa e incrementale (segue una successione di cambiamenti), i cambiamenti devono però essere controllati per prevenire il caos, specialmente in grossi progetti dove il numero di partecipanti è elevato, vi sono tre tipi di iterazioni, cambiamenti relativi a decisioni che **riguardano la decomposizione del sistema in sottosistemi**, cambiamenti relativi a decisioni che **riguardano le interfacce** e cambiamenti relativi a decisioni **da prendere per gestire condizioni di errore** che sono scoperte in fasi avanzate del progetto.
  Tale attività differisce dall'analisi per via dell'assenza di agenti esterni (clienti) che revisionano il lavoro e le scelte fatte, il manager del progetto e gli sviluppatori devono organizzare un processo di revisione per sostituirsi al cliente, in aggiunta per raggiungere gli obiettivi del progetto (design goal) bisogna assicurarsi che il system design sia:
  - **CORRETTO**, ponendosi domande relative a se ogni sottosistema possa essere tracciato su uno use case o una richiesta funzionale, se ogni use case possa essere

mappato su un insieme di sottosistemi, se ogni richiesta non funzionale è analizzata nel system design e se ogni attore abbia una politica di accesso;

- **COMPLETO**, ponendosi domande relative a se le condizioni limite sono giuste, se c'è una rivisitazione degli use case per identificare funzionalità non realizzate nel sistema, se tutti gli use case sono stati esaminati ed è stato assegnato un oggetto control e se ogni aspetto del system design è stato analizzato;
- **CONSISTENTE**, ponendosi domande relative a se è stata aggiunta una priorità ai design goal in conflitto, se ci sono design goal che violano requisiti non funzionali, se ci sono classi o sottosistemi con lo stesso nome e se le collezioni di oggetti sono scambiate in modo consistente tra i sottosistemi;
- **REALISTICO**, ponendosi domande relative a se le nuove tecnologie sono incluse nel sistema, se i requisiti di performance e di affidabilità sono esaminati durante la decomposizione in sottosistemi e se la concorrenza è stata analizzata;
- **LEGGIBILE**, ponendosi domande relative a se i nomi dei sottosistemi sono comprensibili, se entità con nomi simili rappresentano concetti simili e se tutte le entità sono descritte con lo stesso livello di dettaglio.

Il system design è una fase che è incentrata intorno al lavoro fatto **dall'architecture team**, che è responsabile della decomposizione del sistema in sottosistemi e dell'assegnazione ad ogni team delle proprie responsabilità e della selezione degli sviluppatori che devono realizzare le diversi componenti, l'**architetto** riveste il ruolo principale nel system design, assicura la consistenza sia nelle decisioni di design e nello stile delle interfacce sia del design fra i team per la gestione delle configurazioni e del testing, i **liaisons** sono i rappresentati dei team, la cui grandezza è determinata dal numero dei sottosistemi, che lavorano ai diversi sottosistemi, si occupano di raccogliere informazioni da e verso il loro team, negoziano i cambiamenti delle interfacce e durante la fase di design si concentrano sui servizi dei sottosistemi e nella fase di implementazione si concentrano sui dettagli delle API.

M4 – OBJECT DESIGN

OBJECT DESIGN

Durante l'analisi si descrive lo scopo del sistema e si identificano gli oggetti di applicazione, durante il system design viene descritta l'architettura del sistema, la piattaforma hardware e software che permette di selezionare le componenti off-the-shelf, etc., durante l'**object design** chiudiamo il gap tra oggetti di applicazione e componenti off-the-shelf identificando oggetti di soluzione e raffinando gli oggetti esistenti.
L'Object design è il processo che si occupa di **aggiungere dettagli all'analisi dei requisiti** e **prendere decisioni di implementazione**, per ottenere ciò iteriamo nel processo di assegnazione delle operazioni al modello ad oggetti, e risulta essere molto importante perché serve come base dell'implementazione, colui che ha il compito di scegliere fra le diverse metodologie implementative quale applicare per implementare il modello di analisi con l'obiettivo di minimizzare il tempo di esecuzione, la memoria ed altri costi è l'**object designer**.
Le attività dell'Object Design sono:

- **RIUSO**, dove le componenti off-the-shelf identificate durante il system design sono utilizzate nella realizzazione di ogni sottosistema, vengono selezionate librerie di classi ed altre componenti utili per strutture dati e servizi di base, vengono selezionati i Design Pattern per risolvere problemi comuni e per proteggere classi da futuri cambiamenti. Durante tutte queste attività gli sviluppatori devono decidere tra buy-versus-build trade-off (comprare vs. costruire).
  Molte volte le componenti devono essere adattate prima di poter essere utilizzate e ciò si può fare **attraverso oggetti wrapper** o **raffinamenti utilizzando l'ereditarietà**;
- **SPECIFICA DELLE INTERFACCE**, dove i servizi forniti dai sottosistemi vengono specificati in termini di interfacce di classi, incluso operazioni, argomenti, tipi per le firme, ed eccezioni, sono identificati anche ulteriori azioni ed oggetti necessari per trasferire dati tra i sottosistemi.
  Tale attività produce come risultato una specifica completa delle interfacce per ogni sottosistema che è spesso chiamata **API (Application Programmer Interface)** del sottosistema;
- **RISTRUTTURAZIONE**, dove il modello del sistema viene modificato per aumentare il riuso del codice o per soddisfare altri design goal, le tipiche attività sono, **trasformare associazioni binarie in N-arie**, **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** e **aumentare l'ereditarietà ed il packaging modificando classi ed operazioni**.
  Durante tale fase ci si occupa anche di come soddisfare design goal come mantenimento, leggibilità e comprensione del modello del sistema;
- **OTTIMIZZAZIONE**, dove ci si occupa di soddisfare i requisiti di performance del modello di sistema, tale 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 (dare la possibilità di accedere a strati di basso livello)**.

L'attività di object design non è sequenziale, viene svolta infatti in maniera concorrente, ma poiché potrebbero sorgere delle dipendenze, ad esempio componente off-the-shelf può vincolare il tipo di eccezioni di un'operazione, etc., vengono fatte prima le attività di riuso e di specifica delle interfacce per ottenere un modello ad oggetti di design che viene verificato rispetto ai corrispondenti casi d'uso, una volta che il modello è stabile si procede a svolgere le fasi di ristrutturazione ed ottimizzazione, tali fasi se fatte prima delle fasi di riuso e di specifica delle interfacce potrebbero ridurre il numero di oggetti da implementare e quindi aumentare il riuso.

OBJECT DESIGN: SPECIFICARE LE INTERFACCE

Durante l'object design identifichiamo e raffiniamo gli oggetti "solution" per realizzare i sottosistemi definiti durante il system design, la comprensione degli oggetti dunque diviene più approfondita. Nel system design il focus è l'identificazione di grandi parti di lavoro da assegnar ai vari team o sviluppatori, nell'object design invece il focus è la specifica dei confini tra i vari oggetti. Nella **specifica delle interfacce il focus** è comunicare chiaramente e precisamente i dettagli di basso livello degli oggetti del sistema e descrivere precisamente l'interfaccia di ogni oggetto cosi

che non ci sia necessità di lavoro di integrazione per oggetti realizzati da diversi sviluppatori, mentre le attività sono:

- **IDENTIFICARE ATTRIBUTI E OPERAZIONI MANCANTI**
- **SPECIFICARE LE SIGNATURE E LA VISIBILITÀ DI OGNI OPERAZIONE**
- **SPECIFICARE LE PRECONDIZIONI (sotto le quali un'operazione può essere invocata e quelle che determinano un'eccezione)**
- **SPECIFICARE LE POSTCONDIZIONI**
- **SPECIFICARE LE INVARIANTI**

L'**Object Constraint Language (OCL)** consente di specificare precondizioni, postcondizioni e invarianti e le euristiche e linee guida ci consentono di scrivere vincoli leggibili.
Tutti i modelli prodotti fin qui forniscono **una visione parziale del sistema**:

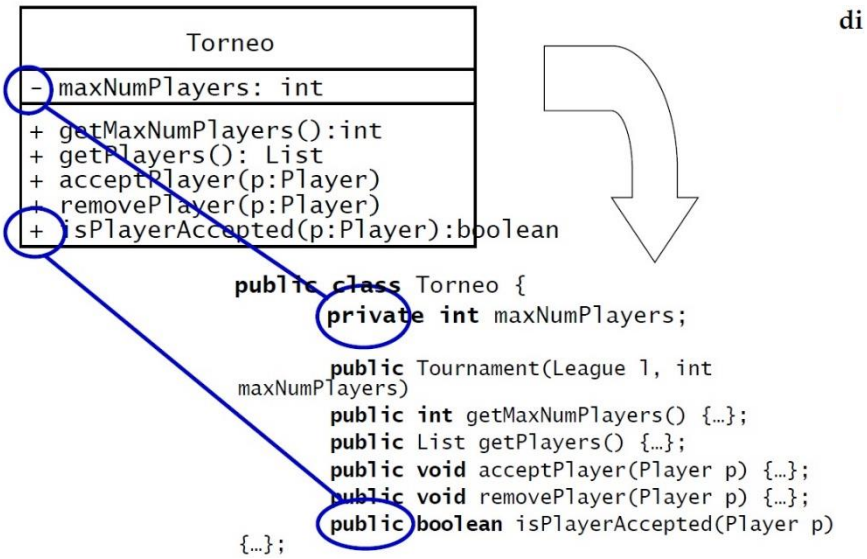
- Il modello ad oggetti di analisi descrive gli oggetti entity, boundary e control che sono visibili all'utente;
- La decomposizione in sottosistemi, dove ognuno di essi fornisce un insieme di servizi (ad alto livello) ad altri sottosistemi, descrive come questi oggetti sono partizionati in pezzi coesi realizzati da diversi team;
- Il mapping Hardware/Software, identifica le componenti che costituiscono la macchina virtuale su cui costruiamo gli oggetti soluzione;
- Use case boundary, descrivono dal punto di vista dell'utente, casi amministrativi ed eccezionali gestiti dal sistema;
- Design Pattern, selezionati durante l'object design reuse, descrivono object design parziali che risolvono questioni specifiche.

L'obiettivo dell'object design è quello di produrre un modello che integri tutte le informazioni in modo coerente e preciso, da tale attività viene rilasciato un documento, l'**ODD (Object Design Document)**, contenente la specifica di ogni classe per supportare lo scambio di informazioni consentendo di prendere decisioni consistenti sia tra i vari sviluppatori che con gli obiettivi di design.
L'attività di identificare attributi e operazioni senza specificare il loro tipo e i loro parametri, svolta durante l'analisi dei requisiti, viene ripresa nell'object design e suddivisa in ben 3 attività:

1. **AGGIUNGERE INFORMAZIONE RELATIVA ALLA VISIBILITÀ**, poiché diversi sviluppatori hanno diverse necessità e non tutti accedono alle operazioni e agli attributi di una classe. UML definisce 3 livelli di visibilità:
    - **Privato** indicata con il simbolo **-**, che indica per un attributo che vi può accedere solo la classe in cui è definito, per un'operazione che può essere invocata solo dalla classe in cui è definita, e che ad entrambi se privati non vi possono accedere sottoclassi o altre classi;
    - **Protetto** indicata con il simbolo **#**, che indica per un attributo o un'operazione che vi può accedere solo la classe in cui sono definiti e ogni suo discendente;
    - **Pubblico** indicata con il simbolo **+**, che indica per un attributo o un'operazione che vi possono accedere tutte le classi (interfaccia pubblica).

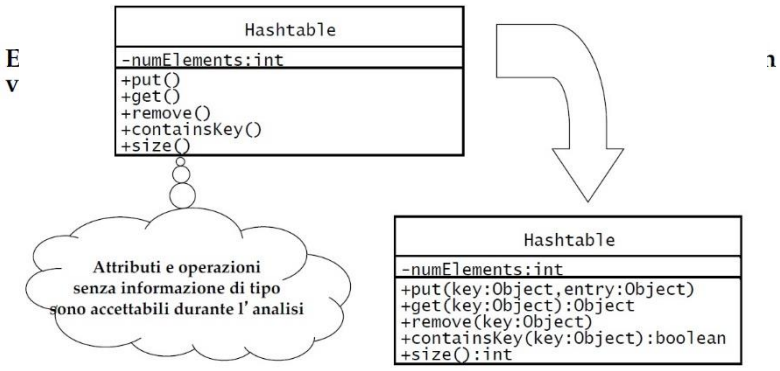
Per l'aggiunta di questa informazione vi sono delle euristiche di cui si consiglia l'adozione, chiamate **Euristiche per Information Hiding**, che suggeriscono di definire attentamente l'interfaccia pubblica per le classi così come per i sottosistemi, applicare sempre il principio di **"Need to know" (bisogno di sapere)**, cioè solo se qualcuno necessita di accedere all'informazione allora si rende possibile questo accesso ma solo attraverso canali ben definiti in modo da tener traccia degli accessi, meno una operazione sa e più bassa sarà la probabilità che sarà influenzata da qualche cambiamento e più facilmente la classe potrà essere cambiata, e trade-off: information hiding vs efficienza, accedere a un attributo privato potrebbe essere troppo lento (ad esempio in sistemi real-time);

Implementazione della visibilità UML in Java



2. **AGGIUNGERE INFORMAZIONE SUI TIPI E SULLE SIGNATURE**, poiché il **tipo** di un attributo fornisce informazione sul range dei valori consentiti e le possibili operazioni
   la **signature** è una tupla che fornisce informazioni similari sui parametri delle operazioni e eventuali valori di ritorno, tali valori hanno una specifica di tipo che vincola il range dei valori che possono assumere;
   Esempio:

2. Aggiungere Informazione di Tipo alle Signature



3. **AGGIUNGERE CONTRATTI**, poiché consentono ai vari sviluppatori di condividere le stesse informazioni sulle classi, si ha bisogno di queste informazioni dato che alle volte

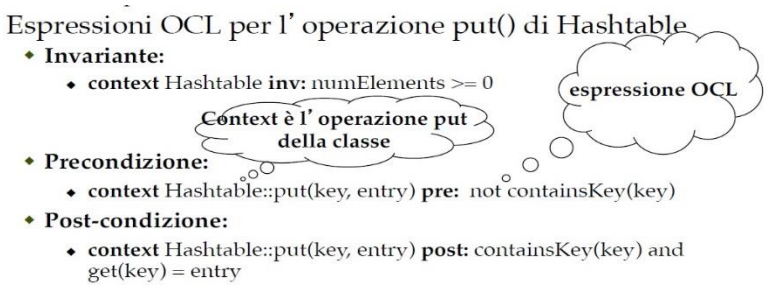
l'informazione di tipo non è sufficiente a specificare il range dei valori consentiti di un attributo, ad esempio:

Es. maxNumPlayers di tipo int può assumere valori negativi

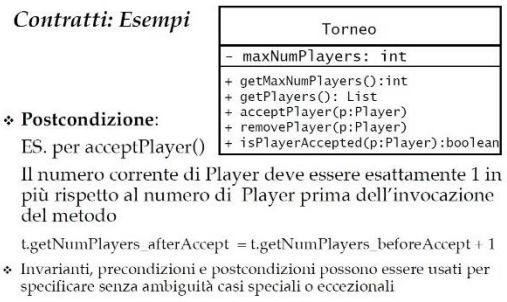
Per specificare ciò si possono aggiungere i contratti, che se aggiunti su una classe consentono a class users, implementors ed extenders di condividere le stesse assunzioni sulla classe.

I contratti includono 3 tipi di vincoli:

- **INVARIANTE**, che è un predicato che è sempre vero per tutte le istanze di una classe, invarianti sono i vincoli associati a classi o interfacce;
- **PRECONDIZIONI**, che sono predicati associati con una specifica operazione e deve essere vera prima che l'operazione sia invocata, vengono usate per specificare vincoli che un chiamate deve soddisfare prima di chiamare un'operazione;
- **POSTCONDIZIONE**, che sono predicati associati con una specifica operazione e devono essere vere dopo che l'operazione è stata invocata, sono utilizzare per

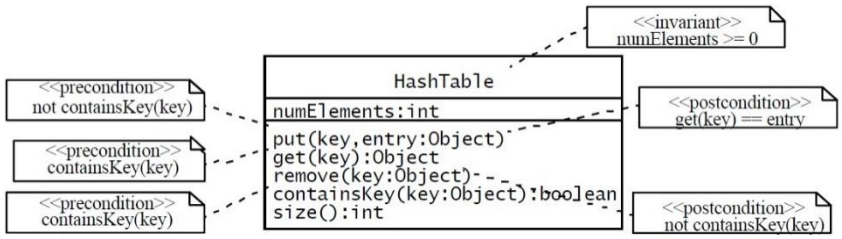


specificare vincoli che l'oggetto deve assicurare dopo l'invocazione dell'operazione.

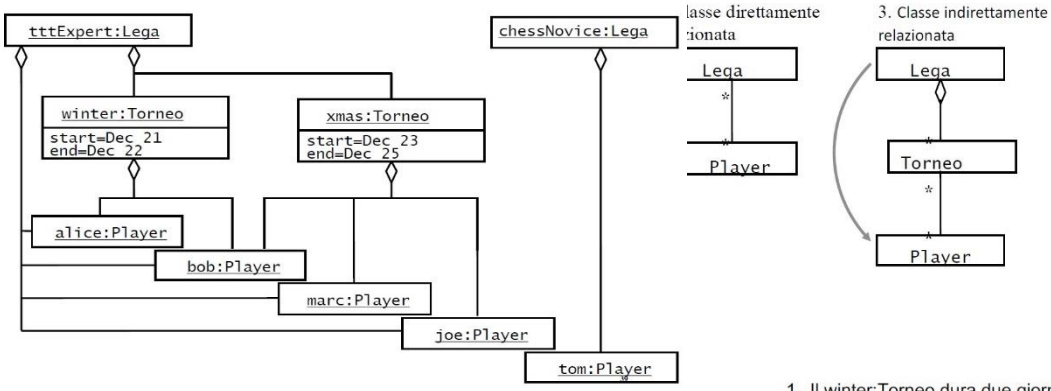


Per esprimere le **Constraints (vincoli)** ne modelli UML si utilizza l'**OCL (Object Constraint Language)**, che non è un linguaggio procedurale cioè non si vincola il control flow, che consente di specificare formalmente i vincoli sugli elementi di un singolo modello (attributi, operazioni, classi) o gruppi di elementi di modello (associazioni e classi partecipanti), un constraint è dunque espresso come un'espressione OCL che ritorna un valore vero o falso.

Un constraint può anche essere illustrato come una nota attaccata all'elemento UML vincolato tramite una relazione di dipendenza



I constraint possono coinvolgere più di una classe, ad esempio abbiamo i seguenti vincoli:
per essere costruiti questi vincoli abbiamo bisogno di effettuare 3 tipi di navigazione nel class diagram:

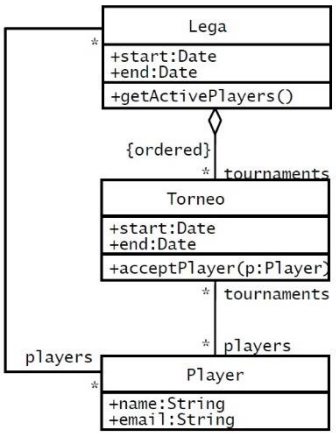


1. Il winter:Torneo dura due giorni, il xmas:Torneo tre giorni, entrambi meno di una settimana
2. Tutti i Players di winter:Torneo e xmas:Torneo sono associati alla tttExpert:Lega. Il Player tom, non fa parte della tttExpert:Lega e non prende parte al alcun Torneo
3. tttExpert:Lega ha 4 Players attivi, mentre la chessNovice:Lega non ne ha nessuno poiché Player tom, non prende parte al alcun Torneo

- La 1) coinvolge un attributo che è locale alla classe di interesse (Torneo)
- La 2) richiede la navigazione di una associazione con una classe direttamente collegata (Players di Torneo, Lega di un Torneo)
- La 3) richiede la navigazione di una serie di associazioni a una classe collegata indirettamente (i Players di tutti i Tornei di una Lega)

Ogni vincolo può essere costruito come combinazione di questi tre casi base

E per meglio comprenderli possiamo istanziare un class diagram per uno specifico gruppo di istanze, chiamato **instance diagram**, con ad esempio 2 Leghe, 2 Tornei e 5 Players:
Quando abbiamo a che fare con associazioni molti a molti alcune espressioni, come ad esempio



1. La durata Pianificata per un Torneo deve essere inferiore a una settimana.
2. Players possono essere accettati in un Torneo solo se sono già registrati con la Lega corrispondente.
3. Il numero di Players attivi in una Lega sono quelli che hanno preso parte ad almeno un Torneo della Lega.

lega.players, possono riferirsi a molti oggetti, dunque OCL fornisce 3 tipologie di **collezioni**:

- **OCL SETS (insiemi)**, usati quando si naviga una singola associazione, ad esempio navigando l'associazione player di winter:Torneo otteniamo l'insieme {alice, bob}, per riferirci ad un'associazione utilizziamo il nome del ruolo (della classe) presente sull'associazione, nel

caso in cui non esita utilizziamo il nome della classe relazionata denotata con la lettera minuscola, ad esempio poiché nel diagramma non è specificata il nome della relazione fra Player e Lega utilizziamo il nome lega per rappresentarla.
Ovviamente se l'associazione ha cardinalità uno avremo un elemento e non un insieme;

- **OCL SEQUENCES (sequenze)**, usati quando si naviga una singola associazione ordinaria;
- **OCL BAGS (multinsiemi)**, usati per accumulare oggetti quando si accede a oggetti correlati in modo indiretto, ad esempio se siamo interessati a determinare quali Player sono attivi nella tttExpert:Lega navighiamo 3 associazioni rispettivamente tttExpert:Lega, winter:Torneo e xmas:Torneo per poi ottenere il bag {alice, bob, bob, marc, joe}, se non siamo interessati al numero di occorrenze nel bag allora il bag può essere sostituito in un insieme utilizzando l'operatore **asSet(collection)**.

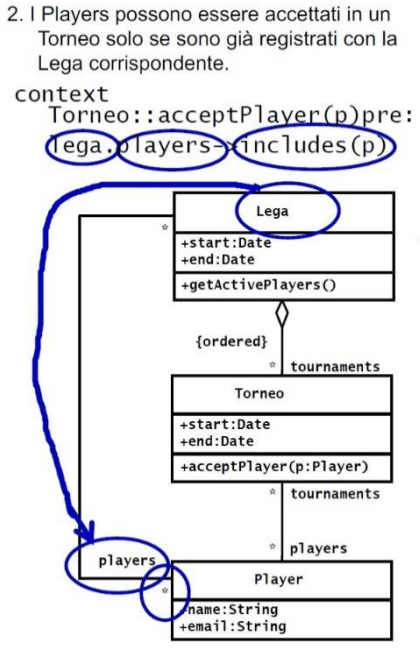
Le **operazioni OCL** per accedere alle collezioni sono:

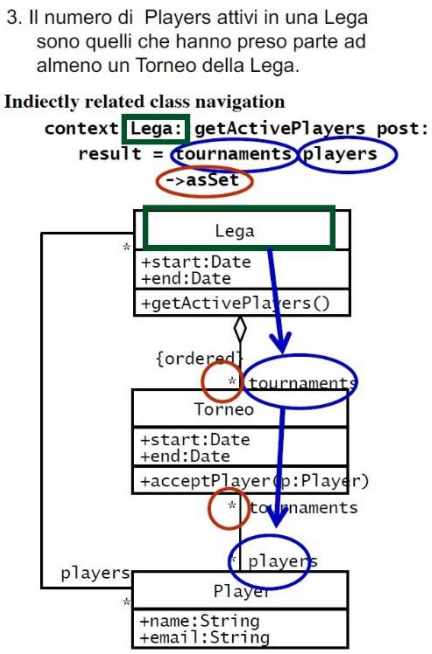
- **→**, operatore per accedere alle collezioni;
- **SIZE**, per restituire il numero di elementi nella collezione;
- **INCLUDES(OBJECT)**, per restituire TRUE se object è nella collezione;
- **SELECT(EXPRESSION)**, per restituire la collezione contenente solo gli elementi della collezione originale per cui expression è TRUE;
- **UNION(COLLECTION)**, per restituire la collezione contenente sia gli elementi della collezione originale sia quelli della collezione specificata come parametro;
- **INTERSECTION(COLLECTION)**, per restituire la collezione contenente solo gli elementi che appartengono sia alla collezione originale sia alla collezione specificata come parametro;
- **asSet(collection)**, per restituire un insieme contente tutti gli elementi che appartengono alla collezione.

OCL supporta anche la **quantificazione** e fornisce a tal proposito gli operatori:

- **forAll(variabile | espressione)**, che è vera se l'espressione è vera per tutti gli elementi nella collezione;
- **exists(variabili | espressione)**, che è vera se esiste almeno un elemento nella collezione per cui l'espressione è vera.

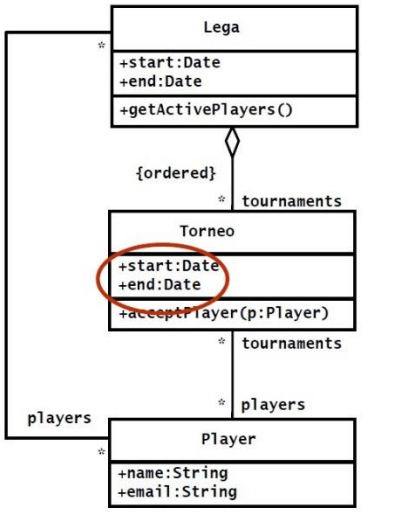
Dunque i vincoli dapprima specificati in OCL verranno rappresentati come segue:





1. La durata Pianificata per un Torneo deve essere inferiore a una settimana.

```
context Torneo inv:
  end - start <= Calendar.WEEK
```



Durate l'Object Design si può incorrere in due problemi di gestione che riguardano **l'aumento della complessità di comunicazione**, poiché il numero di persone che partecipano all'object design aumenta notevolmente è necessario assicurare che le decisioni prese siano in accordo con gli obiettivi di progetto, e la **consistenza con le precedenti decisioni e documenti**, poiché dettagliando e raffinando il modello ad oggetti gli sviluppatori possono rivedere alcune decisioni prese durante le fasi precedenti ed occorre perciò tener traccia di questi cambiamenti ed assicurarsi che tutti i documenti li riflettano in modo consistente.
L'ODD serve dunque per scambiare informazioni sulle interfacce tra i team e come riferimento per il testing, ed è rivolto agli architetti che partecipano al system design, agli sviluppatori che realizzano ogni sottosistema e al tester, e si compone delle seguenti sezioni:

- **INTRODUZIONE**, contenente una breve descrizione dell'analisi dei trade-off realizzati dagli sviluppatori e le convenzioni e linee guida che servono a migliorare la comunicazione che restano invariate.
  Tale sezione è scomposta a sua volta in Object Design Trade-offs, Linee guida per la documentazione delle interfacce, Definizioni acronimi e abbreviazioni e Riferimenti,
- **PACKAGES**, contenente la descrizione della decomposizione di sottosistemi in package e l'organizzazione in file del codice e le dipendenze tra i package e il loro uso;
- **CLASS INTERFACES**, contenente la descrizione delle classi e le loro interfacce pubbliche;
- **CLASS DIAGRAM**;
- **GLOSSARIO**.

Le sezioni Package e Class Interfaces possono essere generate da un tool utilizzando i commenti del codice sorgente, Javadoc genera pagine web dai commenti del codice, gli sviluppatori annotano dichiarazioni di interfacce e classi con commenti tagged, usando i vincoli è anche possibile includere pre e post condizioni nell'header dei metodi, ciò consente di mantenere la consistenza più facilmente tenendo assieme materiale dell'ODD e codice sorgente.

DOCUMENTAZIONE E COMMENTI: JAVADOC

Java SDK contiene uno strumento molto utile, Javadoc, che genera documentazione HTML dal file sorgente, la documentazione delle classi predefinite dei package Java è prodotta in questo modo e i commenti devono cominciare tutti con il delimitatore speciale **/****, il vantaggio è appunto mantenere codice e documentazione nello stesso file, il codice e i commenti si possono aggiornare e la documentazione può essere riprodotta con Javadoc.

Ogni commento di documentazione contiene testo formattato liberamente seguito da **tag**, un tag comincia con il simbolo **@**, ad esempio **@author, @param**, etc., la prima frase del testo deve essere sempre una frase di riepilogo, Javadoc poi genera automaticamente pagine di sommario che estraggono tali frasi, nel testo libero però è possibile utilizzare modificatori HTML.

L'utility Javadoc estrae informazioni relative agli elementi Package, Classi e interfacce pubbliche, Metodi pubblici e protetti e Campi pubblici e protetti, è possibile fornire coommenti per ognuno di tali elementi, ogni commento viene inserito immediatamente sopra la funzione che descrive e comincia con il delimitatore speciale **/**** e termina con un altro delimitatore speciale ***/**.

- **COMMENTI ALLE CLASSI**, devono essere inseriti dopo ogni dichiarazione import prima della definizione della classe altrimenti verranno ignorati da Javadoc

```
/**
   Un oggetto <code>Card</code> rappresenta una carta da gioco,
   come "Regina di cuori". Una carta ha un seme (Cuori, Quadri, Fiori
   o Picche) e un valore (1 = Asso, 2 ... 10, 11 = Fante, 12 = Regina,
   13 = Re).
*/

public class Card
{
...
}
```

- **COMMENTI AI METODI**, devono trovarsi appena prima del metodo che descrivono e si ha la possibilità di usare i tag:
  - **@param variabile descrizione**, che aggiunge una voce alla sezione "parametri" del metodo corrente con una descrizione che può essere anche di più righe con la possibilità di usare tag HTML e tutti i tag @param devono stare assieme;
  - **@return descrizione**, che aggiunge una sezione "return" al metodo corrente con una descrizione che può essere di più righe con la possibilità di usare tag HTML;
  - **@throws classe descrizione**, che aggiunge una nota per indicare che il metodo può lanciare un'eccezione.

```
/**
   Aumenta lo stipendio di un impiegato.
   @param byPercent la percentuale di cui aumentare
                    lo stipendio (es. 10 = 10%)
   @return la quantità di aumento
*/
public double raiseSalary(double byPercent)
{
   double raise = salary * byPercent / 100;
   salary += raise;
   return raise;
}
```

- **COMMENTI AI CAMPI**, è necessario commentare solo i campi pubblici nella maggior parte dei casi le costanti statiche.

```
/**
   Il seme "Cuori" delle carte
*/
public static final int CUORI = 1;
```

I commenti generali sono:

- **@author nome**, genera una voce autore;
- **@version testo**, genera una voce versione;
- **@since testo**, genera una voce "da" ("@since versione 17 -> da versione 17");
- **@deprecated testo**, aggiunge un commento che indica che l'elemento non dovrebbe più essere utilizzato, il testo infatti dovrebbe suggerire la sostituzione dell'elemento citato;
- **@see collegamento**, che aggiunge un collegamento ipertestuale nella sezione "see also", per specificare il collegamento possiamo utilizzare due notazioni: **"string"** e **etichetta**.

Per estrarre i commenti, sia docDirectory la directory dove si desidera che vadano i file HTML di documentazione, bisogna:

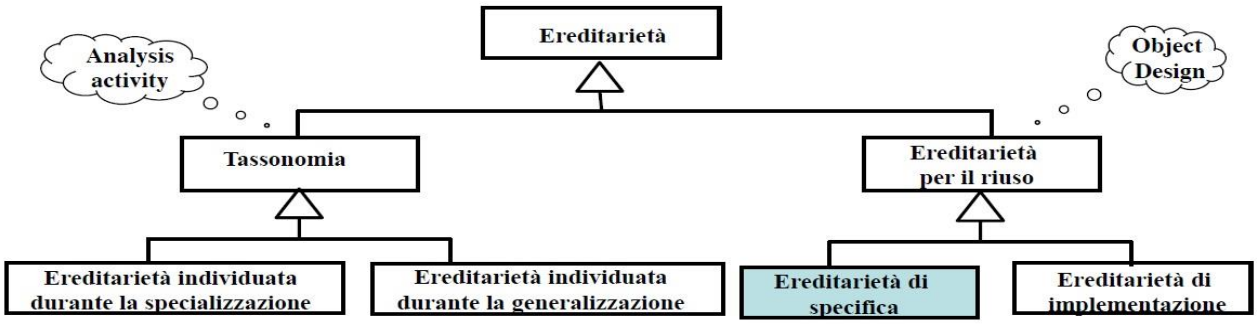
1. Andare nella directory che contiene i file sorgente che si desidera documentare;
2. Eseguire il comando **Javadoc –d docDirectory nomePackage** per un solo package, oppure **Javadoc –d docDirectory nomePackage1 nomePackage2 …** per documentare più package
3. Se i file si trovano nel file predefinito eseguire invece **javadoc –d docDirectory *.java**
4. Se si vuole estrarre nella directory corrente i file HTML basta omettere l'opzione **–d direcotory**.

DESIGN PATTERN

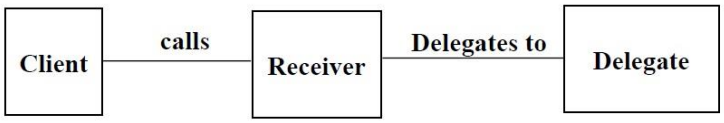
Gli obiettivi dell'attività di riuso sono utilizzare funzionalità già disponibili per realizzare nuove funzionalità e utilizzare la conoscenza acquisita durante precedenti esperienze di progettazione per risolvere un problema corrente, gli strumenti impiegati in tale attività sono:

- **EREDITARIETÀ**, dove le funzionalità sono ottenute per ereditarietà, detta anche **riuso White-Box** perché la struttura interna delle classi antenate è spesso visibile alle sottoclassi, durante l'analisi si utilizza l'ereditarietà per organizzare gli oggetti in una **gerarchia comprensibile (descrizione di tassonomie)** cioè partendo da oggetti astratti i lettori del modello di analisi comprendono le funzionalità chiave del sistema, durante l'object design invece l'utilizzo dell'ereditarietà permette di **ridurre ridondanze**, **migliorare l'estendibilità**, **la modificabilità** e **il riuso del codice**, cioè i comportamenti ridondanti sono fattorizzati in una singola superclasse, riducendo il rischio di introdurre inconsistenze in seguito a futuri cambiamenti, l'ereditarietà è uno strumento molto potente e bisogna prestare attenzione nell'utilizzarla poiché si rischia di ottenere codice più fragile e scorretto utilizzandola dove non occorre.
  L'ereditarietà per il riuso si divide in:

- **EREDITARIETÀ DI SPECIFICA (o d'interfaccia o subtyping)**, che 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.
  Relativo a questa tipologia di ereditarietà, fornendone una definizione formale, è il **principio di Liskov** che dice:
  "Se un oggetto di tipo S può essere sostituito ovunque ci si aspetta con un oggetto di tipo T, allora S è un sottotipo di T"
  Tale principio afferma che un metodo scritto in termini di una superclasse T deve essere in grado di usare istanze di qualunque sottoclasse di T senza sapere se si utilizza la superclasse o una sua sottoclasse, inoltre gli oggetti appartenenti a una sottoclasse devono essere in grado di esibire tutti i comportamenti e le proprietà appartenenti alla superclasse, con la possibilità di averne ulteriori proprie solo ad essa, in modo da essere sostituiti senza intralciare la funzionalità del programma.
  Una relazione di ereditarietà che soddisfa tale principio è chiamata **Ereditarietà stretta**;
- **EREDITARIETÀ D'IMPLEMENTAZIONE (o di classe)**, che definisce l'implementazione di un oggetto in funzione dell'implementazione di un altro oggetto, si eredita da una classe esistente che ossiede tutte le operazioni già implementate.
  I problemi che si hanno con tale tipologia di ereditarietà sono che le operazioni della superclasse sono esposte all'utilizzatore della sottoclasse, alcune operazioni ereditate possono essere utilizzate in modo inaspettato.

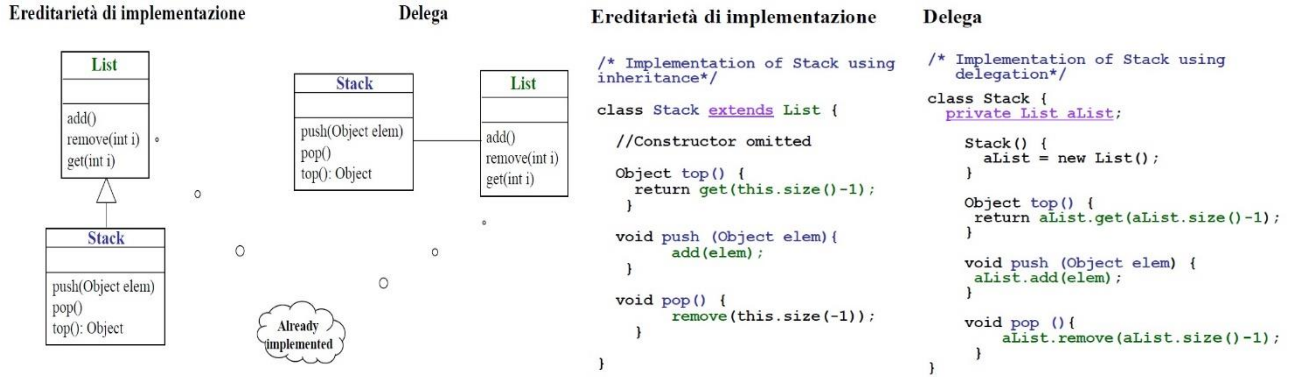


- **COMPOSIZIONE**, dove le nuove funzionalità sono ottenute per aggregazione, detta anche **riuso Black-Box** perché i dettagli implementativi interni agli oggetti non sono visibili all'esterno.
  Un esempio di composizione è la **delega**, essa rappresenta un'alternativa efficace all'ereditarietà d'implementazione, ad esempio due oggetti collaborano per gestire una



  richiesta, un **Client** richiede l'esecuzione di un'operazione all'oggetto **Receiver**, che delega ad un altro oggetto **Delegate** l'esecuzione dell'operazione, in tal modo l'oggetto Client non può utilizzare in modo scorretto l'oggetto Delegate.

Un altro esempio di delega è il seguente:
Usando la delega la classe Stack non include nella sua interfaccia i metodi di List ed il nuovo campo aList è privato, l'utilizzatore della classe Stak non può utilizzare i metodi di List e si ha la possibilità di cambiare la rappresentazione interna di Stack, utilizzando ad esempio la classe Vector invece di List, senza intaccare i client.



La delega non interferisce con le componenti esistenti e porta allo sviluppo di codice più robusto, l'ereditarietà invece è preferibile alla delega in situazioni di subtyping perché porta a design maggiormente estendibili.
I due approcci hanno entrambi i propri pro e contro:
DELEGA:

- **PRO**, l'incapsulamento non è violato, poiché si accede agli oggetti solo attraverso la loro interfaccia, e la flessibilità, poiché 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 che a software statico.

EREDITARIETÀ:

- **PRO**, definita staticamente al momento della compilazione e immediata da utilizzare, poiché è supportata direttamente dal linguaggio di programmazione;
- **CONTRO**, è impossibile cambiare l'implementazione ereditata durante l'esecuzione, e l'implementazione di una sottoclasse diventa strettamente dipendente dalla classe padre, infatti qualsiasi modifica nell'implementazione della classe padre include modifiche nella sottoclasse (entrambe devono essere ricompilate).

- **DESIGN PATTERN**, che sono template di soluzioni, a problemi ricorrenti, impiegati per ottenere riuso e flessibilità.
  Oltre al riuso nella scrittura del software esiste anche il riuso nella progettazione che si fonda su due principi:

  "**Catturare l'esperienza e la saggezza degli esperti al fine di evitare di reinventare ogni volta le stesse cose**"

  L'esperienza è un fatto fondamentale per una buona progettazione, un progettista esperto non parte mai da zero ma riutilizza soluzione che si sono dimostrate valide in passato, la "comunità dei pattern" si prefigge come obiettivo la catalogazione di questi schemi ricorrenti in modo da costruire un dizionario reso fruibile ai progettisti.

"**La modellazione rigorosa del mondo reale conduce a un sistema che riflette le realtà odierne ma non necessariamente quelle di domani**"

Dunque si vuole progettare prodotti software facilmente modificabili ed estendibili per minimizzare il costo di modifiche future, le fonti che inducono alla modifica di un software possono essere molteplici e spesso comuni alla maggior parte dei software, cioè nuovo produttore o nuova tecnologia, nuova implementazione, nuove viste, nuova complessità di dominio ed errori.

Una soluzione è prevedere i cambiamenti e tenerne conto nella progettazione del sistema in quanto tali cambiamenti tendono ad essere simili per molti sistemi.

Il **concetto di pattern** deriva da **Christopher Alexander** esperto nel campo dell'architettura,

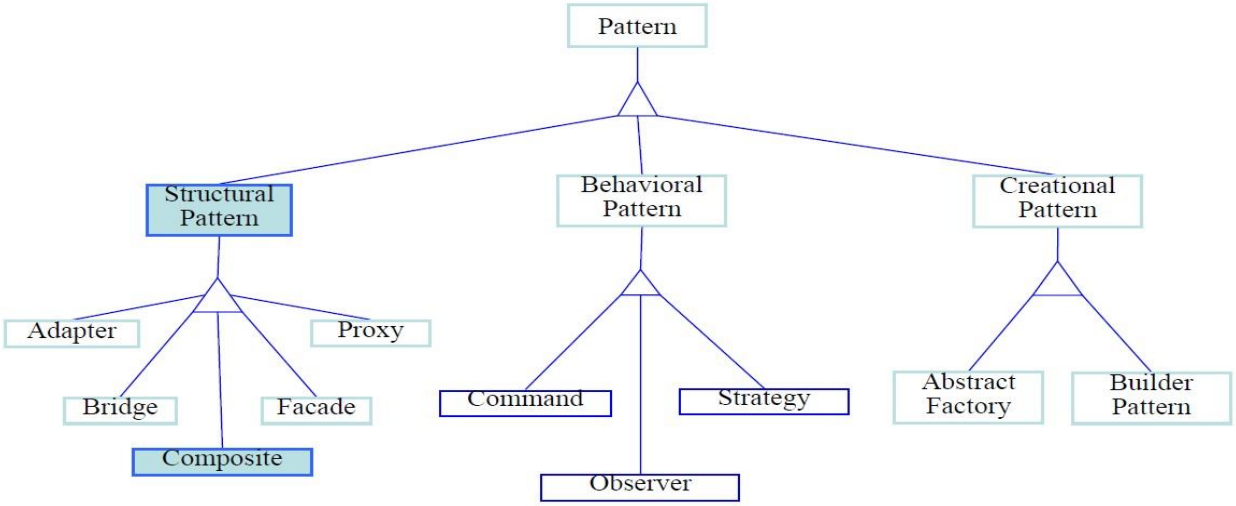
"**Un pattern descrive un problema che ricorre più volte nel nostro ambiente, descrive poi il nucleo della soluzione del problema in modo da poter utilizzare tale soluzione un milione di volte senza mai farlo allo stesso modo**"

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, solitamente sono composti da poche classi, collegate tra loro tramite delegazione ed ereditarietà per fornire una soluzione robusta e modificabile, che possono essere adattate o ridefinite per lo specifico sistema che si vuole realizzare.

Un design pattern è definito mediante 4 componenti principali:

- **NOME**, un nome simbolico che lo identifica;
- **DESCRIZIONE DEL PROBLEMA**, che descrive le situazioni in cui il pattern può essere utilizzato
- **SOLUZIONE**, che descrive gli elementi che costituiscono il progetto, le loro relazioni, responsabilità e collaborazioni;
- **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, come quello **GOF (Gang of Four)** le cui componenti sono **nome e classificazione**, **scopo**, che indica cosa fa il pattern e il suo funzionamento logico, **nomi alternativi**, **motivazione**, che mostra un problema e la soluzione offerta, **applicabilità**, che indica quando può essere applicato il pattern, **struttura**, rappresentazione UML del pattern, **partecipanti**, le classi/oggetti con le proprie responsabilità, **collaborazioni**, come collaborano i partecipanti, **conseguenze**, **implementazione**, **codice di esempio**, **pattern correlati**, relazioni con altri pattern, **utilizzi noti**, esempi consistenti.

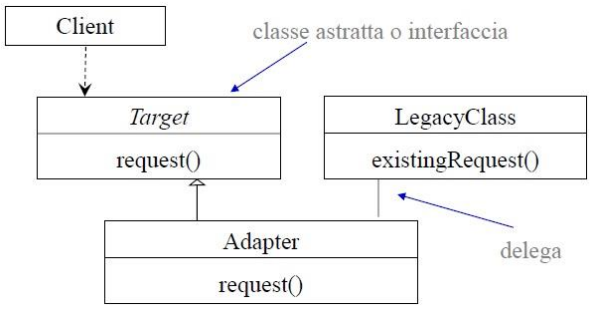


I design pattern vengono classificati in 3 categorie:

- **PATTERN STRUTTURALI (STRUCTURAL PATTERNS)**, che si occupano delle modalità di composizione di classi e oggetti per formare strutture complesse, riducono l'accoppiamento tra due o più classi, incapsulano strutture complesse e introducono una classe astratta per permettere estensioni future.

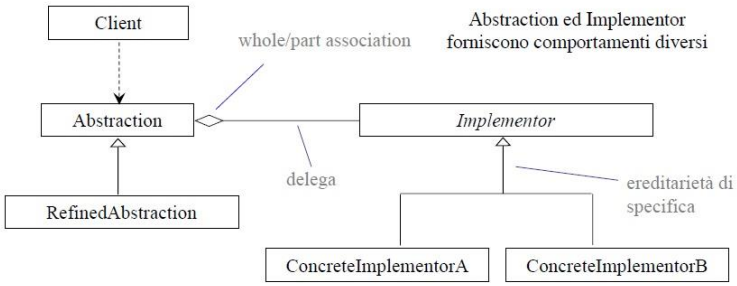
  In questa categoria troviamo i pattern:

  1. **ADAPTER DESIGN PATTERN**, usato quando la complessità del sistema aumenta e il tempo per rilasciare il prodotto diminuisce, il costo dello sviluppo eccede di molto i costi dell'hardware, cosi gli sviluppatori sono incentivati a riutilizzare componenti da progetti precedenti o ad usare componenti off-the-shelf (COTS), che vengono incapsulate in modo da avere una separazione tra il sistema e le componenti minimizzando l'impatto dei nuovi software sul nuovo progetto.
     Tale pattern converte l'interfaccia di una classe in un'interfaccia diversa che il cliente si aspetta, in maniera che classi diverse possano operare insieme nonostante abbiano interfacce incompatibili, ed è applicabile per incapsulare componenti esistenti per riutilizzare componenti precedenti o componenti off-the-shelf (COTS), fornire una nuova interfaccia a componenti legacy esistenti (interface engineering, reengineering).

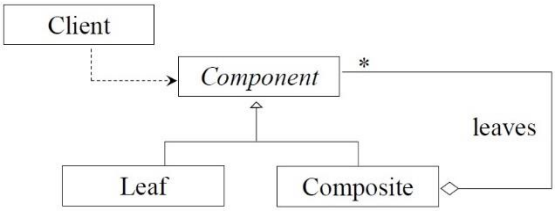


     Le conseguenze che si hanno sono che se il Client utilizza Target allora può utilizzare una qualsiasi istanza dell'adapter in modo trasparente senza necessitare modifche, Adapter lavora con la classe LegacyClass e le sue sottoclassi.
     Tale pattern viene utilizzato quando un'interfaccia e la sua implementazione esistono già e non possono essere modificate.

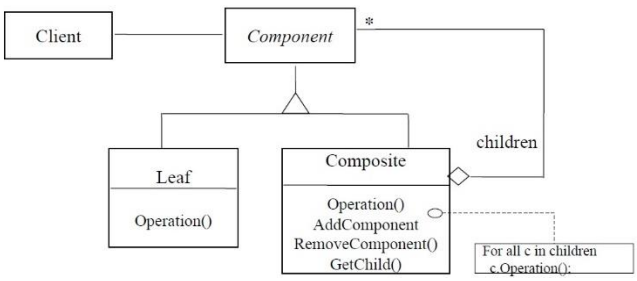
2. **BRIDGE DESIGN PATTERN**, utilizzato quando si ha il problema di sviluppare, testare ed integrare sottosistemi realizzati da differenti sviluppatori in maniera incrementale, per evitare che questa integrazione venga ritardata fino a che tutti i sottosistemi sono stati realizzati si utilizzano implementazioni di stub e si sviluppano diverse implementazioni dello stesso sottosistema, tale pattern consente di sostituire dinamicamente realizzazioni diverse della stessa interfaccia per usi differenti.
   Il problema è separare un'astrazione da un'implementazione cosi che una diversa implementazione possa essere sostituita eventualmente a runtime, tale pattern si rende utile per interfacciare un insieme di oggetti quando non è ancora completamente noto e quando c'è necessità di estendere un sottosistema dopo che il sistema è stato consegnato ed è in esecuzione.



   La soluzione consiste in una classe Abstraction che definisce l'interfaccia visibile al codice Client, Implementator è un'interfaccia astratta che definisce i metodi di basso livello disponibili ad Abtraction, un'istanza di Abstraction mantiene un riferimento alla corrispondente istanza di Implementator ed e entrambe possono essere raffinate indipendentemente;
3. **COMPOSITE DESIGN PATTERN**, tale pattern, ampiamente utilizzato in applicazioni grafiche, compone gli oggetti in strutture ad albero di ampiezza e profondità variabile per rappresentare gerarchie "tutto-parte", permette al client di trattare gli oggetti individuali (foglie) e gli oggetti composti (nodi interni) in maniera uniforme attraverso un'interfaccia comune.

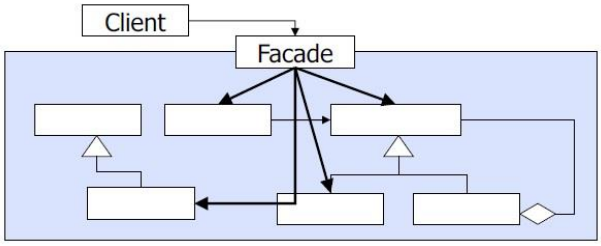


   La soluzione consiste che l'interfaccia Component specifica i servizi che sono condivisi tra Leaf e Composite che ha un'associazione di aggregazione con Component e implementa ogni servizio iterando su ogni Component che contiene, le conseguenze sono che il Client usa sempre lo stesso codice per

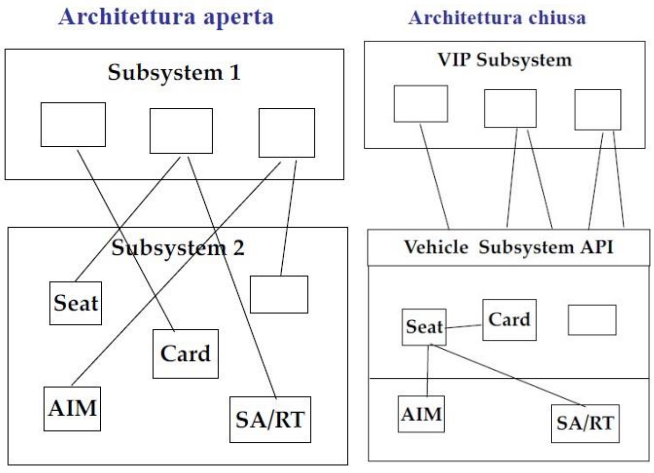


gestire i Leaf o i Composite, i comportamenti specifici di un Leaf possono essere cambiati e possono essere aggiunte nuove classi di Leaf senza cambiare la gerarchia.

4. **FACADE DESIGN PATTERN**, fornisce un'unica interfaccia per accedere ad un insieme di oggetti che compongono un sottosistema, tale pattern viene utilizzato quando tutti i sottosistemi di un software hanno bisogno di utilizzare un modello di facciata che definisce tutti i servizi del sottosistema e consente di fornire un'architettura chiusa.



Le architetture aperte, avendo due sottosistemi comunicanti, permettono ad un sottosistema1 di accedere ad un sottosistema2 chiamando a piacere qualsiasi operazione di una componente o classe, tale architetture hanno il vantaggio di avere un'efficienza elevata ma di contro hanno che non si può essere certi che il chiamante comprenda il funzionamento del sottosistema o le sue relazioni interne e ciò porta ad un uso improprio del sottosistema causando codice non portabile.
Nelle architetture chiuse invece il sottosistema decide esattamente come si accede, non è necessario preoccuparsi di utilizzi impropri da parte dei chiamanti e se viene utilizzata un facade il sottosistema può essere utilizzato in un test d'integrazione iniziale, e si ha il bisogno di descrivere un solo driver, l'interfaccia stessa.



L'utilizzo di tale pattern ha come conseguenze quello di ottenere un'interfaccia unificata per tutte le componenti del sottosistema, ciò porta a rendere il sottosistema può facile da usare, proteggere i client dalle componenti del sottosistema e promuove l'accoppiamento debole tra client e componenti del sottosistema;

5. **PROXY DESIGN PATTERN**, utilizzato quando occorre rinviare l'inizializzazione di un oggetto solo quando questo si rende realmente necessario, tale

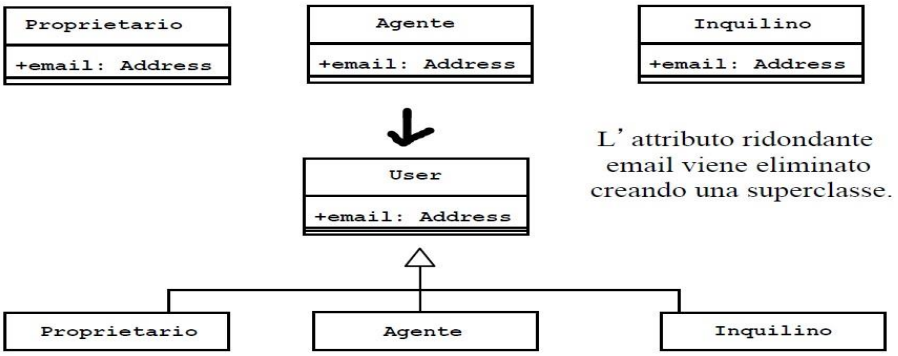
pattern comporta in termini di costi la creazione di oggetti e la loro inizializzazione, ma riduce il costo degli accesi agli oggetti, utilizza un oggetto ulteriore, il Proxy, per dare supporto all'oggetto reale e crearlo solo se l'utente lo richiede.
I pattern proxy possono essere utilizzati per la valutazione lazy e invocazione remota e possono essere implementati con interfacce java, vi sono 3 tipologie di pattern proxy, **REMOTE PROY**, utilizzato quando si ha un rappresentante locale per un oggetto situato in diverso spazio di memoria, la memorizzazione avviene nella cache e perciò è indicato se le informazioni non variano di frequente, **VIRTUAL PROXY**, utilizzato quando l'oggetto è troppo costoso da creare o scaricare e in tal caso il proxy fa da supporto, **PROTECTION PROXY**, utilizzato quando oggetti diversi devono avere diversi diritti di accesso e visualizzazione per lo stesso documento, in tal caso il proxy fornisce il controllo degli accessi all'oggetto reale;

- **PATTERN COMPORTAMENTALI (BEHAVIORAL PATTERNS)**, si occupano di algoritmi e dell'assegnazione delle responsabilità tra oggetti che collaborano tra loro e caratterizzano i flussi di controllo complessi difficili da eseguire a runtime.
  In questa categoria troviamo i pattern:
  - **COMMAND DESIGN PATTERN**, permette di isolare la porzione di codice che effettua un'azione, dal codice che ne richiede l'esecuzione, incapsulandola in una classe Command con l'obiettivo di rendere variabile l'azione del client senza però conoscere i dettagli dell'operazione stessa
  - **OBSERVER DESIGN PATTERN**, chiamato anche "Publish and Subscribe", definisce una dipendenza uno a molti tra gli oggetti in modo che quando un oggetto cambia di stato tutti i suoi dipendenti vengono notificati e aggiornati automaticamente, viene utilizzato per garantire il mantenimento della coerenza
  - **STRATEGY DESIGN PATTERN**; che consente di selezionare un algoritmo a runtime da una famiglia di algoritmi invece di implementare il singolo algoritmo, ciò consente all'algoritmo di variare indipendentemente dal client che lo utilizza.
- **PATTERN CREAZIONALI (CREATIONAL PATTERNS)**, che 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.
  In tale categoria troviamo i pattern:
  - **ABSTRACT FACTORY PATTERN**, che consente di fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le loro classi concrete, le conseguenze dell'applicazione di questo pattern sono l'isolamento delle classi concrete, permette di sostituire facilmente famiglie di prodotti a runtime, promuove la coerenza nell'utilizzo dei prodotti e aggiungere nuove tipologie di prodotti è difficile.
  - **BUILDER DESIGN PATTERN.**

MAPPING MODELS TO CODE

Argomenti di questa lezione **ristrutturazione del modello ad oggetti e ottimizzazione del modello ad oggetti**.
Una volta specificate le interfacce delle classi e raffinata la relazione tra le classi è possibile implementare il sistema che realizza i casi d'uso specificati durante l'analisi dei requisiti e il system design, gli sviluppatori però possono incontrare diversi problemi d'integrazione, come parametri non documentati che possono essere stati aggiunti alle API in seguito a modifiche nei requisiti o attributi addizionali possono essere stati aggiunti all'object model ma non nello schema di memorizzazione persistente (comunicazione inefficiente), con la conseguenza di realizzare codice lontano dal progetto originario e difficile da comprendere.
Gli sviluppatori sesso eseguono **trasformazioni** sul modello ad oggetti, come trasformare localmente il modello degli oggetti per migliorarne la modularità e le prestazioni, trasformare le associazioni del modello ad oggetti in collezioni di riferimenti ad oggetti in quanto i linguaggi di programmazione non supportano il concetto di associazione, scrivere codice per identificare e gestire violazioni dei contratti se il linguaggio di programmazione non supporta i contratti e rivedere la specifica dell'interfaccia per soddisfare nuovi requisiti richiesti dal client, tutte queste attività non sono particolarmente complesse ma presentano aspetti ripetitivi e meccanici che possono indurre lo sviluppatore ad introdurre errori.
Utilizzare un **approccio disciplinato** può 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 e mappare il modello delle classi in uno schema di memorizzazione persistente.
Una **trasformazione** ha lo scopo di migliorare un aspetto del modello e preservare allo stesso tempo tutte le altre proprietà generalmente è **localizzata**, **impatta su un numero ristretto di classi**, **attributi e operazioni** e **viene eseguita in una serie di semplici passi**, vi sono sostanzialmente 4 tipi di trasformazioni:

- **TRASFORMAZIONI DI MODELLO**, che operano sul modello a oggetti, quest'attività riceve come input un modello ad oggetti e da come output un altro modello ad oggetti, ha come scopo quello di semplificare o ottimizzare il modello originale attuando trasformazioni relative all'aggiungere, eliminare o rinominare classi, associazioni o attributi, aggiungendo o eliminando dunque informazioni dal modello.
  Una trasformazione di tale tipo è un'**attività meccanica** ma decidere quale trasformazione applicare e su quali classi farlo richiede esperienza;



- **REFACTORING**, che operano sul codice sorgente migliorandone la **leggibilità** e la **modificabilità** senza modificare però il comportamento del sistema, si focalizza su uno specifico campo o metodo di una classe e per preservare il comportamento del sistema

viene attuato a piccoli passi incrementali intervallati da test che incentivano lo sviluppatore a modificare il codice migliorandolo;

La trasformazione del modello di oggetti vista prima corrisponde ad una sequenza di tre refactoring:

- **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.

Prima del refactoring

```
public class User {
      private String email;
}
public class Proprietario extends User{
   public Proprietario(String email) {
                this.email=email;
                 //…
      }
}
public class Agente extends User{
   public Agente(String email) {
                    this.email=email;
                    //…
      }
}
public class Inquilino extends User{
   public Inquilino(String email) {
                    this.email=email;
                    //…
      }
}
```

Dopo il refactoring

```
public class User {
   private String email;
   public User (String email) {
   this.email=email
   }
}
public class Proprietario extendsUser{
   public Proprietario(String email){
            super(email);
             //…
      }
}
public class Agente extends User{
      public Agente(String email)
   {super(email);//…}
}
public class Inquilino extends User{
      public Inquilino(String email) {
          super(email);
        //…
       }
}
```

- **FORWARD ENGINEERING**, che produce un template del codice sorgente corrispondente al modello ad oggetti, tale attività riceve in input un insieme di elementi del modello e da come output un insieme di istruzioni di codice sorgente ed ha come obiettivi quello di mantenere una forte corrispondenza fra il modello di design ed il codice, ridurre il numero di errori introdotti durante l'implementazione e ridurre gli sforzi di implementazione.
  Alcune importanti considerazioni sono che ogni classe del diagramma UML è mappata in una classe JAVA, la relazione di generalizzazione UML è mappata in una istruzione extends dalla 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, il codice risultate da una trasformazione di questo tipo è sempre lo stesso ma se le classi sono progettate in modo adeguato si introducono meno errori;

Codice sorgente ottenuto con forward engineering

```
public class User {
   private String email;
   public String getEmail() {
         return email;
   }
   public void setEmail(String value){
         email=value;
   }
   public void notify(String msg) {
   //…
   }
//…
}
```

```
public class Proprietario extends User{
    private String CF;
    public String getCF() {
         return CF;
    }
    public void setCF (String value) {
         CF=value;
    }
    //…..
}
```

- **REVERSE ENGINEERING**, che produce un modello a oggetti che corrisponde al codice sorgente, tale attività riceve come input un insieme di elementi di codice sorgente e da come output un insieme di elementi del modello con l'obiettivo di ricreare il modello per un sistema esistente, tale attività è motivata dal fatto che il modello è andato smarrito o non è mai stato realizzato e non è più allineato con il codice sorgente.
  Tale trasformazione è l'inverso del Forward Engineering, cioè si crea una classe UML per ciascuna classe, si aggiunge un attributo per ciascun campo e si aggiunge un'operazione per ciascun metodo, non ricrea necessariamente il modello originario ed è supportata da CASE

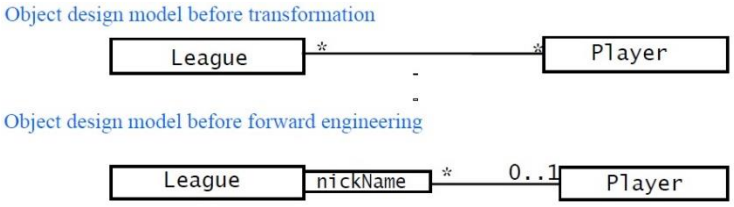
tool ma richiede comunque l'intervento dello sviluppatore per ricostruire un modello il più possibile vicino a quello originario.

Per evitare che le trasformazioni possano indurre errori difficili da rilevare e da correggere è necessario che ogni trasformazione segua i **principi di trasformazione**, cioè:

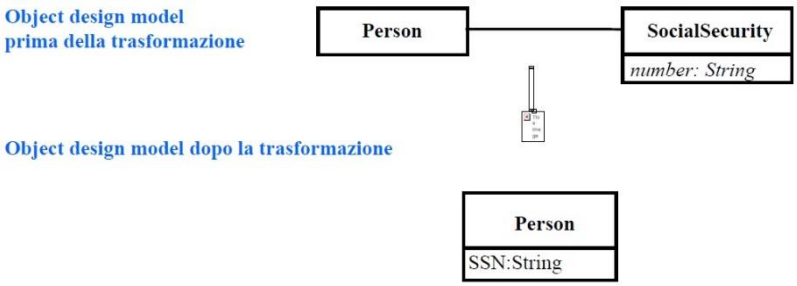
- **Migliori il sistema rispetto ad un singolo obiettivo di design;**
- **Cambi pochi metodi e poche classi alla volta;**
- **Venga applicata in modo isolato rispetto agli altri cambiamenti (uno per volta);**
- **Venga seguita da un passo di validazione che prima di poter passare alla trasformazione successiva valida i cambiamenti di quella attuale.**

Le **attività che coinvolgono trasformazioni** sono:

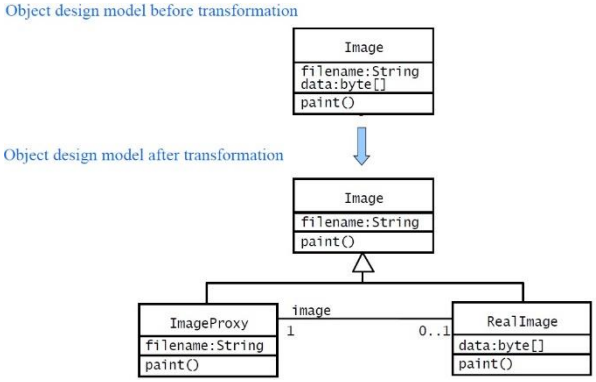
- **OTTIMIZZAZIONE DEL MODELLO DELLE CLASSE**, che viene intrapresa al fine di soddisfare i requisiti di prestazione del modello del sistema, tale attività include la riduzione della molteplicità delle associazioni per velocizzare le richieste, l'aggiunta di associazioni ridondanti per migliorare l'efficienza e l'aggiunta di attributi derivati per migliorare i tempi di accesso agli oggetti, e si divide in attività secondarie, quali:
  - **Ottimizzazione dei cammini di accesso**, e per fare ciò bisogna eliminare il ritardo ottenuto a causa **dell'attraversamento ripetuto di associazioni multiple**, poiché le operazioni più frequenti non dovrebbero richiedere molti attraversamenti di associazioni ma dovrebbero avere un'associazione diretta fra l'oggetto che interroga e l'oggetto interrogato, la soluzione consiste nell'aggiungere l'associazione fra i due oggetti, **dell'attraversamento di associazioni di tipo "molti"**, che si risolve provando a far decrescere il tempo di ricerca riducendo il tipo "molti" al tipo "uno" utilizzando un'associazione qualificata, ma se ciò non è possibile si procede ad ordinare o indicizzare gli oggetti sul lato "molti" per migliorare il tempo di accesso,



    e **della presenza di attributi mal collassati**, questo problema si ha poiché si ha la tendenza durante l'analisi di apportare una modellazione eccessiva individuando delle classi non rilevanti i cui attributi sono chiamati solo dai metodi set e get, per risolverlo si possono spostare gli attributi nella classe chiamante cosi da poterle rimuovere dal modello;
  - **Collassare gli oggetti in attributi**, dopo che il modello è stato ottimizzato diverse volte alcune classi possono avere pochi attributi o comportamenti, se tali classi sono associate ad una sola altra classe allora possono collassare in attributi;



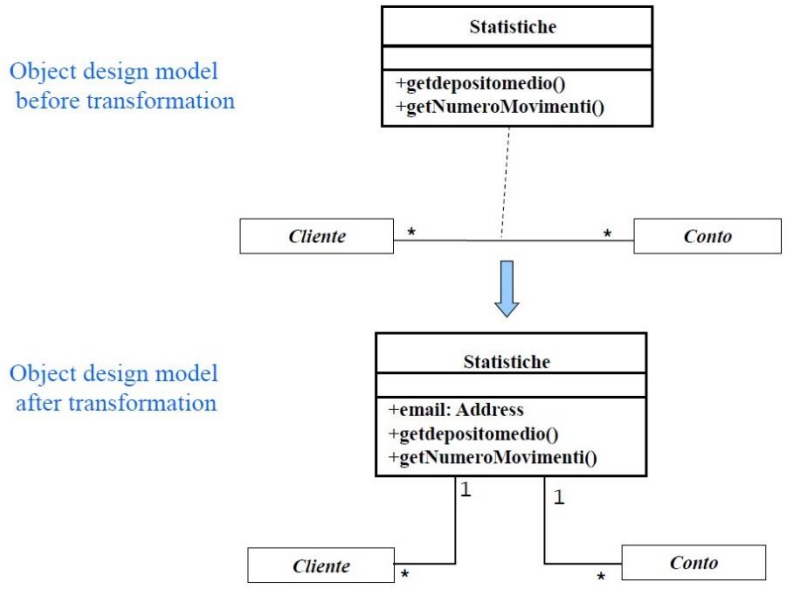
- **Ritardare le elaborazioni costose**, alcuni oggetti sono costosi da creare e dunque la loro creazione può essere ritardata finché il loro contenuto è effettivamente necessario utilizzando il design pattern Proxy;



- **Mantenere in una struttura dati temporanea il risultato di elaborazioni costose**, alcuni metodi sono invocati diverse volte ma il loro risultato non cambia o cambia raramente, ridurre il tempo di elaborazione di tali metodi può portare una riduzione dei tempi di risposta memorizzando il risultato in un attributo privato a discapito della memoria;

- **MAPPING DELLE ASSOCIAZIONI IN COLLEZIONI DI OGGETTI**, attività che si rende necessaria per mappare associazioni in costrutti di codice sorgente, le associazioni sono concetti UML che denotano collezioni di link bidirezionali tra due o più oggetti, i linguaggi di programmazione object oriented non forniscono il concetto di associazione ma forniscono i concetti di **riferimento**, quando un oggetto memorizza un "hendle" verso un altro oggetto (unidirezionali, link 2 oggetti), e di **collezione**, quando possono essere memorizzati i riferimenti a più oggetti, durante l'object design trasformiamo le associazioni in riferimenti considerando la molteplicità e la direzione delle associazioni:
  - **Associazioni uno a uno unidirezionali**, quando una Classe1 chiama le operazioni di una Classe2 ma la Classe2 non chiama mai operazioni della Classe1, ciò si traduce inserendo un attributo classe2 nella Classe1 che referenzia l'oggetto Classe2, **e bidirezionali**, quando una Classe1 chiama le operazioni di una Classe2 e la Classe2 chiama operazioni della Classe1, ciò si traduce inserendo un attributo classe2 nella Classe1 che referenzia l'oggetto Classe2 e inserendo un attributo classe1 nella Classe2 che referenzia l'oggetto Classe1, tali associazioni sono complesse e introducono dipendenze reciproche tra classi;
  - **Associazioni uno a molti**, che non possono essere realizzate utilizzando un singolo riferimento, supponendo che alla Classe1 corrispondano più istanze della Classe2 ciò si traduce inserendo in Classe1 un attributo che rappresenta l'insieme delle istanze di Classe2 (lista, array, etc) e in Classe2 un attributo classe1 che referenzia la Classe1, **e molti a molti**, realizzate inserendo in ambedue le classi coinvolte un insieme di riferimenti che referenziano le classi, cioè in Classe1 un attributo che classe2 che rappresenta l'insieme delle istanze di Classe2 (lista, array, etc) e in Classe2 un attributo che classe1 che rappresenta l'insieme delle istanze di Classe1 (lista, array, etc);;
  - **Classi associative**, che sono utilizzate in UML per contenere gli attributi di un'associazione, per realizzare ciò dapprima trasformiamo la classe associativa in

un oggetto che ha più associazioni binarie poi convertiamo le associazioni binarie in un insieme di attributi referenziati;



- **MAPPING DEI CONTRATTI DELLE OPERAZIONI IN ECCEZIONI**, attività che si rende necessaria per descrivere il comportamento delle operazioni quando i contratti sono violati, un contratto è un vincolo su una classe che deve essere rispettato prima di utilizzare la classe, molti linguaggi object oriented non supportano i contratti e si utilizza il meccanismo delle eccezioni per gestire le violazioni dei contratti.
  Tale attività si compone di ulteriori attività:
  - **Eccezioni in Java utilizzando il meccanismo Try-Throw-Catch**, in Java si solleva un'eccezione con la parola chiave **throw** seguita da un oggetto eccezione che fornisce un posto dove memorizzare informazioni sull'eccezione che usualmente sono semplici messaggi di errore, lanciando un'eccezione si ottiene un duplice effetto, dapprima s'interrompe il flusso di controllo e poi si prosegue a svuotare lo stack delle chiamate finché non si trova un blocco catch che gestisce l'eccezione;
  - **Implementazione di un contratto**, per ogni operazione nel contratto bisogna **controllare le precondizioni**, attivando un test che lancia un'eccezione, prima dell'inizio di ogni metodo, se una precondizione non è verificata, **controllare le postcondizioni**, lanciando un'eccezione, alla fine di ogni metodo, se il contratto è stato violato, **controllare le invarianti**, nello stesso modo delle postcondizioni **e gestire l'ereditarietà**, incapsulando il codice di controllo per precondizioni e postcondizioni in metodi separati che possono essere richiamati dalle sottoclassi.

  Le euristiche che è bene seguire per compiere questa attività dicono che si può omettere il codice di controllo per postcondizioni e invarianti se il codice è scritto da un solo programmatore, per metodi privati e protetti se è ben definita l'interfaccia, bisogna concentrarsi sulle componenti che hanno una unga durata e riutilizzare il codice per il controllo dei vincoli.
- **MAPPING DEL MODELLO DELLE CLASSI IN UNO SCHEMA DI MEMORIZZAZIONE PERSISTENTE**, per realizzare la strategia di memorizzazione persistente prescelta durante il system design alcune classi devono essere mappate in uno schema di memorizzazione ad esempio utilizzando flat file o lo schema del DBMS selezionato, i linguaggi di

programmazione object oriented non forniscono un modo efficiente per memorizzare gli oggetti persistenti quindi si rende necessario mappare tali oggetti in strutture dati che possono essere memorizzate in file o database, se si utilizzano database object oriented non sono necessarie trasformazioni degli oggetti, se invece si utilizzano databse relazionali o file è necessario mappare il modello degli oggetti in uno schema di memorizzazione e fornire un'infrastruttura per convertire gli oggetti in schemi di memorizzazione persistente e viceversa.
Tale attività si divide in attività secondarie quali:

- **Database relazionali**, uno schema è una descrizione dei dati (meta-modello), i db relazionali memorizzano sia lo schema che i dati, questi ultimi vengono memorizzati sotto forma di **tabelle** strutturate in colonne che rappresentano gli **attributi** dove i più importanti fra di essi sono: **chiave primaria**, che identifica univocamente una riga della tabella che rappresenta in java un'istanza dell'oggetto che la tabella rappresenta, e **chiave esterna**, che referenzia la chiave primaria di un'altra tabella.
- **Mappare classi ed attributi**, dove mappiamo ogni classe in una tabella del database, usando lo stesso nome dell'oggetto per garantire la tracciabilità fra le due rappresentazioni, e per ogni attributo aggiungiamo una colonna nella tabella con il nome dell'attributo della classe definendone anche il tipo, ogni riga della tabella rappresenta un'istanza della classe, infine si seleziona la chiave primaria o identificando un insieme di attributi della classe che identifica univocamente l'oggetto oppure aggiungendo un identificatore univoco;
- **Mappare le associazioni**, le associazioni uno a uno e uno a molti sono implementate usando una chiave esterna (**buried association**), in associazioni uno a uno la chiave esterna viene posta in una delle due tabelle, in associazioni uno a molti la chiave esterna viene inserita nella tabella del lato molti.
  Le associazioni molti a molti sono implementate usando una tabella separata, chiamata **tabella associativa**, costituita di due colonne che contengono la chiave esterna di ciascuna classe coinvolta nell'associazione, ogni riga di tale tabella corrisponde ad un collegamento tra due istanze dell'associazione molti a molti.
  Anche le associazioni uno a uno e uno a molti possono essere rappresentate per mezzo di tabelle associative, ciò rende lo schema facilmente modificabile e accresce il numero delle tabelle ed il tempo per attraversare l'associazione, tale scelta va fatta considerando se il tempo di risposta è un fattore critico per il sistema e quando probabilmente l'associazione cambi.
- **Mappare le relazioni di ereditarietà**, i database relazionali non supportano l'ereditarietà e per poterla mappare si utilizzano due meccanismi, **mapping verticale**, simile al mapping di associazioni uno a uno, cioè ogni classe è rappresentata da una tabella e utilizza un chiave esterna per collegare la tabella corrispondente ad una sottoclasse con quella corrispondente alla superclasse, tale tecnica risulta modificabile ma meno efficace, **e mapping orizzontale**, dove gli attributi sella superclasse sono ricopiati in tutte le sottoclassi e la superclasse viene eliminata, tale tecnica risulta più modificabile ma meno efficace.

Le euristiche per effettuare trasformazioni sono:

- **Utilizzare sempre il medesimo strumento per effettuare le trasformazioni**

- **Mantenere traccia dei contratti nel codice sorgente e non solo nel modello di object design**
- **Utilizzare gli stessi nomi per gli stessi oggetti**
- **Utilizzare delle linee guida per le trasformazioni**

Le trasformazioni devono essere documentate per poter essere riapplicate in caso di cambiamenti nel modello di object design o nel codice sorgente, diversi ruoli cooperano per selezionare, documentare e applicare le trasformazioni:

- **Il "core architect"**, seleziona le trasformazioni che devono essere applicate sistematicamente;
- **L' "architecture liaison"**, responsabile della documentazione dei contratti associati alle interfacce dei sottosistemi;
- **Lo "sviluppatore"**, responsabile di eseguire l'insieme delle convenzioni stabilite dal "core architect" ed di applicare le trasformazioni e converte il modello di object in codice sorgente.

M5 – TESTING

TESTING

Il testing consiste nel trovare le differenze tra il comportamento atteso specificato attraverso il modello del sistema e il comportamento osservato dal sistema implementato, vi sono 4 tipologie di testing:

- **UNIT TESTING**, che consiste nel trovare le differenze tra object design model e la componente corrispondente;
- **STRUCTURAL TESTING**, che consiste nel trovare le differenze tra system design e un sottoinsieme integrato di sottoinsiemi;
- **FUNCTIONAL TESTING**, che consiste nel trovare le differenze tra use case model e il sistema;
- **PERFORMANCE TESTING**, che consiste nel trovare le differenze tra requisiti non funzionali e le performance del sistema reale;

Il testing dovrebbe essere realizzato da sviluppatori che non sono stati coinvolti nelle attività precedenti, analisi, design e implementazione, dette di costruzione per il sistema, poiché è un'attività che va in contrasto con quelle costruttive perché tenta di rompere il sistema.
Tale attività ha dunque come obiettivo quello di progettare test per provare il sistema e rivelarne problemi, con lo scopo di massimizzare il numero di errori scoperti per consentire agli sviluppatori di correggerli.

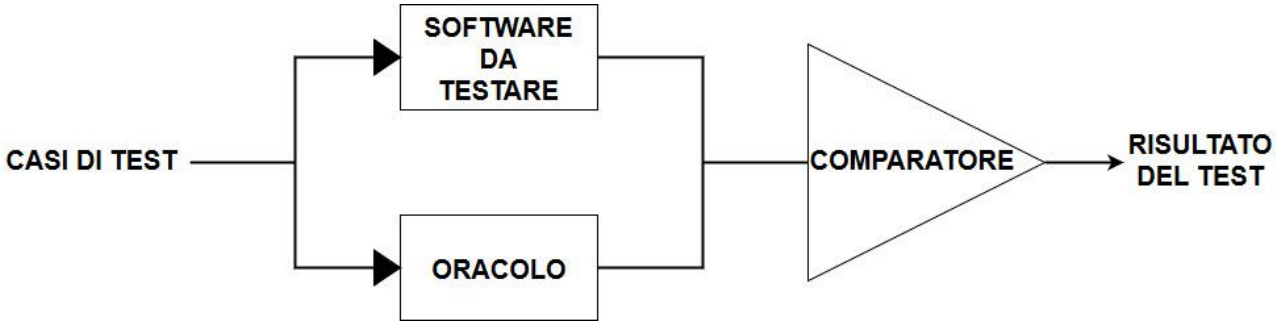
"Program testing can be used to show the presence of bugs, but never to show their absence" (Dijkstra 1972)

La terminologia che si utilizza in quest'attività comprende termini quali:

- **AFFIDABILITÀ**, che indica la misura di successo con cui il comportamento osservato di un sistema è conforme ad una certa specifica del relativo comportamento;
- **FALLIMENTO (FAILURE)**, che indica qualsiasi deviazione del comportamento osservato dal componente specificato;

- **STATO DI ERRORE (ERRORE)**, che indica che il sistema è in uno stato in cui ogni ulteriore elaborazione porta ad un fallimento;
- **DIFETTO (BUG/FAULT)**, che indica la causa meccanica o algoritmica di un errore.

In generale non tutti i fault generano failure, una failure può essere generata da più fault e un fault può generare diverse failure, quando non è importante distinguere fra fault e failure si può usare il termine **defect (difetto)** per riferirsi appunto sia alla causa (fault) che all'effetto (failure).
Un **test** è formato da un insieme di **casi di test**, atti a testare il programma attraverso un insieme di possibili dati in input, la cui esecuzione consiste nell'esecuzione del programma per tutti i casi di test e se ha successo si rilava uno o più malfunzionamenti del programma.
Per effettuare un test bisogna conoscere il comportamento atteso per poterlo confrontare con quello osservato, a conoscere i comportamenti per ogni caso di prova è l'**oracolo**, particolare meccanismo software utilizzato in queste fasi per determinare l'esito di un test, l'oracolo umano si basa sulle specifiche o sul giudizio, l'oracolo automatico invece è generato dalle specifiche (formali), stesso software ma sviluppato da altri e versione precedente (test di regressione).



Gli oracoli automatici si rendono essenziali poiché il test di applicazioni grandi e complesse può richiedere milioni di casi di test e la dimensione dello spazio si uscita può eccedere le capacità umane che sono poco affidabili anche per uscite di piccole dimensioni.
Dunque per essere capaci di stabilire con certezza l'entità dell'esito di un test bisogna dapprima specificare il comportamento desiderato, gli **Algorithmic fault** si hanno quando si ha una cattiva comunicazione tra i team oppure l'implementazione della specifica da parte di un team risulta sbagliata, i **Mechanical Fault** si hanno quando bensì l'implementazione rispetti le specifiche del RAD potrebbe risultare non allineata nella messa in opera a causa di fattori esterni come ad esempio il fallimento della virtual machine.
Alcuni esempi di Fault ed errori sono, per i **Fault nella specifica delle interfacce**, disallineamento tra quello di cui il client ha bisogno e cosa gli viene offerto, disallineamento fra i requisiti e l'esecuzione, per gli **Algorithmic Fault**¸ inizializzazione mancane, errori di ramificazione (uscita preventiva o ritardata da loop), prova mancante per zero, per i **Mechanical Fault** che sono molto difficili da trovare, documentazione non aderente alle condizioni reali e alle procedure operative, e per gli **Errori**, errori scaturiti da sovraccarico, errori limite e di capacità ed errori legati al throughput o alla performance.
I difetti possono essere trattati adottando diversi approcci:

- **VERIFICATION**, dove si assume un ambiente ipotetico che non corrisponde all'ambiente reale, tale prova potrebbe non essere corretta poiché omette i vincoli importanti;
- **MODULAR REDUNDANCY**, attività molto costosa;
- **DECLARING A BUG TO BE A "FEATURE"**, attività non molto buona;
- **PATCHING**, attività che rallenta il rendimento;
- **TESTING**, attività preferibile dove il testing non è mai abbastanza buono.

Per aumentare l'**affidabilità** di un sistema software vengono utilizzate alcune tecniche, quali:

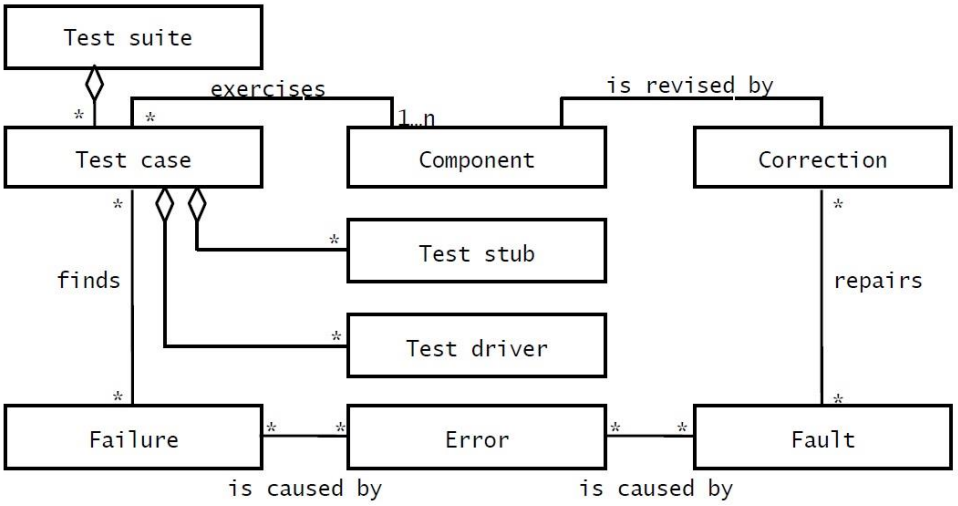
- **FAULT AVOIDANCE (evitare i difetti)**, dove si utilizzano tecniche per rilevare i difetti senza eseguire il sistema tentando appunto di prevenirne l'inserimento prima della realizzazione del sistema, tali tecniche includono metodologie di sviluppo, cioè si utilizza una buona metodologia di programmazione per ridurre la complessità, gestione delle configurazioni, cioè si utilizza il controllo delle versioni per evitare sistemi incoerenti, e verifica, cioè viene applicata la verifica per evitare bug algoritmici,
- **FAULT DETECTION (rilevare i guasti)**, dove si utilizzano tecniche come debugging e testing che sono tecniche rispettivamente non controllate e controllate, usate durante il processo di sviluppo per indentificare stati di errore e il difetto posto alla base prima di procedere al rilascio del sistema, in molti casi sono utilizzate anche dopo il rilascio del sistema.
  Tali tecniche includono:
  - **REVIEW**, che consiste in un'ispezione manuale di alcuni o tutti gli aspetti del sistema senza eseguire realmente il sistema e risulta essere una tecnica molto efficace, quasi l'85% dei fault vengono rilevati grazie ad essa.
    Ci sono due tipi di review, **walkthrough**, dove lo sviluppatore presenta informalmente le API, il codice, la documentazione associata delle componenti al team di review che commenta sul mapping modelli-codice usando il RAD e **inspection**, simile al walkthrough ma con la differenza che la presentazione delle componenti è formale, lo sviluppatore, che interviene solo se sono necessari chiarimenti, infatti non può presentare gli artefatti questo viene fatto dal team di review che è responsabile del controllo delle interfacce e del codice rispetto ai requisiti e controlla l'efficienza degli algoritmi con le richieste non funzionali;
  - **DEBUGGING**, dove si assume che il bug possa essere trovato partendo da un failure non pianificato, lo sviluppatore attraversa una sequenza di stati senza errore fino ad arrivare ad identificare uno stato di errore, successivamente si determina il bug algoritmico o meccanico che ha causato questo stato. Tale tecnica è attuata per garantire correttezza o performance;
  - **TESTING**, tecnica, usata nel processo di sviluppo, che tenta di creare stati di fallimento o errore in modo pianificato consentendo allo sviluppatore di trovare fault nel sistema prima che esso sia rilasciato al cliente, dunque il testing ha successo quando il fault viene trovato.
    La caratteristica di un buon modello per il testing è che contiene test case che identificano fault e che dovrebbero includere un range ampio di valori di input incluso input invalidi e condizioni limite, tale approccio richiede molto tempo anche per sistemi di piccole dimensioni
- **FAULT TOLLERANCE (tolleranza ai guasti)**, dove si assume che un sistema possa essere realizzato con bug e che i fallimenti del sistema possano essere gestiti effettuando il recovery a run-time.

Testare qualsiasi modulo non banale o qualsiasi sistema è pressoché impossibile, vi sono infatti **limiti teorici**, che riguardano problemi d'interruzione, il settore del testing è tormentato da problemi **indecidibili**, cioè problemi per cui è possibile dimostrare che non esiste algoritmi per risolverlo, e stabilire se l'esecuzione di un programma termina a fronte di un input arbitrario è un problema indecidibile, e **limiti pratici**, proibitivi in termini di tempo e costi.

Per sviluppare un test efficace è necessario avere una comprensione dettagliata del sistema, conoscenza delle tecniche di prova e abilità nell'applicare queste tecniche in modo efficace ed efficiente, i test vengono svolti in maniera molto più efficace se fatti da tester indipendenti poiché spesso chi progetta il sistema sviluppa un atteggiamento mentale troppo positivo verso il software convincendosi che questo funzioni in un determinato modo, utilizzando data set che fanno funzionare il programma, quando poi non è così.
Il **testing esaustivo**, si basa sull'esecuzione per tutti i possibili ingressi, ha come obiettivo quello di dimostrare la correttezza ma generalmente è impossibile da realizzare poiché anche per programmi banali si arriva a tempi superiori a quelli del big bang, il testing **termina** quando il programma si può ritenere analizzato a sufficienza riferendosi a criteri come **criterio temporale**, periodo di tempo predefinito, **criterio di costo**, sforzo allocato predefinito, **criterio di copertura**, percentuale predefinita degli elementi di un modello di programma e legato ad un criterio di selezione dei casi di test, e **criterio statistico**, predefinito.
Una **componente** è una parte del sistema che può essere isolata per essere testata, un **test case** è un insieme di input e di risultati attesi che esercitano una componente con lo scopo di causare fallimenti e rilevare fault, esso ha 5 attributi che sono **NAME**, per distinguere i test case tra loro e viene determinato, secondo euristiche, a partire dal nome della componente o del requisito che si sta testando, **LOCATION**, che descrive dove il test case può essere trovato, generalmente un path name oppure un URL al programma da eseguire e il suo input, **INPUT**, che descrive l'insieme di dati in input o comandi che l'attore del test case deve inserire, **ORACLE**, che descrive il comportamento atteso del test case, **LOG**, che è l'insieme delle correlazioni del comportamento osservato con il comportamento atteso per varie esecuzione del test.
Eseguire test case su una componente o una combinazione di componenti richiede che le componenti testate siano isolate dal resto del sistema, per garantire ciò vengono utilizzati **Test Driver**, che è una implementazione parziale di una componente che dipende dalla componente testata (componenti che chiamano la componente testata), e **Test Stub**, che è una implementazione parziale di componenti da cui la componente testata dipende (componenti chiamate dalla componente testata), che hanno il compito di sostituire le parti mancanti del sistema.
Un test stub deve fornire la stessa API del metodo della componente simulata 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, dunque la sua implementazione non è cosa facile poiché non è sufficiente scrivere un test stub che semplicemente stampa un messaggio attestante della sua inizializzazione, la componente chiamata deve svolgere una mansione e se il test stub non simula correttamente questa mansione la componente da testare potrebbe avere comunque dei failure nonostante l'assenza di fault, dunque si deve trovare il giusto compromesso tra implementazione del test stub accurata e sostituzione del test stub con la componente reale.
Una volta che i test sono identificati e descritti si determinano le relazioni fra questi espresse tramite **aggregation**, usata quando un test case può essere decomposto in un insieme di subtest, e **precedence**, usata quando un test case deve precedere un altro test case, un buon modello di test deve contenere poche relazioni per velocizzare il processo i testing.
I test case sono classificati in **blackbox**, focalizzati sul comportamento I/O e incuranti della struttura interna della componente, e **whitebox**, focalizzati sulla struttura interna della

componente, indipendentemente dall'input ogni stato nel modello dinamico dell'oggetto e ogni integrazione tra gli oggetti viene testata.
Lo **scaffolding** indica i programmi ausiliari e le classi che ci consentono di testare una determinata componente del programma, quando viene dato un input alla componente da testare l'oracolo conosce l'output previsto che viene poi confrontato con quello effettivo per identificare failure, la necessità di driver o stub dipende dalla posizione del componente nell'architettura del ssitema.
Quando i test sono stati eseguiti e i fallimenti sono stati rilevati gli sviluppatori cambiano la componente per eliminare il fault sospettato, una **correzione** è un cambiamento di una componente atto a riparare il fault, tale cambiamento potrebbe portare a nuovi fault e per gestire tali difetti vengono utilizzate alcune tecniche, quali:

- **PROBLEM TRACKING**, include la documentazione di ogni fallimento, stato di errore, e bug rilevato, la sua correzione, e la revisione delle componenti coinvolte nel cambiamento, tale tecnica consente di restringere la ricerca di nuovi fault;
- **REGRESSION TESTING**, riesecuzione dei test precedenti subito dopo il cambiamento, per assicurare che le funzionalità che lavorano prima della correzione non sono state influenzate, tale tecnica risulta particolarmente importante quando si utilizza un approccio iterativo allo sviluppo come nel caso OO;
- **RATIONAL TESTING**, include la documentazione delle motivazioni alla base dei cambiamenti e le relazioni dietro le motivazioni nella revisione della componente, tale tecnica consente agli sviluppatori di evitare di introdurre nuovi fault analizzando le assunzioni utilizzate per la costruzione della componente.



Il **MANAGING TESTING** è l'attività che ci dice come:

- Gestire le testing activity per minimizzare le risorse necessarie, poiché alla fine del testing gli sviluppatori dovranno rilevare e successivamente 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,
- Come pianificare le testing activity (diagramma di PERT che mostra le dipendenze),
- Come documentare le attività, attraverso la produzione di 4 documenti, il **TEST PLAN**, che si focalizza sugli aspetti manageriali del testing documentando la portata, l'approccio, le risorse e il programma delle testing activity, ed individua i requisiti e i componenti da testare, il **TEST CASE SPECIFICATION**, attraverso il quale si documenta ogni test, e contiene gli input, i driver, gli stub e gli output previsti dai test oltre alle attività da eseguire, il **TEST INCIDENT REPORT**, con il quale si documenta l'esecuzione di ogni test case, si forniscono i

dettagli sul risultato effettivo rispetto a quello previsto e altre informazioni intese a chiarire il motivo per cui un test ha avuto esito negativo, e il **TEST REPORT SUMMARY**, che elenca tutti i fallimenti rilevati durante i test, gli sviluppatori assegnano una priorità a ciascun errore e pianificano le modifiche nel sistema e nei modelli;

- I ruoli assegnati durante il testing.

Le attività di testing dunque si dividono in:

- **COMPONENT INSPECTION**, che ha lo scopo di trovare i fault in una componente individuale attraverso l'ispezione manuale del codice sorgente.
  Le ispezioni vengono condotte da team di sviluppatori, incluso l'autore della componente, un moderatore e uno o più revisori che trovano bug nella componente, prima o dopo il testing delle unità.
  La metodologia proposta per le ispezioni è quella di **Fagan** che si scompone in varie fasi:
  - **OVERVIEW**, dove l'autore presenta l'obiettivo e lo scope della componente e i goal dell'ispezione;
  - **PREPARATION**, dove i revisori analizzano l'implementazione della componente;
  - **INSPECTION MEETING**, dove un reader legge il codice sorgente e spiega ciò che dovrebbe fare, il team di ispezione analizza la componente ed evidenzia eventuali **fault** di cui il moderatore tiene traccia;
  - **REWORK**, dove l'autore rivede la componente;
  - **FOLLOW-UP**, dove il moderatore controlla la qualità del rework e determina la componente che deve essere ispezionata di nuovo;

  Tale metodologia è percepita come time-consuming dunque il processo di ispezione è stato poi rivisitato da **Parnas** che propone una metodologia, la **ACTIVE DESIGN REVIEW**, dove nella fase di preparation i revisori oltre che analizzare l'implementazione della componente ne individuano anche i fault per poi compilare un questionario che testa la loro comprensione della componente e viene abolita la fase di inspection meeting poiché l'autore incontra separatamente ogni revisore per collezionare feedback sulla componente.
- **USABILITY TESTING**, che ha lo scopo di trovare le differenze tra il sistema e le aspettative dell'utente per quanto riguarda l'uso del sistema, dunque testare la comprensibilità del sistema da parte dell'utente.
  Tale attività si rende necessaria perché gli utenti potrebbero trovare problemi durante l'utilizzo delle interfacce e può essere attuata attraverso tre differenti tecniche:
  - **SCENARIO TEST**, dove viene presentato ad uno o più utenti uno scenario immaginario che deve essere il più realistico possibile, gli sviluppatori valutano la bontà del modello, quanto velocemente questi utenti comprendono lo scenario e come reagiscono alla descrizione del nuovo sistema, vengono molto utilizzati strumenti come mock-up e storyboard.
    Tale metodologia ha il vantaggio di essere estremamente economica da realizzare e da ripetere ma ha lo svantaggio che gli utenti non possono interagire direttamente con il sistema avendo dati fissi;
  - **PROTOTYPE TEST**, dove agli utenti finali viene mostrato una parte del software che implementa gli aspetti chiave del sistema, si implementa completamente uno use case e vengono utilizzati prototipi funzionali per valutare le richieste cruciali (**VERTICAL PROTOTYPE**) oppure si implementa un singolo layer nel sistema che

rappresenta l'interfaccia per molti use case senza fornire funzionalità (**HORIZONTAL PROTOTYPE**).

Il vantaggio di questa metodologia è che i prototipi forniscono una vista realistica del sistema all'utente e il prototipo può essere concepito per collezionare informazioni dettagliate ma lo svantaggio è che richiede un impegno maggiore nella costruzione rispetto agli scenari cartacei;

- **PRODUCT TEST**, che risulta essere molto simile al prototype test con la differenza che viene utilizzata una versione funzionale del sistema, il test può essere affrontato solo dopo che una buona parte del sistema è stata sviluppata e ciò richiede che il sistema sia facilmente modificabile;

Tutte e tre le tecniche sono basate sull'approccio degli esperimenti controllati che si compone di varie attività, quali, sviluppo degli obiettivi del test, selezione di un campione rappresentativo di utenti finali, una simulazione 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 e raccomandazioni su come migliorare il sistema.

- **UNIT TESTING**, che ha lo scopo di trovare i fault isolando una componente individuale usando test stub e test driver, e esercitando la componente tramite un test case.
  Tale attività si focalizza dunque sui building block del sistema (oggetti e sottosistemi) e vine applicata poiché si riduce la complessità concentrandosi su una sola unit, trovare i bug risulta più facile poiché sono coinvolte poche componenti e si possono testare diverse unità in parallelo.
  Le unità candidate per il test sono prese dal modello ad oggetti e dalla decomposizione in sottosistemi, l'insieme minimale di oggetti da considerare sono praticamente quelli partecipanti negli use case, i sottosistemi devono essere testati dopo che ogni sua classe e oggetto sono stati testati individualmente.
  Per tale attività è possibile attuare due tipologie di testing:
  - **BLACKBOX TESTING (FUNZIONALE)**, che si focalizza sul comportamento di I/O, non preoccupandosi della struttura interna della componente, se per ogni input dato (impossibile generare tutti i possibili input) l'output corrisponde a quello predetto allora la componente supera il test.

    L'**equivalence testing** ha come obiettivo quello di ridurre il numero di test case effettuando un partizionamento sulla base dell'equivalenza cioè si dividono le condizioni di input in classi di equivalenza, che vengono selezionate seguendo delle linee guida dove si considera dapprima la validità dell'input per un range di valori, si selezionano i test case da 3 classi di equivalenza sotto dento e sopra il range, poi successivamente si considera la validità dell'input a seconda dell'appartenenza ad un insieme discreto, si selezionano i test case da due classi di equivalenza valore discreto valido e non valido, e si scelgono i test case per ognuna di essa.

    Molto utile risulta essere il **PARTIZIONAMENTO SISTEMATICO** che viene attuato per partizionare il dominio di input in modo che da tutti i punti del dominio ci si attende lo stesso comportamento.

    Equivalence Class Testing, basato sul prodotto cartesiano del sottoinsieme delle partizioni (ad esempio si hanno 3 input A,B e C partizionati in A1, A2, A3, B1, B2,

B3, B4, C1 e C2 il numero di test case sarà 24 (n partizioni A x n partizioni B x n partizioni C)) e testa tutte le interazioni di classe, viene utilizzato per dare un senso al test, indovinare il comportamento sottostante ed evitare la ridondanza grazie alle classi disgiunte ed è adatto quando i dati di input sono definiti in termini di intervalli o di insiemi di valori discreti, esso ha due varianti **Weak Equivalence Class Testing**, dove si sceglie un rappresentante per ogni classe di equivalenza, e **Strong**

Il **boundary testing** è un caso speciale di equivalence testing dove invece di selezionare gli elementi nella classe di equivalenza si selezionano gli elementi ai confini della classe, tali elementi possono essere rappresentati supponendo di avere una funzione F con due variabili x1 e x2, gli elementi che vengono considerati sono quelli a <= x1 <= b e c <= x2 <= d.

L'idea alla base consiste di tenere i valori delle variabili al loro minimo, appena sopra il minimo, un valore nominale, appena al di sotto del loro massimo e al loro massimo (in pratica variabili min, min+, nom, max- e max), mantenere tali valori tranne quello nominale per lasciare che assuma il suo valore estremo.

Nel **Worst Case Testing** si considera il caso peggiore in cui più variabili hanno un valore estremo, ciò si risolve facendo il prodotto cartesiano di { min, min+, nom, max- e max}.

Entrambe le tecniche soffrono dello svantaggio di non esplorare le combinazioni di dati in input che molte volte sono la causa di un bug.

- **WHITEBOOX TESTING (STRUTTURALE)**, che si focalizza sulla struttura interna della componente, indipendentemente dall'input ogni stato del modello dinamico dell'oggetto e ogni integrazione tra gli oggetti viene testata, tale tipologia di testing gode della proprietà di **completezza** poiché ogni statement della componente viene testato almeno una volta.

  Vi sono quattro tipi di whitebox testing **STATEMENT TESTING**, dove si testano i singoli statement della componente, **LOOP TESTING**, dove si definiscono i casi di test in modo da assicurare che il loop che deve essere saltato completamente, il loop da eseguire esattamente una volta e loop da eseguire più di una volta, **PATH TESTING**, con il quale ci assicuriamo che tutti i path nel programma siano eseguiti, **BRANCH TESTING**, con il quale ci assicuriamo che ogni possibile uscita da una condizione sia testata almeno una volta;

Nel whitebox testing il numero potenzialmente infinito di percorsi deve essere testato, essi spesso verificano ciò che viene fatto invece di ciò che dovrebbe essere fatto ed è impossibile rilevare casi d'uso mancanti, nel blackbox testing invece vi è una possibile esplosione combinatoria di casi di test, spesso non è chiaro se i casi di test ricoprono un particolare errore e non è possibile scovare casi d'uso esterni.

Nella unit testing entrambe le attività sia black che white si rendono necessarie poiché sono gli estremi di un continuum test e qualsiasi scelta dei casi di test si trova nel mezzo e dipende da numeri di possibili percorsi logici, numero dei dati in input, quantità di calcolo e complessità di algoritmi e strutture dati.

I quattro step per il testing sono:

- **Selezionare cosa deve essere misurato**, completezza dei requisiti, codice testato per affidabilità e design testato per la coesione;

- **Decidere come è fatto il test**, ispezione del codice, whitebox, blackbox e selezionare la strategia di test d'integrazione;
- **Sviluppare i casi di test**, che sono un insieme di dati di test o situazioni che verranno utilizzate per esercitare l'unità in fase di test o circa l'attributo che viene misurato;
- **Cercare il test oracle**, un oracolo contenente i risultati previsti per una serie di test case che deve essere annotato prima che avvenga il test vero e proprio.

Per selezionare i casi di test si può procedere in tal modo:

- **Utilizzare le conoscenze di analisi sui requisiti funzionali**, use case, dati in input attesi, dati in input non validi;
- **Utilizzare le conoscenze di progettazione relative alla struttura del sistema, agli algoritmi, alle strutture dati (whitebox)**, strutture di controllo come rami, anelli, e strutture dati come record, array, etc.;
- **Utilizzare le conoscenze di implementazione sugli algoritmi**, forzare la divisione per zero.

- **INTEGRATION TESTING**, che ha lo scopo di trovare fault integrando differenti componenti insieme, quando i bug in ogni componente sono stati rilevati e riparati le componenti sono pronte per essere integrate in sottosistemi più grandi, lo scopo di questa attività è quello di rilevare bug che non sono stati determinati durante lo unit testing focalizzandosi su un insieme di componenti che sono integrate, due o più componenti sono integrate e analizzate e quando dei bug sono rilevati nuove componenti possono essere aggiunte per riparare i bug, sviluppare test stub e test driver per un test integration è time-consuming, l'ordine in cui le componenti sono integrate può influenzare lo sforzo richiesto per l'integrazione, vi sono diverse strategie di testing per ordinare le componenti da testare originariamente concepite per una decomposizione gerarchica del sistema, ogni componente appartiene a una gerarchia di layer ordinati in base all'associazione "Call", quali:
  - **BIG BANG INTEGRATION**, dove le componenti sono dapprima testate individualmente e poi insieme un singolo sistema, sebbene sia semplice è molto costoso poiché se un test scopre un fallimento è impossibile distinguere i fallimenti nelle interfacce dai fallimenti all'interno delle componenti;
  - **BOTTOM UP INTEGRATION**, dove i sottosistemi nel layer più in basso della gerarchia sono testati individualmente, poi i prossimi sottosistemi che sono testati sono quelli che "chiamano" i sottosistemi testati in precedenza, si ripete questo passo finché tutti i sottosistemi sono testati, i test driver sono molto utilizzati per simulare le componenti dei layer più in alto che non sono stati ancora integrati, Tale strategia non è buona per sistemi decomposti funzionalmente perché testa i sottosistemi più importanti per ultimi ma è utili per integrare sistemi Object-oriented, real-time e con rigide richieste sulle piattaforme;
  - **TOP DOWN INTEGRATION**, dove sono testati prima i layer al top o i sottosistemi di controllo poi vengono combinati tutti i sottosistemi che sono chiamati dai sottosistemi testati e quindi testa la collezione risultante di sottosistemi, questa attività viene iterata fino a quando tutti i sottosistemi non sono integrati e testati, molto utilizzati sono i test stub per simulare le componenti dei layer bottom che non sono state ancora testate.

Tale strategia ha il vantaggio che i test cases possono essere definiti in termine delle funzionalità del sistema e si possono riutilizzare nelle varie iterazioni e lo svataggio che scrivere gli stub è un'attività costosa e difficile poiché devono consentire tutte le possibili condizioni da testare e dunque possono essere richiesti in gran quantità specialmente se il sistema contiene molti metodi, una soluzione a questi problemi consiste nel testare individualmente ogni layer della decomposizione prima di fare il merge dei layer con lo svantaggio però di avere la necessità sia di test stub che test driver;

- **SANDWICH TESTING**, combina l'uso di top-down strategy con bottom-up strategy, il sistema viene visto come se avesse 3 layer un layer target nel mezzo, un layer sopra il target e uno sotto il target, il testing converge al layer target che viene selezionato nel caso in cui ci siano più di 3 layer tentando di minimizzare il numero di stub e di driver.
  Tale strategia ha il vantaggio che il test dei layer al top e al bottom possono essere fatti in parallelo ma si ha il problema che i sottosistemi del target layer non vengono testati individualmente prima dell'integrazione, una soluzione a questo problema sarebbe modificare l'approccio sandwich testando i 3 layer individualmente prima di combinarli in test incrementali con gli altri, i test individuali dei layer consistono nell'effettuare test del layer al top con stub per il layer al target, test per il layer al target con stub e driver rispettivamente per i layer al bottom e al top e test del layer al bottom con driver per il layer target, i test dei layer combinati consistono in 2 test il layer al top accede al layer target, può riusare i test del layer target dai test individuali, sostituendo i driver con le componenti dei layer al top e il layer 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.

Dunque gli step dell'integration testing sono:

- **In base alla strategia d'integrazione**, selezionare un componente da testare;
- **Mettere insieme il componente selezionato**, eseguire eventuali correzioni preliminari necessarie per rendere operativo il test di integrazione;
- **Effettuare test funzionali**, definire casi di test che esercitano tutti i casi d'uso del componente selezionato;
- **Eseguire test strutturali**, definire i casi di test che esercitano il componente selezionato;
- **Eseguire test delle prestazioni**;
- **Tenere un registro dei casi di test e delle attività di test**;
- **Ripetere i passaggi da 1 a 7 fino a quando non viene testato l'intero sistema.**

- **SYSTEM TESTING**, che si focalizza sul sistema completo, i suoi requisiti funzionali e non funzionali e il suo ambiente, assicura che il sistema completo sia conforme ai requisiti funzionali e non funzionali che hanno un impatto notevole sul testing del sistema poiché più sono espliciti più facili sono da testare, la qualità degli use case influenza il functional testing, la qualità della decomposizione influenza lo structure Testing e la qualità dei requisit non funzionali e dei vincoli influenza il performance testing.
  Tale attività si scompone in attività secondarie, quali:

- **FUNCTIONAL TESTING**, che ha come obiettivo quello di testare le funzionalità del sistema, quello che si fa è essenzialmente il blackbox testing, il sistema viene trattato come un blackbox, i test case sono progettati dal documento dei requisiti e si focalizza sugli use case, i test case per le unità possono anche essere riusati ma devono essere scelti quelli rilevanti pe l'utente finale e che hanno una buona possibilità di fallimento
- **PILOT TESTING**, i sistemi piloti sono utili quando il sistema è costruito senza un insieme id richieste specifiche o senza un particolare cliente in mente, e dunque all'utente non viene fornita nessuna linea guida, vi sono due tipologie di pilot testing, **Alpha testing**, test pilota con utenti che esercitano il sistema nell'ambiente di sviluppo, e **Beta testing**, test di accettazione del sistema realizzato da un numero limitato di utenti nell'ambiente di utilizzo;
- **PERFORMANCE TESTING**, che ah come obiettivo quello di tentare di rompere il sistema e si suddivide in **stress testing**, che punta a stressare il sistema riguardante il carico di lavoro, **volume testing**, che verifica cosa succede se vengono gestite grandi quantità di dati, **configuration testing**, che testa le varie configurazioni software e hardware, **compatibility test**, che verifica la compatibilità coni sistemi esistenti, **security testing**, che prova a violare i requisiti di sicurezza, **timing testing**, che valuta i tempi di risposta per eseguire una funzione, **environmental test**, che verifica la tolleranza per calore, umidità, movimento, portabilità, **quality testing**, che verifica l'affidabilità, manutenibilità e disponibilità del sistema, **recovery testing**, che verifica la risposta del sistema alla presenza di errori o alla perdita di dati, **human factor testing**, che verifica l'interfaccia utente con l'utente;
- **ACCEPTANCE TESTING**, dove il cliente valuta il sistema utilizzando 3 metodologie **Benchmark test**, il cliente prepara un insieme di testa case che rappresentano le condizioni tipiche sotto cui il sistema dovrà operare, **Competitor testing**, il nuovo sistema è testato contro un sistema esistente, **Shadow testing**, una forma di testing a confronto, il nuovo sistema e il sistema legacy sono eseguiti in parallelo e i loro output sono confrontati.
  Se il cliente è soddisfatto il sistema viene accettato;
- **INSTALLATION TESTING**, dopo che il sistema è stato accettato viene installato nell'ambiente di utilizzo rispettando in pieno le richieste del cliente, in molti casi il test di installazione ripete i test case eseguiti durante il functional testing e il performance testing nell'ambiente di utilizzo, quando il cliente è soddisfatto il software viene formalmente rilasciato ed è pronto per l'utilizzo.

CATEGORY PARTITION

L'obiettivo del functional testing è quello di trova discrepanze fra il comportamento effettivo di una funzione del sistema e il comportamento desiderato descritto nelle specifiche funzionali del sistema. I test devono essere eseguiti per tutte le funzioni del sistema e devono essere progettati per massimizzare le possibilità di trovare errori nel software, il test funzionale può essere derivato da 3 fonti **specifiche del software**, **informazioni di design** o **codice stesso.**
L'approccio standard, il **Partition**, ha come idea principale quella di partizionare il dominio di input della funzione sottoposta a test e quindi selezionare i dati di test per ogni classe della partizione, il problema però è che tutte le tecniche esistenti mancano di sistematicità.

Il metodo **CATEGORY PARTITION** ha come idee principali quella che la specifica del test è una rappresentazione uniforme delle informazioni di test per una funzione, può essere facilmente modificata e fornisce al tester un modo logico per controllare il volume dei test, utilizzare un **generator tool** per aiutare fornendo un modo automatico per produrre test approfonditi e per evitare test impossibili o indesiderabili, e che il metodo enfatizza sia la copertura delle specifiche che degli aspetti di rilevamento degli errori del test.
La strategia per generare i test case si compone di alcuni passi:

1. Trasformare le **specifiche** del software in modo da renderle più concise e strutturate;
2. Decomporre le specifiche in **unità funzionali** in modo da testarle individualmente;
3. Identificare i **parametri** e le **environment condition** (condizioni dell'ambiente);
4. Cercare le **categorie** che caratterizzano ciascun parametro e le condizioni ambientali (solitamente una proprietà o caratteristica principale di un parametro o di una condizione ambientale);
5. Ogni categoria dovrebbe essere suddivisa in **scelte** distinte, ogni scelta all'interno di una categoria è un insieme di valori simili che possono essere assunti dal tipo di informazioni nella categoria;

Lista categorie, una lista di scelte per ogni categoria

⇓

Specifica formale del test

6. **Test frames**, un insieme di scelte uno per ogni categoria;
   Test cases, frame di prova con valori specifici per ciascuna scelta;
   Test scripts, sequenza si casi di test correlati;

Per ridurre i casi di prova vengono adottati dei metodi, che hanno come obiettivo quello di ridurre il numero esponenziale di combinazioni considerando dolo i casi più significativi, quali:

- **PARTIZIONE DELLE CATEOGIRE**, basato sull'identificazione di categorie di input e relazioni di incompatibilità, **PASSO 1 -** il sistema viene diviso in funzioni che possono essere testate indipendentemente, il metodo individua i parametri di ogni funzione e per ogni parametro individua categorie distinte, oltre ai parametri possono essere considerati anche oggetti dell'ambiente, le categorie sono le principali proprietà o caratteristiche, le categorie vengono ulteriormente suddivise in modo in scelte nello stesso modo in cui si applica la partizione in classi di equivalenza (possibili valori), normal values, boundary values, special vaues , error values; **PASSO 2 –** 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 generati Test frames che consistono di combinazioni valide di scelte nelle categorie che vengono poi convertiti in test data;
- **TABELLE DELLE DECISIONI**, basato sulla costruzione di tabelle per identificare i casi più significativi, le motivazioni del suo utilizzo sono che: aiuta ad esprimere test requirements in una forma direttamente comprensibile e usabile, supporta la generazione automatica o manuale di casi di test, una particolare risposta o un sottoinsieme di risposte deve essere selezionato valutando molte condizioni corrispondenti, ed è l'ideale per descrivere situazioni in cui un numero di combinazioni di azioni sono prese in corrispondenza di insiemi di condizioni variabili, come ad esempio nei sistemi di controllo.

Le tabelle hanno una struttura suddivisa in una sezione delle condizioni che contiene condizioni e combinazioni di queste, una condizione esprime relazioni tra variabili di decisione, e una sezione delle azioni che mostra le risposte che devono essere prodotte quando le corrispondenti combinazioni di condizioni sono vere, il limite di tale approccio è che le azioni risultanti sono determinate dai valori correnti delle variabili di decisione, e sono indipendenti dall'ordine di input e dall'ordine in cui le condizioni sono valutate, e possono apparire più di una volta ma ogni combinazione di condizioni è unica.
Per n condizioni ci possono essere 2 alla n varianti (combinazioni uniche di condizioni e azioni) ma per fortuna ci sono meno varianti esplicite e in aiuto ci vengono i valori "**dont care**", che riducono il numero di varianti considerando i casi in cui gli input sono necessari ma non hanno effetto, possono essere omessi o casi mutuamente esclusivi;

- **GRAFO CAUSA EFFETTO**, basato sulla costruzione di grafi per ridurre il numero di combinazioni in base agli effetti delle diverse combinazioni, tale approccio mira a creare combinazioni interessanti di test data, individua cause, (condizioni di input, stimoli) che devono essere stabilite in modo tale da essere o vere o false, ed effetti (output, cambiamenti di stato del sistema), specifica esplicitamente i vincoli (ambientali, esterni) u cause ed effetti ed aiuta a selezionare i più significativi sottoinsiemi di combinazioni input-output e costruisce tabelle di decisione più piccole.
  I grafi sono strutturati associando ad un nodo ogni causa ed ogni effetto, i nodi causa ed effetto sono piazzati su lati opposti di un foglio, l'interconnessione di nodi causa ed effetto e produzione di un grafo booleano, una linea da una causa ad un effetto indica che la causa + una condizione necessaria per l'effetto, una singola cause può essere necessaria per diversi effetti e un singolo effetto può avere molte cause necessarie, se un effetto ha due o più cause la relazione logica tra le cause è espressa inserendo operatori and e or logici tra le due linee, una causa la cui negazione è necessaria per un effetto è espressa etichettando la linea con l'operatore not logico, possono essere usati nodi intermedi per semplificare il grafo e la sua costruzione;
- **PAIRWAISE COMBINATORIAL TESTING**, che si divide in due tipologie:
  - **Pairwise (and n-way) combinatorial testing**, combina i valori sistematicamente ma non esaustivamente, avvengono molte relazioni solo tra due o pochi parametri o catteristiche;
  - **Pairwise combination (invece che esaustive)**, genera combinazioni che efficientemente coprono tutte le coppie di classi, la maggior parte delle failure sono causate da valori singoli o combinazioni di pochi valori coprendo coppie, riduce il numero di casi di test ma consente di rivelare la maggior parte dei difetti.