1. INTRODUZIONE

1.1 Software

Le economie di tutti i paesi sviluppati sono dipendenti dal **software**, che rappresenta una delle voci rilevanti del PIL di questi paesi e, ogni giorno, un numero sempre più alto di sistemi contengono software o dipendono da esso. La sua evoluzione è avvenuta attraverso varie fasi. Si è partiti da una fase di **Arte**, durante la quale le applicazioni venivano sviluppate da singole persone e utilizzate dagli stessi sviluppatori, per passare ad una fase di **Artigianato**, durante la quale le applicazioni venivano sviluppate da piccoli gruppi specializzati per un cliente. Tuttavia, i costi per tenere una propria unità informatica erano troppo alti e si è giunti così alla fase di **Industria**: il software è diffuso in diversi settori, sono cresciute le dimensioni, la complessità e la criticità delle applicazioni, si è sviluppato un vero e proprio mercato con relativa concorrenza, è nata la necessità di migliorare la produttività e la qualità, i progetti vengono gestiti e il software è in continua evoluzione. Insomma, oggi, il software è ovunque!

Una prima distinzione su cui prestare attenzione è quella tra *Programma* e *Prodotto software*.
Per un programma, l'autore è anche l'utente, non è documentato, quasi mai è testato e non c'è progetto. In altre parole non serve un approccio formale al suo sviluppo.
Un prodotto software, invece, è usato da persone diverse da chi lo ha sviluppato. Si tratta di un prodotto industriale, il cui costo è circa 10 volte il costo del corrispondente programma e, ovviamente, richiede un approccio formale allo sviluppo. Questi prodotti si possono dividere in due categorie:

- Prodotti generici: sistemi stand-alone prodotti da una organizzazione e venduti a un mercato di massa. Sono quelli meno costosi perché la spesa per lo sviluppo è ammortata nel tempo (ad esempio, Microsoft Office).
- Prodotti specifici: sistemi commissionati da uno specifico utente e sviluppati specificatamente per quest'ultimo da un qualche contraente.

La fetta maggiore della spesa mondiale è nei prodotti generici, ma il maggior sforzo di sviluppo è chiaramente nei prodotti specifici.

Di seguito sono riportate varie definizioni formali del termine *Software*:

- *Dizionario della lingua italiana* (Devoto, Oli - 1971):
  Software: il corredo dei linguaggi e dei programmi di cui è munito un sistema elettronico per l'elaborazione dei dati.

- *Oxford Advanced Learner's Dictionary* (1992):
  Software: dati, programmi, e tutto ciò che non è parte di un computer ma usato quando si lavora con quest'ultimo.

- *Standard IEEE (Institute of Electrical and Electronics Engineers) 610.12-1990*:
  Software: l'insieme dei programmi, delle procedure ed eventuali documentazioni allegate, e dei dati pertinenti le operazioni di un sistema computerizzato.

Per *Prodotto Software*, quindi, si intende l'insieme completo dei programmi, delle procedure con eventuali documentazioni allegate e dei dati progettati dagli sviluppatori per un utente e tutto ciò che viene rilasciato a quest'ultimo.

In altre parole, il software non è solo il codice, ma tutti gli "artefatti" che lo accompagnano e che sono prodotti durante l'intero processo di sviluppo: codice, documentazione, casi di test, specifiche di progetto, procedure di gestione, manuali utente…
Data la sua natura, il software è completamente diverso da ogni altro prodotto industriale classico e, in generale, di ingegneria: è intangibile, malleabile, ad alta intensità di lavoro umano, spesso costruito ad hoc, invece che assemblato, e manutenzione significa cambiamento.

Durante il processo evolutivo del software, sono stati sfatati molti miti:
- *Management*:
  - "Con computer potenti e moderni risolviamo tutti i problemi".
  - "Se siamo in ritardo possiamo recuperare aumentando il numero di programmatori".
- *Cliente*:
  - "Un'affermazione generica degli scopi è sufficiente per cominciare a scrivere programmi".
  - "I mutamenti nei requisiti di un progetto si gestiscono facilmente grazie alla flessibilità del software".
- *Programmatore*:
  - "Una volta messo in opera il programma, il lavoro è finito".
  - "Il solo prodotto di un progetto concluso è il programma".

Queste affermazioni, oggi, sono ritenute tutte errate.

1.2 Ingegneria del Software

I problemi che si possono incontrare nella produzione del software dipendono da vari fattori: **costi**, **ritardi e abbandoni**, **affidabilità**.

1.Costi

Il software ha costi elelevati! Principalmente si tratta dei costi, espressi in mesi/uomo, delle risorse umane: ore lavoro o manpower (il più dominante), hardware, software e risorse di supporto. Anche il testing influisce su questi costi, impiegandone fino al 40%, si preferisce infatti non ritardarlo, perché alla fine costa sempre di più. Inoltre la manutenzione costa più dello sviluppo: per sistemi che rimangono a lungo in esercizio i costi di manutenzione possono essere svariate volte il costo di produzione!
In genere, la produttività media è da 300 a 1000 linee di codice rilasciate per mese/uomo.
Nello scenario attuale, se un'azienda tiene una persona a lavorare per un anno (anno/uomo), al cliente costa $100.000, cioè $8.000 per mese/uomo. Un cliente, infatti, paga da $8 a $25 per linea di codice rilasciata e per un prodotto di dimensioni medio-piccole (circa 50.000 linee di codice) da $500.000 a $1.250.000. E la manutenzione è ancora più gravosa: se una linea di codice costa $25, al cliente, un intervento su una linea di codice in manutenzione può costare fino a $1.000!

2.Ritardi e abbandoni

Si possono verificare ritardi nelle consegne, con conseguente aumenti dei costi. Eccone alcuni esempi:

- Il progetto di un sistema di comando e controllo per la US Air Force fu stimato dalla azienda vincitrice per la fornitura intorno ai $400.000, costo rinegoziato successivamente a $700.000, poi a $2.500.000 e infine a $3.200.000. In altre parole, il costo finale divenne maggiore di circa 10 volte la stima iniziale e con un notevole ritardo rispetto alla stessa stima!! (Da un report del 1981).

- Una azienda nel settore della grande distribuzione di prodotti aveva richiesto un sistema che, stima iniziale, sarebbe stato sviluppato in 9 mesi al prezzo di $250.000 (da un report del 1989). Due anni dopo, e dopo una spesa di $2.500.000, il lavoro non era stato ancora completato e fu stimato che erano necessari altri $3.600.000!! Risultato: il progetto fu abbandonato!
- Da un report USA del 1989 risulta che su 600 aziende contattate, più del 35% avevano progetti "runaway", cioè in ritardo o fuori dal budget e dallo schedule stimati.

3.Affidabilità

Il software è spesso inaffidabile e molti malfunzionamenti sono rilevati durante l'operatività del sistema. Da un'analisi del ministero della difesa USA risulta che più del 70% di tutti i malfunzionamenti in sistemi con complicati e sofisticati apparati meccanici, elettrici, idraulici, etc., sono dovuti al software (vedi l'Arianne 5 e la sonda MARINER).

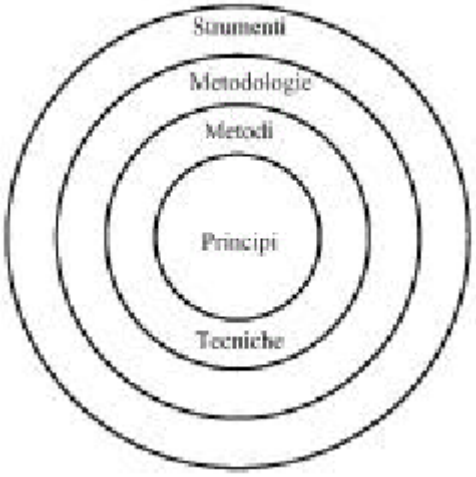
Da queste considerazioni nasce la necessità di un approccio ingegneristico alla produzione software, per poter sviluppare il *giusto* prodotto, al *giusto* costo, nel tempo *giusto* e con la *giusta* qualità.

Lo scopo dell'**Ingegneria del Software** riguarda la costruzione di software di grandi dimensioni, di notevole complessità e sviluppati tramite lavoro di gruppo. Progetti software di questo tipo hanno tipicamente versioni multiple, una lunga durata e frequenti cambiamenti volti ad eliminare difetti, ad adattare il prodotto a nuovi ambienti e volti a introdurre miglioramenti e nuove funzionalità.
Uno dei "guru" dell'Ingegneria del software, *Parnas*, ne ha dato la seguente definizione: "Costruzione multi-persona di software multi-versione".
Uno degli obiettivi fondamentali dell'Ingegneria del Software è quello di *contestualizzare* il software. Infatti, la maggior parte del SW è collocata all'interno di un "sistema" misto HW/SW e l'obiettivo finale di chi produce è creare tale sistema in maniera da soddisfare globalmente i requisiti dell'utente. Da qui segue il coinvolgimento nella definizione dei requisiti del sistema.
E' quindi essenziale la conoscenza del dominio applicativo per un efficace sviluppo del SW, perché lo stesso è utile solo se riesce a condensare nei suoi algoritmi la conoscenza di tale dominio applicativo. Contrariamente, invece, il SW potrebbe essere inutile o dannoso (ad esempio, il sistema di controllo di un aeroplano).

Sono state date varie definizioni formali di Ingegneria del Software:

- Da *"Standard Glossary of Software Engineering Terminology"* (1993) dell'*IEEE* (*Institute of Electrical and Electronics Engineers*): Applicazione di una strategia sistematica, disciplinata e misurabile allo sviluppo, esercizio e manutenzione del software (programmi, procedure, regole e associata documentazione, dati).
- La disciplina tecnologica e manageriale che riguarda la produzione sistematica e la manutenzione dei prodotti software che vengono sviluppati e modificati entro i tempi e i costi preventivati (D. Farley).

In generale, l'ingegneria del software si occupa dei ***metodi***, delle ***metodologie***, dei ***processi*** e degli ***strumenti*** per la gestione professionale (sviluppo, manutenzione, ritiro) del software:



Si basa su **Principi**, quali:

Rigore: concetto primitivo (precisione, accuratezza);
Formalità: oltre il rigore (fondamento matematico);
Separazione di aspetti diversi: affrontare separatamente i vari lati 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;
Anticipazione del cambiamento: la progettazione deve favorire l'evoluzione del SW;
Generalità: tentare di risolvere il problema nella sua accezione più generale;
Incrementalità: lavorare per passi successivi.

Metodo (o tecnica): procedimento generale per risolvere classi di problemi specificati di volta in volta (linee guida o regole che governano le attività; il metodo dei minimi quadrati, il metodo di Montecarlo, il metodo di Newton, come fare il brodo di carne, come fare il lesso, ...).

Metodologia: insieme di principi, metodi ed elementi di cui una o più discipline si servono per garantire la correttezza e l'efficacia del proprio procedere (ad esempio la metodologia della macerazione carbonica permette di ottenere vini novelli, freschi, profumati, …).

Strumento (tool): un artefatto, un sistema per fare qualcosa in modo *migliore* (ad esempio, il frullatore per fare la maionese, un cavatappi per aprire una bottiglia), insomma, un supporto SW pratico all'applicazione.

Procedura: una combinazione di **strumenti** e **metodi** che, assieme, permettono di produrre un certo prodotto (ad esempio, la ricetta della Saker Torte: montare a neve le chiare di 4 uova ...).

Paradigma: un particolare approccio o filosofia per fare qualcosa (ad esempio, lo stile della cucina, noi riconosciamo la cucina Francese, quella Italiana, quella Cinese, …).

Anche la definizione di un **processo** può essere differente:

- Un processo è un particolare *metodo* per *fare qualcosa* costituito da una sequenza di passi che coinvolgono attività, vincoli e risorse (*Pfleeger*).
- Un processo è una particolare metodologia operativa che nella tecnica definisce le singole operazioni fondamentali per ottenere un prodotto industriale (*Zingarelli*).

Ma che cosa è un **Processo software**? *Sommerville* lo definisce come un metodo per sviluppare del software. In generale, un Processo software è un insieme organizzato di *attività* che sovrintendono

alla costruzione del prodotto da parte del team di sviluppo, utilizzando metodi, processi, 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*: informale, semi-formale o formale (*maturità del processo*).
Lo Standard IEEE 610.12-1990 definisce il ***Processo di sviluppo del software*** come "il processo mediante il quale le richieste dell'utente vengono tradotte in un prodotto software. Il processo include traduzione delle richieste dell'utente in requisiti software, trasformazione dei requisiti software in progetto, implementazione del progetto in codice, testing del codice, e a volte, installazione e collaudo del software per la messa in esercizio. Queste attività possono sovrapporsi o essere eseguite iterativamente. Vedi anche: sviluppo incrementale, prototipazione rapida, modello a spirale, modello a cascata".

Secondo *B. Boehm* i *Progetti* possono essere suddivisi in classi:

Classe	**Linee di codice**	**Persone (media)**	**Mesi**
Small	2.000	1-2	4-5
Intermediate	8.000	2-6	8-9
Medium	32.000	6-16	14
Large	128.000	16-51	24
Very Large	512.000	60-157	41-42
…	…	…	…

In uno scenario del genere, le necessità del *Project Management* sono: *project planning* (definire attività, risorse da allocare, tempi), *project monitoring e controllo*, *metriche di prodotto e di processo* per misurare e controllare...
A tali scopi esistono sistemi software che forniscono un supporto automatico per le attività di un processo software, detti **CASE** (*Computer-Aided Software Engineering*). Si dividono in:

- *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 e dizionari dei dati per gestire entità del progetto.

- *Lower-CASE*: Strumenti che supportano le attività delle fasi finali del processo, come programming, testing e debugging. Includono generatori di GUI per la costruzione di interfacce utente, debuggers per supportare la ricerca di program fault e traduttori automatici per generare nuove versioni di un programma.

2. CICLO DI VITA DEL SOFWARE

2.1 Overview

Secondo lo Standard IEEE 610.12-1990, il **Ciclo di Vita del Software** è il periodo di tempo che inizia quando il software viene concepito e termina quando il prodotto non è più disponibile per l'uso da molto tempo. Tipicamente, include le fasi di concepimento, requisiti, progetto, implementazione, test, installazione e collaudo, una fase operativa e di manutenzione, e a volte, una fase di ritiro. Queste fasi possono sovrapporsi o essere eseguite iterativamente.

Questo concetto è differente da quello di **Ciclo di Sviluppo del Software**: il periodo di tempo che inizia con la decisione di sviluppare un prodotto software e termina quando il prodotto è completato. Questo ciclo, tipicamente, include le fasi di requisiti, progetto, implementazione, test, e a volte, una fase di installazione e collaudo. Queste fasi possono sovrapporsi o essere eseguite iterativamente, a seconda dell'approccio scelto per lo sviluppo. Inoltre, questi termini sono a volte usati sia per indicare un lungo periodo di tempo, sia per indicare il periodo che termina quando il software non subisce più variazioni da molto, oppure per indicare l'intero Ciclo di Vita del Software.

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

Molti autori, tuttavia, usano i termini processo di sviluppo del software e ciclo di vita del software come sinonimi.

Esistono vari modelli di CVS, nati negli ultimi 30 anni: Waterfall (cascata), Prototyping, Incremental delivery (approcci evolutivi), Spiral model. La definizione di questi modelli è influenzata da molti aspetti: specificità dell'organizzazione produttrice, know-how, area applicativa e particolare progetto, strumenti di supporto, diversi ruoli produttore/committente.

In un'ottica di alto livello, comunque, un CVS consta delle seguente fasi:

- *Definizione*: si occupa del *cosa*. In particolare si occupa di determinare i requisiti, le informazioni da elaborare, le funzioni e le prestazioni attese, il comportamento del sistema, le interfacce, i vincoli progettuali, i criteri di validazione.
- *Sviluppo*: si occupa del *come*. In altre parole, si occupa di definire il progetto, l'architettura software, la strutturazione dei dati, delle interfacce e dei dettagli procedurali, di tradurre il progetto nel linguaggio di programmazione e dei collaudi.
- *Manutenzione*: si occupa delle *modifiche*, come correzioni, adattamenti, miglioramenti, prevenzione.

2.2 Modello a Cascata (Waterfall)

Modello popolare negli anni '70, grazie a Royce, nato come reazione al "code and fix" originario e trova ispirazione dall'industria manifatturiera. E' un modello sequenziale lineare, cioè una progressione sequenziale (in cascata) di fasi, senza ricicli, per meglio controllare tempi e costi.

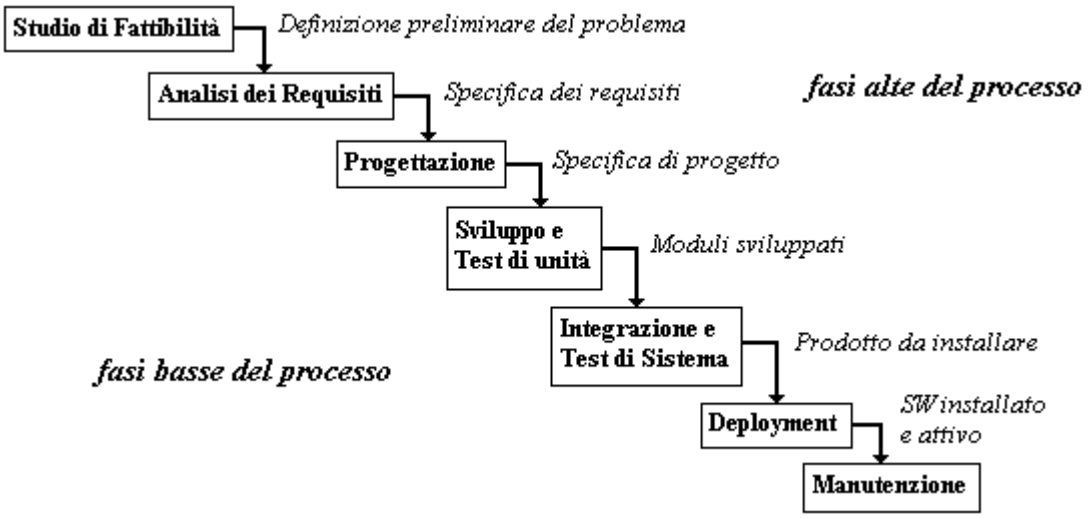
Definisce e separa le varie fasi e le attività del processo (non c'è, o è minimo, l'overlap fra le fasi), prevede uscite intermedie rappresentate da *semilavorati* del processo (documentazione cartacea e programmi) formalizzati in struttura e contenuti. Inoltre, consente un controllo dell'evoluzione del processo mediante attività trasversali alle diverse fasi.
Come già accennato, l'organizzazione delle fasi è sequenziale:

- ogni fase raccoglie un insieme di attività omogenee per metodi, tecnologie, skill del personale, etc.;
- ogni fase è caratterizzata dalle attività (task), dai prodotti di tali attività (diliverables), dai controlli relativi (quality control measures);
- la fine di ogni fase è un punto rilevante del processo (*milestone*);
- i semilavorati in output da una fase sono input per la fase successiva;
- i prodotti di una fase vengono "congelati", ovvero non sono più modificabili se non innescando un processo formale e sistematico di modifica.

Un modello a cascata è paragonabile a un modello di processo industriale, ad esempio la costruzione di una casa. Utilizzando il modello a cascata:

- *Specifica Requisiti*: definizione dei requisiti e vincoli del sistema (…vorrei una casa su 2 piani, con autorimessa e cantina…);
- *Progetto di Sistema*: produrre un modello cartaceo (…planimetrie, assonometrie,…);
- *Progetto di Dettaglio*: modelli dettagliati delle parti (progetto dell'infrastruttura elettrica, idrica, calcoli statici travi, …);
- *Costruzione*: realizzare il sistema;
- *Test dei Componenti*: verifica le parti separatamente (impianto elettrico, idraulico, portata solai, …);
- *Test di Integrazione*: integra le parti (il riscaldamento funziona…);
- *Test di Sistema*: verifica che il sistema rispetti le specifiche richieste;
- *Installazione*: avvio e consegna del sistema ai clienti (ci vanno a vivere);
- *Manutenzione*: (cambio della guarnizione al lavandino che perde, …).

In generale, il Modello a Cascata può essere sviluppato mediante le seguenti fasi:



Fasi alte del processo:

- *Studio di fattibilità*: consiste nella valutazione preliminare dei costi e dei benefici, varia a seconda della relazione committente/produttore. L'obiettivo è stabilire se avviare il progetto, individuare le possibili opzioni e le scelte più adeguate, valutare le risorse umane e finanziare necessarie. L'output di questa fase è il *documento di fattibilità*, che contiene: la definizione preliminare del problema, gli scenari e le strategie alternative di soluzione, i costi, i tempi e le modalità di sviluppo per ogni alternativa.

- *Analisi dei requisiti*: analisi completa dei bisogni dell'utente e del dominio del problema, coinvolgendo il committente e ingegneri del SW. L'obiettivo è descrivere le funzionalità e le caratteristiche di qualità che l'applicazione deve soddisfare (il "*che cosa*"). L'output di questa fase è il *documento di specifica dei requisiti*: manuale d'utente e piano di *acceptance test* del sistema.

- *Progettazione*: definizione di una struttura opportuna per il SW, scomponendo il sistema in componenti e moduli: allocazione delle funzionalità ai vari moduli e definizione delle relazioni fra i moduli. Durante questa fase si effettua una distinzione tra *architectural design*, cioè struttura modulare complessiva (componenti), e *detailed design*, cioè dettagli interni a ciascuna componente. L'obiettivo è il "*come*", e l'output è il *documento di specifica di progetto* (è possibile l'uso di linguaggi/formalismi per la progettazione).

Fasi basse del processo:

- *Programmazione (Sviluppo) e test di unità*: ogni modulo viene codificato nel linguaggio scelto e testato isolatamente.

- *Integrazione e test di sistema*: composizione dei moduli nel sistema globale e verifica del corretto funzionamento del sistema. Mediante l'*α-test* il sistema viene rilasciato internamente al produttore, con il *β-test* viene invece rilasciato a pochi e selezionati utenti.

- *Deployment*: distribuzione e gestione del software presso l'utenza.

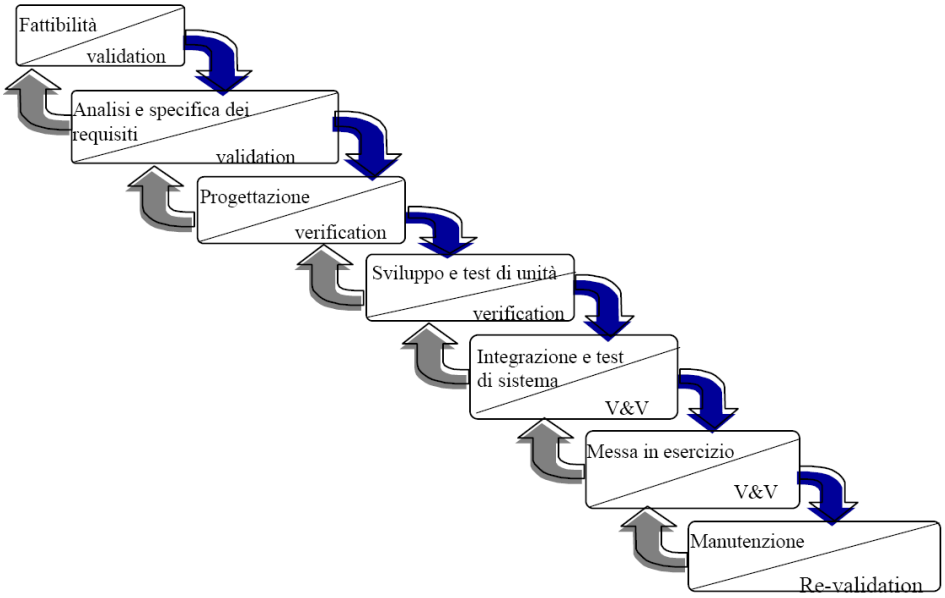
- *Manutenzione*: riguarda l'evoluzione del SW e segue le esigenze dell'utenza. Comporta ulteriore sviluppo, per cui racchiude in sé nuove iterazioni di tutte le fasi precedenti.

Tra i **pro** del Modello a Cascata dobbiamo riconoscere: la definizione di molti concetti utili (semilavorati, fasi, ecc.); il fatto che ha rappresentato un punto di partenza importante per lo studio dei processi SW; è facilmente comprensibile e applicabile.
D'altra parte, ha come **contro** il fatto che l'interazione con il committente c'è solo all'inizio e alla fine, infatti i requisiti sono congelati alla fine della fase di analisi, ma i requisiti dell'utente sono spesso imprecisi ("l'utente sa quello che vuole solo quando lo vede"). Inoltre il nuovo sistema software diventa installabile solo quando è totalmente finito: né l'utente né il management possono giudicare prima del completamento del sistema sull'adesione alle proprie aspettative.
Nella realtà le cose sono diverse! Di norma le specifiche del prodotto sono incomplete e inconsistenti, e l'applicazione evolve durante tutte le fasi, cioè non esiste una netta separazione tra le fasi di specifica, progettazione e produzione. Inoltre gli overlap (sovrapposizioni) e i ricicli esistono! In alcuni casi, è auspicabile sviluppare prima una parte del sistema e poi completarlo. E poi il software non si consuma: fare manutenzione non significa sostituire componenti! Tuttavia, questa fase non può essere considerata marginale.

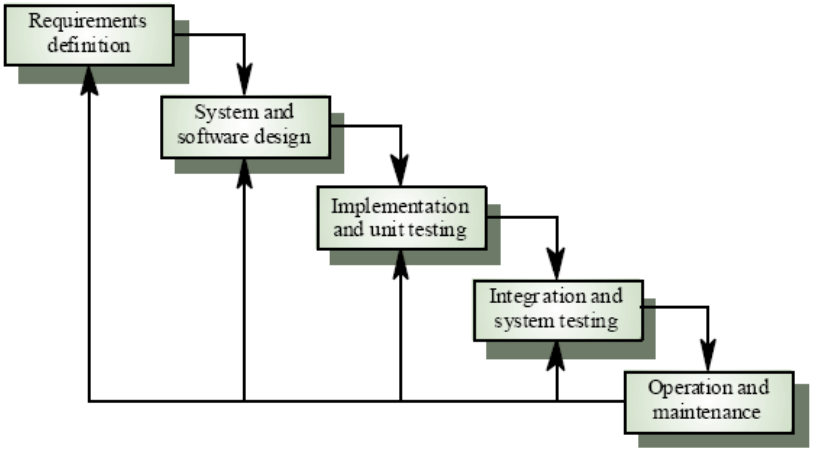
Da queste critiche al Modello a Cascata sono derivati altri cicli di vita che cercano di ovviare al problema dell'instabilità dei requisiti, che tengono in considerazione l'esistenza dei ricicli, e in cui è data maggiore enfasi sulla manutenzione (modelli evolutivi). I più rilevanti sono: Varianti del modello a cascata, Modello trasformazionale, Modelli con prototipo ("do it twice"), Modelli evolutivi ed incrementali, Modelli basati su riuso, Meta-modello a spirale.

2.3 Variante del Modello a Cascata: V&V e Retroazione (Feedback)

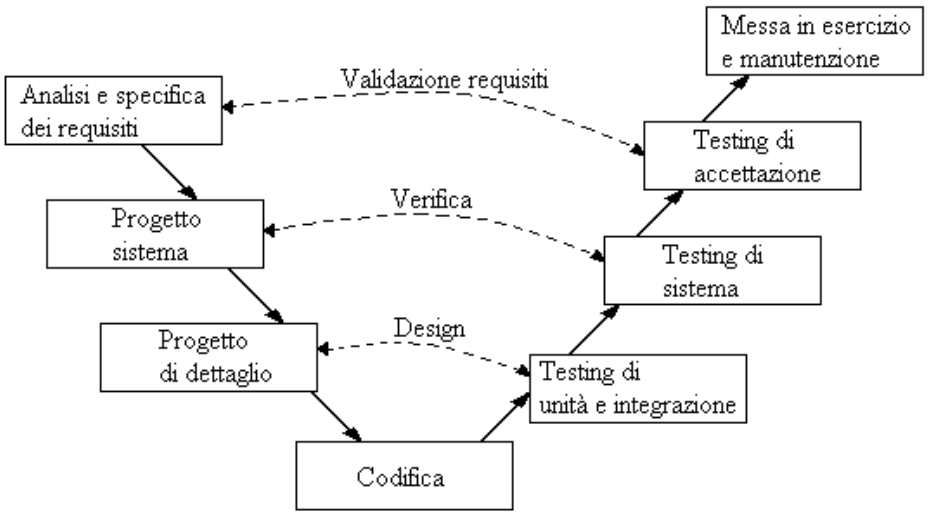
V&V sta per *Verification & Validation*, che secondo *B.Boehm* significano rispettivamente: "stiamo costruendo il prodotto nella giusta maniera?" e "stiamo costruendo il prodotto giusto?". In altre parole, *Verifica* significa stabilire la verità della corrispondenza tra un prodotto software e la sua specifica, e *Convalida* significa stabilire l'appropriatezza di un prodotto software rispetto alla sua missione operativa.



In un processo di questo tipo, dei Feedback possono essere inviati ad una qualsiasi delle fasi precedenti:



2.4 Modello a V



Le attività di sinistra sono collegate a quelle di destra intorno alla codifica. Se si trova un errore in una fase a destra si ri-esegue il pezzo della V collegato. Ovviamente, si può iterare migliorando requisiti, progetto, codice.

2.5 Modelli basati su prototipo

Un *prototipo* aiuta a comprendere i requisiti o a valutare la fattibilità di un approccio. In pratica, si realizza una prima implementazione (prototipo), più o meno incompleta, da considerare come una "prova", con lo scopo di accertare la fattibilità del prodotto e validare i requisiti. Dopo la fase di utilizzo del prototipo si passa alla produzione della versione definitiva del sistema SW mediante un modello che, in generale, è di tipo waterfall.
In parole povere, quindi, il prototipo è un mezzo attraverso il quale si interagisce con il committente per accertarsi di aver ben compreso le sue richieste, per specificarle meglio e per la valutare la fattibilità del prodotto. Esistono due modalità di prototipazione:

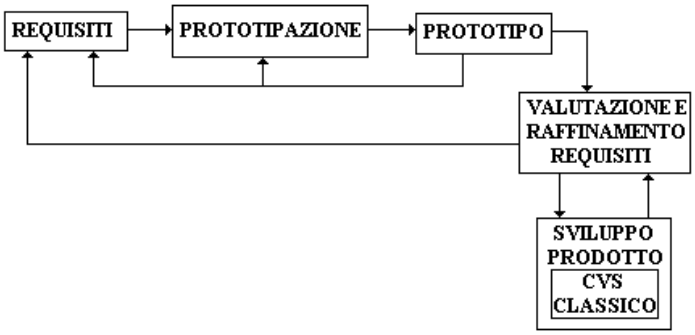
- **mock-ups**: produzione completa dell'interfaccia utente. Consente di definire con completezza e senza ambiguità i requisiti (si può, già in questa fase, definire il manuale di utente);
- **breadboards**: implementazione di sottoinsiemi di funzionalità critiche del sistema software, non nel senso della fattibilità ma in quello dei vincoli pesanti che sono posti nel funzionamento del sistema software (carichi elevati, tempo di risposta, …), senza le interfacce utente. Produce feedbacks su come implementare la funzionalità (in pratica si cerca di conoscere, prima di garantire).

E più in generale:

- *Prototipazione "throw-away"*: pervenire ad una migliore comprensione dei requisiti del prodotto da sviluppare. Lo sviluppo dovrebbe avviarsi con la parte dei requisiti meno compresa.
- *Prototipazione "esplorativa"*: pervenire ad un prodotto finale partendo da una descrizione di massima e lavorando a stretto contatto con il committente. Lo sviluppo dovrebbe avviarsi con la parte dei requisiti meglio compresa.

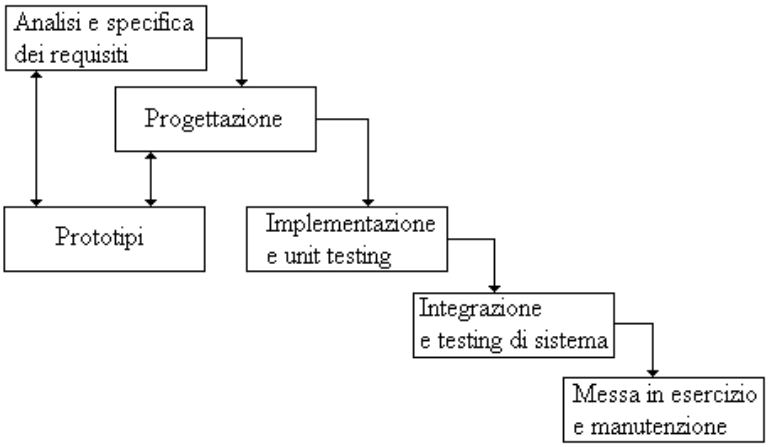
Da tenere ben presente, comunque, che il prototipo è uno strumento di identificazione dei requisiti dell'utente, pertanto è per sua natura incompleto, approssimativo e realizzato utilizzando parti già possedute o routines stub. In altre parole, dopo il suo utilizzo, il prototipo **deve** essere gettato!

Modello di prototipazione throw-away:



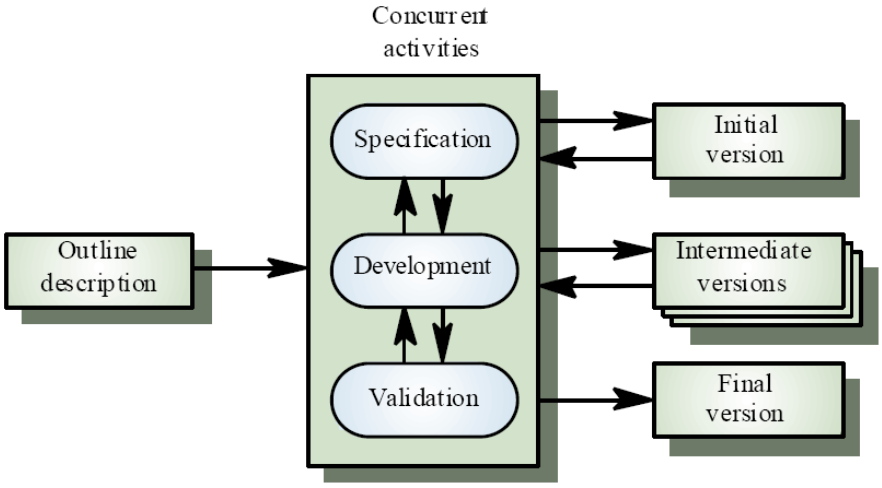
2.6 Modello Misto Cascata/Prototipi

Si utilizzano prototipi preliminari. L'obiettivo è lavorare con i clienti ed evolvere i prototipi verso il sistema finale. Si dovrebbe però avere un'idea chiara, oltre che dei requisiti ben compresi.



2.7 Sviluppo evolutivo

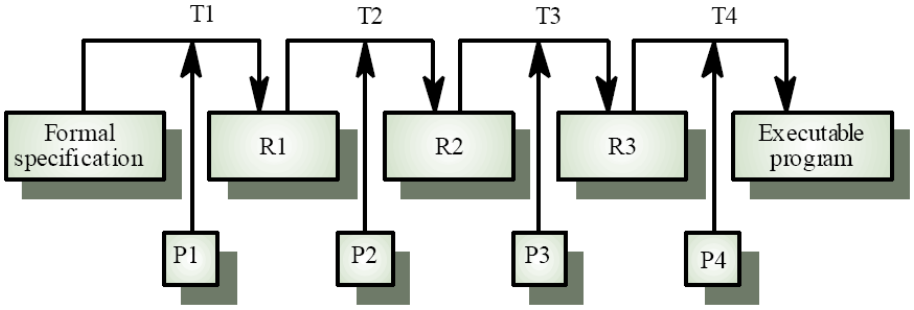
Si parte da una descrizione di massima e si sviluppa una prima versione, che viene migliorata e raffinata attraverso versioni intermedie, fino ad ottenere la versione finale:



Nell'utilizzare questo modello, però, bisogna prestare molta attenzione: si può avere perdita di visibilità del processo di sviluppo, spesso il prodotto finito è scarsamente strutturato, sono richieste competenze specifiche nell'uso di linguaggi di prototipazione rapida (RAD), si può avere perdita di visibilità del processo da parte del management.
E' un modello, tuttavia, utile per sviluppare sistemi interattivi di taglia medio-piccola, o per sviluppare parti di sistemi più grandi (ad esempio una UI), o per sviluppare sistemi con un breve ciclo di vita.

2.8 Trasformazioni formali

Si basano sulla trasformazione di una "specifica matematica" in un programma eseguibile, attraverso trasformazioni che permettono di passare da una rappresentazione all'altra, preservando la correttezza:



Nella figura, T1, T2, T3, T4 sono le trasformazioni, R1, R2, R3 sono le rappresentazioni e P1, P2, P3, P4 sono le dimostrazioni di correttezza delle corrispondenti trasformazioni.
Anche questo modello presenta particolari problematiche: richiede competenze e skill specifici per l'applicazione delle tecniche; presenta difficoltà nella specifica formale di parti del sistema (ad esempio l'interfaccia utente); ha costi di sviluppo in genere più elevati; il committente non comprende le specifiche formali. La sua applicabilità è giustificata nello sviluppo di sistemi critici (safety o security).

2.9 Modelli di sviluppo a componenti (o basati su Riuso)

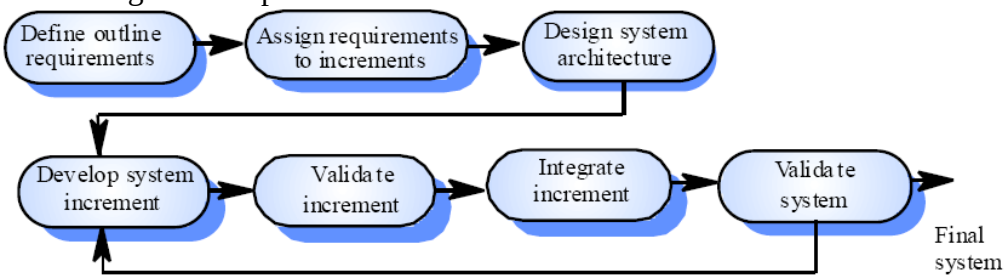
Sono modelli basati sul riuso sistematico di componenti e producono sistemi software integrati da componenti esistenti o sistemi COST (Commericial-off-the-shelf). Esempi sono il *Rapid*

application development e il *Full reuse model* (Basili, 1990). Quest'ultimo prevede repository (conservazione) di componenti riusabili a diversi livelli di astrazione, prodotti durante le diverse fasi del ciclo di vita (specifiche, progetti, codice, test case, …), e durante lo sviluppo di un nuovo sistema è previsto il riuso di componenti esistenti e il popolamento delle repository con nuove componenti. Questi modelli sono particolarmente adatti per lo sviluppo di software object-oriented.

2.10 Iterazioni del processo: Sviluppo incrementale e a spirale

I requisiti sono sempre soggetti a modifiche nel corso dello sviluppo. Questo comporta iterazioni di "rework" soprattutto nelle fasi iniziali. Tali iterazioni possono essere applicate a qualsiasi modello di processo di sviluppo. Due approcci fondamentali, in tal senso, sono lo *Sviluppo incrementale* e lo *Sviluppo a spirale*.

Lo **Sviluppo incrementale** risolve la difficoltà di produrre l'intero sistema in una sola volta, nel caso di grandi progetti SW. Queste difficoltà possono essere sia del produttore che del committente, che potrebbe non avere l'immediata disponibilità finanziaria necessaria per l'intero progetto. Il prodotto viene consegnato con più rilasci.



Le fasi del processo sono completamente realizzate e il sistema così progettato viene decomposto in *sottosistemi* (**incrementi**) che vengono implementati, testati, rilasciati, installati e messi in manutenzione secondo un piano di priorità in tempi diversi. Diventa fondamentale la fase, o l'insieme di attività, di integrazione di nuovi sottosistemi prodotti con quelli già in esercizio.
Uno dei vantaggi che si hanno nell'utilizzo del Modello incrementale è dovuto alla possibilità di anticipare da subito, alcune funzionalità al committente, perché ciascun incremento corrisponde al rilascio di una parte delle funzionalità. In tal modo i requisiti a più alta priorità per il committente vengono rilasciati per primi e si ha minore rischio di un completo fallimento del progetto. Un altro vantaggio è quello di poter effettuare un Testing più esaustivo: i rilasci iniziali agiscono come prototipi e consentono di individuare i requisiti per i successivi incrementi; i servizi a più alta priorità sono anche quelli che vengono maggiormente testati.
Basati su un approccio differente dai Modelli incrementali, ma accomunati dal prevedere più versioni successive del sistema, sono i **Modelli iterativi** (*evolutivi*): ad ogni istante dopo il primo rilascio esiste un sistema versione N in esercizio ed un sistema N+1 in sviluppo. Con lo Sviluppo incrementale, ogni versione aggiunge nuove funzionalità/sottosistemi, mentre con lo Sviluppo iterativo, da subito sono presenti tutte le funzionalità/sottosistemi che vengono successivamente raffinate e migliorate.

Con il **Modello a spirale**, si ha la formalizzazione del concetto di iterazione: il riciclo è un fondamento per questo modello. Il processo viene rappresentato come una spirale, piuttosto che come sequenza di attività, in cui ogni giro della spirale rappresenta una fase del processo. Le fasi non sono predefinite, ma vengono scelte in accordo al tipo di prodotto e ognuna di esse prevede la scoperta, la valutazione e il trattamento esplicito dei "*rischi*". Si tratta di un **meta-modello**, cioè si

ha la possibilità di usare uno o più modelli (il ciclo a cascata si può vedere come un caso particolare, con una sola iterazione). Ogni quarto della spirale si riferisce alle seguenti operazioni:

1.determinare obiettivi, vincoli ed alternative;
2.valutare le alternative, identificare e risolvere i rischi;
3.sviluppare e verificare il prossimo prodotto;
4.pianificare la fase seguente.

In un modello del genere, il compito di chi gestisce (il manager) è minimizzare i rischi, che sono di varie tipologie: personale inadeguato, scheduling, budget non realistico, sviluppo del sistema sbagliato, … Questo compito è importante perché il rischio, che tra l'altro può provocare ritardi e costi imprevisti, è insito in tutte le attività umane ed è una misura dell'incertezza sul risultato dell'attività. Inoltre è collegato alla quantità e qualità delle informazioni disponibili: meno informazione si hanno più alti sono i rischi.
Oltre al Modello incrementale e al Modello a spirale, è interessante anche l'approccio basato su **Estreme programming**. Senza scendere nei dettagli, si tratta di un approccio recente allo sviluppo del software basato su iterazioni veloci che rilasciano piccoli incrementi delle funzionalità.
Questo approccio, prevede una partecipazione più attiva del committente al team di sviluppo, e consente il miglioramento costante e continuo del codice (verifica e adeguamento in tempi estremamente ridotti).

2.11 Considerazioni conclusive

Il Modello a Cascata comporta alti rischi nello sviluppo di sistemi nuovi, non familiari per problemi di specifica e progetto. D'altra parte, i rischi sono bassi nello sviluppo di applicazioni familiari con tecnologie note.
I modelli basati su Prototipazione, invece, prevedono bassi rischi per le nuove applicazioni perché specifica e sviluppo vanno di pari passo. Prevedono, d'altro canto, rischi più alti per la mancanza di un processo definito e visibile.
Il modello Trasformazionale, infine comporta alti rischi per le tecnologie coinvolte e le professionalità richieste.
Dovendo effettuare una scelta, per sistemi o sottosistemi di cui si ha una buona conoscenza, si può adottare il modello a Cascata, perché la fase di analisi dei rischi ha costi limitati. Requisiti stabili e sistemi critici per le persone o cose (safety critical), invece, possono essere sviluppati con approcci Trasformazionali. Zone non completamente specificate e interfacce d'utente possono impiegare il modello a Prototipi.
Ognuno dei modelli visti, inoltre, comporta una differente visibilità del processo di sviluppo:

Modello	Visibilità del processo
Modello a Cascata	Buona visibilità, ogni attività produce dei rilasci.
Sviluppo Evolutivo	Poca visibilità, è sconveniente produrre documenti durante le rapide iterazioni.
Trasformazioni Formali	Buona visibilità, i documenti devono essere prodotti da ogni fase per consentire l'avanzamento del processo.
Sviluppo basato su riuso	Visibilità moderata, si può imporre la produzione di documenti che descrivono i componenti riusati e riusabili.
Modello a spirale	Buona visibilità, ogni segmento e ogni anello della spirale dovrebbero produrre qualche documento.

Esempio: Documenti del Modello a cascata

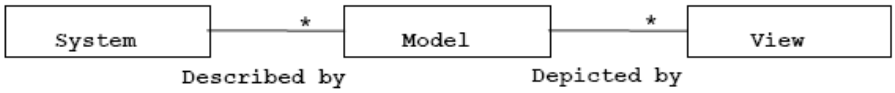
Activity	**Output documents**
Requirements analysis	Feasibility study, Outline requirements
Requirements definition	Requirements document
System specification	Functional specification, Acceptance test plan Draft user manual
Architectural design	Architectural specification, System test plan
Interface design	Interface specification, Integration test plan
Detailed design	Design specification, Unit test plan
Coding	Program code
Unit testing	Unit test report
Module testing	Module test report
Integration testing	Integration test report, Final user manual
System testing	System test report
Acceptance testing	Final system plus documentation

3.1 Terminologia

La *modellazione* consiste nella costruzione di un'astrazione della realtà. Le astrazioni sono delle semplificazioni, perché ignorano dettagli irrilevanti e rappresentano sono quelli rilevanti. Ovviamente, cosa è rilevante o irrilevante dipende dall'obiettivo del modello. Il motivo per il quale si modella il software è dovuto a vari fattori:

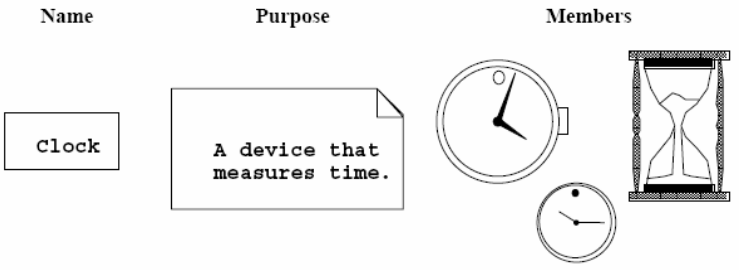
- il software è divenuto sempre più complesso: Windows XP, ad esempio, è costituito da 40 milioni di linee di codice e, un singolo programmatore non può gestire questo ammontare di codice in tutta la sua interezza;
- un codice non è facilmente comprensibile dagli sviluppatori che non lo hanno scritto;
- abbiamo bisogno di rappresentazioni più semplici per sistemi complessi.

Un **sistema** è un insieme organizzato di parti comunicanti, ognuna delle quali è un sottosistema. Un **modello** è un'astrazione che descrive un sottoinsieme di un sistema. Una **vista** (**view**) descrive particolari aspetti di un modello. Una **notazione** è un insieme di regole grafiche o testuali per descrivere viste. Viste e modelli di un singolo sistema possono sovrapporsi l'uno con l'altro.
In UML:



Di seguito riportiamo una carrellata di definizioni, che ci servono per acquisire una certa terminologia riguardante la modellazione a oggetti.

Un **fenomeno** è un oggetto nel mondo di un dominio che possiamo percepire (ad es. "La lezione a cui stiamo assistendo"). Un **concetto**, invece, descrive le proprietà dei *fenomeni* che sono comuni (ad es. "Lezione di Ingegneria del Software"). I concetti sono delle 3-tuple, hanno infatti un *Nome* (per poterli distinguere da altri concetti), un *Obiettivo* (proprietà che determinano se un fenomeno è un membro di un concetto), dei *Membri* (l'insieme dei fenomeni che sono parte del concetto).

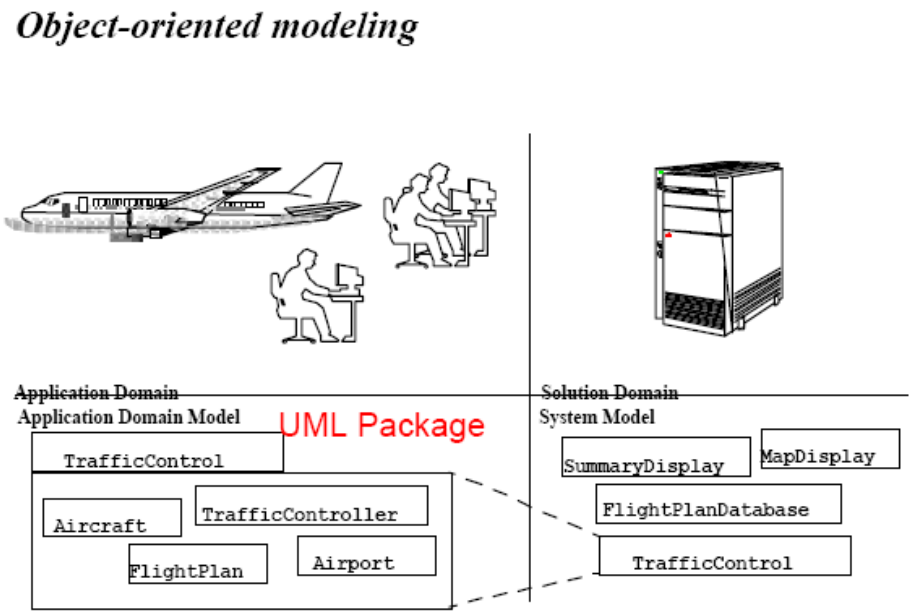


Possiamo allora definire l'**Astrazione** come una classificazione dei fenomeni in concetti e la **Modellazione** come lo sviluppo di astrazioni per rispondere a specifiche domande a riguardo di un insieme di fenomeni, ignorando dettagli irrilevanti.

Riferendoci al software, un **Tipo** è un'astrazione nel contesto dei linguaggi di programmazione, ha un nome unico per distinguersi da altri tipi e denota un insieme di valori che sono i membri del tipo di dati (Nome: int, Obiettivo: numero intero, Membri: 0, -1, 1, 2, -2, …). Un membro di un tipo

specifico si dice **Istanza**. Quindi, il tipo di una variabile rappresenta tutte le possibili istanze che la variabile può assumere. Possiamo allora concludere che Tipo e Istanze corrispondono a Concetti e Fenomeni.
Un **Tipo di dati Astratto** è un tipo speciale, la cui implementazione è nascosta dal resto del sistema e una **Classe** è un'astrazione nel contesto dei linguaggi orientati agli oggetti. Come un tipo di dati astratto, una classe incapsula sia lo *stato* (variabili) che il *comportamento* (metodi) (es.: Classe Vector). Diversamente dai tipi di dati astratti, le classi possono essere definite in termini di altre classi attraverso il meccanismo dell'**ereditarietà**.

Il **Dominio Applicativo** (Analisi dei Requisiti) è l'ambiente in cui opera il sistema. Il **Dominio delle Soluzioni** (System Design, Object Design) è l'insieme delle tecnologie disponibili per costruire il sistema.



3.2 Cosa è UML

UML è l'acronimo di **Unified Modeling Language**, ed è uno standard emergente per la modellazione del software orientato agli oggetti. E' il risultato della convergenza delle notazioni dei 3 principali metodi object-oriented: *OMT* (*James Rumbaugh*), *OOSE* (*Ivar Jacobson*), *Booch* (*Grady Booch*). E' supportato da diversi tool CASE, come *Rational ROSE* e *TogetherJ*. Con UML si può modellare l'80% della maggioranza dei problemi usando circa solo il 20% di UML.

Permette di costruire:

- **Use Case Diagrams**: descrivono le funzionalità del sistema dal punto di vista dell'utente;
- **Class Diagrams**: descrivono la struttura statica del sistema (oggetti, attributi, associazioni);
- **Sequence Diagrams**: descrivono il comportamento dinamico tra gli attori e il sistema e, tra gli oggetti e il sistema;
- **State Chart Diagrams**: descrivono il comportamento dinamico di oggetto individuale (essenzialmente sono macchine a stati finiti);
- **Activity Diagrams**: modellano i comportamenti dinamici di un sistema, in particolare il workflow (essenzialmente sono flowchart).

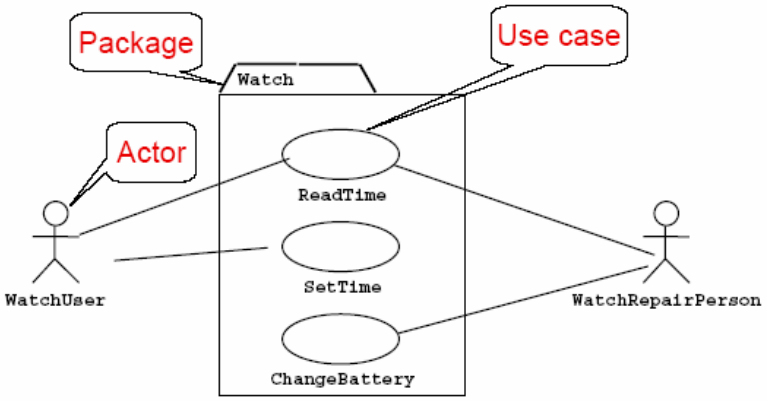
Ulteriori notazioni previste da UML saranno introdotte successivamente: *Implementation Diagrams* (*Component Diagrams* e *Deployment Diagrams*) e *Object Constraint Language* (*OCL*).

In UML si utilizzano le seguenti convenzioni:

- i rettangoli sono classi o istanze;
- gli ovali sono funzioni o casi d'uso;
- le istanze sono denotate con i nomi sottolineati (`myWatch:SimpleWatch`);
- i tipi sono denotati con nomi non sottolineati (`SimpleWatch`);
- i diagrammi sono grafi in cui i nodi sono le entità e gli archi sono relazioni tra le entità.

3.3 Use Case Diagrams

Sono usati durante la fase di requirements elicitation (raccolta dei requisiti) per rappresentare il comportamento esterno. L'attore rappresenta i tipi di utenti del sistema. I casi d'uso rappresentano una sequenza di interazioni per un tipo di funzionalità. Il **modello dei caso d'uso** è l'insieme di tutti i casi d'uso, ed è una completa descrizione delle funzionalità del sistema e del suo ambiente.



Un **attore** modella un'entità esterna che comunica col sistema: utenti, sistemi esterni, ambiente fisico. Ha un nome unico e una descrizione opzionale. (Es.: Passeggero: Una persona nel treno).
Un **caso d'uso** rappresenta una classe di funzionalità fornite dal sistema mediante un flusso di eventi e consiste di un nome unico, attori partecipanti, condizioni d'entrata, flusso di eventi, condizioni di uscita, requisiti speciali.

Esempio di Caso d'uso:

Name: `Purchase ticket`

Participating actor: `Passenger`

Entry condition:
- `Passenger` standing in front of ticket distributor.
- `Passenger` has sufficient money to purchase ticket.

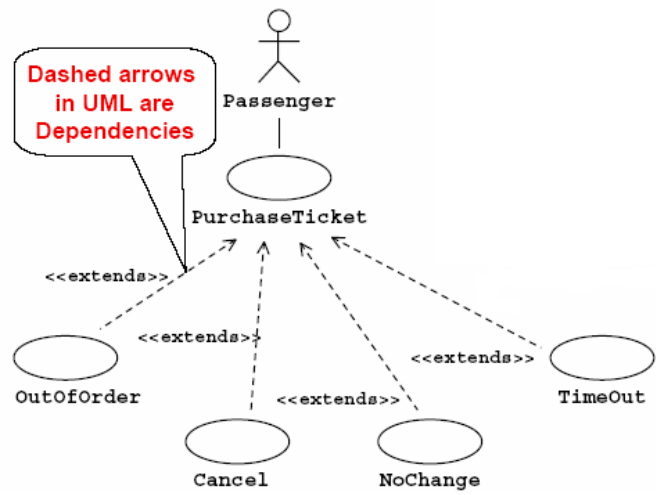
Exit condition:
- `Passenger` has ticket.

Event flow:
1. `Passenger` selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. `Passenger` inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

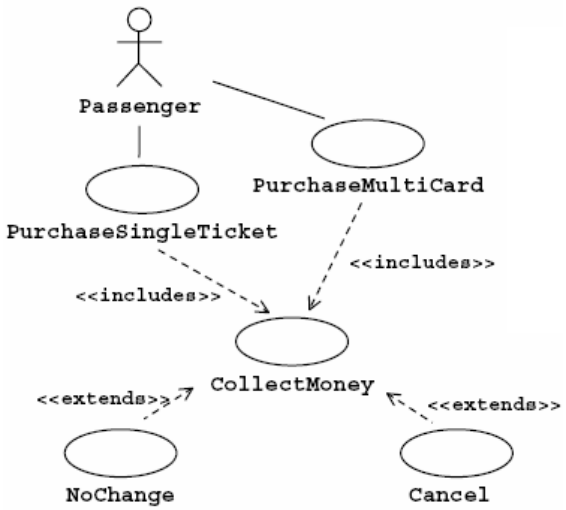
Anything missing?

Exceptional cases!

Un caso d'uso può estenderne un altro aggiungendo eventi. La relazione `<<extends>>` rappresenta, quindi, casi invocati eccezionalmente o raramente, cioè indica che il caso d'uso **può** includere il comportamento di un altro caso d'uso. Il flusso di eventi eccezionali è tratto dal flusso principale, per chiarezza. I casi d'uso che rappresentano flussi eccezionali possono estendere più di un caso d'uso e la direzione di una relazione `<<extends>>` è verso il caso d'uso esteso.
Un caso d'uso che estende, viene indicato nella entry condition.

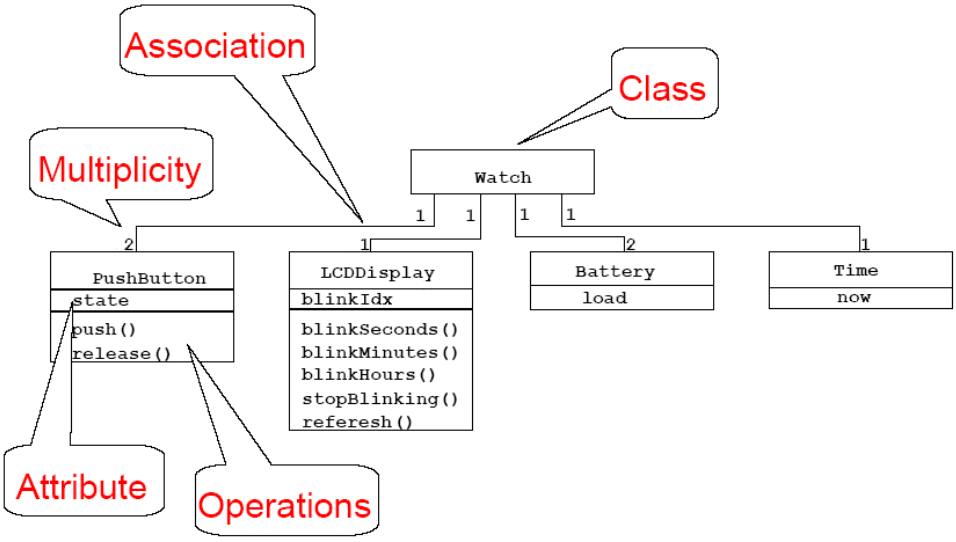


La relazione `<<includes>>` rappresenta il comportamento che è tratto dal caso d'uso per il riuso, non perché è un eccezione. La sua direzione è verso l'utilizzo del caso d'uso. Serve a eliminare ridondanza: più casi d'uso hanno lo stesso comportamento, allora se ne può creare uno che rappresenta questo comportamento e che viene incluso dagli altri nei loro flussi di eventi. Il caso d'uso incluso si indica nei requisiti speciali.



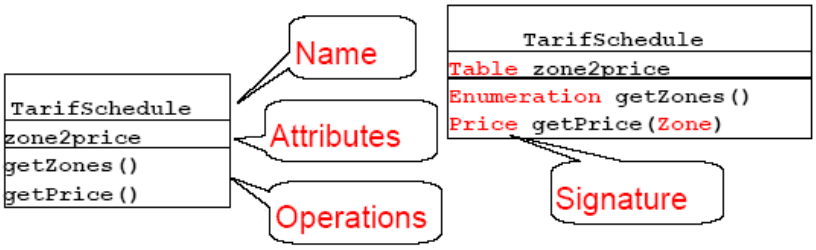
Riassumendo, quindi, i *Diagrammi dei Casi d'uso* rappresentano i comportamenti esterni e sono utili come indici dei casi d'uso. Le descrizioni dei casi d'uso forniscono il succo del modello, non i Diagrammi dei Casi d'uso. Tutti i casi d'uso hanno bisogno di essere descritti per fare in modo che il modello sia utile.

3.4 Class Diagrams

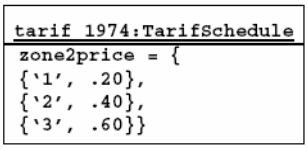


Il Class Diagrams rappresenta la struttura del sistema e viene usato: durante l'analisi dei requisiti per modellare i concetti del dominio del problema; durante la progettazione del sistema per modellare i sottosistemi e le interfacce; durante l'object design per modellare le classi.

Una classe rappresenta un concetto e incapsula stati (attributi) e comportamenti (operazioni). Ogni attributo ha un tipo, ogni operazione ha una signature e il nome della classe è la sola informazione obbligatoria.



Un'istanza, invece rappresenta un fenomeno, ha un nome sottolineato e può contenere la classe dell'istanza. Gli attributi sono rappresentati con i loro valori.

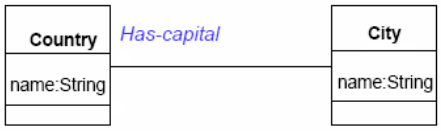


A questo punto nasce spontanea la domanda: qual è la differenza tra attore, classe e istanza?

Un attore è un'entità esterna al sistema da modellare e che interagisce con esso. Una classe è un'astrazione che modella un'entità nel dominio del problema e deve essere modellata all'interno del sistema. Un oggetto è una specifica istanza di una classe.

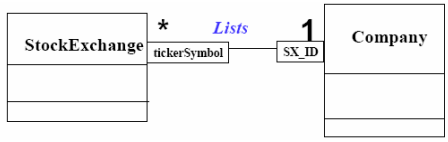
Per indicare relazioni tra le classi si usano le **Associazioni**. La *molteplicità* indicata all'estremità di un'associazione denota quanti riferimenti ad altri oggetti può avere l'oggetto sorgente.

Associazione **1-a-1** Associazione **1-a-Molti**



Associazione **Molti-a-Molti**

Associazione **Molti-a-1**



Collegamenti (**Link**) e **Associazioni** stabiliscono delle relazioni tra oggetti e classi. Un *Link* è una connessione tra due oggetti, mentre un'*Associazione* è sostanzialmente è un *mapping* bidirezionale (1-a-1, molti-a-1, …) e descrive un insieme di link allo stesso modo in cui una classe descrive un insieme di oggetti.

Esempio

Definizione del Problema: Uno stock exchange (borsa valori) ha una lista di molte compagnie. Ogni compagnia è identificata univocamente da un ticker symbol.

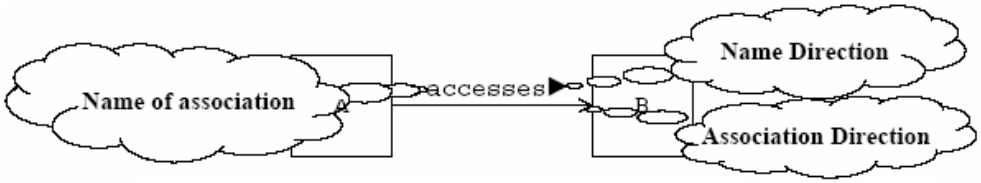
Class Diagram

Java Code

```
public class StockExchange
{
 private Vector m_Company = new Vector();
};

public class Company
{
 public int m_tickerSymbol;
 private Vector m_StockExchange = new Vector();
};
```

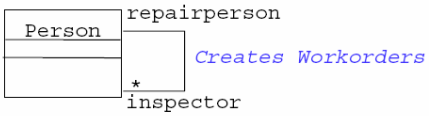
Un'associazione tra due classi è per default un mapping bidirezionale: la classe A può accedere alla classe B e viceversa (entrambe le classi giocano il ruolo di agente). Se vogliamo far diventare A un client e B un server, si può far diventare l'associazione unidirezionale. La freccia punta al server:



La Classe A (il "client") accede alla classe B (il "server"). B è anche detta *navigabile*.

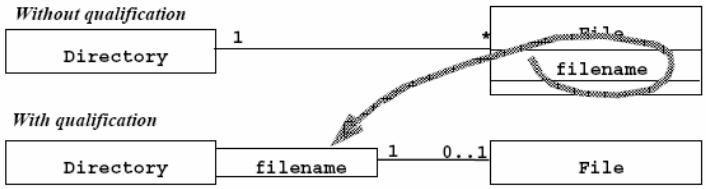
Ogni lato di un'associazione può essere etichettato con un ruolo. Il **nome di un ruolo** è il nome che identifica univocamente un lato dell'associazione. E' scritto accanto alla linea dell'associazione vicino alla classe che ha tale ruolo. Li usiamo necessariamente per le associazioni tra due oggetti della stessa classe e per distinguere tra due associazioni tra la stessa coppia di classi. Non li usiamo

invece se c'è solo una singola associazione tra una coppia di classi distinte, perché i nomi delle classi bastano allo scopo.
Ad esempio, una persona assume il ruolo di riparatore rispetto a un'altra persona, che assume il ruolo di ispettore rispetto alla prima:

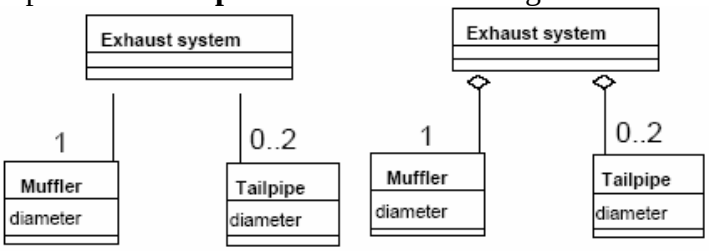


- *Ruolo Client*: Un oggetto che può operare sotto altri oggetti ma che non è mai operato da altri.
- *Ruolo Server*: Un oggetto che non opera mai sotto altri oggetti, ma che è operato da altri.
- *Ruolo Agent*: Un oggetto che gioca il ruolo di Client e Server ed è di solito creato per eseguire del lavoro per mezzo di un attore o di altro agente.

Per ridurre la molteplicità di un'associazione si possono usare i **qualificatori**:

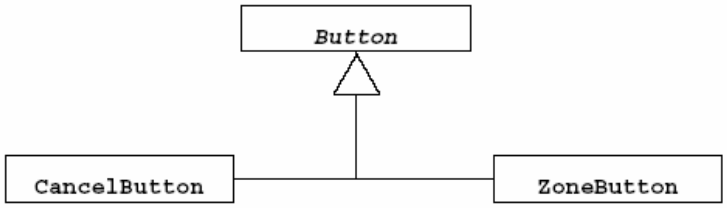


Un **aggregazione** è un caso speciale di associazione che denota una gerarchia di "*consiste di*".
L'**aggregato** è la classe padre e le **componenti** sono le classi figlie:



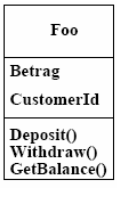
Un *diamante pieno* indica una **composizione**, una forma forte di aggregazione dove i componenti non possono esistere senza l'aggregato.

L'**ereditarietà** semplifica il modello eliminando ridondanza:

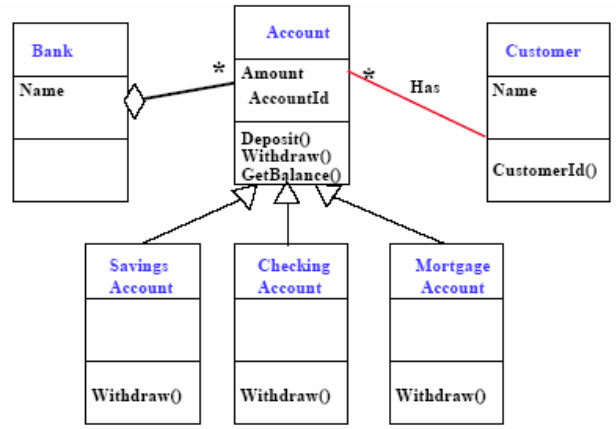


Le classi figlie ereditano gli attributi e le operazioni della classe padre.

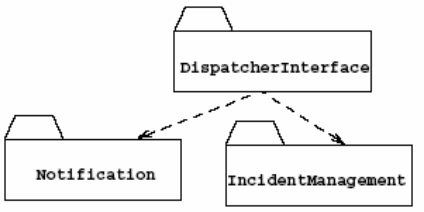
A questo punto siamo in grado di costruire un **modello a oggetti**.
L'identificazione di una classe avviene grazie a Nome, Attributi e Metodi.



Il nome è molto importante, è bene sceglierlo accuratamente (Foo è il nome giusto?). Una volta individuata la classe, trovare nuovi oggetti, ed eventualmente iterare nomi, attributi e metodi. Successivamente trovare le associazioni tra gli oggetti, etichettare le associazioni e determinarne la molteplicità. Iterare più volte il discorso:

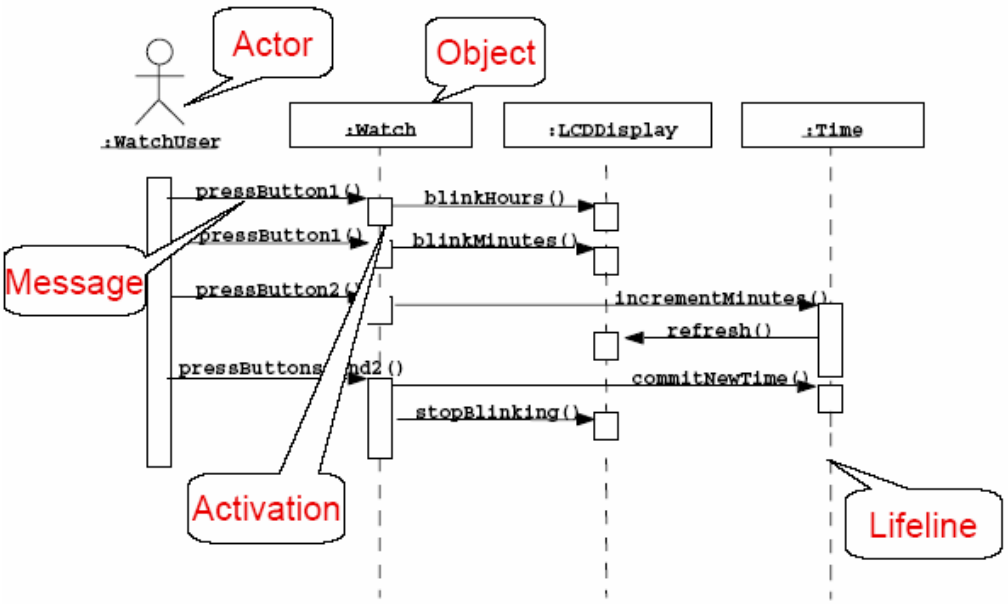


Un **Package** è un meccanismo di UML per organizzare gli elementi in gruppi (di solito non è un concetto del dominio applicativo). In pratica, i package sono dei raggruppamenti con cui è possibile organizzare i modelli UML per aumentare la loro leggibilità.

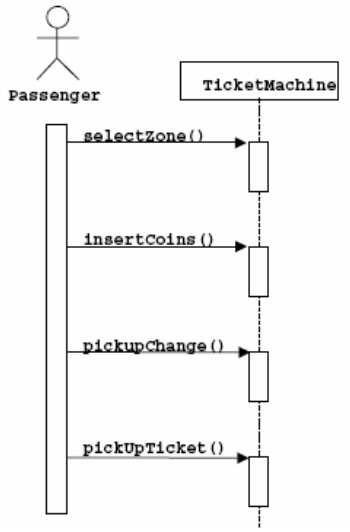


Un sistema complesso può essere decomposto in sottosistemi, ognuno dei quali è modellato con un package.

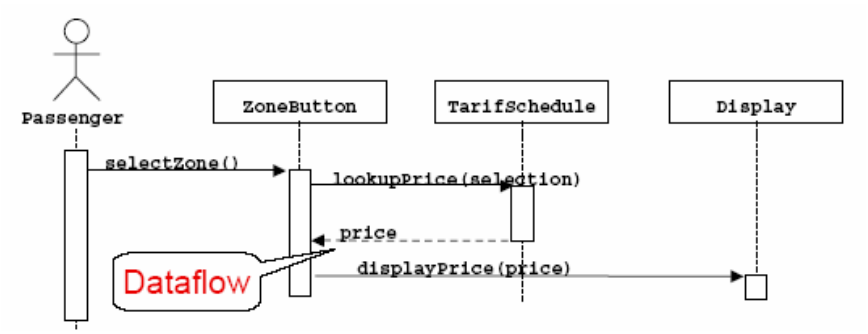
3.5 Sequence Diagram



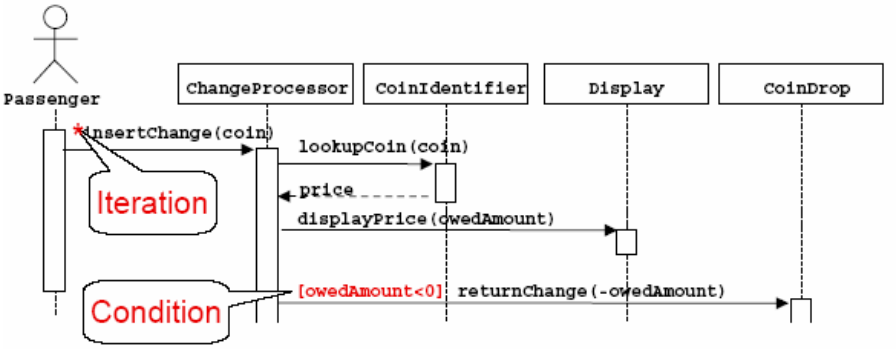
Rappresentano il comportamento mediante interazioni e sono usati durante l'analisi dei requisiti per raffinare le descrizioni dei casi d'uso e per trovare nuovi oggetti, e durante il system design per raffinare le interfacce dei sottosistemi. Le classi sono rappresentate da colonne, i messaggi da frecce, le attivazioni da rettangoli stretti e le lifeline da linee tratteggiate.



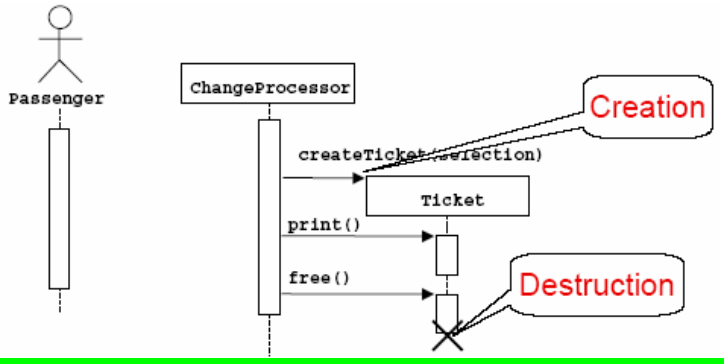
La sorgente di una freccia indica l'attivazione che ha inviato il messaggio. Un attivazione è lunga quanto tutte le attivazioni annidate. Le linee tratteggiate orizzontali indicano il flusso dei dati e quelle verticali le lifelines.



Le iterazioni sono denotate da un * che precede il nome del messaggio e le condizioni sono denotate da espressioni booleane tra [ ] precedenti il nome del messaggio.



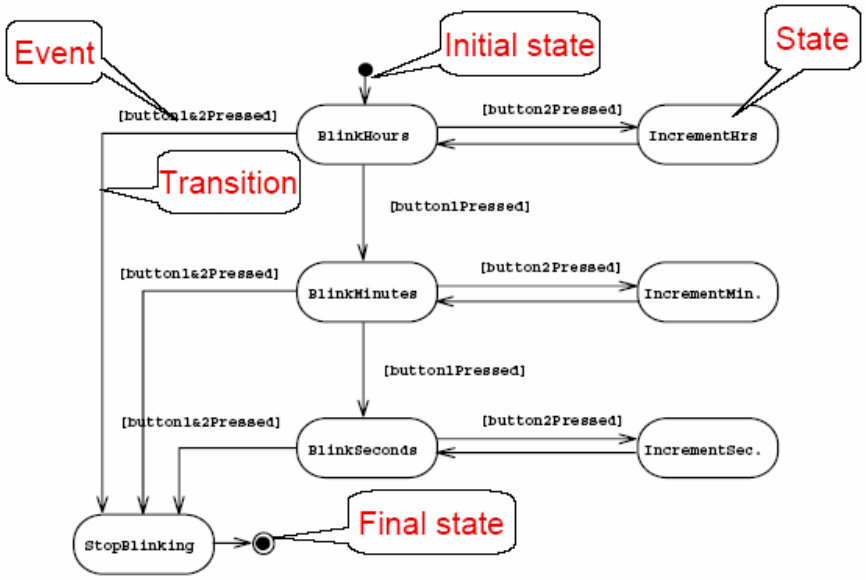
La creazione è denotata con una freccia che punta all'oggetto e la distruzione con un marcatore X alla fine della distruzione dell'attivazione. In ambienti di garbage collection, la distruzione può essere usata per indicare la fine della vita utile di un oggetto.



Riassumendo, quindi, un *Sequence Diagram UML* rappresenta il comportamento in termini di interazioni ed è utile per trovare gli oggetti mancanti. In altre parole completa il class diagram (che rappresenta la struttura). E' dispendioso in termini di tempo ma vale la pena costruirlo.

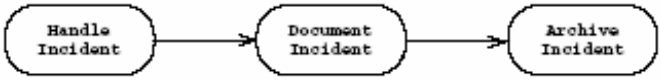
3.6 State Chart Diagram

Rappresenta il comportamento mediante stati e transizioni.



3.7 Activity Diagram

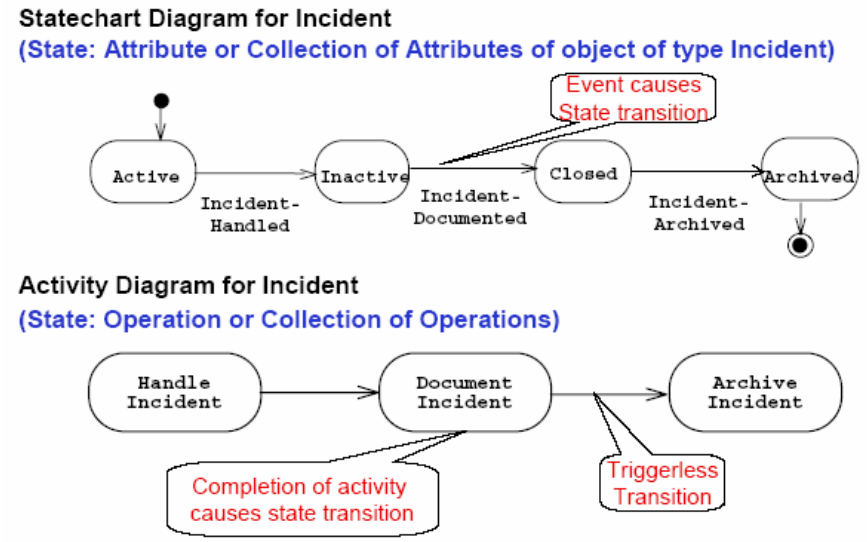
Un activity diagram mostra il flusso di controllo all'interno del sistema.



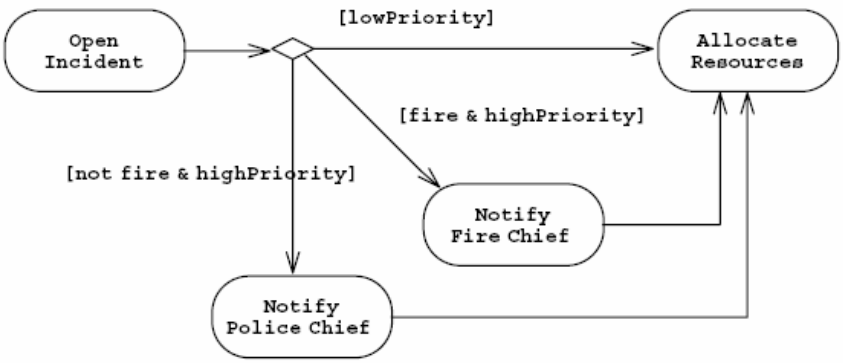
E' un caso speciale di uno *State Chart Diagram* in cui gli stati sono le attività ("funzioni"). Possiamo averne di due tipi:

- **Action state**: non può essere decomposto ulteriormente e rappresenta gli avvenimenti istantanei rispetto al livello di astrazione usato nel modello;
- **Activity state**: può essere decomposto ulteriormente e l'attività è modellata da un altro activity diagram.

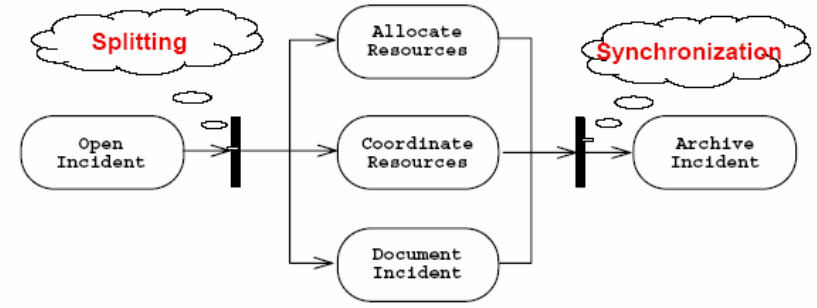
State Chart e Activity Diagram a confronto:



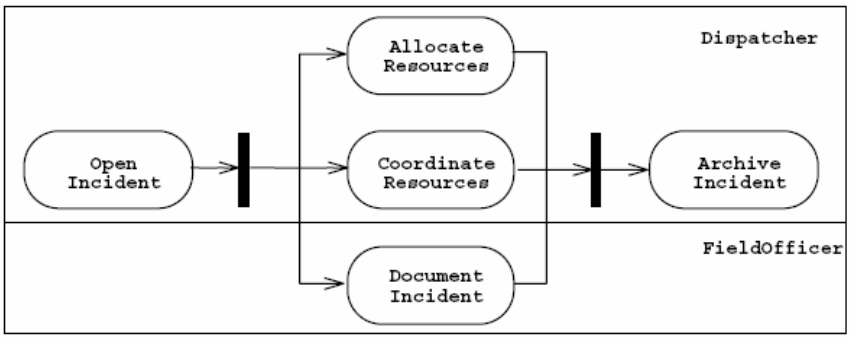
Con l'activity diagram si modellano le **decisioni**:



si modella la **concorrenza**, *sincronizzando* attività multiple e usando lo *splitting* del flusso di controllo in thread multipli:



Inoltre le azioni possono raggruppate in **swimlanes**, in modo da indicare gli oggetti o il sottosistema che implementa l'azione:



3.8 Considerazioni conclusive

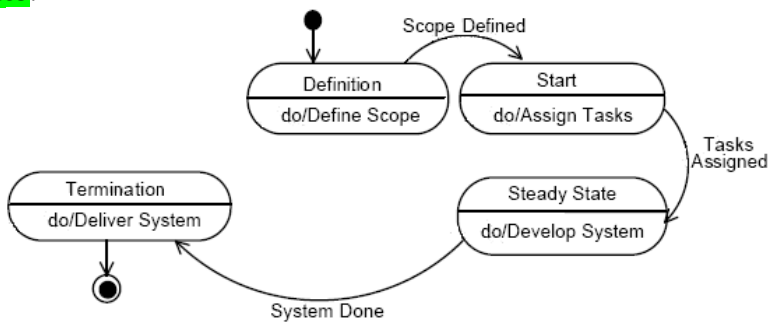
Cosa dovrebbe essere fatto prima? Codice o Modello? E' relativo! Con una Forward Engineering si crea il codice dal modello, usando una progettazione detta a *Greenfield*, cioè, che parte da zero e si può spaziare ovunque. Con una Reverse Engineering si crea il modello dal codice e la progettazione è detta a Interfacce o *Re-ingegneristica*. Con una Roundtrip Engineering si realizza un misto delle prime due ed è utile quando i requisiti, le tecnologie e gli schedule cambiano frequentemente.
In ogni caso, UML è un ottimo strumento per ottenere i modelli. fornisce una vasta varietà di notazioni per rappresentare molti aspetti dello sviluppo software: è potente, ma il linguaggio è complesso; se usato in modo improprio può generare modelli illeggibili; si possono commettere facilmente errori se si usano troppe caratteristiche "esotiche" (è meglio usarlo nella maniera più facile).

4. ORGANIZZAZIONE DEL PROGETTO E COMUNICAZIONE

4.1 Organizzazione del Progetto

Le componenti di un progetto sono:

- **Work product**: ogni elemento prodotto dal progetto, come un pezzo di codice, modelli, o documenti. Questi lavori prodotti per il cliente sono chiamati **deliverables** (consegne).
- **Schedule**: specifica quando i lavori sul progetto dovrebbero essere portati a termine.
- **Partecipanti**: ogni persona partecipante al progetto. A volte queste persone sono dette *membri del progetto* (*project member*).
- **Task**: il lavoro che deve essere svolto dai partecipanti al progetto per creare un work product.
- **Definizione del Progetto**: formale (ad esempio, si firma un contratto) o informale (ad esempio, si promette di fare qualcosa).
- **Tipo del progetto**: dipende soprattutto dai deliverables (ad esempio, progetto software o progetto del sistema).
- **Dimensione del progetto**: da piccola a grande.
- **Stati del progetto**.



Un'aspetto particolarmente importante, inoltre, è la **Comunicazione**: nello sviluppo di grandi sistemi, si spenderà più tempo nella comunicazione che nella codifica. A tal proposito, UML permette ai partecipanti al progetto di costruire modelli del sistema e discutere (comunicare) a riguardo. Questa comunicazione può essere pianificata (schedulata) o meno.
La *Comunicazione pianificata* (schedulata) serve a diffondere le informazioni che i partecipanti devono usare (ispezione del problema, meeting di stato, peer review, client e project review, realease). La *Comunicazione non pianificata*, invece, serve in momenti critici e in casi in cui si ha bisogno di ulteriori informazioni (richieste di chiarimenti, richieste di cambi, emettere una risoluzione).
Comunque, i modelli non sono la sola informazione di cui si ha bisogno quando si comunica, in un progetto: un ingegnere del software ha bisogno di imparare i cosiddetti *soft skills* (scrittura di testi tecnici, leggere documentazione, comunicare, collaborare, gestire, presentare).
In generale, durante la fase iniziale del progetto, bisogna:

- *tenere un meeting iniziale*: i partecipanti al progetto apprendono dal cliente il problema che deve essere risolto e lo scopo del sistema che deve essere sviluppato;
- *costruire un team*: i partecipanti sono assegnati a un team sulla base delle loro capacità e dei loro interessi;
- *tenere sessioni di training*: i partecipanti che non sono preparati per eseguire le attività richieste vengono ulteriormente "addestrati";
- *costruire l'infrastruttura di comunicazione*: questa supporta eventi di comunicazione pianificata e non;

- *estendere l'infrastruttura di comunicazione*: vengono stabilite liste di bollettini addizionali e un portale del team per il particolare progetto;
- *tenere un primo meeting sullo stato del team*: i partecipanti al progetto sono tenuti a condurre meeting di stato, memorizzare tali informazioni e diffonderle agli altri membri del progetto;
- *comprendere la review (revisione) dello schedule*: contiene un insieme di milestone di alto livello per comunicare i risultati del progetto in forma di review (revisioni) al project manager e al cliente.

L'organizzazione del progetto, inoltre, può essere basata su team o in maniera gerarchica.

Un **team** è un piccolo insieme di partecipanti al lavoro sulla stessa attività o compito. Un **gruppo**, invece, è un insieme di persone che sono assegnate a un compito comune, ma che lavorano individualmente, senza alcun bisogno di comunicazione per portare a termine la porzione di lavoro loro assegnata. Un **comitato**, infine, è composto da persone che lavorano insieme per revisioni, portare critiche e proposte. Generalmente ci sono tre modalità di interazione, in una organizzazione **team-based**:

- **Reporting**: è usata per fare resoconti sulle informazioni di stato. Ad esempio, uno sviluppatore informa, mediante un rapporto, un altro sviluppatore che è pronta una *API (Application Programmer Interface)*, oppure il leader di un team informa il project manager, sempre mediante un rapporto, che un task (compito) assegnato non è stato ancora completato.

- **Decision**: è usata per diffondere decisioni. Ad esempio, il leader di un team decide che uno sviluppatore deve pubblicare una API, un project manager decide che una consegna deve essere affrettata, oppure un tipo di decisione è la risoluzione di un problema.

- **Comunicazione**: è usata per scambiare tutti gli altri tipi di informazioni necessarie per prendere decisioni, o che sono utili allo stato. Ad esempio, lo scambio dei requisiti o dei modelli di progetto o la creazione di un argomento di supporto a un *proposal* (proposta). Anche un invito a pranzo è una comunicazione.

In una organizzazione gerarchica, invece, la struttura dei report è gerarchica: sia le informazioni di stato che le decisioni sono unidirezionali. La decisioni sono sempre prese dalla "radice" dell'organizzazione e trasmesse, mediante le associazioni, alle "foglie" dell'organizzazione. Le informazioni di stato sono generate dalle "foglie" e comunicate alla "radice" mediante rapporti.

Questa organizzazione però comporta dei problemi. Alcune decisioni tecniche necessitano di essere prese localmente dagli sviluppatori, ma dipendono da informazioni conosciute da altri sviluppatori di altri team. Usando questa struttura per comunicare le decisioni e i progressi, il processo può rallentare sensibilmente. La soluzione è di scambiare informazioni attraverso una struttura di comunicazione addizionale, che permette ai partecipanti di comunicare direttamente l'uno con l'altro. In questi casi, spesso la comunicazione è delegata a uno sviluppatore, chiamato **liaison** (legame, connessione), che è responsabile di portare le informazioni avanti e indietro (struttura di comunicazione **liaison-based**). In una struttura del genere di solito è prevista la presenza di un team di liaison, detto *Cross-functional team* che non lavora direttamente su un sottosistema.
Se gli sviluppatori hanno la possibilità di comunicare direttamente tra loro, la struttura di comunicazione e dell'organizzazione è detta **peer-based**.

A prescindere dall'organizzazione scelta, ogni partecipante al progetto ha un particolare ruolo.

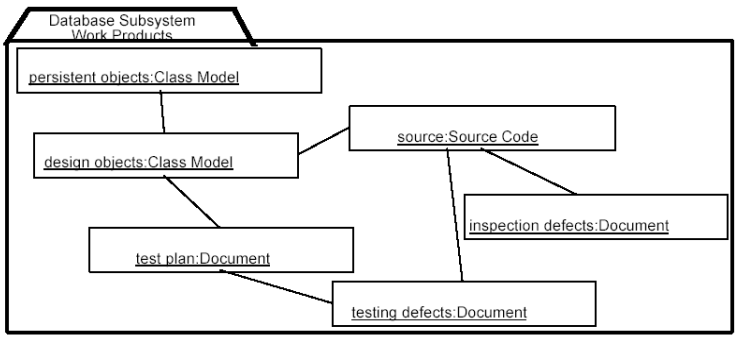
Un **ruolo** definisce l'insieme dei compiti tecnici e manageriali assunti da un partecipante o da un team. In una organizzazione team-based i compiti sono assegnati alle persone o ai team in base ai ruoli.

I tipi di ruoli in un progetto software possono essere di:

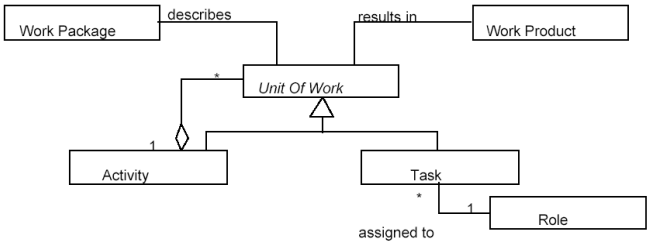
- *Management*: riguardanti l'organizzazione e l'esecuzione del progetto, all'interno di vincoli (ad esempio, project manager e team leader);
- *Development*: riguardanti la specifica, la progettazione e la costruzione dei sottosistemi. Questi ruoli includono l'analista, l'architetto di sistema, il progettista a oggetti, l'implementatore e il tester;
- *Cross-functional (liaison)*: riguardanti il coordinamento tra i team (API Engineer, Document Editor, Configuration Manager, Tester);
- *Consulant (consulenza)*: riguardanti il provvedimento di supporti temporanei in settori in cui i partecipanti al progetto sono poco esperti (cliente, utente finale, esperti del dominio applicativo ed esperti del dominio delle soluzioni).

Sulla base dei ruoli, i partecipanti al progetto lavoreranno ai task. Un **task** è un assegnamento ben definito di un lavoro per un ruolo. Il project manager o il leader di un team assegna un task a un ruolo. Il partecipante cui è assegnato il ruolo lavora al task, e il manager monitorizza i suoi progressi e il completamento. Gruppi di task correlati sono detti **attività**.
L'elemento tangibile, risultato da un task è detto **work product** (un object model, un class diagram, un pezzo di codice, un documento o parti di documenti, …), che è soggetto a deadline e suggerisce altri task. Rappresenta un importante artefatto manageriale: si può accertare il loro sviluppo, e l'inizio dei task dipende da altri work product. Ogni work product che deve essere sviluppato per il cliente è detto **deliverable** (consegna). Un sistema software, e l'allegata documentazione, di solito costituisce un insieme di deliverable. I work product che non visibili dal cliente, invece, sono chiamati **internal work product**.

Esempio di work product:



Tutta la specifica del lavoro che deve essere effettuato per completare un task o un'attività è descritto in un **work package**, che è costituito da: nome e descrizione del task, risorse necessarie alla sua realizzazione, dipendenze sull'input (work product prodotti da altri task) e sull'output (work product prodotti dal task in questione).



L'organizzazione dei task avviene in base allo **Schedule**, che è il *mapping* dei task nel tempo: a ogni task è assegnato un tempo di inizio e uno di fine, in modo da pianificare le *deadline* per ogni deliverable. Le due notazioni a diagrammi più usate per lo schedule sono i diagrammi *PERT* e i diagrammi di *Gantt*. Un diagramma di Gantt è un grafico a barre in cui l'asse orizzontale rappresenta il tempo e quello verticale i vari task da eseguire. Un diagramma PERT rappresenta uno schedule come un grafo aciclico di task. L'inizio e la durata del task sono usati per computare il percorso critico (*critical path*), che rappresenta il percorso più lungo possibile nel grafo. La lunghezza di questo path corrisponde allo schedule più corto possibile, assumendo sufficienti risorse per portare a termine, in parallelo, task indipendenti.
Chiaramente, i task sul critical path sono quelli più importanti, perché un ritardo in ognuno di questi task produrrà un ritardo in tutto il progetto.

4.2 Modalità di Comunicazione

La **modalità di comunicazione** è il tipo di informazione scambiata che definisce obiettivi e scopi. Può essere **schedulata** e quindi si tratta di una comunicazione pianificata, oppure **event driven** (guidata da eventi), e quindi non pianificata.

In una **comunicazione schedulata**, si possono utilizzare i seguenti:

- **Definizione del problema**: di solito è schedulata all'inizio del progetto e ha come obiettivo quello di presentare gli scopi, i requisiti e i vincoli. (Esempio: presentazione del cliente).
- **Project Review**: sono focalizzati sul modello del sistema e vengono schedulati intorno alle milestone del progetto e ai deliverable. Il loro obiettivo è verificare lo stato e revisionare il modello e la decomposizione del sistema e le interfacce del sottosistema. (Esempi: Analysis Review, System Design Review).
- **Client Review**: sono focalizzati sui requisiti e di solito schedulati dopo la fase di analisi. L'obiettivo è riassumere al cliente le variazioni dei requisiti.
- **Walkthrough**: si tratta di Peer Review informali e devono essere schedulati da ogni team. L'obiettivo è migliorare la qualità del sottosistema (Esempio: gli sviluppatori presentano il sottosistema ai membri del team, informale, peer-to-peer).
- **Inspection**: si tratta di Peer Review formali, schedulati dal project manager, ed hanno come obiettivo l'accettazione dei requisiti. (Esempio: Test di accettazione del cliente, mediante una sorta di dimostrazione del sistema finale all'utente).
- **Status Review**: schedulati ogni settimana, con l'obiettivo di individuare scostamenti dallo schedule e correggerli, o identificare nuovi problemi. (Esempio: sessioni di stato nei regolari meeting settimanali).
- **Brainstorming**: schedulati ogni settimana, con l'obiettivo di generare e valutare un gran numero di soluzioni al problema. (Esempio: sessioni di discussione nei regolari meeting settimanali).
- **Release**: di solito schedulate dopo ogni fase, hanno l'obiettivo di fornire come *Baseline* il risultato di ogni attività di sviluppo software. Includono Software Project Management Plan

(SDMP), Requirement Analysis Document (RAD), System Design Document (SDD), Object Design Document (ODD), Test Manual (TM), User Manual (UM).

- **Postmortem Review**: schedulata alla fine del progetto, ha come obiettivo la descrizione delle "lezioni" imparate (errori commessi, tecniche apprese, …).

In una **comunicazione Event driven**, invece, si utilizzano i seguenti:

- **Request for clarification**: le richieste di chiarimenti rappresentano la grossa quantità di comunicazione tra sviluppatori, clienti e utenti. (Esempio: uno sviluppatore può richiedere un chiarimento a riguardo di una frase nel problem statement).
- **Request for change:** un partecipante riporta un problema e propone una soluzione. Le richieste di cambiamento sono spesso formalizzate quando la dimensione del progetto è sostanziale. (Esempio: un partecipante riporta un problema nel condizionatore d'aria di un'aula e suggerisce di cambiarlo).
- **Issue resolution**: scelta di una singola soluzione a un problema per il quale sono state proposte varie soluzioni. Si usano *issue base* per collezionare problemi e proposte.

4.3 Meccanismi di Comunicazione

Il **meccanismo di comunicazione** è il tool o la procedura che si usa per trasmette le informazioni. Può essere **Sincrona**, cioè chi invia e chi riceve sono disponibili allo stesso tempo, oppure **Asincrona**, per cui i due interlocutori non comunicano nello stesso tempo.

I meccanismi di **comunicazione sincrona** sono:

- **Segnali di fumo**.
- **Hallway conversation**: tipo di conversazione faccia a faccia, che supporta conversazioni non pianificate, richieste di chiarimenti e di cambiamenti. I vantaggi sono dovuti al fatto che è un meccanismo economico ed efficiente per risolvere semplici problemi. D'altra parte, informazioni importanti possono essere andare perdute e possono verificarsi incomprensioni quando la conversazione è riportata ad altri.
- **Meeting**: realizzati faccia a faccia, per telefono o in video conferenza, e supportano conversazioni pianificate, client review, project review, status review, peer review, postmortem review, brainstorming, issue resolution. Sono meccanismi efficienti per risolvere problemi, e ottenere l'unanimità, ma costosi (in termini di persone e risorse), difficili da gestire e con essi è difficile ottenere risultati efficienti.
- **Questionari e Interviste strutturate**: supportano problem definition e postmortem review. Aiutano chiarire incomprensioni riscontrate durante la fase di requirements elicitation a basso costo per l'utente. Lo svantaggio è che sono difficili da progettare.
- **Synchronous groupware**: supportano conversazioni non pianificate, client review, project review, peer review, brainstorming, issue resolution. Il vantaggio è che le persone comunicano allo stesso tempo anche stando in posti differenti, tuttavia diventa difficile coordinare gli utenti.

I meccanismi di **comunicazione asincrona**, invece, sono:

- **E-Mail**: supportano release, change request, brainstorming. Sono ideali per la modalità di comunicazione event-driven e per trasmettere annunci. Lo svantaggio è che le e-mail inviate al di fuori dal contesto possono essere facilmente incomprese, inviate a persone sbagliate, perse o non lette dal ricevente.
- **Newsgroup**: supportano release, change request, brainstorming. Sono adatti per notificare e discutere tra persone che condividono interessi comuni e sono economici (disponibili shareware).

- **World Wide Web**: supporta release, change request, inspection. Il vantaggio è che fornisce agli utenti una metafora ipertestuale: i documenti contengono collegamenti ad altri documenti. Tuttavia, non è facile supportare la rapida evoluzione dei documenti.
- **Lotus Notes**: ogni utente vede lo spazio delle informazioni come un insieme di database, contenenti documenti composti da un insieme di campi. Gli utenti collaborano creando, condividendo e modificando documenti. Supporta release, change request, brainstorming e ha come vantaggio quello di fornire un eccellente meccanismo di controllo degli accessi e di replicazione dei database. Lo svantaggio è che si ha a che fare con formati proprietari, e inoltre è costoso.

Consiglio: un modo di realizzare comunicazione significativa in un progetto è mediante: meeting

settimanali, project review, comunicazione online (forum di discussione, email, web).

4.4 Ruoli in un meeting

Primary facilitator: è il responsabile dell'organizzazione del meeting e della sua esecuzione. Inoltre, scrive l'agenda che descrive gli obiettivi e gli scopi del meeting, e la distribuisce ai partecipanti.
Minute taker: è il responsabile della registrazione del meeting, identifica elementi e problemi e li riporta ai partecipanti.
Time keeper: è responsabile di tenere traccia del tempo.

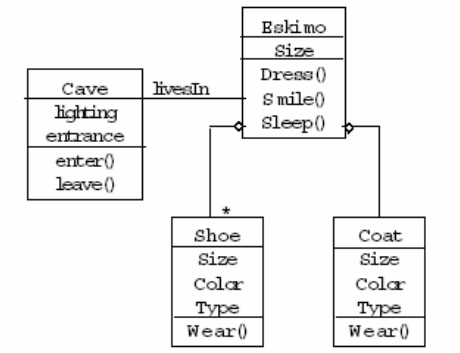
4.5 APPENDICE

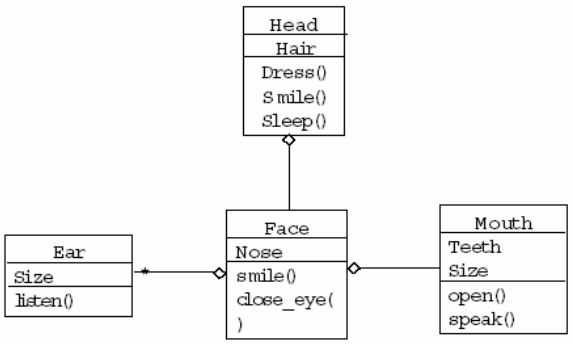
Realizzare un **Document Review**, significa:
- compilare un form di revisione;
- allegare il documento che deve essere revisionato;
- distribuire il form di revisione ai revisori;
- attendere i commenti dei revisori;
- ottenere i commenti di revisione;
- creare degli elementi di azione dai commenti selezionati;
- rivisitare il documento e distribuire la versione revisionata;
- iterare il ciclo di revisione.

Tutto ciò può essere fatto usando il portale e *ADAMS*.

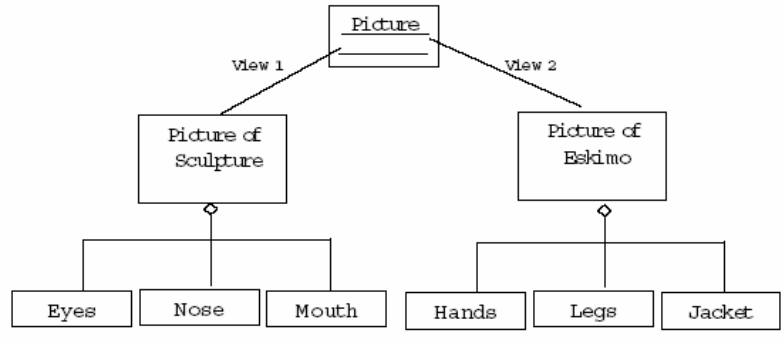
5.1 Overview

Cosa è? Un'eschimese o la testa di un indiano? I modelli a oggetti sarebbero:

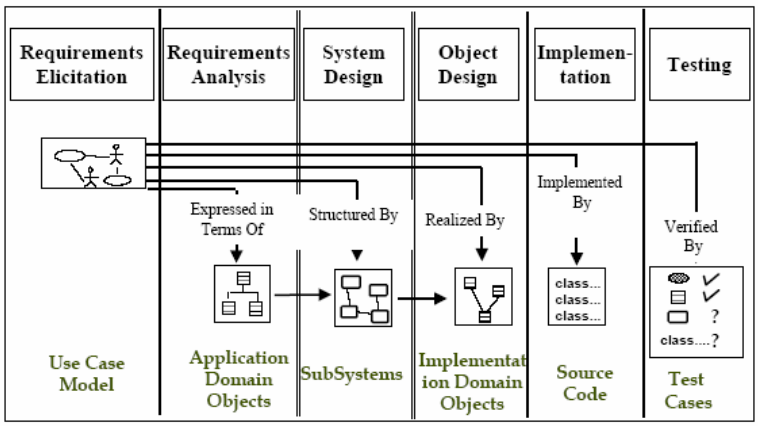




Dal punto di vista di un artista, invece:



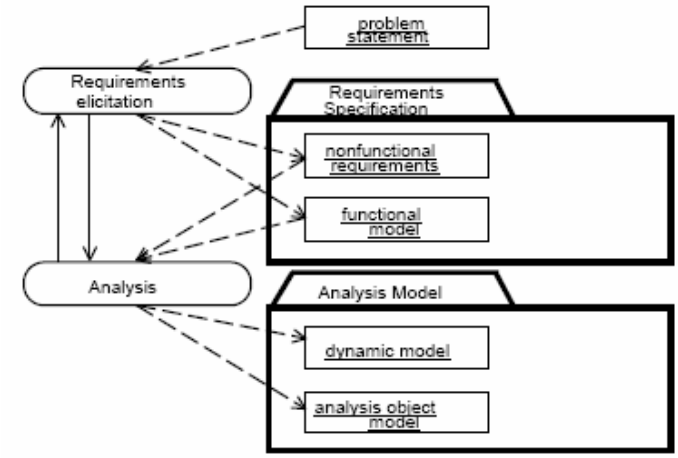
Abbiamo tre modi di gestire la complessità: **Astrazione**, **Decomposizione** (con la tecnica *Divide and Conquer*) e **Gerarchia** (con la tecnica *Layering*). Esistono due modalità per gestire la Decomposizione: *orientazione a oggetti* e *decomposizione funzionale* (comporta, però, un codice non manutenibile). A seconda dell'obiettivo del sistema, possono essere trovati oggetti differenti.
Il metodo migliore comunque è di iniziare con una descrizione delle funzionalità (Modello dei Casi d'uso) e successivamente di proseguire cercando gli oggetto (object model). Per capire di quali attività e modelli abbiamo bisogno, dobbiamo conoscere il ciclo di vita del software: l'insieme delle attività e delle relazioni tra queste, che sono di supporto allo sviluppo di un sistema software. Le domande solite che ci si pone sono: Quali attività dovrei scegliere per il progetto software? Quali sono le dipendenze tra queste? Come posso schedulare le attività? Qual è il risultato di ogni attività?



Il primo passo nello stabilire i requisiti è senza dubbio quello di identificare il sistema. Lo sviluppo di un sistema non si effettua a partire da immagine fissa dello scenario (dominio)! Bisogna rispondere a due domande: Come si possono identificare gli obiettivi di un sistema? Cruciale è la definizione dei confini del sistema: cosa è interno e cosa è esterno ad esso? Le risposte a tali domande sono da cercarsi nel **processo dei requisisti**, che consiste di due attività:

- **Requirements Elicitation** (fase di raccolta dei requisiti): definizione del sistema in termini comprensibili dal cliente ("Descrizione del Problema");
- **Requirements Analysis** (Analisi dei Requisiti): specifica tecnica del sistema in termini comprensibili dagli sviluppatori ("Specifica del Problema").

Il Requirements Process può essere riassunto come:



5.2 Il Problem Statement

L'output della fase di Requirements Elicitation è il documento di **specifica dei requisiti**, usa un linguaggio naturale ed è derivato dal Problem Statement. Durante la fase di Analisi dei Requisiti, questo documento viene formalizzato e strutturato per produrre il **Modello di Analisi**, che usa notazioni formali o semi-formali, ad esempio quelle di un linguaggio grafico come UML. Entrambi i modelli focalizzano l'attenzione sui requisiti dal punto di vista dell'utente del sistema, e il punto di partenza è sempre il Problem Statement.

Il **Problem Statement** (Descrizione del Problema) è sviluppato dal cliente come una descrizione del problema che dovrebbe affrontare il sistema (è anche detto Statement of Work). Un buon Problem Statement descrive: la situazione corrente, le funzionalità che il nuovo sistema dovrebbe supportare, l'ambiente in cui il sistema sarà distribuito, le consegne attese dall'utente, le date di

rilascio, un insieme di criteri di accettazione. In maniera più formale, un Problem Statement contiene:

- *Situazione corrente*: il problema da risolvere.
- Descrizione di uno o più *scenari*.
- *Requisiti*: funzionali, non funzionali e vincoli ("pseudo requisiti").
- *Project Schedule*: i punti rilevanti (milestones) che riguardano l'interazione con il cliente, includendo anche deadline per lo sviluppo del sistema.
- *Target environment*: l'ambiente in cui il software sviluppato verrà sottoposto a uno specifico insieme di test di sistema.
- *Criteri di Accettazione per il Cliente*: i criteri per i test di sistema.

La **situazione corrente** descrive un problema corrente, ad esempio il tempo di risposta del gioco degli scacchi è troppo lento oppure, voglio giocare ma non trovo giocatori del mio livello. Siamo in una situazione quindi in cui è cambiato qualcosa, nel dominio applicativo o in quello delle soluzioni. I cambi nel dominio applicativo si hanno con l'introduzione di nuove funzionalità e i cambi nel dominio delle soluzioni si hanno quando appare una nuova soluzione, ad esempio grazie a nuove tecnologie disponibili.

5.3 I requisiti

I **requisiti** possono essere di 3 tipi differenti:

- **Requisiti funzionali**: descrivono le interazioni tra il sistema e i suoi ambienti, indipendentemente dall'implementazione (ad esempio un giocatore deve avere la possibilità di definire un nuovo gioco).
- **Requisiti non funzionali**: descrivono aspetti del sistema non direttamente legati al suo comportamento funzionale (ad esempio, il tempo di risposta deve essere inferiore a 1 secondo).
- **Vincoli** ("pseudo requisiti): imposti dal cliente o dall'ambiente in cui opera il sistema (ad esempio, il linguaggio di implementazione deve essere Java).

Di solito non vengono considerati requisiti: la struttura del sistema, la tecnologia di implementazione, la metodologia di sviluppo, l'ambiente di sviluppo, il linguaggio di implementazione, la riusabilità.
I requisiti, inoltre, devono essere **validati**. Questo è un momento critico nel processo di sviluppo e, di solito, avviene dopo il requirements engineering o l'analisi dei requisiti (a volte anche alla consegna, mediante test di accettazione del cliente). I criteri usati sono:

- *Correttezza*: i requisiti devono rappresentare la vista del cliente.
- *Completezza*: tutti i possibili scenari, in cui il sistema può essere usato, devono essere descritti, includendo anche comportamenti eccezionali dell'utente o del sistema.
- *Consistenza*: non devono esserci requisiti funzionali o non funzionali in contraddizione.
- *Chiarezza (non ambiguità)*: esattamente un sistema è definito.
- *Realismo*: i requisiti possono effettivamente essere implementati e consegnati all'interno di vincoli.
- *Traceability* (attribuibilità): ogni funzione del sistema deve essere attribuibile a un corrispondente insieme di requisiti funzionali.
- *Verificabilità:* una volta costruito il sistema, è sempre possibile progettare test per dimostrare la sua corrispondenza alla specifica dei requisiti.

I problemi con la validazione dei requisiti sono dovuti ai cambiamenti molto rapidi dei requisiti durante la fase di requirements elicitation. A tal proposito, esistono dei tool di supporto alla gestione dei requisiti: memorizzano i requisiti in repository condivisi, forniscono accesso multi utente, creano automaticamente documenti di specifica del sistema a partire dal repository, permettono di cambiare la gestione, forniscono traceability attraverso il ciclo di vita del progetto. Un esempio è *RequisitPro from Rational*.

5.4 Scenari e Casi d'uso

La fase di Requirements Elicitation si differenzia sulla base del tipo di progetto. In una **Greenfield Engineering** lo sviluppo comincia da zero, non esiste nessun sistema a priori e i requisiti sono ottenuti dall'utente finale e dal cliente. Nasce, perciò, a partire dai bisogni dell'utente. In una **Re-Enigeering** si ri-progetta e/o re-implementa un sistema esistente usando nuove tecnologie. Nasce, quindi, dalle tecnologie disponibili.
In una **Interface Engineering**, infine, si forniscono i servizi di un sistema esistente in un nuovo ambiente. L'esigenza nasce, quindi, dalle tecnologie disponibili o dai nuovi bisogni di mercato.

Pertanto, la fase di *Requirements Elicitation*, risulta essere un'attività molto interessante e che richiede collaborazione di persone con differenti esperienze: utenti con conoscenze del dominio applicativo, sviluppatori con conoscenze del dominio delle soluzioni dal punto di vista progettuale o implementativi, … E' una fase che fa da ponte tra l'utente e gli sviluppatori, fornendo **Scenari**, cioè esempi di utilizzo del sistema in termini di sequenze di interazioni tra l'utente e il sistema, e **Casi d'uso**, cioè astrazioni che descrivono una classe di scenari.
Usiamo i casi d'uso perché sono assolutamente comprensibili all'utente: modellano il sistema dal punto di vista dell'utente (requisiti funzionali), definendo ogni possibile flusso di eventi attraverso il sistema e descrivendo l'interazione tra gli oggetti. Sono, perciò, strumenti utili alla gestione del progetto e possono essere la base di alcuni prodotti di sviluppo come Manuali utente, System design e Object design, implementazione, specifica dei test, test di accettazione del cliente e sono un'eccellente base per lo *sviluppo incrementale e iterativo*.
Gli scenari, invece, sono una descrizione narrativa di cosa fanno le persone e mostrano come questi provano a usare le applicazioni e i sistemi computerizzati. Sono descrizioni concrete e informali di una singola caratteristica del sistema usato da un singolo attore e possono essere impiegati in modi differenti, durante il ciclo di vita di un software. Ce ne sono di vari tipi:

- *As-is*: usati nella descrizione della situazione corrente, e di solito per progetti di re-engineering. L'utente descrive il sistema.
- *Visionary*: usati per descrivere un sistema futuro, di solito per progetti di greenfield engineering e reengineering. Spesso non possono essere realizzati da utenti o sviluppatori solamente.
- *Evaluation*: descrivono operazioni dell'utente rispetto alle quali il sistema verrà poi valutato.
- *Training*: istruzioni passo passo che guidano un utente inesperto attraverso il sistema, in parole povere si tratta di tutorial.

Il problema è come individuare questi scenari. Una cosa è certa: non aspettarsi descrizioni verbali dal cliente, se il sistema non esiste (greenfield engineering) e non aspettarsi informazioni precise se il sistema esiste. Piuttosto, è bene utilizzare un approccio dialettico (evolutivo, incrementale), per aiutare il cliente a formulare i requisiti, e per ottenere aiuto dallo stesso per comprenderli. Bisogna tenere presente, inoltre, che i requisiti evolvono durante lo sviluppo degli scenari.
In generale, un'euristica per trovare gli scenari si basa sul porsi (o sul porre al cliente) le seguenti domande: Qual è la prima operazione che il sistema ha bisogno di eseguire? Quali dati l'attore creerà, memorizzerà, cambierà, eliminerà o aggiungerà nel sistema? Quali cambiamenti esterni il sistema ha bisogno di conoscere? Su quali cambiamenti o eventi l'attore del sistema dovrà essere informato?

Attenzione però a non contare solo sui questionari! Insistere sull'osservazione delle operazioni che effettua il sistema esistente (interface engineering o reengineering) e porre domande all'utente finale non solo sul contratto software. In ogni caso, aspettarsi resistenza e cercare di superarla.

Dopo aver definito uno scenario, bisogna trovare un caso d'uso che specifica tutte le possibili istanze dell'operazione che descrive lo scenario. Il caso d'uso deve essere descritto in maniera dettagliata: descrivere le entry condition, il flusso di eventi, le exit condition, le eccezioni, i requisiti speciali (vincoli e requisiti non funzionali).
In particolare, un caso d'uso è un flusso di eventi nel sistema, che include le interazioni con gli attori. E' avviato da un attore, ha un nome, una condizione di terminazione e in notazione grafica è rappresentato da un ovale contenente il nome del caso d'uso. Il **Modello dei Casi d'uso** è l'insieme di tutti i casi d'uso che specificano le complete funzionalità del sistema.
Un'euristica per trovare i casi d'uso si basa sui seguenti concetti:

- Selezionare una sottile fetta verticale del sistema, ad esempio uno scenario e discuterne in dettaglio con l'utente per comprendere lo stile di interazione da lui preferito.
- Selezionare una fetta orizzontale, ad esempio più scenari, per definire lo scopo del sistema, discutendone con l'utente.
- Utilizzare prototipi illustrati (mock-up) come supporto visuale.
- Cercare di capire cosa fa l'utente, osservando i lavori e mediante i questionari.

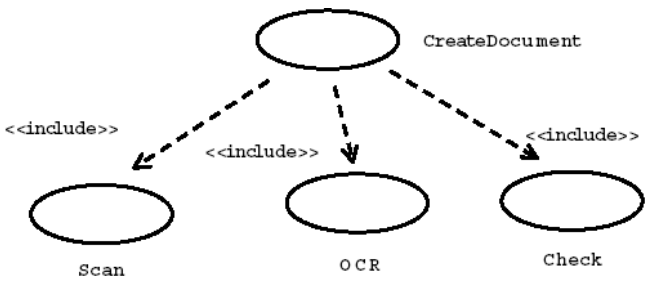
La costruzione di un caso d'uso avviene mediante i seguenti passi: dare un nome al caso d'uso, trovare gli attori, concentrarsi sul flusso di eventi, tirare fuori le eccezioni, identificare e annotare ogni requisito speciale. Formalmente, deve contenere:

- *Nome* del caso d'uso.
- *Attori*: descrizione degli attori coinvolti nel caso d'uso.
- *Entry condition*: usare una frase sintetica come "Questo caso d'uso inizia quando…".
- *Flusso di eventi*: in forma libera, mediante linguaggio naturale informale.
- *Exit condition*: usare una frase sintetica come "Questo caso d'uso termina quando…".
- *Eccezioni*: descrivere cosa accade se qualcosa va male.
- *Requisiti speciali*: lista dei requisiti non funzionali e dei vincoli.

Un *modello dei casi d'uso* consiste di casi d'uso e **relazioni** tra loro. Queste relazioni possono essere **Dipendenze** o **Generalizzazioni**. La Generalizzazione indica la situazione in cui un caso d'uso astratto ha differenti specializzazioni. Le dipendenze si dividono in **Inclusioni** (un caso d'uso usa un altro caso d'uso) e **Estensioni** (un caso d'uso estende un altro caso d'uso).

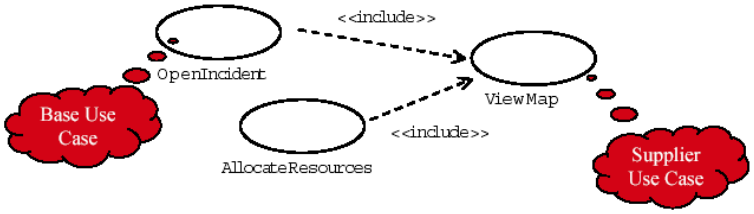
<<include>> per realizzare Decomposizioni funzionali

Siamo nella situazione in cui una funzione nel problem statement originale è troppo complessa per poter essere risolta immediatamente. La soluzione è descrivere la funzione come l'aggregazione di un insieme di funzioni più semplici. Il caso d'uso associato è decomposto in casi d'uso più piccoli:



<<include>> per ottenere il riuso di funzionalità già esistenti

Siamo nella situazione in cui le funzioni esistono già e ci chiediamo come riusarle. La soluzione è usare una relazione di inclusione tra un caso d'uso A e un caso d'uso B per indicare che un'istanza di A esegue tutte i comportamenti descritti in B, cioè "A delega B". Ad esempio, il caso d'uso "ViewMap" descrive un comportamento che può essere usato dal caso d'uso "OpenIncident":



Da notare che il caso di base non può esistere da solo: è sempre chiamato con il caso d'uso supplier (fornitore).

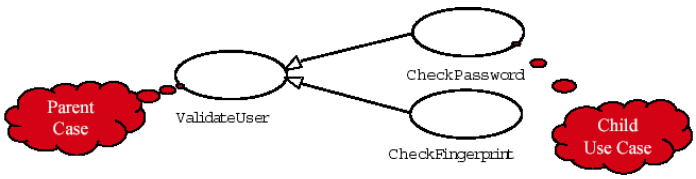
La relazione <<extend>>

Siamo nella situazione in cui la funzionalità nel problem statement originale ha bisogno di essere estesa. La soluzione è l'utilizzo di una relazione di estensione dal caso d'uso A al caso d'uso B per indicare che B è un'estensione di A. Ad esempio, il caso d'uso "ReportEmergency" è completo di per sé, ma può essere esteso dal caso d'uso "Help" per uno specifico scenario in cui l'utente richiede aiuto:

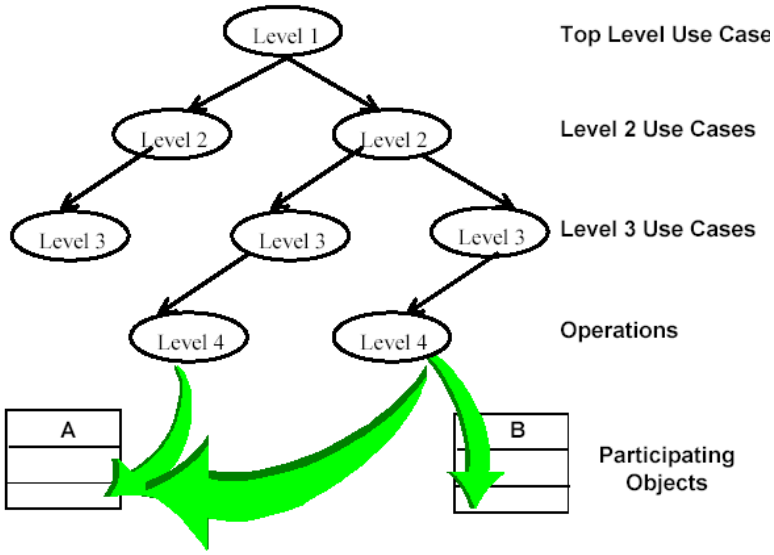
Da notare che in una relazione di estensione, il caso d'uso di base può essere eseguito senza il caso d'uso di estensione.

La relazione di Generalizzazione

Il problema che si affronta è che si hanno comportamenti comuni tra i casi d'uso e si vuole fattorizzarli. La soluzione è usare un'associazione di generalizzazione tra i casi d'uso per fattorizzare i comportamenti comuni. I casi d'uso "figli" ereditano il comportamento e il significato del caso d'uso "padre" e aggiungono o ignorano alcuni comportamenti. Ad esempio, consideriamo il caso d'uso "ValidateUser", responsabile di verificare le identità degli utenti. Il cliente potrebbe richiedere due realizzazioni: "CheckPassword" e "CheckFingerprint", cioè un controllo sulla password e uno sull'impronta digitale.



5.5 Dai casi d'uso agli oggetti



Bisogna trovare gli oggetti partecipanti ai casi d'uso. Per ogni caso d'uso eseguire i seguenti passi:

- Trovare i termini che gli sviluppatori o gli utenti usano per chiarire e far comprendere il flusso di eventi. Iniziare sempre con i termini dell'utente e poi negoziare: FieldOfficerStationBoundary o FieldOfficerStation? IncidentBoundary o IncidentForm? EOPControl o EOP?
- Identificare le reali entità di cui il sistema ha bisogno per tenerne traccia. Ad esempio, FiledOfficer, Dispatcher, Resource.
- Identificare le reali procedure di cui il sistema ha bisogno per tenerne traccia. Ad esempio, EmergencyOperationPlan.
- Identificare sorgenti e pozzi di dati. Ad esempio, Printer.
- Identificare artefatti per interfacce. Ad esempio, PoliceStation.
- Fare un'analisi testuale per trovare oggetti aggiuntivi (usando la tecnica di Abott).
- Modellare il flusso di eventi con un sequence diagram.

Successivamente costruire un dizionario dei dati (o glossario) e, infine, fare Cross-Checking sui casi d'uso e sugli oggetti partecipanti.

5.6 Il modello FURPS+

FURPS+ è l'acronimo di "Functional, Usability, Reliability, Performance and Supportabiliy" (il + indica categorie addizionali). E' un modello usato per suddividere in categorie i requisiti non funzionali e i vincoli, detti anche *requisiti di qualità*.

Categorie di requisiti non funzionali:

- L'**usabilità** riguarda la facilità con cui un utente impara a operare, a preparare gli input e interpretare gli output di un sistema o di un componente (le convenzioni adottate per l'interfaccia utente, la visibilità dell'help on line, il livello della documentazione utente,…).
- L'affidabilità (**reliability**) è la capacità di un sistema o di un componente ad eseguire le funzioni richieste sottostando a condizioni per un determinato periodo di tempo (scarsi tempi di fallimenti, capacità di identificare specificati fallimenti o di resistere a specifici attacchi alla sicurezza,…). Recentemente è stata rimpiazzata da **dependability** (lealtà), che è la proprietà di

un sistema computerizzato per cui la fiducia può legittimamente essere posta al servizio che fornisce. Questa proprietà include reliability, robustezza e sicurezza.
- I requisiti di **Performance** sono riguardanti attributi quantificabili del sistema, come tempo di risposta, throughput, disponibilità e accuratezza.
- I requisiti di **Supportabilità** sono riguardanti la facilità di effettuare cambiamenti al sistema dopo lo sviluppo, includendo ad esempio l'adattabilità, la manutenibilità, e l'internazionalizzazione. Lo standard ISO 9126 sulla qualità del software rimpiazza questa categoria con: **manutenibilità** e **portabilità**.

Categorie di pseudo requisiti:
- I **requisiti di implementazione** sono vincoli sull'implementazione del sistema, includenti l'uso di specifici tool, linguaggi di programmazione, o piattaforme hardware.
- I **requisiti sulle interfacce** sono vincoli imposti dai sistemi esterni, includenti sistemi esistenti e formati di interscambio.
- I **requisiti sulle operazioni** sono vincoli sull'amministrazione e la gestione del sistema.
- I **requisiti di packaging** sono vincoli sulla reale consegna del sistema (ad esempio vincoli sul supporto di installazione per montare il software).
- I **requisiti legali** sono riguardanti la licenza, le regolamentazioni e i certificati di rilascio.

Sulla base di queste definizioni ecco come il FURPS+ suggerisce di identificare i requisiti non funzionali:
- *Usabilità*: Qual è il livello di esperienza dell'utente? Quale standard di interfaccia utente gli è familiare? Quale documentazione dovrebbe essergli fornita?
- *Reliability*: Quanto fidato, disponibile e robusto dovrebbe essere il sistema? Il riavvio del sistema in casi di fallimento è accettabile? Quanti dati può perdere il sistema? Come dovrebbe gestire le eccezioni? Ci sono requisiti di safety? Ci sono requisiti di sicurezza?
- *Performance*: Come dovrebbe rispondere il sistema? Qualche operazione dell'utente richiede tempi critici? Quanti utenti concorrenti dovrebbe supportare? Quanto è grande una tipica memorizzazione dei dati per sistemi di questo tipo? Qual è il massimo tempo di attesa accettato dall'utente?
- *Supportabilità*: Quali sono le estensioni prevedibili del sistema? Chi mantiene il sistema? Ci sono piani per portarlo in differenti ambienti software o hardware?
- *Implementazione*: Ci sono vincoli sulla piattaforma hardware? I vincoli sono imposti dal team di manutenzione? Sono imposti dal team di testing?
- *Interfaccia*: Il sistema dovrà interagire con altri sistemi esistenti? Come sono i dati esportati/importati nel sistema? Quali standard usati dal cliente dovrà supportare il sistema?
- *Operazioni*: Chi mantiene il sistema in esecuzione?
- *Packaging*: Chi installa il sistema? Quante installazioni sono previste? Ci sono vincoli sui tempi di installazione?
- *Lagali*: Che licenza deve avere il sistema? Qualche assicurazione è rilasciata per i fallimenti del sistema? Ci sono diritti d'autore o diritti di licenza contratti per l'utilizzo di specifici algoritmi o componenti?

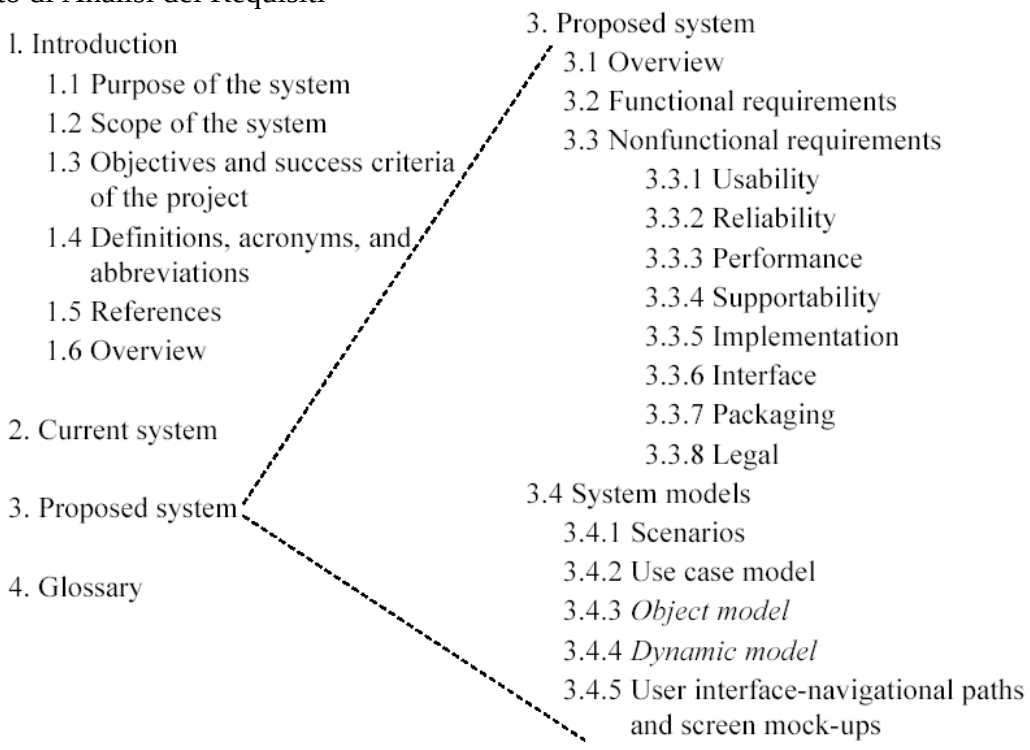
5.7 Gestire la fase di Requirements Elicitation

Durante la fase di Requirements Elicitation, ci sono due metodi per raccogliere informazioni, prendere decisioni con gli utenti e i clienti e per gestire le dipendenze tra i requisiti e gli altri artefatti:

- *Joint Application Design (JAD)*: focalizza l'attenzione sull'ottenere il consenso tra sviluppatori, utenti e clienti. Viene usato, quindi, per negoziare le specifiche con il cliente. Gli utenti, i clienti e gli sviluppatori presentano i loro punti di vista, e bisogna tenerne conto. Bisogna, tuttavia, ascoltarne anche altri e successivamente si può negoziare per giungere a una soluzione accettabile. Il documento di specifica dei requisiti, quindi, è sviluppato insieme ai diversi **stackholder** (utenti, clienti, sviluppatori).

- *Traceability*: focalizza l'attenzione sul registrare, strutturare, collegare, raggruppare e mantenere le dipendenze tra i requisiti e le dipendenze tra questi e gli altri work product. Bisogna, perciò seguire il ciclo di vita di ogni requisito, controllare che il sistema sia completo, che sia conforme ai suoi requisiti, per tenere traccia della logica sottostante le scelte e valutare l'impatto dei cambiamenti. Bisogna, inoltre, fare cross-referencing tra documenti, modelli e artefatti di codice e usare semplici tool (fogli elettronici, word processor) per memorizzare le dipendenze. A tal proposito esistono tool specializzati: *Rational RequisitePro, Telelogic DOORS*.

Documento di Analisi dei Requisiti

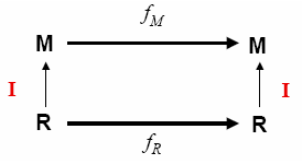


Riassumendo, la fase di Requirements Elicitation serve a costruire un modello funzionale del sistema che sarà poi usato durante l'analisi per costruire un object model e un dynamic model. Le attività della Requirements Elicitation sono: identificare gli attori, gli scenari, i casi d'uso, le relazioni tra questi, raffinare i casi d'uso, identificare requisiti non funzionali e identificare gli oggetti partecipanti.

6. ANALISI DEI REQUISITI (OBJECT MODELING)

6.1 Overview

Consideriamo la **Realtà** R e il **Modello** M. La realtà è fatta di cose reali, persone, processi che avvengono nel tempo e relazioni tra queste cose. Il modello, invece, è l'astrazione da cose, persone, processi e relazioni (realmente esistenti o solo pensate) tra queste astrazioni.
Usiamo i modelli per fare astrazione dai dettagli nella realtà (in modo da poter ottenere conclusioni complicate nella realtà ma con semplici passi nel modello), per fare discernimento tra passato e presente, e per fare previsioni sul futuro.
In generale, un modello è buono se le relazioni che sono valide nella realtà R lo sono anche nel modello M, cioè se il seguente diagramma è commutativo:



dove I mappa la realtà nel modello, f_m rappresenta le relazioni tra le astrazioni in M e f_r le relazioni tra le cose reali in R.
I modelli, tuttavia, sono **falsificabili**. Nel medioevo, le persone credevano nella realtà, in ciò che vedevano, ma i modelli della realtà non possono essere veri: un modello è sempre un'approssimazione, lo costruiamo in base alle nostre conoscenze o alle conoscenze attuali. A tal proposito, vale il **Principio di Falsificazione** (di Popper): *possiamo solo costruire modelli dalla realtà, che sono "veri" finché non troviamo un contro esempio*.
Nonostante questo principio, però, spesso continuiamo ad usare il modello, anche dopo aver dimostrato la sua falsità, perché magari funziona bene nella maggioranza dei casi.
Il principio di falsificazione è alla base dello sviluppo software: l'obiettivo dei prototipi, delle revisioni e dei test di sistema è proprio quello di falsificare il sistema software.
Inoltre, la modellazione è iterativa: possiamo pensare a un modello come la realtà e costruire un altro modello a partire dal primo (con astrazione ancora maggiore) e lo sviluppo software si basa su questo concetto, infatti, è una trasformazione di modelli: analisi, progetto, implementazione, testing.

6.2 Le Attività dell'Object Modeling

L'obiettivo principale è trovare le astrazioni più importanti. Se troviamo quelle sbagliate, non c'è problema: si può iterare e correggere il modello! I passi da seguire sono:

1. identificare le classi (basandosi sulla fondamentale assunzione che stiamo cercando astrazioni);
2. trovare gli attributi;
3. trovare i metodi;
4. trovare le associazioni tra le classi.

In ogni caso l'importante è iterare.
L'**identificazione delle classi** è un passo cruciale nella modellazione orientata agli oggetti. Bisogna identificare i confini e le entità importanti del sistema, e l'assunzione di base è: possiamo cercare classi per un nuovo sistema software (Forward Engineering) o per un sistema esistente (Reverse Engineering). Chiaramente gli oggetti non vengono trovati osservando un immagine statica, fissa, della scena o del dominio. Anzi, il dominio applicativo deve essere analizzato e, a seconda

dell'obiettivo del sistema, potrebbero essere trovati oggetti differenti. Per identificare gli obiettivi del sistema, ci servono gli **Scenari** e i **Casi d'uso**.
In generale, le componenti di un **Object Model** sono:

- *Classi*.
- *Relazioni*: generiche (associazioni) o canoniche, cioè aggregazioni (parti di gerarchia) e generalizzazioni (tipi di gerarchia)
- *Attributi*: bisogna individuarli, e possono essere anche applicazioni specifiche; inoltre attributi in un sistema possono essere classi in un altro e, quindi, si devono trasformare in classi.
- *Operazioni*: bisogna individuarle, possono essere Generiche (Get/Set, General world knowledge, design patterns) o del Dominio (Dynamic model, Functional model).

Ma qual è la differenza tra Oggetto e Classe? Un **oggetto** (istanza) è esattamente una cosa, mentre una **classe** descrive un gruppo di oggetti con proprietà simili. Un **Object diagram** è una notazione grafica per modellare oggetti, classi e relazioni tra loro. Esempi di object diagram sono i **class diagram** (utili per tassonomie, pattern, schemi,…), che sono template per descrivere molte istanze di dati, e gli **instance diagram** (utili per discutere degli scenari, casi di test ed esempi), che sono particolari insiemi di oggetti correlati tra loro. A tal proposito, *Poeidon* e *Argo UML* sono ottimi CASE tool per costruire object diagram, in particolare class diagram.

Quindi, identificare le classi è un punto centrale della modellazione a oggetti. Lo si può fare seguendo vari approcci:

- *Approccio al Dominio applicativo*: si richiede agli esperti del dominio applicativo di identificare le astrazioni più rilevanti.
- *Approccio sintattico*: si parte dai casi d'uso e si estrapolano dal flusso di eventi gli oggetti partecipanti. Successivamente, si analizzano i sostantivi e i verbi (*tecnica di Abbot*) per identificare le componenti del modello a oggetti.
- *Approccio Design pattern*: utilizza modelli di progetto riusabili.
- *Approccio basato sui componenti*: identifica le classi esistenti nel dominio delle soluzioni.

In particolare, è utile:

- informarsi sul dominio del problema, osservando il cliente;
- utilizzare conoscenze generali e intuizione;
- osservare il flusso di eventi e trovare gli oggetti partecipanti ai casi d'uso;
- provare a stabilire una tassonomia;
- applicare conoscenze di progettazione: distinguere tipi differenti di oggetti, e utilizzare modelli di progetto;
- fare un'analisi sintattica del problem statement, degli scenari o dei flussi di eventi. Si può usare l'**Analisi Testuale di Abbot** (1983), anche nota come "analisi sostantivi-verbi", secondo la quale i sostantivi sono buoni candidati a essere classi e i verbi sono buoni candidati a essere operazioni.

Esistono poi vari modi per **trovare gli oggetti**.

- Investigazione sintattica con la tecnica di Abbot: nel problem statement (proposto inizialmente, ma raramente funziona se il problem statement è grande, in genere più di 5 pagine) o nel flusso di eventi dei casi d'uso.
- Usare varie sorgenti di conoscenza: conoscenze applicative (interviste all'utente finale e a esperti per determinare le astrazioni del dominio applicativo), conoscenze progettuali (astrazioni riusabili nel dominio delle soluzioni), conoscenze generiche (usare anche queste e l'intuizione).

- Formulazione degli scenari (in linguaggio naturale): descrizione del concreto utilizzo del sistema.
- Formulazione dei casi d'uso (in linguaggio naturale o in UML): descrizione delle funzionalità mediante attori e flusso di eventi.

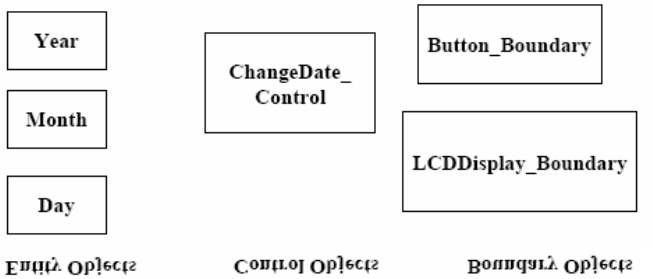
In generale, considerare un caso d'uso e osservare il flusso di eventi:
- trovare i termini che gli sviluppatori o gli utenti usano per chiarire e far comprendere il flusso di eventi;
- osservare i nomi ricorrenti (ad esempio, Incident);
- identificare le reali entità di cui il sistema ha bisogno e tenerne traccia (ad esempio, FiledOfficer, Dispatcher, Resource);
- identificare le reali procedure di cui il sistema ha bisogno e tenerne traccia (ad esempio, EmergencyOperationPlan);
- identificare sorgenti e pozzi di dati (ad esempio, Printer);
- identificare artefatti per interfacce (ad esempio, PoliceStation)

Sicuramente alcuni oggetti mancheranno e bisogna comunque trovarli, modellando il flusso di eventi con un sequenze diagram. Importante è anche usare i termini degli utenti.

Possiamo, inoltre, fare una classificazione sui tipi di oggetti:
- **Entity object**: rappresentano l'informazione persistente tracciata dal sistema (oggetti del dominio applicativo).
- **Boundary object**: rappresentano l'interazione tra l'utente e il sistema.
- **Control object**: rappresentano le operazioni eseguite dal sistema.

Avere tre tipi di oggetti porta i modelli a essere molto più flessibili ai cambiamenti. Le interfacce del sistema cambiano molto più dei controlli, che a loro volta cambiano molto più del dominio applicativo. Nel modello architetturale *MVC* (Model/View/Controller), il dominio applicativo è rappresentato dai Model object, visualizzato dai View object e manipolato dai control object.

Esistono poi delle convenzioni per dare i nomi ai tipi di oggetti. UML fornisce diversi meccanismi per estendere il linguaggio e in particolare per stereotipare, in modo da presentare nuovi elementi modellanti. Le convenzioni raccomandate sono di distinguere tra tipi di oggetti in base alla loro sintassi: quelli che terminano con il suffisso "_Boundary" sono boundary object, quelli che terminano con il suffisso "_Control" sono control object. Gli entity object non hanno alcun suffisso legato al loro nome.

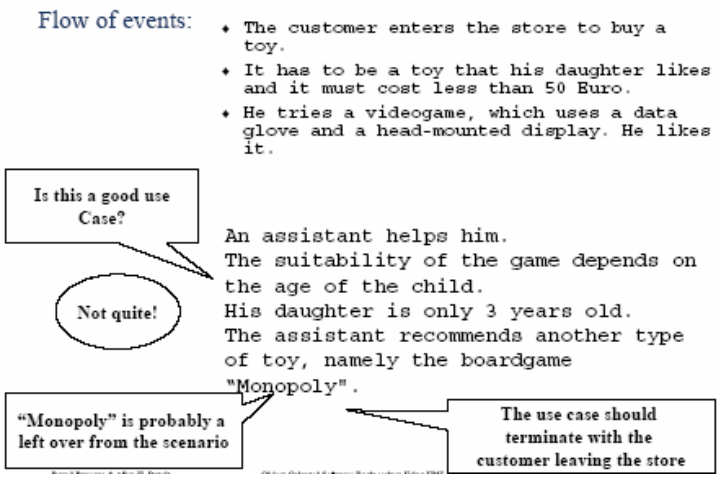


La tecnica di Abbot indica di mappare parti del testo in componenti del modello nella seguente maniera:

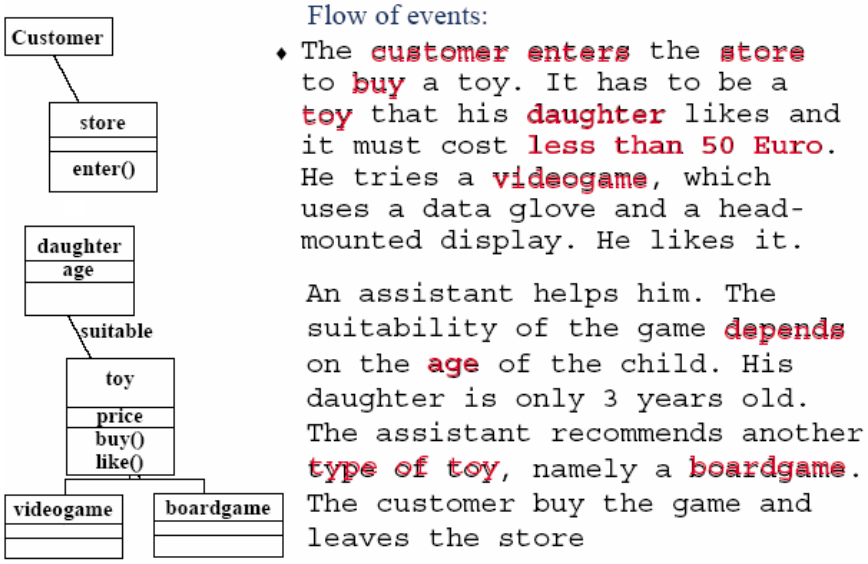
Parole	**Componenti del Modello**	**Esempi**
Nome proprio	Oggetto	Jim Smith
Nome comune	Classe	Giocattolo, bambola
Verbo che indica azione	Metodo	Comprare, raccomandare
Verbo essere	Ereditarietà	Is-a (kind-of)

Verbo avere	Aggregazione	Has an
Verbo di modo	Vincolo	Deve essere
Aggettivo	Attributo	Età di 3 anni
Verbo transitivo	Metodo	Entrare
Verbo intransitivo	Metodo (evento)	Dipende da

Esempio:



Generazione di un class diagram dal flusso di eventi:



Riassumendo, l'ordine delle attività nella modellazione è il seguente:

1. formulare un po' di scenari con l'aiuto dell'utente finale e/o di esperti del dominio applicativo;
2. estrapolare i casi d'uso dagli scenari, con l'aiuto di esperti del dominio applicativo;
3. analizzare il flusso di eventi, ad esempio con l'analisi testuale di Abbot;
4. generare il diagramma delle classi, che si ottiene mediante i seguenti passi:
   4.1 identificazione delle classi (analisi testuale, esperti del dominio applicativo);
   4.2 identificazione degli attributi e delle operazioni (a volte prima che le classi siano trovate!);
   4.3 identificazione delle associazioni tra le classi;
   4.4 identificazione delle molteplicità delle associazioni;
   4.5 identificazione dei ruoli;
   4.6 identificazione dei vincoli.

Inoltre, è bene tener presente alcuni suggerimenti: migliorare la leggibilità del class diagram, gestire la modellazione a oggetti, prevedere differenti utenti del class diagram, non inserire più 7 classi, al massimo 8-9, nel package, aggiungere le tassonomie (relazioni, associazioni, molteplicità, …) in un diagramma separato. Un'euristica per la modellazione a oggetti in pratica è :

- organizzare meeting per identificare gli oggetti;
- trovare per prima cosa gli oggetti giusti;
- provare a differenziarli tra entità, interfacce e control object;
- trovare le associazioni e le loro molteplicità (molteplicità insolite di norma conducono a nuovi oggetti o categorie);
- identificare le eriditarietà: osservare le tassonomie, categorizzare;
- identificare le aggregazioni.;
- fare brainstorming, e iterare, iterare, …

6.3 Ruoli differenti durante lo sviluppo

Perché usiamo il class diagram? L'obiettivo principale è quello di descrivere le proprietà statiche di un sistema. Lo usa il cliente e l'utente finale, perché di solito focalizzano di più sulle funzionalità del sistema. Vengono usati anche dagli esperti del dominio applicativo, per modellarlo e dagli sviluppatori durante lo sviluppo del sistema, durante l'analisi, il system design, l'object design e durante l'implementazione.
In accordo all'attività di sviluppo, gli sviluppatori giocano ruoli differenti: Analista, Progettista del Sistema, Progettista di dettaglio, Implementatore. In piccoli sistemi, alcuni di questi ruoli non esistono, o sono rivestiti dalla stessa persona. In ogni caso, ognuno di questi ruoli consente di vedere il modello da ottiche differenti. Prima di descrivere questi punti di vista, è bene distinguere tra due tipi di classi che appaiono nel class diagram: le **classi del dominio applicativo** e quelle **del dominio delle soluzioni**.
Il dominio applicativo è il dominio del problema (servizi finanziari, meteorologia, gestione degli incidenti, architettura,…). Le classi di questo dominio sono astrazioni in quel dominio. Se modelliamo applicazioni di business, queste classi sono chiamate anche *business object*.

Il dominio delle soluzioni, invece, è il dominio che aiuta nella soluzione del problema (telecomunicazione, database, costruttori di compilatori, sistemi operativi,…). Le classi di questo dominio sono astrazioni introdotte per motivi tecnici, perché aiutano a risolvere il problema (alberi, tabelle hash, scheduler,…).

In questo scenario, l'**Analista**, ad esempio, è interessato solo alle classi del dominio applicativo: le associazioni tra queste sono relazioni tra astrazioni nel dominio applicativo, oppure l'uso di ereditarietà nel modello riflette tassonomie nel dominio applicativo (una **tassonomia** è una gerarchia di astrazioni). L'analista non è interessato all'esatta signature delle operazioni e alle classi del dominio delle soluzioni.
Il **Designer**, invece, focalizza la sua attenzione sulla soluzione del problema, che è appunto il dominio delle soluzioni. La progettazione consiste di molti task (decomposizione in sottosistemi, selezione della piattaforma hardware, gestione dei dati del sistema, …).
Un importante problema di progettazione è la specifica delle interfacce: il progettista descrive le interfacce di classi (object design) e sottosistemi (system design) e il suo obiettivo è l'usabilità e la riusabilità di queste interfacce. *Design-usability* significa che le interfacce sono usabili da più classi possibile nel sistema, mentre *Design-reusability* significa che la definizione delle interfacce deve essere tale da fare in modo che le interfacce possano essere usate anche in altri (futuri) sistemi software (ciò comporta la costruzione di librerie di classi).

Ulteriormente differente, può essere il punto di vista dell'**Implementatore**. Ce ne sono di tre tipi:

- *Class implementor*: implementa le classi, scegliendo le strutture dati appropriate (per gli attributi), gli algoritmi (per le operazioni), e realizza l'interfaccia delle classi in un linguaggio di programmazione.
- *Class extender*: estende le classi in sottoclassi che servono per nuovi problemi o nuovi domini applicativi.
- *Class-user (cliente)*: è il programmatore che vuole usare una classe esistente (ad esempio, una classe da una libreria o da un altro sistema), oppure è l'utente della classe, che è solo interessato alla signature delle operazioni e delle precondizioni della classe, sotto cui può essere invocata. Pertanto, il class-user non è tanto interessato all'implementazione della classe.

Perché distinguiamo tra questi differenti usi delle classi? I modelli spesso non distinguono tra classi del dominio applicativo e del dominio delle soluzioni, perché il linguaggi di modellazione come UML sono stati progettati per poter fornire l'utilizzo di entrambi i tipi nello stesso modello. Si preferirebbe comunque non avere classi del dominio delle soluzioni nel modello di analisi.
Inoltre, alcuni sistemi non distinguono tra specifica e implementazione di una classe, questo perché i linguaggi di programmazione orientati agli oggetti permettono simultaneamente l'uso di specifiche e implementazioni. Si preferirebbe che il progetto del modello a oggetti non contenesse implementazioni.
Distinguiamo tra questi punti di vista differenti, perché la chiave per creare sistemi software di alta qualità è proprio l'esatta distinzione tra classi del dominio applicativo e del dominio delle soluzioni, e tra specifica delle interfacce e specifica delle implementazioni.

6.4 Considerazioni conclusive

Nasce spontanea una domanda: tra i vari aspetti della modellazione a oggetti, come Operazione, Signature, Metodo, quale bisogna usare e quando?

- *Operazione*: una funzione o una trasformazione applicata agli oggetti in una classe. Tutti gli oggetti in una classe condividono la stessa operazione. Si usa nella fase di analisi.
- *Signature*: numero e tipi di argomenti, tipi di valore dei risultati. Si usa nella fase di object design.
- *Metodo*: implementazione di un'operazione per una classe. Si usa nella fase di implementazione.
- *Operazione polimorfica*: la stessa operazione applicata a differenti classi.

I class diagram sono spesso parti di modelli, li troviamo nel modello di Analisi, che modella il dominio applicativo e nei modelli di System Design e Object Design che modellano il dominio delle soluzioni. A seconda del nostro ruolo, vediamo gli oggetti e i modelli da differenti prospettive. Spesso siamo interessati solo ad aspetti limitati di un modello. Inoltre, a seconda del nostro ruolo e del modello, abbiamo differenti interpretazioni per differenti costrutti UML: differenti interpretazioni per associazioni, attributi ed ereditarietà.
Diamo uno sguardo a queste interpretazioni. Il **modello di Analisi** è costruito durante la fase di analisi e i principali stakeholder sono l'utente finale, il cliente, l'analista. Il diagramma contiene solo classi del dominio applicativo. Questo modello è alla base della comunicazione tra analisti, esperti del dominio applicativo e utenti finali del sistema.
Il **modello dell'object design** (a volte chiamato anche **specification model**) è, invece, creato durante la fase di object design. I principali stackholder sono gli specificatori di classi, gli implementatori delle classi e gli utenti delle classi. Il class diagram contiene classi del dominio applicativo e delle soluzioni. Questo modello è alla base della la comunicazione tra progettisti e implementatori.

7. ANALISI DEI REQUISITI (DYNAMIC MODELING)

7.1 Overview

Abbiamo già visto come individuare le classi per il modello a oggetti. A questo punto, vediamo come identificarle nel **dynamic model**. Gli eventi in un *sequence diagram* così come le azioni e le attività nello *state chart diagram* sono candidati ad essere le operazioni pubbliche delle classi. Le activity line nel sequence diagram sono anch'esse candidate a essere oggetti.
I diagrammi per la modellazione dinamica sono di due tipi: **Interaction diagram**, che descrivono il comportamento dinamico tra gli oggetti e gli **Statechart diagram** che descrivono il comportamento dinamico di un singolo oggetto.
Gli **interaction diagram** si dividono a loro volta in:

- **Sequence diagram**: mostrano il comportamento dinamico di un insieme di oggetti organizzati in sequenze di tempo. Buoni per specifiche e complessi scenari in real-time.
- **Collaboration diagram**: mostrano le relazioni tra gli oggetti, ma non mostrano il tempo.

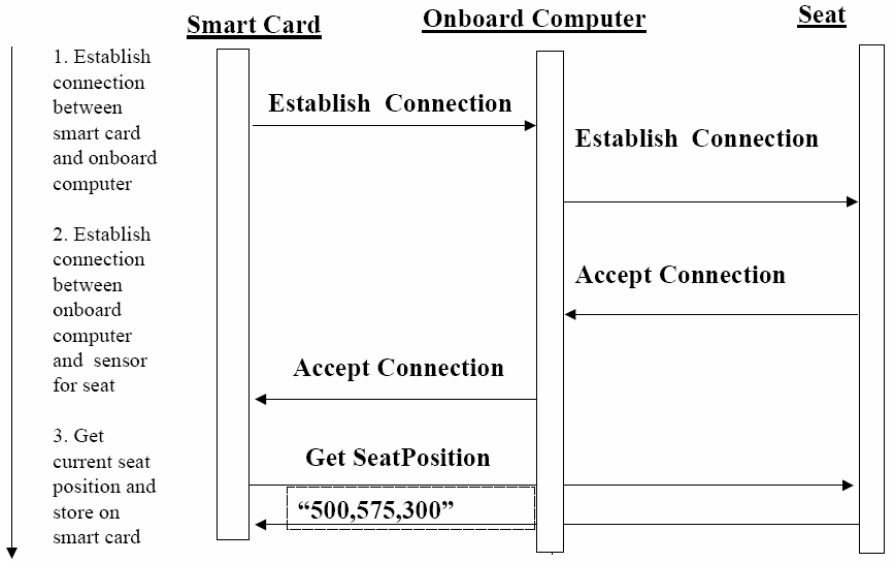
Gli **statechart diagram**, invece, sono macchine a stati che descrivono le risposte di un oggetto di una determinata classe agli stimoli percepiti dall'esterno (*eventi*). Un tipo speciale di questi diagrammi sono gli **Activity diagram**, in cui ogni stato è uno *stato action*.

Un **modello dinamico** è una collezione di più statechart diagram, uno per ogni classe con comportamento dinamico rilevante. L'obiettivo di questo modello è quello di individuare e fornire metodi per il modello a oggetti. Per ottenerlo, si comincia dai casi d'uso o dagli scenari (dai flussi di eventi) e si modellano le interazioni tra gli oggetti mediante il sequence diagram, per poi modellare i comportamenti dinamici di ogni singolo oggetto, mediante gli statechart diagram.
Si parte perciò dal flusso di eventi. Ma cosa è un **evento**? E' qualcosa che accade ad un certo punto. Gli eventi possono essere in relazione con altri: casualmente correlati (prima, dopo) o casualmente non correlati (concorrenti). Inviano informazioni da un oggetto a un altro e possono essere raggruppati in classi con una struttura gerarchica. Il termine "evento", quindi, è spesso usato in due modi differenti, per far riferimento a un'istanza di una classe di eventi o ad attributi di una classe di eventi.

7.2 Sequence Diagram

Dal flusso di eventi nel caso d'uso o scenario, quindi, si procede al **sequence diagram**. Un sequence diagram è una descrizione grafica degli oggetti che partecipano al caso d'uso o allo scenario, usando la notazione dei grafi aciclici direzionati. E' chiaro, allora, che gli oggetti possono essere identificati oltre che durante la modellazione a oggetti, anche durante la modellazione dinamica. Un'euristica per ottenere un sequence diagram è basata sulle seguenti osservazioni: un evento ha sempre uno trasmettitore e un ricevitore; la rappresentazione degli eventi è a volte chiamata *messaggio*; trovare questi messaggi per ogni evento significa trovare gli oggetti partecipanti al caso d'uso.

Esempio di sequence diagram:



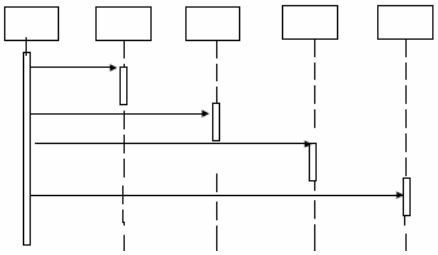
Ecco le regole per la costruzione di un buon sequence diagram:

- *Layout*: la prima colonna dovrebbe corrispondere all'attore che ha iniziato il caso d'uso, la seconda ai *boundary object* e la terza ai *control object* che gestiscono il resto del caso d'uso.
- *Creazione*: i control object sono creati nella fase di inizializzazione di un caso d'uso e i boundary object sono creati dai control object.
- *Accesso*: agli *entity object* accedono i control e i boundary object, e gli entity object non dovrebbero mai chiamare i boundary e i control object. Quest'ultima proprietà permette di poter condividere entity object tra vari casi d'uso e permette agli entity object di essere immuni ai cambiamenti tecnologici introdotti nei boundary object.

Sui sequence diagram, quindi, possiamo intravedere la struttura dei casi d'uso, e questa struttura ci può aiutare a determinare in che modo è decentralizzato il sistema. Distinguiamo tra due strutture: *Fork* e *Stair diagram*.

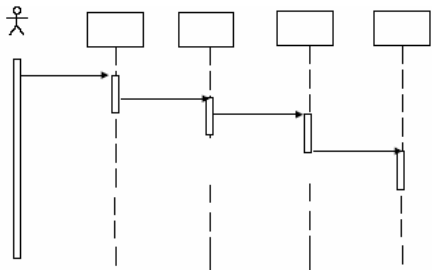
Fork diagram

La maggior parte dei comportamenti dinamici si trovano in un singolo oggetto, di solito nel control object. Questo conosce tutti gli altri oggetti e spesso li usa per inviare richieste e comandi.



Stair diagram

Il comportamento dinamico è distribuito. Ogni oggetto delega alcune responsabilità ad altri e ha solo poche informazioni sugli altri oggetti, ma sa quale di questi può aiutarlo in uno specifico comportamento.

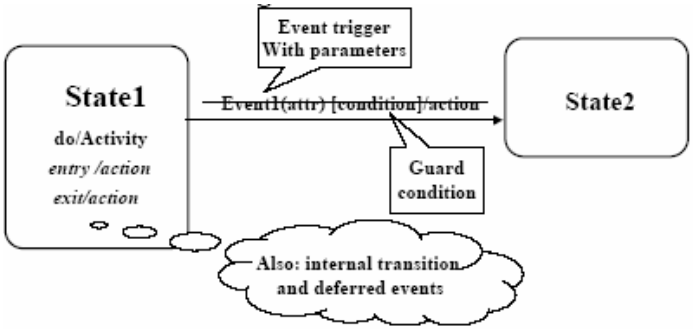


Quale di questi due diagrammi dovrebbe essere usato? I fan dell'object-oriented affermano che la struttura Stair è migliore, perché la maggioranza delle responsabilità è distribuita. Comunque, ciò non è sempre vero. La migliore euristica prevede *strutture di controllo decentralizzate*, in cui le operazioni hanno una forte connessione e vengono eseguite sempre nello stesso ordine, oppure

strutture di controllo centralizzate (supportano meglio i cambiamenti), in cui le operazioni possono cambiare ordine e quelle nuove possono essere inserite come il risultato di nuovi requisiti.

7.3 Statechart diagram

La notazione è basata sul lavoro di *Harel*, e sono state aggiunte poche modifiche orientate agli oggetti. Come si evince dalla figura, uno **statechart diagram** UML può essere mappato in una macchina a stati finiti.



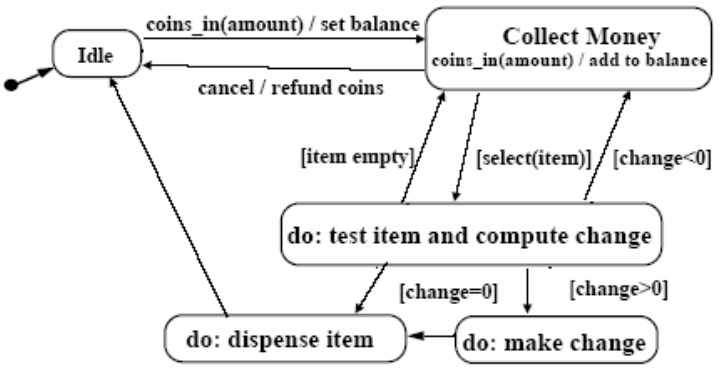
In particolare, si tratta di grafi in cui i nodi sono gli stati e gli archi direzionati sono transizioni etichettate dai nomi degli eventi. Nel diagramma, c'è sempre uno stato iniziale (identificato da un pallino) e uno finale (identificato da un pallino cerchiato). Una transizione rappresenta un cambio di stato a seguito di un evento, di una condizione o del tempo. Ogni qualvolta si verifica un evento, si valuta la *condizione di guardia*, e se è vera, si effettua la transizione. Una transizione che non lascia lo stato di partenza è detta *transizione interna*.
Distinguiamo tra due tipi di operazioni in uno statechart:

- *attività*: operazione che richiede tempo per essere completata ed è associata agli stati;
- *azione*: operazione istantanea, atomica, associata agli eventi e agli stati e avviene a seguito di una Entry, una Exit, una Internal Action.

Uno statechart diagram relaziona eventi e stati per una classe (un modello a oggetti con un insieme di oggetti ha un insieme di state diagram).

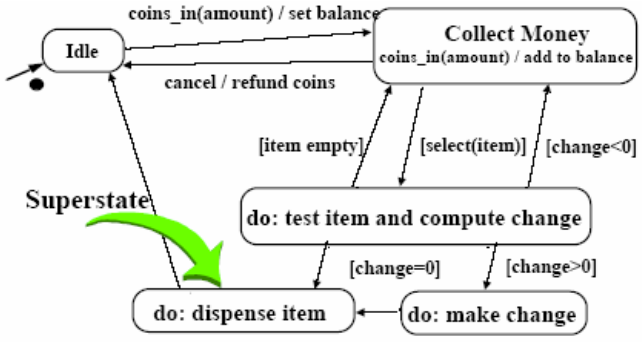
Uno **stato** è l'astrazione degli attributi di una classe ed ha una durata. In altre parole è l'aggregazione di diversi attributi di una classe. Fondamentalmente è una classe di equivalenza contenente tutti quegli attributi che assumono valori e collegamenti che non hanno bisogno di essere distinti fino a dove interessa la struttura di controllo del sistema.

Esempio di statechart diagram:

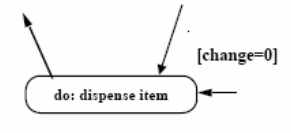


Diagrammi di stato annidati

Le attività negli stati sono elementi compositi che denotano altri diagrammi di stato di livello più basso. Un diagramma di stato lower-level corrisponde a una sequenza di stati lower-level ed eventi che sono invisibili al diagramma di livello superiore. Insiemi di sottostati in un diagramma di stato annidato denotano un **superstato** e sono racchiusi in un grande ovale, anche detto *contour* (contorno, profilo).



Nell'esempio, '*dispense item*' è un'attività atomica:



che possiamo espandere in una attività composta:

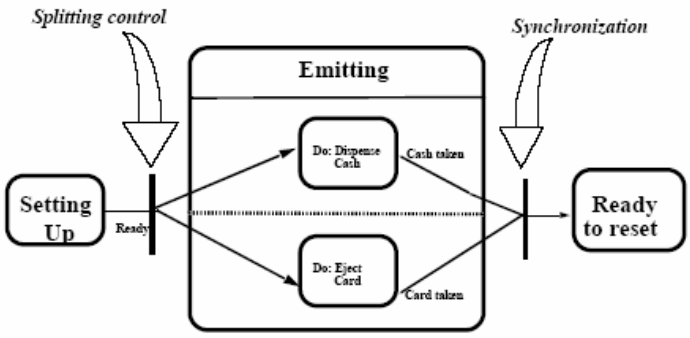
L'obiettivo dei superstati è, quindi, quello di evitare i *modelli a spaghetti* e di ridurre il numero di linee in un diagramma di stati. Le transizioni da altri stati al superstato entrano nel primo sottostato del superstato, e quelle da un superstato verso altri stati sono ereditate da tutti i sottostati (eridarietà di stati), ad eccezione delle transizioni che hanno bisogno di essere eseguite alla fine dell'attività associata allo stato (come nell'esempio precedente).

Modellare la concorrenza

Questi diagrammi sono utilizzati anche per modellare la concorrenza, che può essere di due tipologie differenti:

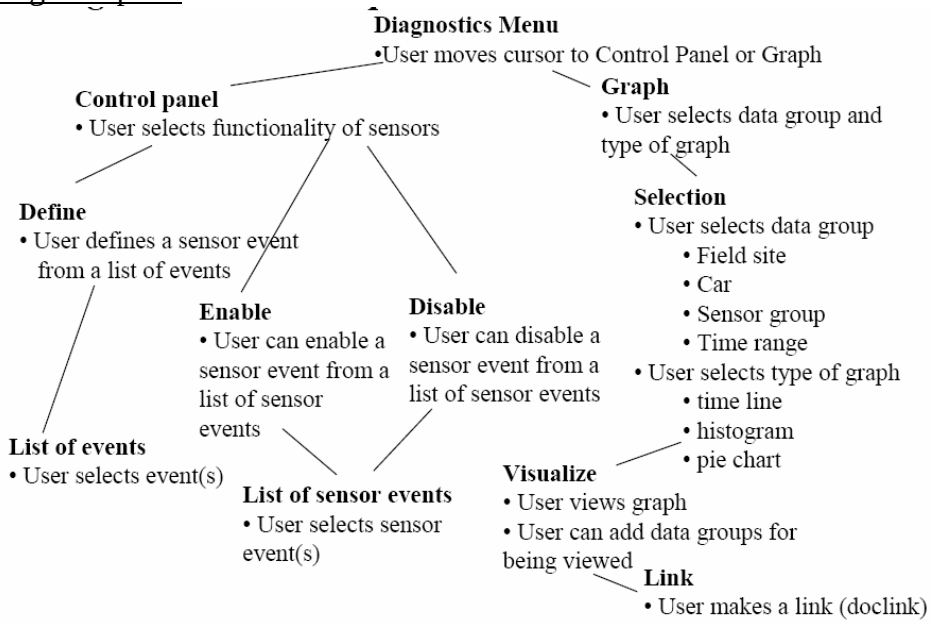
- *Concorrenza di sistema*: gli stati di tutto il sistema sono l'aggregazione di diagrammi di stato, uno per ogni oggetto. Ogni diagramma di stato è eseguito concorrentemente ad altri.
- *Concorrenza di oggetti*: un oggetto può essere partizionato in sottoinsiemi di stati (attributi e collegamenti) tali che ognuno di essi ha il suo sottodiagramma. Lo stato dell'oggetto consiste di un insieme di stati, uno stato da ogni sottodiagramma. I diagrammi di stato sono divisi in sottodiagrammi mediante linee marcate.

Esempio di concorrenza tra oggetti:



Modellazione dinamica delle interfacce utente
Gli statechart diagram possono essere usati per progettare interfacce utente, per questo sono anche detti **Navigation Path**. Gli stati sono i nomi delle schermate: la struttura grafica delle schermate associata agli stati aiuta quando si presenta il modello dinamico di un'interfaccia utente. Le attività e le azioni sono indicate con dei pallini sotto il nome della schermata (spesso è mostrata solo l'azione exit). Le transizioni di stato sono il risultato dell'azione exit, del click su un bottone, di una scelta dal menù, di movimenti del cursore. Questo approccio è ottimo per progettare interfacce utente in sistemi web-based.

Esempio di navigation path:



Riassumendo, quindi, gli statechart diagram aiutano a identificare i cambiamenti in un oggetto individuale nel tempo, mentre i sequence diagram aiutano ad identificare le relazioni temporali tra oggetti nel tempo e la sequenza delle operazioni come risposta a uno o più eventi.

Suggerimenti pratici per modellare dinamicamente:

- Costruire i modelli dinamici solo per le classi con comportamenti dinamici significativi, in modo da evitare la "paralisi" dell'analisi.
- Considerare solo gli attributi rilevanti, usando anche astrazione se necessario. Osservare la granularità dell'applicazione quando si decidono le azioni e le attività.
- Ridurre l'ingombro di notazione, provando a mettere azioni in stati contenitore (tenendo conto che azioni identiche sugli eventi conducono allo stesso stato).

7.4 Sommario: Analisi dei Requisiti

1. Quali sono le trasformazioni? => **Modellazione funzionale**: creare gli scenari e i diagrammi dei casi d'uso, parlando col cliente, osservando, prendendo informazioni sul passato, facendo esperimenti.

2. Qual è la struttura del sistema? => **Modellazione a oggetti**: creare i class diagram (identificare gli oggetti, le associazioni, le molteplicità, gli attributi, le operazioni).

3. Qual è il suo comportamento? => **Modellazione dinamica**: creare i sequence diagram (identificare trasmettitore e ricevitore, mostrare la sequenza degli eventi scambiati tra gli oggetti, identificare le dipendenze tra eventi e la concorrenza tra gli eventi) e creare gli statechart diagram (solo per gli oggetti dinamicamente interessanti).

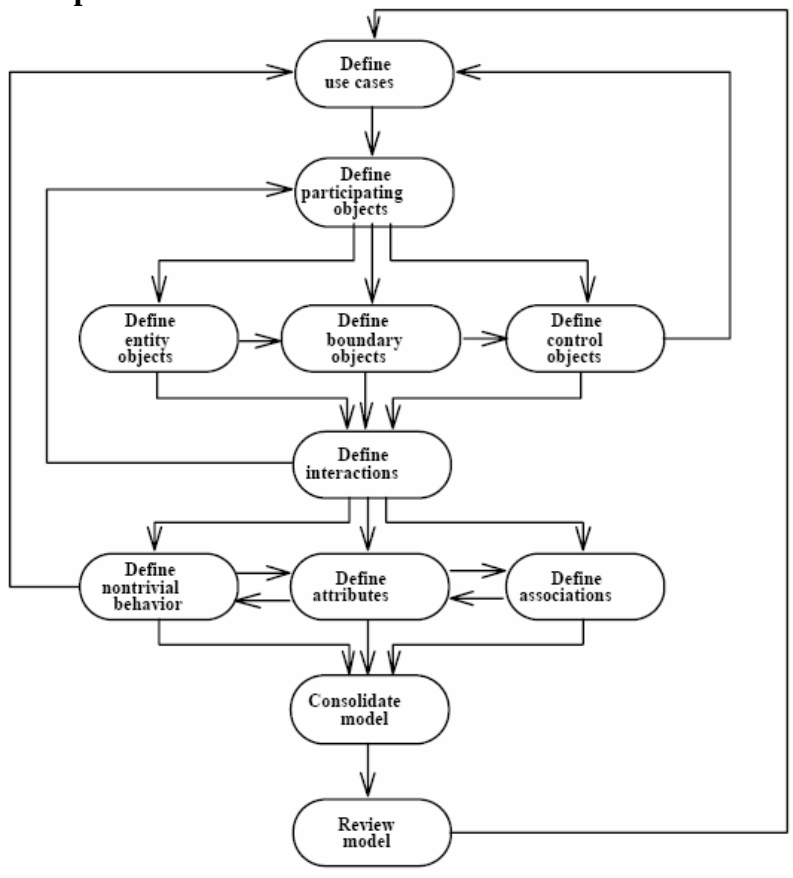
In altre parole, fare analisi significa:

1. *analizzare il problem statement*: identificare i requisiti funzionali, non funzionali e i vincoli (pseudo requisiti);
2. *costruire il modello funzionale*: sviluppare i casi d'uso per illustrare i requisiti funzionali;
3. *costruire il modello dinamico*: sviluppare i sequence diagram per illustrare le interazioni tra gli oggetti e sviluppare gli statechart diagram per gli oggetti con comportamenti interessanti;
4. *costruire il modello a oggetti*: sviluppare i diagrammi delle classi per mostrare la struttura del sistema.

Analisi collaborativa

Un **sistema** è una collezione di sottosistemi che forniscono servizi. L'analisi di questi servizi è fornita da un insieme di team che producono i modelli per i loro sottosistemi. L'integrazione dei modelli dei sottosistema nel modello completo del sistema viene effettuata dall'*architecture team*.
Si utilizzano anche **checklist** (liste di controllo) come integrazione all'analisi: Tutte le classi sono menzionate nel dizionario? I nomi dei metodi sono consistenti con i nomi delle azioni, delle attività, degli eventi e dei processi?…
Si utilizzano controlli basati su assunzioni fatte da ogni servizio: si controllano metodi mancanti, classi, associazioni pendendi,…

Activity diagram UML per fare analisi:



Template del documento di analisi dei requisiti (RAD)

1. Introduction
2. Current system
3. Proposed system
   - 3.1 Overview
   - 3.2 Functional requirements
   - 3.3 Nonfunctional requirements
   - 3.4 Constraints ("Pseudo requirements")
   - 3.5 System models
     - 3.5.1 Scenarios
     - 3.5.2 Use case model
     - 3.5.3 Object model
       - 3.5.3.1 Data dictionary
       - 3.5.3.2 Class diagrams
     - 3.5.4 Dynamic models
     - 3.5.5 User interface
4. Glossary

Di seguito, il dettaglio della sezione 3.5 e della sezione 3.3:

3.5.1 Scenarios
- **As-is scenarios, visionary scenarios**

3.5.2 Use case model
- **Actors and use cases**

3.5.3 Object model
- **Data dictionary**
- **Class diagrams (classes, associations, attributes and operations)**

3.5.4 Dynamic model
- **State diagrams for classes with significant dynamic behavior**
- **Sequence diagrams for collaborating objects (protocol)**

3.5.5 User Interface
- **Navigational Paths, Screen mockups**

3.3.1 User interface and human factors
3.3.2 Documentation
3.3.3 Hardware considerations
3.3.4 Performance characteristics
3.3.5 Error handling and extreme conditions
3.3.6 System interfacing
3.3.7 Quality issues
3.3.8 System modifications
3.3.9 Physical environment
3.3.10 Security issues
3.3.11 Resources and management issues

Le domande che si possono porre per ottenere i requisiti non funzionali sono le seguenti:

3.3.1 User interface and human factors:
- Che tipo di utente userà il sistema?
- Saranno più di un tipo ad usarlo?
- Che tipo di esperienze saranno richieste ad ogni tipo di utente?
- E' particolarmente importante che il sistema sia facile da imparare?
- E' particolarmente importante che gli utenti siano protetti dal commettere errori?
- Che tipi di periferiche input/output per le interfacce con l'uomo sono disponibili, e che caratteristiche hanno?

3.3.2 Documentation
- Che tipo di documentazione è richiesta?
- A quale pubblico sono indirizzate le documentazioni?

3.3.3 Hardware considerations

- Che hardware utilizza il sistema proposto?
- Che caratteristiche deve avere, comprese dimensioni di memoria e spazi di memorizzazione ausiliari?

3.3.4 Performance characteristics

- Ci sono vincoli sulla velocità, sul throughput o sui tempi di risposta del sistema?
- Ci sono vincoli sulla dimensione o sulla capacità dei dati che processerà il sistema?

3.3.5 Error handling and extreme conditions

- Come dovrebbe rispondere il sistema ad errori di input?
- Come dovrebbe rispondere il sistema in condizioni estreme?

3.3.6 System interfacing

- Gli input possono arrivare anche da sistemi esterni?
- Gli output possono essere indirizzati anche a sistemi esterni?
- Ci sono restrizioni sui formati o sui supporti che devono essere usati per l'input o per l'output?

3.3.7 Quality issues

- Quali sono i requisiti per la reliability (affidabilità)?
- Il sistema deve intercettare i fallimenti?
- Dopo quanto tempo il sistema deve essere riavviato a seguito di un fallimento?
- Quanto è accettabile che il sistema sia down per un periodo di 24 ore?
- E' importante che il sistema sia portabile?

3.3.8 System modifications

- Quali parti del sistema sono le maggiori candidate a subire modifiche successive?
- Che tipo di modifiche ci si deve aspettare?

3.3.9 Physical environment

- Dove verrà installato l'equipaggiamento del sistema?
- Si troverà in uno o più posti?
- Le condizioni ambientali saranno in qualche modo fuori dell'ordinario (ad esempio, temperature insolite, vibrazioni, campi magnetici,…)?

3.3.10 Security issues

- L'accesso ai dati o al sistema deve essere controllato?
- La sicurezza fisica è un problema?

3.3.11 Resources and Management issues

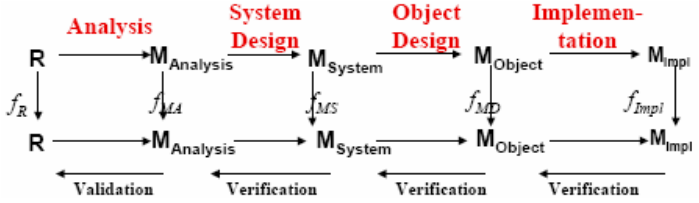
- Con che frequenza verrà fatto il backup del sistema?
- Chi ne sarà responsabile?
- Chi è il responsabile dell'installazione del sistema?
- Chi sarà il responsabile della sua manutenzione?

Per quanto riguarda infine i *vincoli* (pseudo requisiti), si fa riferimento a qualsiasi restrizione ponga il cliente sul dominio delle soluzioni. Ad esempio: la piattaforma finale deve essere IBM/360 (è un mainframe), il linguaggio di implementazione deve essere COBOL, deve essere usato uno standard

particolare per la documentazione, deve essere usato un dataglove, deve essere usato ActiveX, il sistema deve essere interfacciato con un lettore di fogli perforati, …

Verifica e Validazione dei modelli

La **verifica** è l'equivalente del controllo tra la trasformazione tra due modelli: abbiamo due modelli, la trasformazione tra questi è corretta? La **validazione** è differente. Non abbiamo due modelli, ma dobbiamo confrontarne uno con la realtà, che potrebbe anche essere un sistema artificiale, come un sistema già esistente.



Le tecniche che si usano per validare sono basate su revisioni formali o informali, e includono i seguenti controlli: **correttezza**, **completezza**, **consistenza**, **ambiguità**, **realismo**.

Controllare la *consistenza* significa fare controlli incrociati tra le classi, per verificare la consistenza dei nomi, degli attributi e dei metodi. Controllare la *completezza* significa identificare associazioni pendenti, che non puntano a niente, identificare classi definite due volte, identificare classi mancanti (a cui si fa riferimento magari in un sottosistema ma non definite da nessuna parte). Controllare le *ambiguità* significa controllare nomi scritti male, classi con gli stessi nomi ma con significati differenti, etc.

A tal proposito, è importante modellare anche le **Checklist** per le revisioni:

- Il modello è corretto? Un modello è corretto se rappresenta il sistema dal punto di vista dell'utente.
- Il modello è completo? Un modello è completo se è stato descritto ogni scenario del sistema, comprese le eccezioni.
- Il modello è consistente? Un modello è consistente se non ha componenti in contraddizione.
- Il modello è non ambiguo? Un modello è non ambiguo se descrive un sistema (una sola realtà), non molti.
- Il modello è realistico? Un modello è realistico se può essere implementato senza problemi.

Inoltre, un problema che si può incontrare durante la modellazione è che, di norma, descriviamo il modello di un sistema da differenti punti di vista (class diagram, use case, sequence diagram, statechart diagram). Si ha, quindi, il bisogno di controllare l'equivalenza di queste viste nel migliore dei modi. Per farlo bisogna eseguire dei controlli sintattici sui modelli:

- controllare la consistenza dei nomi delle classi, degli attributi, dei metodi in differenti sottosistemi;
- identificare associazioni pendenti;
- identificare classi doppiamente definite;
- identificare classi mancanti, menzionate in un modello ma non definite in nessun documento;
- controllare le classi con gli stessi nomi ma con significati differenti.

L'importante è non fidarsi dei CASE tool per tali controlli, perché molti di questi tool non li effettuano tutti.

Project Agreement (accordo)

Al termine di tutte le fasi, si passa al **project agreement**. Il project agreement rappresenta l'accettazione del modello di analisi (o di parte di esso) da parte del cliente, fornito dal documento di analisi dei requisiti. Il cliente e gli sviluppatori convergono a una singola idea e concordano sulle funzioni e le caratteristiche che il sistema dovrà avere. In più, concordano su: una lista di requisiti

prioritari, una revisione del processo, una lista di criteri che saranno usati per accettare o rifiutare il sistema, uno schedule e un budget.
Per quanto riguarda la priorità dei requisiti, questa può suddivisa in 3 categorie:

- **Alta** (*requisiti core*): bisogna tenerne conto durante le fasi di analisi, di progettazione e di implementazione. Una caratteristica ad alta priorità deve essere dimostrata con successo durante l'accettazione da parte del cliente.

- **Media** (*requisiti opzionali*): bisogna tenerne conto durante le fasi di analisi e di progettazione. Di solito l'implementazione e la dimostrazione avviene alla seconda iterazione del processo di sviluppo del sistema.

- **Bassa** (*requisiti di fantasia*): bisogna tenerne conto durante la fase di analisi, ma si tratta di scenari "visionari". Illustrano come il sistema verrà usato in futuro, anche se non esistono ancora le tecnologie di cui si avrebbe bisogno.

Chiaramente, a seconda delle esigenze e degli scopi del progetto, si possono prevedere ulteriori categorie di priorità, ad esempio altissima, medio-alta, medio-bassa, bassissima, e così via.

8. SYSTEM DESIGN (DECOMPOSIZIONE DEL SISTEMA)

8.1 Overview

Il **System Design** (progettazione del sistema) è la trasformazione del modello di analisi nel **System Design Model** (modello di progettazione del sistema).
Perché il Design è così difficile? A differenza dell'Analisi, che focalizza sul dominio applicativo, il Design focalizza sul dominio delle soluzioni. Le conoscenze di Design, quindi, sono in continua evoluzione e le ragioni per cui si prendono decisioni di design cambiano molto rapidamente. In generale, comunque, gli sviluppatori devono raggiungere dei compromessi fra i vari obiettivi di design che sono spesso in conflitto gli uni con gli altri e devono anticipare molte decisioni relative alla progettazione, pur non avendo una chiara idea del dominio della soluzione.
Lo **scopo** del System Design è quello di costruire un "ponte" tra il sistema desiderato e quello esistente, in maniera maneggevole, usando la tecnica *Divide and Conquer*, in maniera da modellare il sistema come un insieme di sottosistemi.
In generale, il System Design è costituito da una serie di attività:

- Identificare gli obiettivi di design: gli sviluppatori identificano e definiscono le priorità delle qualità del sistema.
- Progettare la decomposizione del sistema in sottosistemi: basandosi sugli use case e sui modelli di analisi, gli sviluppatori decompongono il sistema in parti più piccole, utilizzando stili architetturali standard.
- Raffinare la decomposizione in sottosistemi per rispettare gli obiettivi di design: la decomposizione iniziale di solito non soddisfa gli obiettivi di design, per questo gli sviluppatori devono raffinarla finché gli obiettivi non sono soddisfatti.

In termini più dettagliati, queste attività sono:

1. Definire gli obiettivi di design (definizione, trade off).
2. Decomporre il sistema in sottoinsiemi più piccoli (strati, partizioni, usando accoppiamenti e coesioni) che possono essere realizzati da team individuali.
3. Gestire la concorrenza, identificando thread.
4. Scegliere le strategie di mapping hardware/software.
5. Scegliere le strategie relative alla gestione dei dati persistenti (oggetti persistenti, file, database, strutture dati).
6. Gestire globalmente le risorse: politiche di controllo degli accessi e politiche di sicurezza.
7. Scegliere il controllo software: monolitico, event-driven, thread, processi concorrenti.
8. Gestire le boundary condition (inizializzazione, terminazione, fallimenti).

Il modello di analisi descrive il sistema dal punto di vista degli attori e serve come base fra il cliente e gli sviluppatori. Non contiene informazioni sulla struttura interna del sistema, sulla sua configurazione hardware e, in generale, su come il sistema dovrebbe essere realizzato.
L'analisi dei requisiti fornisce in output il modello dei requisiti. Ecco come usare questo modello per il System Design:

- Requisiti non funzionali => (1) Definizione degli obiettivi di design.
- Modello funzionale => (2) Decomposizione del sistema (selezione di sottosistemi basandosi sui requisiti funzionali, su coesione e su accoppiamento).
- Modello a oggetti => (4) Mapping hardware/software e (5) Gestione dei dati permanenti.
- Modello dinamico => (3) Gestione della concorrenza, (6) Gestione globale delle risorse, (7) Controllo software.

- Decomposizione in sottosistemi => (8) Gestione delle boundary condition.

I prodotti del system design, quindi, sono:
- Obiettivi di design, descrivono la qualità del sistema.
- Architettura software, descrive: la decomposizione del sistema in termini delle responsabilità dei sottosistemi, in modo che ogni sottosistema possa essere assegnato ad un team e realizzato indipendentemente; le dipendenze fra sottosistemi; l'hardware associato ai vari sottosistemi; le politiche relative al flusso di controllo, al controllo degli accessi e alla memorizzazione dei dati.
- Boundary use case, descrivono la configurazione del sistema, le scelte relative allo startup, allo shutdown ed alla gestione delle eccezioni.

8.2 Identificare gli Obiettivi di Design

Questo è il primo passo del system design, ed è effettuato per identificare le qualità su cui deve essere focalizzato il sistema. Molti design goals possono essere ricavati dai requisiti non funzionali o dal dominio applicativo, altri sono forniti dal cliente. E' comunque importante formalizzarli esplicitamente, perché ogni importante decisione di design deve essere presa seguendo lo stesso insieme di criteri.
E' possibile selezionare questi obiettivi di design da una lunga lista di qualità desiderabili. I criteri sono organizzati in cinque gruppi:

- *Criteri di Performance*
  Includono i requisiti imposti sul sistema in termini di spazio e velocità: tempo di riposta, throughput, memoria.

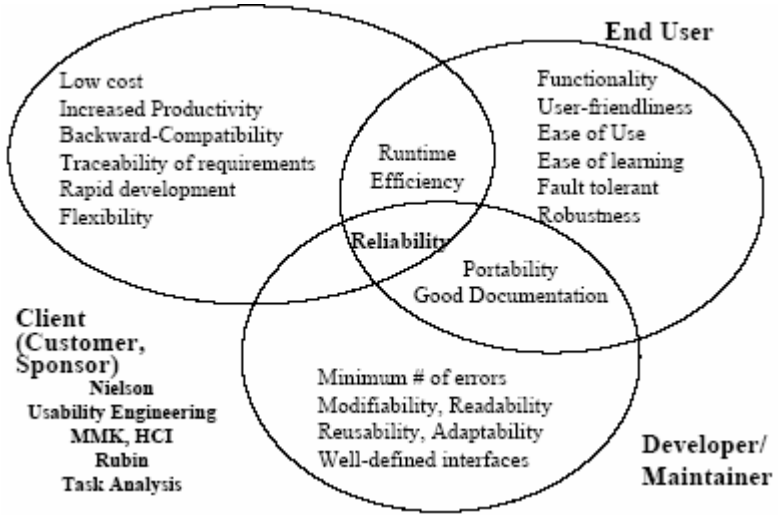
- *Criteri di Affidabilità*
  Riguardano i requisiti che deve avere il sistema per minimizzare i crash e le loro conseguenze: robustezza, affidabilità, disponibilità, tolleranza ai fault, sicurezza, fidatezza.

- *Criteri di Costo*
  Includono i costi per sviluppare il sistema, per metterlo in funzione e per amministrarlo: costo dello sviluppo del sistema iniziale, costo relativo all'installazione del sistema e training degli utenti, costo per convertire i dati del sistema precedente (viene applicato quando è richiesta la compatibilità con il sistema precedente, cioè la backward compatibility), costo di manutenzione, costo di amministrazione.

- *Criteri di Mantenimento*
  Determinano quanto deve essere difficile modificare il sistema dopo il suo rilascio: estendibilità, modificabilità, adattabilità, portabilità, leggibilità, tracciabilità dei requisiti.

- *Criteri di End User*
  Includono qualità che sono desiderabili dal punto di vista dell'utente, ma che non sono state coperte dai criteri di performance e affidabilità: utilità, usabilità.

Relazioni tra gli obiettivi di progettazione:



In realtà, quando si definiscono questi obiettivi, spesso solo un piccolo sottoinsieme di essi può essere tenuto in considerazione. Ad esempio, non è realistico sviluppare software che sia simultaneamente sicuro e costi poco. Gli sviluppatori, perciò, devono dare delle priorità agli obiettivi di design, tenendo conto anche di aspetti manageriali, quali il rispetto dello schedule e del budget. In genere, quindi, si utilizzano dei trade-off tra i principali obiettivi di design, ad esempio:

Funzionalità *vs.* Usabilità
Costo *vs*. Robustezza
Efficienza *vs.* Portabilità
Sviluppo rapido *vs.* Funzionalità
Costo *vs.* Riusabilità
Compatibilità all'indietro *vs.* Leggibilità

Dal punto di vista pratico, la definizione di questi obiettivi di design è da ricercare principalmente tra i requisiti non funzionali, che possono fornire indizi per l'uso dei Design Patterns (modelli di progetto): rileggere il problem statement e usare indizi testuali (come con la tecnica di Abbot durante l'Analisi) per identificare i design pattern. Riportiamo di seguito alcuni esempi, ma solo nei capitoli successivi sarà più chiaro il concetto di Design Pattern:

- "produttore indipendente", "dispositivo indipendente", "deve supportare una famiglia di prodotti" → *Abstract Factory Pattern*;
- "deve interfacciarsi con un oggetto esistente" → *Adapter Pattern*;
- "deve potersi interfacciare con vari sistemi, alcuni dei quali saranno sviluppati in futuro", "un primo prototipo deve essere dimostrato" → *Bridge Pattern*;
- "struttura complessa", "deve avere profondità e ampiezza variabili" → *Composite Pattern*;
- "deve interfacciarsi a un insieme di oggetti esistenti" → *Façade Pattern*;
- "la sua locazione deve essere trasparente" → *Proxy Pattern*;
- "deve essere estendibile", "deve essere scalabile" → *Observer Pattern*;
- "deve fornire una politica indipendente dal meccanismo" → *Strategy Pattern*;

8.3 Decomposizione del Sistema

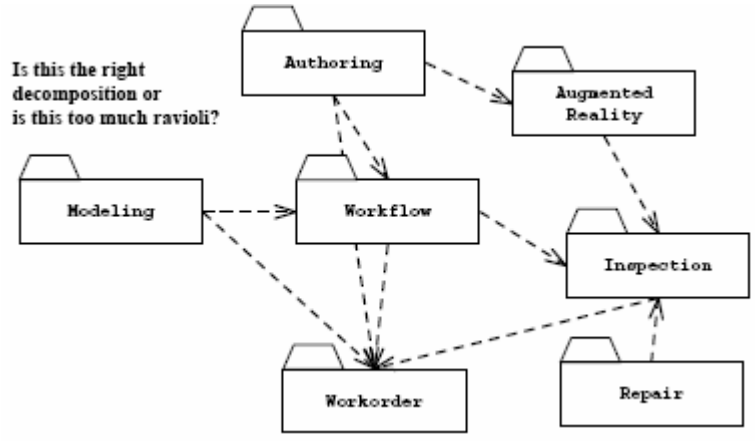
Un **Sottosistema** (Package in UML) è una collezione di classi, associazioni, operazioni, eventi e vincoli che sono in relazione tra loro. Origini dei sottosistemi sono gli oggetti e le classi UML.
I **Servizi** (dei Sottosistemi) sono gruppi di operazioni fornite dai sottosistemi. Trovano origine dai casi d'uso dei sottosistemi e sono specificati dalle **Interfacce dei Sottosistemi**. Queste interfacce specificano le interazioni e il flusso di informazioni da e per i *Subsystem Boundary* (confini dei sottosistemi), ma non interni ai sottosistemi. Dovrebbero essere ben definite e piccole, e spesso sono chiamate **API**: *Application Programmer's Interface* (ma questo termine dovrebbe essere usato durante l'implementazione, non durante il System Design).

In particolare, definiamo:

- **Servizio:** un insieme di operazioni correlate che condividono un obiettivo comune. Servizi di notifica dei sottosistemi sono: LookupChannel(), SubscribeToChannel(), SendNotice(), UnsubscribeFromChannel(). I Servizi sono definiti in fase di System Design.
- **Interfaccia di un Sottosistema:** insieme di operazioni correlate, con signature completamente specializzate. Queste interfacce sono definite durante l'Object Design e sono anche dette API.

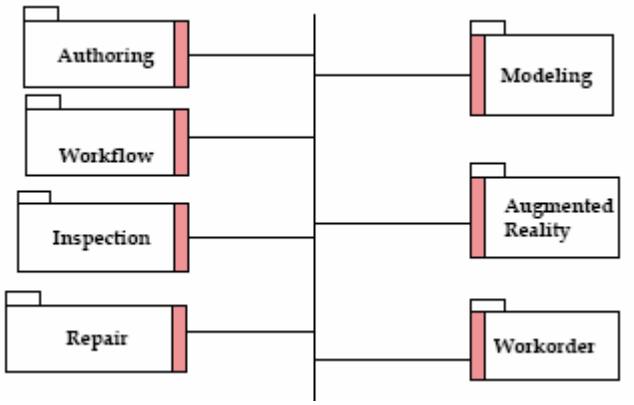
Per ridurre la complessità della soluzione, si decompone il sistema in parti più piccole, in Sottosistemi. Il problema è come selezionare questi sottosistemi. I criteri per sceglierli si basano sull'osservazione che molte interazioni dovrebbero essere interne ai sottosistemi, piuttosto che attraverso i confini (Alta coesione). A tal proposito, il problema primario è del tipo "Che tipo di servizio è fornito dai sottosistemi (interfacce dei sottistemi)?" e quello secondario è del tipo "I sottosistemi possono essere ordinati gerarchicamente (layers)? Che tipo di modello è adatto per descrivere i layers (strati) e le partizioni?"

Esempio di Decomposizione di un sistema:



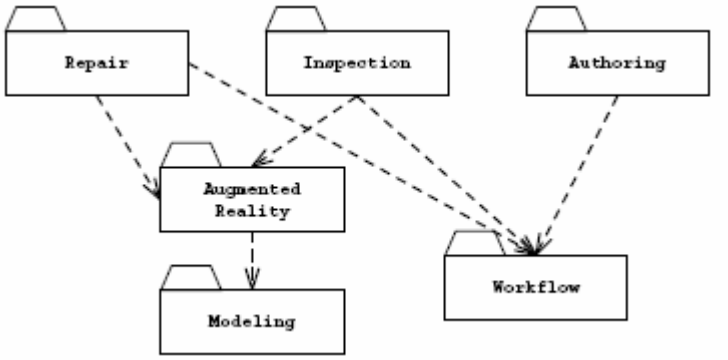
Un **Oggetto Interfaccia di Sottosistema** fornisce un servizio: si tratta di un insieme di metodi pubblici forniti dal sottosistema e l'interfaccia del sottosistema descrive tutti i metodi dell'oggetto interfaccia del sottosistema. E' consigliabile usare un Facade Pattern per questi oggetti.

Esempio: Un sistema visto come un insieme di sottosistemi comunicanti attraverso un bus software



La linea colorata rappresenta l'oggetto interfaccia di un sottosistema, che pubblica il servizio, cioè l'insieme dei metodi, forniti dal sottosistema.

Esempio: Un'architettura a 3 strati



Qual è la relazione tra Modeling e Authoring? C'è bisogno di altri sottosistemi?

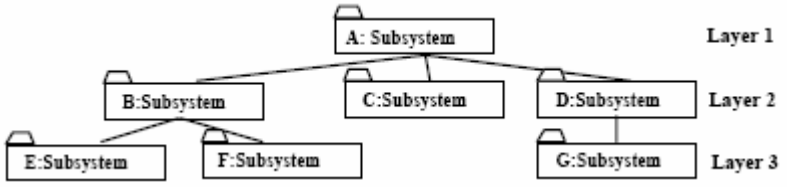
Coupling e Cohesion (Accoppiamento e Coesione)

L'obiettivo è ridurre la complessità mentre avvengono cambiamenti. La **Coesione** misura le dipendenze tra le classi, quindi **Alta Coesione** significa che le classi nel sottosistema eseguono task simili e sono in relazione con ogni altra classe mediante associazioni, mentre **Bassa Coesione**, significa che la maggioranza delle classi sono ausiliare ed eterogenee con nessuna associazione.
Il **Coupling** (Accoppiamento), invece, misura le dipendenze tra i sottosistemi. **Alto Coupling** significa che i cambiamenti a un sottosistema avranno alto impatto sugli altri (cambi del modello, ricompilazione massiccia, etc.), mentre **Basso Coupling** significa che un cambiamento in un sottosistema non incide sugli altri.
I sottosistemi dovrebbero avere la massima coesione e il minimo coupling possibile: Come possiamo realizzare l'alta coesione e come possiamo realizzare il basso coupling?

Partizioni e Strati (Layers)

Il *partizionamento* e la *stratificazione* sono tecniche usate per ottenere un basso coupling. Un sistema grande è di solito decomposto in sottosistemi usando sia gli strati che le partizioni.
Le **Partizioni** dividono verticalmente un sistema in svariati sottosistemi indipendenti (o debolmente accoppiati) che forniscono servizi allo stesso livello di astrazione.
Uno **Strato** è un sottosistema che fornisce servizi di sottosistema a uno strato più alto (livello di astrazione): uno strato può dipendere solo da quelli più bassi e non ha conoscenza di quelli più alti.

Esempio: Decomposizione del Sottosistema in Strati

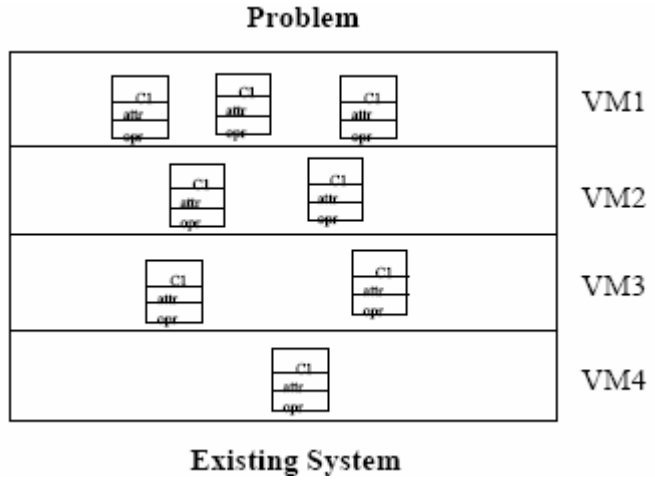


Euristica per la decomposizione del sottosistema: non più di 7 (più o meno 2) sottosistemi, perché troppi sottosistemi aumentano la coesione ma anche la complessità (più servizi), e non più di 4 (più o meno 2) strati (la migliore scelta è per 3).

Le relazioni tra gli strati possono essere del tipo:
- lo strato A "chiama" lo strato B (a runtime);
- lo strato A "dipende" dallo strato B (dipendenze sul "make", a tempo di compilazione).

Le relazioni tra le partizioni possono essere del tipo:
- i sottosistemi hanno mutua ma non profonda conoscenza degli altri;
- la partizione A "chiama" la partizione B e la partizione B "chiama" la partizione A.

Un esempio di sistema stratificato è dato dalla **Virtual Machine** (Dijkstra, 1965). Secondo Dijkstra, 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. In altre parole è un sottosistema, ed è connesso alle macchine virtuali di livello più alto e a quelle di livello più basso mediante associazioni del tipo "fornisce servizi per". Le macchine virtuali possono implementare due tipi di architettura software: **Aperta** e **Chiusa**.
In un'**architettura chiusa**, ogni strato può invocare solo operazioni allo strato immediatamente inferiore, per questo, la stratificazione è detta *opaca*. Gli *obiettivi* di design sono l'alta manutenibiltà e la flessibilità.
In un'**architettura aperta**, invece, ogni strato può invocare operazioni a ogni altro strato inferiore, per questo la stratificazione è detta *trasparente*. L'*obiettivo* di design è l'efficienza a runtime.

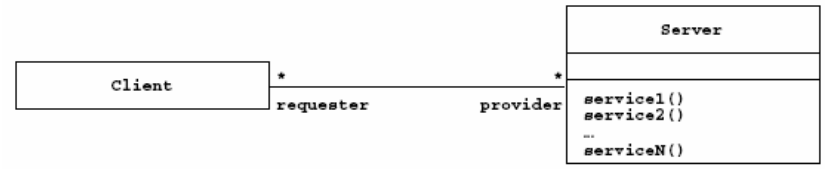
Le principali *proprietà di un sistema stratificato* sono:
- I sistemi stratificati sono preferibili perché gerarchici: la gerarchia riduce la complessità (mediante il low coupling).
- Le architetture chiuse sono più portabili.
- Le architetture aperte sono più efficienti.

- Se un sottosistema è uno strato, è spesso chiamato virtual machine.

8.4 Stili di Architetture Software

Un'architettura software include scelte relative a: decomposizione in sottositemi, flusso di controllo globale, gestione delle boundary condition, protocolli di comunicazione tra i sottosistemi.
La decomposizione in sottosistemi è l'identificazione dei sottosistemi, dei servizi e delle relazioni tra questi. La sua specifica è un'operazione critica: è difficile modificarla quando lo sviluppo è partito, perché bisognerebbe modificare molte interfacce di sottosistemi. *Pattern* per architetture software, cioè stili architetturali che possono essere usati come base di architetture software, sono: *Client/Server*, *Peer-To-Peer*, *Repository*, *Model/View/Controller*, *Pipe e Filtri*.

Stile Architetturale Client/Server
Uno o più *server* forniscono servizi a istanze di altri sottosistemi detti *client*. I client chiamano il server che realizza alcuni servizi e restituisce un risultato. Chiaramente i client conoscono l'interfaccia del server (i suoi servizi) e i server non hanno bisogno di conoscere quelle dei client.
In generale, gli utenti interagiscono solo con il client e la risposta è immediata.



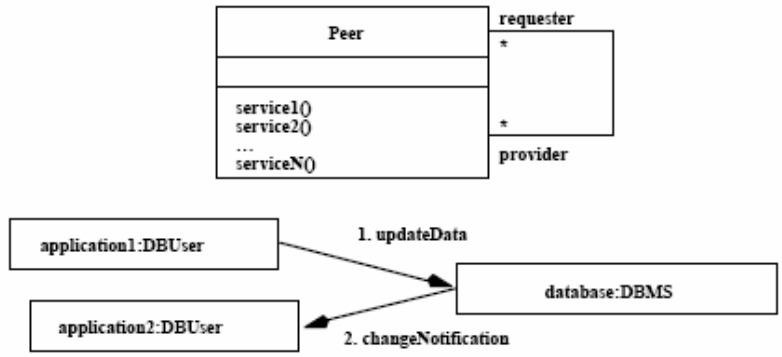
Questo tipo di architettura è spesso utilizzata in sistemi basati su database, dove il front-end è l'applicazione utente (il client) e il back-end gestisce l'accesso e la manipolazione del database (il server). Le funzioni realizzate dal client sono: interfacce utente customized, elaborazione dei dati al front-end, inizio delle chiamate a procedure remote, accesso al database sul server attraverso la rete.
Le funzioni realizzate dal database server sono: gestione centralizzata dei dati, integrità dei dati e consistenza del database, sicurezza del database, operazioni concorrenti (accessi multipli), elaborazione centralizzata (ad esempio archiviazione).
Gli *obiettivi* di design per Sistemi Client/Server sono:

- Portabilità del Servizio: il server può essere installato su una varietà di macchine e sistemi operativi e le funzioni in una varietà di ambienti di rete.
- Trasparenza, Locazione-Trasparenza: il server potrebbe essere distribuito, ma dovrebbe fornire un singolo servizio "logico" all'utente.
- Performance: il client dovrebbe essere in grado di supportare task interattivi display-intensive e il server dovrebbe fornire operazioni CPU-intensive.
- Scalabilità: il server dovrebbe avere capacità disponibili a gestire un grosso numero di client.
- Flessibilità: il sistema dovrebbe essere usabile con una varietà di interfacce utente e periferiche finali.
- Affidabilità: il sistema dovrebbe sopravvivere a problemi nei nodi o nei link di comunicazione.

I problemi con questo tipo di architettura sono dovuti al fatto che i sistemi stratificati non forniscono comunicazione peer-to-peer, di cui si ha spesso bisogno. Ad esempio, un database riceve query da un'applicazione ma invia anche notifiche all'applicazione quando i dati sono cambiati.

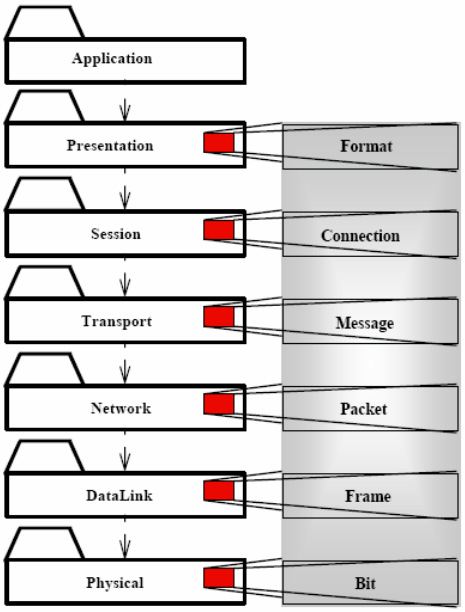
Stile Architetturale Peer-to-Peer

E' una generalizzazione dell'Architettura Client/Server, dove però i client possono essere server e i server possono essere client. E' molto più complessa a causa della possibilità di deadlock.



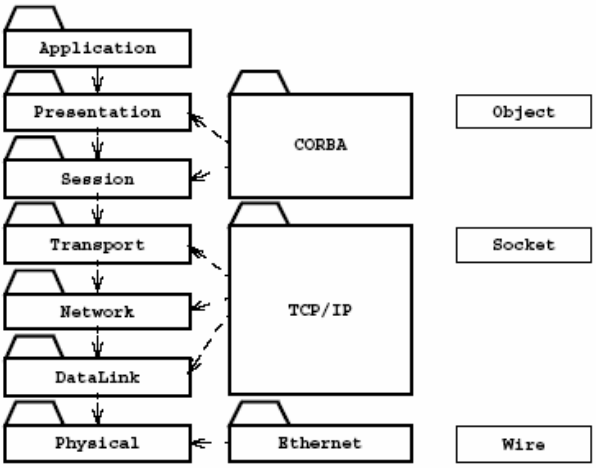
Un esempio di stile architetturale Peer-To-Peer è il **Modello di Riferimento OSI** (Open System Interconnection) di **ISO** (*International Standard Organization*). Questo modello è un'architettura software chiusa, e definisce 7 strati di protocolli di rete e precisi metodi di comunicazione tra questi. Gli strati sono:

- *Livello Fisico*: rappresenta l'interfaccia hardware alla rete e permette di inviare e spedire *bit* sul canale.
- *Livello Datalink*: permette di inviare e ricevere *frame* senza errori usando i servizi del livello Fisico.
- *Livello di Rete*: responsabile dell'affidabilità della trasmissione e dell'instradamento in una rete.
- *Livello di Trasporto*: responsabile della trasmissione affidabile tra due end-point (questa è l'interfaccia vista dai programmatori Unix quando trasmettono su socket TCP/IP).
- *Livello di Sessione*: si occupa di stabilire una connessione, compresa l'autenticazione.
- *Livello di Presentazione*: fornisce servizi di trasformazione dei dati, come scambio di byte e cifratura.
- *Livello di Applicazione*: è il sistema che stiamo progettando (a meno che non si tratti di uno stack di protocolli). Questo strato è spesso stratificato a sua volta.



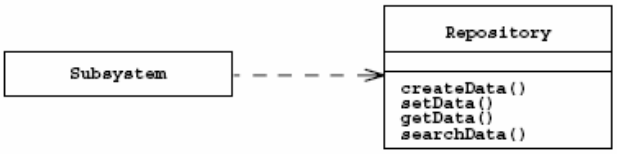
Si può pensare a ogni strato come a un package UML contenente un insieme di oggetti.

Esistono poi dei **Middleware** che implementano più strati del modello, in modo da permettere di focalizzare maggiormente sul Livello di Applicazione:



Stile Architetturale a Repository

I sottosistemi accedono e modificano i dati in una singola struttura chiamata *repository*, e sono loosely coupled (interagiscono solo attraverso il repository). Il flusso di controllo è dettato dal repository centrale (un cambiamento nei dati memorizzati) o dai sottosistemi (lock del repository, primitive di sincronizzazione).



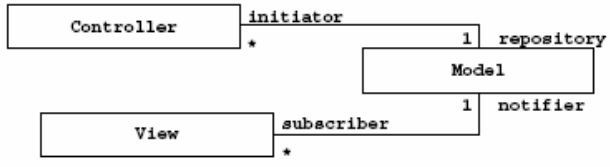
I repository sono adatti per applicazioni con task di elaborazione dati che cambiano di frequente. Una volta definito un repository centrale, possono essere facilmente definiti nuovi servizi sottoforma di sottosistemi addizionali. Tuttavia, il repository centrale può facilmente diventare un collo di bottiglia per aspetti sia di prestazione che di modificabilità. Inoltre, il coupling tra i sottosistemi e il repository è alto, ed è quindi difficile cambiare il repository senza impattare su tutti i sottosistemi.

Questa architettura è anche nota come **Architettura Blackboard**. Un primo sistema di questo tipo è *Hearsay II Speech Understanding System*.

Stile Architetturale Model/View/Controller

In questa architettura i sottosistemi sono classificati in 3 tipi differenti:

- Sottosistema **Model**: mantiene la conoscenza del dominio applicativo.
- Sottosistema **View**: visualizza all'utente gli oggetti del dominio applicativo.
- Sottosistema **Controller**: responsabile della sequenza di interazioni con l'utente e di notificare ai View i cambiamenti nel modello.

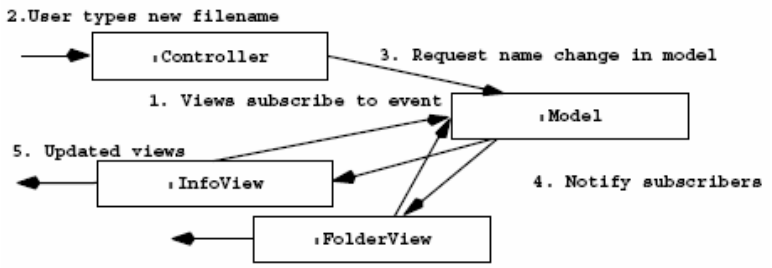


I sottosistemi Model sono sviluppati in modo che non dipendano da alcun sottosistema View o Controller. Cambiamenti nel loro stato sono propagati ai sottosistemi View attraverso un protocollo *"subscribe/notify"*.

Il modello MVC è un caso speciale di Architettura Repository: il sottosistema Model implementa la struttura dati centrale e il sottosistema Controller gestisce esplicitamente il flusso di controllo: ottiene gli input dall'utente e manda messaggi al modello. Il Viewer visualizza il modello.

Il motivo per cui si separano Model, View e Controller è che le interfacce utente sono soggette a cambiamenti più spesso di quanto avviene per la conoscenza del domino applicativo (e quindi del Model).
Questo modello è appropriato per i sistemi interattivi, specialmente quando si utilizzano viste multiple di uno stesso Model. Tuttavia, introduce lo stesso collo di bottiglia visto per lo stile architetturale Repository.

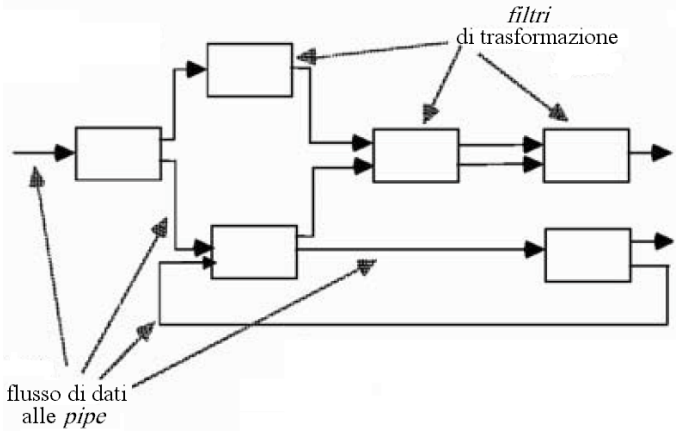
Esempio:



InfoView e FolderView sottoscrivono i cambi al modello File quando sono creati. L'utente digita il nuovo nome del file e il Controller manda la richiesta al Model. Quest'ultimo cambia il nome del file e notifica i subscriber del cambiamento. Entrambi InfoView e FolderView sono aggiornati in modo tale che l'utente veda i cambi in modo consistente.

Pipe e Filtri

Il modello Unix si basa su questo concetto:



I *filtri* (sottosistemi) ricevono e inviano dati attraverso le loro *pipe* di input e output (sono le associazioni tra i sottosistemi), ignorando l'esistenza e l'identità di ogni altro filtro.

9. ATTIVITA' DEL SYSTEM DESIGN

9.1 Overview

Gli obiettivi guidano le decisioni che gli sviluppatori devono prendere, specialmente quando sono necessari dei compromessi. Abbiamo visto come gli sviluppatori dividono il sistema in sottoinsiemi per gestire la complessità, in modo che lo sviluppo di ogni sottosistema possa essere assegnato ad un team e realizzato in modo indipendente. Perché questo sia possibile, però, bisogna far fronte a determinate *scelte* che si presentano quando il sistema viene decomposto: Identificazione della Concorrenza, Mapping Hardware/Software, Gestione dei Dati Persistenti, Gestione Globale delle Risorse, Scelta del Controllo Software, Boundary Condition.
Ognuna di queste **Attività del System Design**, "corregge" la decomposizione del sistema, in modo da risolvere particolari problematiche. Una volta completate queste attività, possono essere definite le interfacce dei sottosistemi.

9.2 Identificare la Concorrenza

In questa attività bisogna identificare i processi concorrenti e gestire i problemi di concorrenza. Gli *obiettivi* di design sono i tempi di risposta e le performance.
Un **Thread di controllo** è un path attraverso un insieme di diagrammi di stato in cui, in ogni istante, un solo oggetto è attivo. Un thread rimane in un diagramma di stato fino a quando un oggetto invia un evento a un altro oggetto e attende un altro evento.
Quando un oggetto effettua un invio non bloccante di un evento, si utilizza un **Thread splitting**.
Due oggetti sono ***concorrenti*** se possono ricevere eventi nello stesso istante senza interagire. Gli oggetti concorrenti dovrebbero essere assegnati a thread di controllo differenti e gli oggetti con attività mutuamente esclusive possono essere raggruppati in un singolo thread di controllo.

Le domande tipiche a cui si devono cercare risposte per risolvere questioni di concorrenza sono del tipo: Quali oggetti dell'Object Model sono indipendenti? Quali tipi di thread di controllo sono identificabili? Il sistema fornisce accesso a utenti multipli? Può una singola richiesta del sistema essere decomposta in richieste multiple? Queste richieste possono essere gestite in parallelo?

I sistemi concorrenti possono essere implementati su ogni sistema che fornisce concorrenza *fisica* (hardware) o *logica* (software).

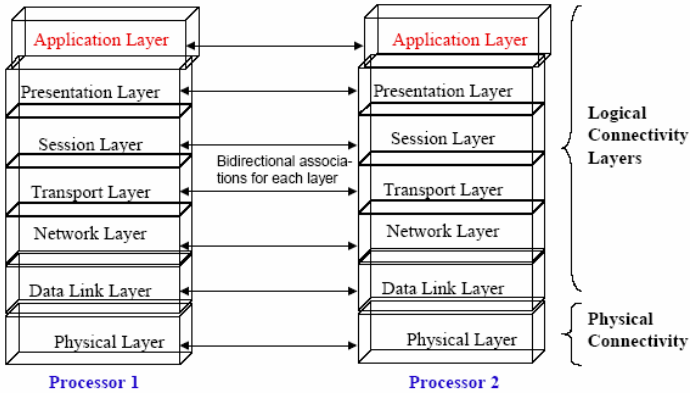
9.3 Mapping Hardware/Software

Questa attività introduce due domande: Come realizzare i sottosistemi: Hardware o Software? Come mappare il modello a oggetti sull'hardware e sul software scelti?
Ci si chiede, quindi, come mappare gli oggetti e le associazioni nella realtà: gli oggetti su processori, memoria, input/output, e le associazioni sulla connettività.
Molte delle difficoltà della progettazione di un sistema nascono dall'incontro tra l'hardware imposto dall'esterno e i vincoli software.

Quando si mappano gli oggetti, ci si trova davanti alle seguenti problematiche:

- Questioni relative ai processori: Il tasso di computazione è troppo alto per un singolo processore? Possiamo velocizzare l'elaborazione distribuendo i task tra più processori? Quanti processori sono richiesti per mantenere uno stato di carico stabile?
- Questioni relative alla memoria: C'è abbastanza memoria per gestire le richieste?
- Questioni relative all'I/O: C'è bisogno di componenti hardware extra per poter gestire il tasso di generazione dei dati? Il tempo di risposta eccede la larghezza di banda di comunicazione disponibile tra i sottosistemi o tra un task e un dispositivo hardware?

Quando si mappano le associazioni tra sottosistemi, invece, bisogna descrivere la connettività fisica dell'hardware: spesso il Livello Fisico del Modello di Riferimento OSI. Le tipiche problematiche da affrontare sono: Quali associazioni nel modello a oggetti sono mappate in connessioni fisiche? Quale delle relazioni client-supplier nel modello di analisi/design corrisponde alle connessioni fisiche?
Bisogna, inoltre, descrivere la connettività logica (associazioni tra sottosistemi): identificare associazioni che non vengono mappate direttamente in connessioni fisiche. Come potrebbero esse implementate queste associazioni?



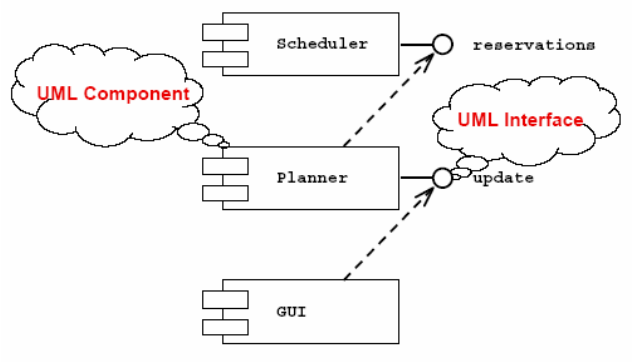
In generale, comunque, le domande tipiche a cui si devono cercare risposte per risolvere questioni di mapping Hardware/Software sono del tipo: Qual è la connessione tra le unità fisiche? E' ad Albero, a Stella, a Matrice, ad Anello? Qual è il protocollo di comunicazione appropriato tra i sottosistemi? Certe funzionalità sono già disponibili in hardware? Certi task richiedono locazioni specifiche per controllare l'hardware o per permettere operazioni concorrenti (spesso vero per sistemi *embedded*, cioè per sistemi in cui l'hardware e il software sono strettamente legati)? Questioni generali relative alle performance del sistema: Qual è il tempo di risposta desiderato?

Se, invece, l'**architettura è distribuita**, abbiamo bisogno di descrivere al meglio l'architettura di rete, e quindi il sottosistema di comunicazione. Le domande a cui bisogna rispondere sono del tipo: Qual è il mezzo trasmissivo? Ethernet? Wireless? Qual è il QoS (Quality of Service)? Che tipo di protocollo di comunicazione può essere usato? Le interazioni potrebbero essere asincrone, sincrone o bloccanti? Qual è la richiesta di larghezza di banda disponibile tra i sottosistemi?

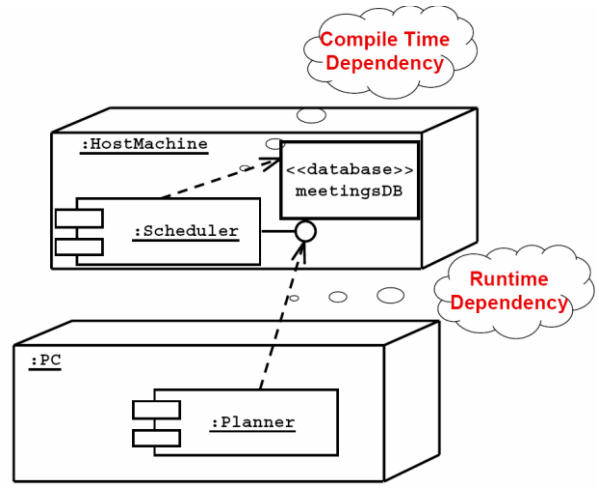
Progettare i Sottosistemi con UML
Durante il Sistem Design dobbiamo modellare la struttura statica e dinamica. A tal proposito, si utilizzano per le strutture statiche i *Component Diagram*, che mostrano la struttura al "design time" o al "compile time", e per le strutture dinamiche i *Deployment Diagram*, che mostrano la struttura al "run time". Da notare il tempo di vita dei componenti: alcuni esistono solo al "design time", altri fino al "compile time", alcuni solo al "link" o al "run time".

Un **Component Diagram** è un grafo di componenti connesse mediante relazioni di dipendenza, e mostra le dipendenze tra componenti software: codice sorgente, librerie linkabili, eseguibili. Le dipendenze sono mostrate con frecce tratteggiate dal componente client al componente fornitore (supplier), e i loro tipi sono specifici dei linguaggi di implementazione. Un Component Diagram può essere usato anche per mostrare le dipendenza su un *façade*: si usano le frecce tratteggiate verso le corrispondenti interfacce UML.

Esempio:



I **Deployment Diagram**, invece, sono utili per mostrare il progetto di un sistema dopo aver effettuato scelte in merito a: Decomposizione in Sottosistemi, Concorrenza, Mapping Hardware/Software. Si tratta di grafi di nodi connessi mediante associazioni di comunicazione. I nodi vengono mostrati come box 3D e possono contenere istanze di componenti, e le componenti possono contenere oggetti, per indicare che un oggetto è parte di una componente.



9.4 Gestione dei Dati Persistenti

Alcuni oggetti nei modelli necessitano di essere persistenti. Un oggetto persistente può essere realizzato mediante:

- *Strutture dati*, se i dati possono essere volatili.
- *File*: che sono economici, semplici, e forniscono memorizzazione permanente. Sono, tuttavia, strutture di basso livello (read, write) e può esserci necessità di aggiungere codice alle applicazioni, in modo da fornire opportuni livelli di astrazione.
- *Database*: potenti, facili da portare e supportano letture e scritture multiple.

Chiaramente scegliamo i file se i dati sono voluminosi, se abbiamo molti dati grezzi, se c'è necessità di mantenere i dati solo per un breve periodo di tempo e se la densità di informazione è bassa. Viceversa, scegliamo di usare un database se i dati richiedono accessi a livelli di dettaglio più raffinato, mediante utenti multipli, se i dati devono essere portati attraverso piattaforme multiple (sistemi eterogenei), se più programmi hanno accesso ai dati e se la gestione degli stessi richiede molte infrastrutture.
Un **Database Management System** (*DBMS*) contiene meccanismi per descrivere dati, gestire la memorizzazione persistente e per fornire meccanismi di backup. Fornisce accesso concorrente ai dati memorizzati e contiene informazioni sui dati (*meta-data* o *data schema*).

Quando si scegli un database bisogna considerare varie problematiche relative a:

- *Spazio di memorizzazione*: i database richiedono circa il triplo dello spazio occupato dai dati reali.
- *Tempo di risposta*: i database sono caratterizzati da limiti sull'I/O e sulla comunicazione (database distribuiti). Il tempo di risposta è anche influenzato dal tempo di CPU, dalle contese in lock e dai ritardi dovuti alle frequenti schermate di visualizzazione.
- *Modalità di Locking*: il lock può essere *pessimistico*, per cui si effettua il lock prima di accedere all'oggetto e si effettua il rilascio al termine dell'operazione, oppure *ottimistico*, per cui le letture e le scritture possono avvenire liberamente (alta concorrenza!). In quest'ultimo caso, quando l'attività è stata completata, il database controlla se è avvenuta una contesa. In tal caso, tutto il lavoro viene perso.
- *Amministrazione*: grossi database richiedono specifici staff di supporto per istituire politiche di sicurezza, gestire lo spazio su disco, prevedere backup, monitorare le performance, adeguare le messe a punto.

I **Database Object-Oriented** supportano tutti i concetti fondamentali della modellazione a oggetti: classi, attributi, metodi, associazioni, ereditarietà. Per mappare un modello a oggetti in un database orientato agli oggetti, bisogna: individuare gli oggetti persistenti, realizzare una normale analisi dei requisiti e un object design, creare singoli indici di attributi per ridurre i colli di bottiglia per le performance e realizzare il mapping (in modo specifico per i prodotti disponibili in commercio).

I **Database Relazionali**, invece, sono basati sull'algebra relazionale e i dati vengono presentati come tabelle 2-dimensionali, con un numero specifico di colonne e un arbitrario numero di righe. Si usa come *Chiave Primaria* una combinazione di attributi che identifica univocamente una riga nella tabella (ogni tabella dovrebbe avere solo una chiave primaria) e la *Chiave Esterna*, invece, è un riferimento alla chiave primaria di un'altra tabella. Il linguaggio standard per definire e manipolare questo tipo di tabelle è *SQL*.
I principali database commerciali, inoltre, supportano vincoli: *Integrità referenziale*, per esempio, significa che i riferimenti a entrate in altre tabelle esistono realmente.

In generale, le domande tipiche a cui si devono cercare risposte per risolvere questioni relative alla gestione dei dati sono del tipo: I dati dovrebbero essere distribuiti? Il database dovrebbe essere estendibile? Quanto spesso si accede al database? Qual è il tasso di richiesta atteso (per le query)? Nel caso peggiore? Qual è la dimensione di una tipica richiesta? Nel caso peggiore? I dati hanno bisogno di essere archiviati? Il system design tenta di nascondere la locazione dei database (trasparenza della locazione)? C'è necessità di una singola interfaccia per accedere ai dati? Qual è il formato delle query? Il database dovrebbe essere relazionale oppure object-oriented?

9.5 Gestione globale delle Risorse e della Sicurezza

In sistemi multi-utente, attori differenti hanno accesso a diverse funzionalità e dati. Durante l'analisi modelliamo questi accessi differenti associando differenti casi d'uso a differenti attori. Durante il system design modelliamo questi accessi differenti esaminando l'object model, per determinare quali oggetti sono condivisi tra gli attori. A seconda dei requisiti di sicurezza del sistema, possiamo anche definire il modo in cui gli attori vengono autenticati al sistema e come certi dati dovrebbero essere crittografati.
In altre parole, bisogna descrivere i diritti di accesso per differenti classi di attori (controllo degli accessi statico), bisogna descrivere come gli oggetti si difendono da accessi non autorizzati e bisogna descrivere le politiche di sicurezza (autenticazione degli attori e cifratura dei dati).

Modelliamo gli accessi alle classi con una **Matrice di Accesso**: le righe rappresentano gli attori del sistema e le colonne rappresentano le classi di cui vogliamo controllare l'accesso. Un **Diritto di Accesso** è un'entrata nella matrice degli accessi ed elenca le operazioni che possono essere effettuate dagli attori sulle istanze delle classi.
Si possono utilizzare varie implementazioni della Matrice degli Accessi:

- *Tabella globale degli accessi (Global Access Table)*: rappresenta esplicitamente ogni cella nella matrice come una tupla *(attore, classe, operazione)*. Determinare se un attore ha accesso a uno specifico oggetto richiede la verifica della tupla corrispondente. Se non c'è la tupla, l'accesso è negato.
- *Lista di controllo degli accessi (Access Control List)*: associa una lista di coppie *(attore, operazione)* a ogni classe che può essere acceduta. Ogni volta che un oggetto è acceduto, la sua lista degli accessi è controllata per il corrispondente attore e operazione. (Esempio: lista degli ospiti ad un party).
- *Capability*: associa una coppia *(classe, operazione)* a un attore. Consente ad un attore di ottenere l'accesso ad un oggetto della classe descritta nella capability stessa. (Esempio: un invito per un party).

In generale, le domande tipiche a cui si devono cercare risposte per risolvere questioni relative al controllo degli accessi sono del tipo: Il sistema ha bisogno di un meccanismo di autenticazione? Se si, qual è lo schema di autenticazione? Username e password? Lista di controllo degli accessi? Tickets? Basato su Capability?

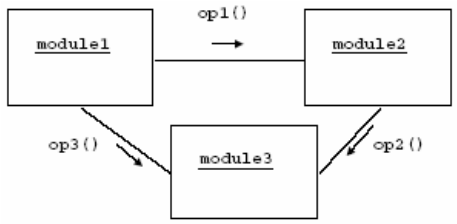
Inoltre: Qual è l'interfaccia utente per l'autenticazione? Il sistema ha bisogno di un name server di rete? Come diviene noto un servizio al resto del sistema? A runtime? A tempo di compilazione? Mediante la Porta? Mediante il Nome?

9.6 Stabilire il Controllo Software

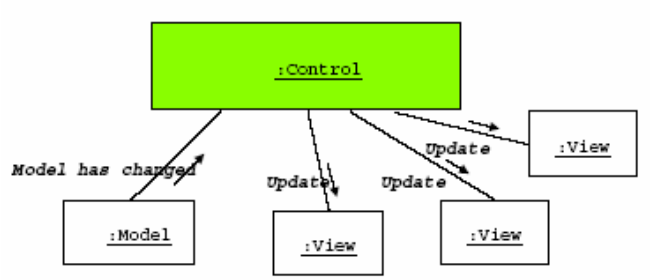
Si può decidere di scegliere un controllo del flusso implicito o esplicito:
- controllo implicito (linguaggi dichiarativi, non procedurali): sistemi basati su regole, programmazione logica.
- controllo esplicito (linguaggi procedurali): centralizzato o decentralizzato.

Il *Controllo Centralizzato*, può essere **Procedure-driven** o **Event-driven**. Nel primo caso il controllo risiede nel codice del programma (ad esempio il programma principale chiama le procedure dei sottosistemi). Si tratta di un controllo semplice e facile da costruire, ma difficile da

mantenere. Nel secondo caso, invece, il controllo risiede in un *dispatcher* che chiama le funzioni mediante *callback*. E' molto flessibile, buono per progettare interfacce utente grafiche ed è facile da estendere.



Esempio di Procedure-Driven Control



Esempio di Sistema Event-Based: MVC

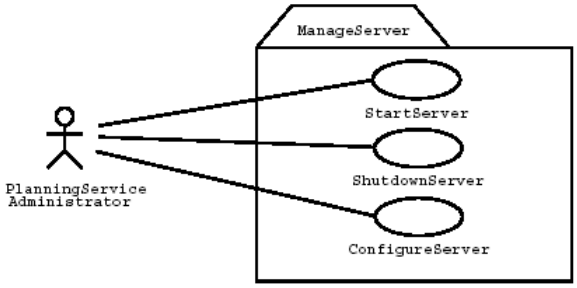
Quando si parla, invece, di *Controllo Decentralizzato*, il controllo risiede in vari oggetti indipendenti (supportato da alcuni linguaggi). E' possibile, in questo caso, velocizzare l'elaborazione mediante parallelizzazione, aumentando, però, l'overhead di comunicazione. (Esempio: Sistema Message-based).
A questo punto nasce spontanea la domanda: Si dovrebbe usare un design centralizzato o decentralizzato? In generale, si parte dal modello di analisi e si analizza il modo in cui i control object partecipano ai sequence diagram. Se il sequence diagram sembra avere una struttura fork, si utilizza un Design Centralizzato, viceversa, in caso di struttura stair, si utilizza un Design Decentralizzato.
Nel caso di Design Centralizzato, un control object o un sottosistema ("spider") controlla tutto. Il vantaggio è che i cambiamenti nella struttura di controllo sono facili da apportare, lo svantaggio è che il singolo control object può diventare un collo di bottiglia per le performance.
Nel caso di Design Decentralizzato, il controllo è distribuito tra più di un control object. Il vantaggio è che si adatta meglio allo sviluppo object-oriented, ma lo svantaggio è che le responsabilità sono distribuite.

9.7 Boundary Condition

La maggioranza degli sforzi del system design sono indirizzati al comportamento *steady-state* (stato stabile). Nella fase di system design bisogna anche determinare la configurazione, l'avvio e la terminazione del sistema che si sta sviluppando. In particolare:

- *Configurazione*: per ogni oggetto persistente, identificare in quale caso d'uso viene creato o distrutto (o archiviato). Per ogni oggetto che non viene creato o distrutto in qualche caso d'uso, aggiungere un caso d'uso invocato dall'amministratore del sistema.
- *Inizializzazione (start-up)*: descrive come il sistema o una componente è portato da uno stato non inizializzato a uno stato stabile (*startup use case*).
- *Terminazione*: descrive quali risorse sono rilasciate e quali sistemi vengono notificati al momento della terminazione del sistema o di una componente (*termination use case*).
- *Failure (gestione delle eccezioni)*: le cause che li provocano possono essere molte (bug, errori, problemi esterni, come l'alimentazione elettrica) e buoni system design devono prevedere *fatal failure* (*failure use case*). I casi d'uso eccezionali estendono i casi d'uso più rilevanti.

Ad esempio, può capitare durante il system design, di individuare un sottosistema addizionale, ad esempio un server. Per questo nuovo sottosistema c'è bisogno di definire dei casi d'uso, in maniera particolare quelli di gestione del server (start-up, shutdown, etc…):



Riassumendo, le Boundary Condition vengono meglio modellate mediante casi d'uso con attori e oggetti: l'attore, in questo caso, è spesso l'amministratore del sistema e i casi d'uso interessanti sono quelli che riguardano lo start-up di un sottosistema, lo start-up dell'intero sistema, la terminazione di un sottosistema, gli errori e le failure di un sottosistema o componente.

In generale, le domande tipiche a cui si devono cercare risposte per risolvere questioni relative alle Boundary Condition sono del tipo:

- Inizializzazione: Come si avvia il sistema? A quali dati bisogna accedere allo start-up? Quali servizi devono essere registrati? Cosa fanno le interfacce utente all'avvio del sistema? Come si presentano all'utente?
- Terminazione: I singoli sottosistemi possono terminare? Se un singolo sottosistema termina, gli altri vengono notificati? In che modo gli aggiornamenti locali vengono comunicati al database?
- Failure: Come si comporta il sistema quando un nodo o un link di comunicazione fallisce? Ci sono link di comunicazione di backup? In che modo il sistema recupera da una failure? E' differente dall'inizializzazione?

10.1 Overview

Nei *Sistemi Centralizzati*, le applicazioni vengono eseguite da un singolo processore o comunque su un singolo computer che costituisce la sola componente autonoma del sistema. Questa componente è condivisa tra differenti utenti e tutte le sue risorse sono sempre accessibili. La componente, quindi, costituisce il "singolo punto di controllo" e il "singolo punto di failure" dell'intero sistema. Questa è una situazione tipica per i sistemi Mainframe.
Oggi, lo scenario è diverso: virtualmente, tutti i sistemi basati su computer sono ***Sistemi Distribuiti***. Le informazioni da elaborare sono distribuite tra diversi computer piuttosto che confinate su una singola macchina. In questa situazione, quindi, l'**Ingegneria del Software Distribuito** diventa davvero importante.
I vantaggi riscontrabili in un Sistema Distribuito sono ovvi: condivisione di risorse, larghe vedute, concorrenza, scalabilità, tolleranza ai fault, trasparenza. D'altra parte si hanno degli svantaggi, dovuti a: complessità, sicurezza, gestibilità, imprevedibilità.
Le Architetture Distribuite che tratteremo sono:

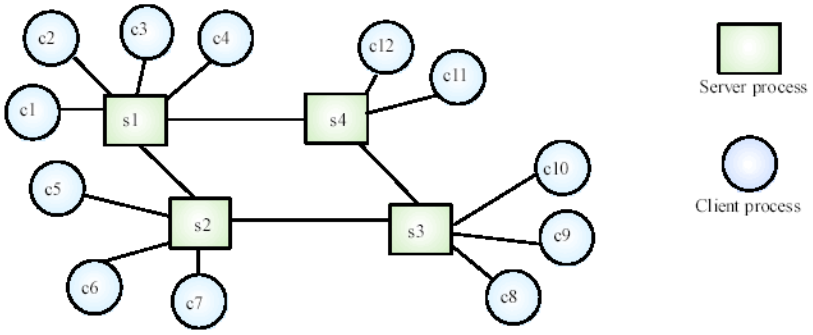
- *Architetture Client-Server*: i servizi distribuiti vengono chiamati dai client. I server, che forniscono i servizi, sono trattati in maniera differente dai client, che usano i servizi.
- *Architetture ad Oggetti Distribuiti*: non c'è distinzione tra client e server. Ogni oggetto nel sistema può fornire ed usare servizi di altri oggetti.

Un **Middleware** è un software che gestisce e supporta le differenti componenti di un sistema distribuito. In poche parole, risiede al centro (*middle*) del sistema. Di solito è "*off-the-shelf*" piuttosto che un software scritto specificatamente. Esempi: monitor di elaborazioni di transazioni, convertitori di dati, controller di comunicazione.

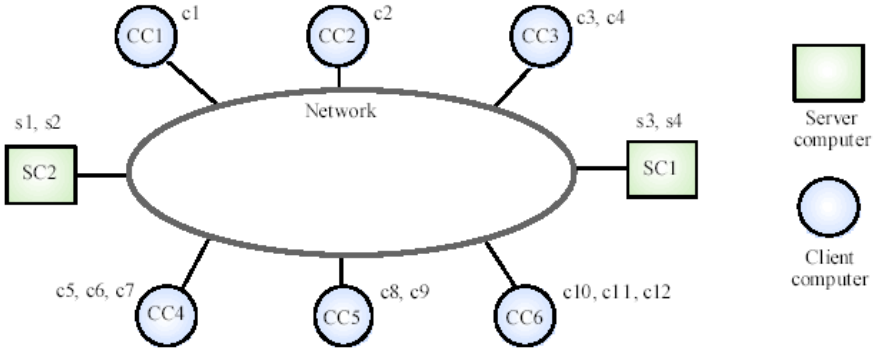
10.2 Architetture Client-Server

L'applicazione è modellata come un insieme di servizi forniti da server e un insieme di client che usano tali servizi. I client conoscono i server ma i server non hanno bisogno di conoscere i client Chiaramente, client e server sono processi logici e il mapping dei processori ai processi non deve essere necessariamente 1:1.

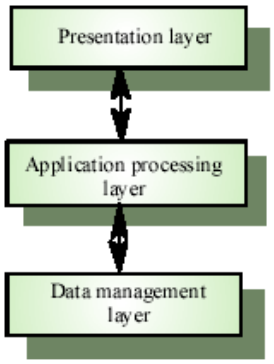
Esempio di Sistema Client-Server:



Esempi di computer in una rete C/S:



Architettura ad Applicazione Stratificata



Livello Presentazione: si occupa di presentare i risultati di una computazione agli utenti del sistema e di raccogliere gli input degli utenti.

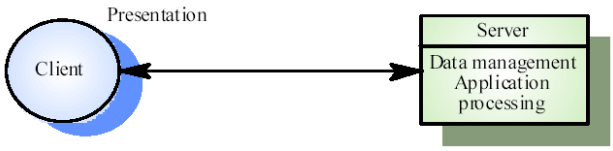
Livello Elaborazione Applicazione: si occupa di fornire all'applicazione specifiche funzionalità, ad esempio, in un sistema bancario le funzionalità sono quelle di apertura di un conto, di chiusura di un conto, etc…

Livello Gestione Dati: si occupa della gestione dei database di sistema.

Application layers e Sistemi Client-Server

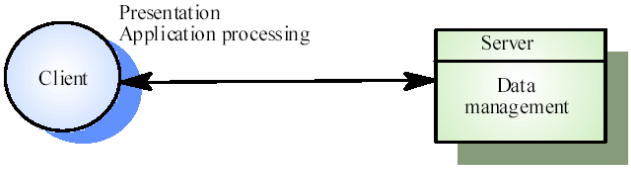
Modello Thin-Client

Usato quando sistemi esistenti migrano all'architettura client-server. Il sistema esistente agisce da server e sui client è implementata solo l'interfaccia grafica. In altre parole, tutta l'elaborazione delle applicazioni e la gestione dei dati sono eseguite dal server. Il client è responsabile semplicemente di eseguire la presentazione del software. Un grosso svantaggio è che viene introdotto un pesante carico di elaborazione sia sul server che sulla rete.

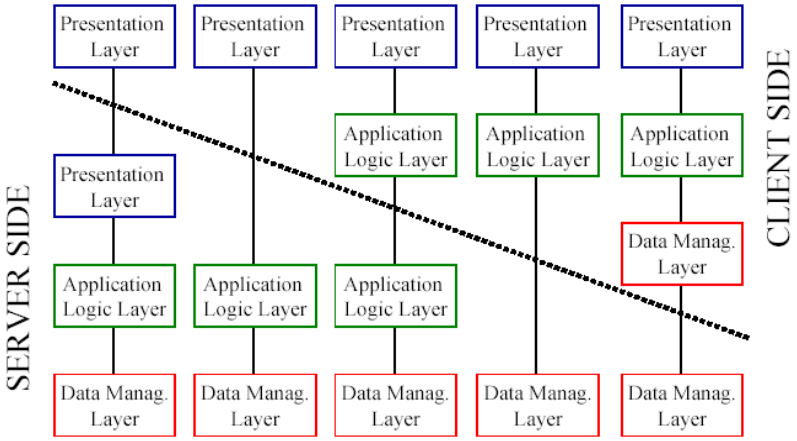


Modello Fat-Client

Il server è responsabile solo della gestione dei dati, mentre il software del client implementa la logica applicativa e le interazioni con gli utenti del sistema. Quindi, la maggioranza dell'elaborazione è delegata ai client visto che l'elaborazione dell'applicazione è eseguita localmente. E' un modello maggiormente adatto per i nuovi sistemi C/S dove le capacità dei sistemi client sono in noto aumento. E' più complesso del modello Thin-Client, specialmente per quanto riguarda la gestione, ad esempio, le nuove versioni dell'applicazione devono essere installate su tutti i client.

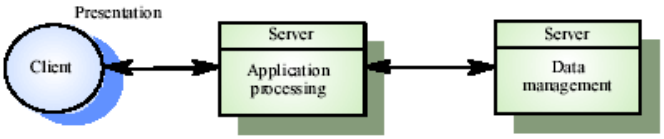


Stili Client-Server

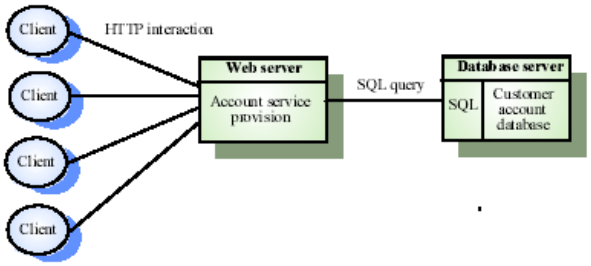


Architetture Three-tier (tre strati)

In un'architettura di questo tipo, ogni livello applicativo dell'architettura può essere eseguito su un processore separato. Permette di avere migliori performance dell'approccio thin-client ed è più semplice dell'approccio fat-client. Se è richiesta un'architettura più scalabile, si possono aggiungere server extra.

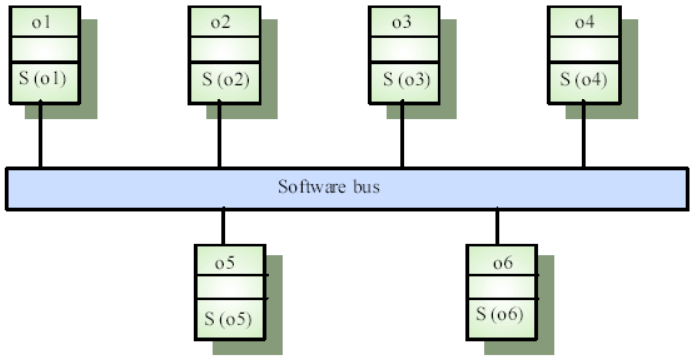


Esempio: sistema bancario on-line



10.3 Architetture ad Oggetti Distribuiti

Non c'è distinzione in un'architettura di questo tipo tra client e server. Ogni entità distribuibile è un oggetto che fornisce servizi ad altri oggetti e riceve servizi da altri oggetti. La comunicazione tra gli oggetti avviene attraverso un sistema middleware chiamato *Object Request Broker* (**Software Bus**).



I vantaggi che si hanno con l'utilizzo di questa architettura sono:

- I progettisti del sistema hanno la possibilità di ritardare decisioni su dove e come i servizi dovrebbero essere forniti.
- Si ha a che fare con un'architettura "molto aperta", che permette di aggiungere ad essa nuove risorse quando richiesto.
- Il sistema che si costruisce è flessibile e scalabile.
- E' possibile, quando richiesto, riconfigurare il sistema dinamicamente, per gli oggetti che si sono spostati lungo la rete.

Un'architettura del genere può essere utilizzata come un *modello logico*, che permette di strutturare e organizzare il sistema. In questo caso, si deve pensare a come fornire le funzionalità applicative solamente in termini di servizi e combinazioni di questi.
In alternativa, si può usare questa architettura come un *flessibile approccio all'implementazione di sistemi client-server*. Il modello logico del sistema è un modello client-server, ma sia i client che i server vengono realizzati come oggetti distribuiti che comunicano fra loro, attraverso un software bus.

11. SYSTEM DESIGN (REVIEW E GESTIONE)

11.1 Revisione del System Design

Come l'Analisi, il System Design è un'attività evolutiva ed iterativa. A differenza dell'analisi, però, non ci sono agenti esterni, come il cliente, a revisionare le successive iterazioni e ad assicurare la migliore qualità. In ogni caso, molte attività nel system design introducono revisioni al modello di analisi, quindi i clienti e gli utenti vengono comunque portati dietro nel processo, a causa di tali revisioni.

Questa attività di incremento della qualità è ancora necessaria, e i project manager e gli sviluppatori devono riorganizzare il processo di revisione in modo da sostituire il cliente. I revisori, in questo caso, possono essere sviluppatori non coinvolti nel system design oppure sviluppatori di altri progetti. In aggiunta, per raggiungere gli obiettivi di design identificati durante il system design, bisogna assicurare che il modello di design sia *corretto*, *completo*, *consistente*, *realistico* e *leggibile*. In particolare:

- Il modello del system design è **corretto** se il modello di analisi può essere mappato su di esso. A tal proposito, ci si pone le seguenti domande: Ogni sottosistema può essere tracciato su un caso d'uso o su un requisito non funzionale? Ogni caso d'uso può essere mappato su un insieme di sottosistemi? Ogni obiettivo di design può essere tracciato su un requisito non funzionale? Ogni requisito non funzionale è trattato nel system design model? Ogni attore ha una politica di accesso? Ogni politica di accesso è consistente con il requisito non funzionale di sicurezza?
- Il modello è **completo** se ogni requisito (e ogni questione di design) è stato perseguito. A tal proposito, ci si pone le seguenti domande: Sono state gestite le boundary condition? C'è stata una rivisitazione dei casi d'uso per identificare le funzionalità mancanti nel system design? Tutti i casi d'uso sono stati esaminati ed assegnati ad un control object? Sono stati analizzati tutti gli aspetti del system design (ad esempio, allocazione hardware, memorizzazione persistente, controllo degli accessi, codice esistente, boundary condition)? Tutti i sottosistemi hanno una definizione?
- Il system design model è **consistente** se non contiene contraddizioni. Ci si pone, a tal proposito, le seguenti domande: E' stata assegnata una priorità ai design goal in conflitto? Qualche design goal viola un requisito non funzionale? Ci sono più sottosistemi o classi con lo stesso nome? Le collezioni di oggetti vengono scambiate in maniera consistente tra i sottosistemi?
- Il modello è **realistico** se il sistema corrispondente può essere implementato. Le domande che ci si pone sono: E' inclusa qualche nuova tecnologia o componente nel sistema? E' stata valutata l'appropriatezza o la robustezza di queste tecnologie o componenti? Come? Sono stati revisionati i requisiti di performance e di affidabilità nel contesto della decomposizione in sottosistemi? Sono state trattate le questioni relative alla concorrenza (ad esempio, contese e deadlock)?
- Il modello è **leggibile** se sviluppatori non coinvolti nel system design possono comprenderlo. Le domande che ci si pone sono: I nomi dei sottosistemi sono comprensibili? Entità (come sottosistemi, classi) con nomi simili, rappresentano concetti simili? Tutte le entità sono descritte allo stesso livello di dettaglio?

11.2 Gestione del System Design

Documentare il System Design

Il System Design è documentato dal **System Design Document** (**SDD**), che descrive: l'insieme dei Design Goal del progetto, la Decomposizione in sottosistemi (con UML class diagram), il Mapping Hardware/Software (con UML deployment diagram), la Gestione dei dati, i Meccanismi di controllo del flusso, le Boundary Condition.
Questo documento viene usato per definire le interfacce tra i team di sviluppo e funge da riferimento quando c'è bisogno di rivisitare le decisioni prese a livello architetturale.
Gli utilizzatori dell'SDD includono il project manager, gli architetti di sistema (come sviluppatori partecipanti al system design), gli sviluppatori che progettano e implementano ogni sottosistema.
La struttura del System Design Document è la seguente:

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Design goa1s
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Current software architecture
3. Proposed software architecture
   - 3.1 Overview
   - 3.2 Subsystem decomposition
   - 3.3 Hardware/software mapping
   - 3.4 Persistent data management
   - 3.5 Access control and security
   - 3.6 Global software control
   - 3.7 Boundary conditions
4. Subsystem services

La *Sezione 1* (Introduzione) contiene una breve panoramica dell'architettura software e degli obiettivi di design. Le *sezioni 1.1*, *1.2*, *1.3* e *1.5* sono simili a quelle presenti nel RAD.
La *Sezione 1.4* (Riferimenti), contiene riferimenti ad altri documenti e informazioni sulla tracciabilità (ad esempio, requisiti nel documento di analisi correlati, riferimenti a sistemi esistenti, vincoli che impattano sull'architettura software).
La *Sezione 2* (Architettura software corrente) contiene l'architettura del sistema che stiamo per sostituire. Se non esiste un sistema precedente, questa sezione può essere rimpiazzata da un'analisi dell'architettura di sistemi esistenti, simili a quello che si sta progettando. Lo scopo di questa sezione è quello di esplicitare le informazioni precedenti che gli architetti di sistema hanno usato, le loro assunzioni, le problematiche in comune con il nuovo sistema, che dovranno essere affrontate.
La *Sezione 3* (Architettura software proposta): documenta il system design model del nuovo sistema. La *Sezione 3.1* (Overview), presenta una panoramica dell'architettura software e descrive brevemente l'assegnamento delle funzionalità a ogni sottosistema. La *Sezione 3.2* (Decomposizione in sottosistemi), descrive la decomposizione in sottosistemi e le responsabilità di ognuno di essi. E' questo il principale prodotto del system design.
La *Sezione 3.3* (Mapping Hardware/Software) descrive come i sottosistemi vengono assegnati all'hardware e alle componenti *"off-the-shelf"*. Elenca anche le problematiche introdotte da nodi multipli e dal riuso del software.

La *Sezione 3.4* (Gestione dei dati persistenti) descrive i dati persistenti memorizzati dal sistema e l'infrastruttura di gestione richiesta per essi. Questa sezione tipicamente include la descrizione dei *data-schemes*, la scelta di un database e la descrizione dell'incapsulamento del database.
La *Sezione 3.5* (Controllo degli accessi e sicurezza) descrive il modello utente del sistema in termini di una *matrice degli accessi*. Questa sezione descrive anche le problematiche di sicurezza, come la scelta di un meccanismo di autenticazione, l'uso di crittografia e la gestione delle chiavi.
La *Sezione 3.6* (Controllo software globale) descrive come viene implementato il controllo software globale. In particolare, questa sezione dovrebbe descrivere come vengono iniziate le richieste e come vengono sincronizzati i sottosistemi. Dovrebbe elencare e trattare le problematiche di sincronizzazione e concorrenza.
La *Sezione 3.7* (Boundary condition) descrive lo start-up, lo shutdown e i comportamenti errati del sistema. Se vengono scoperti nuovi casi d'uso per l'amministratore del sistema, questi dovrebbero essere inclusi nel documento di analisi dei requisiti, non in questa sezione.
La *Sezione 4* (Servizi dei sottosistemi) descrive i servizi forniti da ogni sottosistema, in termini di operazioni. Anche se questa sezione è di solito vuota o incompleta nelle prime versioni dell'SDD, e serve come riferimento per i team a riguardo dei limiti tra i loro sottosistemi. L'interfaccia di ogni sottosistema è derivata da questa sezione e dettagliata nell'*Object Design Document*.

Assegnare le responsabilità

Il system design in sistemi complessi è incentrato intorno all'**architecture team**, che è un *cross-functional team* costituito da architetti che definiscono la decomposizione in sottosistemi e da sviluppatori selezionati che implementeranno i sottosistemi. A questo punto è chiaro che il system design include persone esposte alle conseguenze delle decisioni prese per il design stesso.
L'architecture team inizia a lavorare appena il modello di analisi diventa stabile e continua a farlo fino alla fine della fase di integrazione. Ciò incentiva questo team ad anticipare problemi che verrebbero incontrati durante l'integrazione.
Il numero dei sottosistemi determina la dimensione dell'architecture team. Per sistemi complessi, ne viene introdotto uno per ogni livello di astrazione. In tutti i casi, dovrebbe esserci un ruolo integrativo nel team che assicuri la consistenza e la comprensibilità dell'architettura per ogni singolo individuo.
I ruoli principali, che fanno parte del team sono:

- L'**Architetto**, che riveste il ruolo principale nel system design. Assicura la consistenza nelle decisioni di design e nello stile delle interfacce, assicura la consistenza del design tra i team per la gestione della configurazione e del testing, in particolare nella formulazione delle politiche di gestione della configurazione e delle strategie per l'integrazione del sistema. E' il leader del cross-functional architecture team.
- Gli **Architecture Liason** sono i membri dell'architecture team e sono rappresentativi dei team che lavorano ai diversi sottosistemi. Raccolgono le informazioni da e verso i loro team e negoziano i cambiamenti nelle interfacce. Durante il system design, si concentrano sui servizi dei sottosistemi e durante la fase di implementazione si concentrano sulla consistenza delle API.
- I ruoli di **Document Editor**, **Configuration Manager** e **Reviewer** sono gli stessi dell'analisi.

Comunicazione nel System Design

La comunicazione durante il system design dovrebbe essere meno difficoltosa che durante l'analisi: le funzionalità del sistema sono state definite, i partecipanti al progetto hanno avuto esperienze simili e da ora dovrebbero conoscere meglio ogni altro collega. Tuttavia, la comunicazione è ancora difficile, a causa di nuove sorgenti di complessità:

- *Dimensioni*: il numero di problemi da affrontare aumenta non appena gli sviluppatori avviano il design, così come aumenta il numero di elementi da manipolare (ogni pezzo di funzionalità

richiede molte operazioni su molti oggetti). Inoltre, gli sviluppatori esaminano, spesso contemporaneamente, più design e più tecnologie di implementazione.

- *Cambiamenti*: la decomposizione in sottosistemi e le interfacce dei sottosistemi sono in costante cambiamento e i termini usati dagli sviluppatori per nominare parti differenti del sistema evolvono costantemente. Se i cambiamenti sono rapidi, gli sviluppatori non possono discutere a riguardo di una stessa versione dei sottosistemi, il che provoca molta confusione.
- *Livello di astrazione*: le discussioni a riguardo dei requisiti possono diventare concrete grazie all'utilizzo di interfacce mok-up e analogie con sistemi esistenti. Quelle a riguardo dell'implementazione diventano concrete quando sono disponibili i risultati dell'integrazione e dei test. Le discussioni nel system design, però, raramente sono concrete, visto che i risultati del design sono tangibili solo più tardi, durante l'implementazione e il testing.
- *Riluttanza a confrontare problemi*: il livello di astrazione per la maggioranza delle discussioni può anche essere semplificato, per ritardare la risoluzione di problemi complessi. Una tipica risoluzione per i problemi di controllo è spesso: "rivisitiamo il problema durante l'implementazione". Laddove è usualmente desiderabile ritardare certe decisioni di design, come la scelta delle strutture dati interne e degli algoritmi usati da ogni sottosistema, alcune decisioni che hanno impatto sulla decomposizione in sottosistemi e sulle interfacce dei sottosistemi non dovrebbero essere ritardate.
- *Goal e Criteri in conflitto*: singoli sviluppatori spesso ottimizzano differenti criteri. Uno sviluppatore esperto in interfacce utente sarà propenso all'ottimizzazione del tempo di risposta e uno sviluppatore esperto in database potrebbe ottimizzare il throughput. Questi obiettivi in conflitto, specialmente se impliciti, portano gli sviluppatori a spingere la decomposizione del sistema in direzioni differenti, ottenendo inconsistenze.

Per rendere effettiva ed efficace la comunicazione, quindi bisogna:

- *Identificare e assegnare delle priorità agli obiettivi di design per il sistema e renderli espliciti*: se gli sviluppatori interessati al system design hanno accesso a questo processo, avranno più tempo assegnato per questi design goal. I design goal fungono anche da strumento oggettivo con cui si possono valutare le decisioni.
- *Rendere disponibile la corrente versione della decomposizione del sistema a tutti gli interessati*: un documento distribuito via Internet è sicuramente una rapida maniera di distribuzione. L'uso di strumenti di gestione della configurazione per mantenere il system design document, è di aiuto agli sviluppatori nell'identificazione dei cambiamenti recenti.
- *Mantenere un glossario up-to-date*: come nell'analisi, definire esplicitamente i termini riduce le incomprensioni. Quando si identificano e modellano i sottosistemi, è bene fornire definizioni in aggiunta ai nomi: un diagramma UML con solo i nomi dei sottosistemi non è sufficiente a supportare l'effettiva comunicazione. Quindi, una breve e sostanziale definizione dovrebbe accompagnare il nome di ogni sottosistema e di ogni classe.
- *Confrontare i problemi di design*: ritardare le decisioni di design può essere un bene nel caso in cui sono richieste molte informazioni prima di effettuare la decisione. Questo approccio, comunque, può prevenire il confronto tra difficili problemi di design. Prima di intavolare una problematica, dovrebbero essere esaminate e descritte diverse alternative possibili, per giustificare il ritardo. Questo assicura che la questione può essere ritardata senza impattare sulla decomposizione del sistema.
- *Iterare*: dissertazioni scelte nella fase di implementazione possono migliorare il system design. Ad esempio, nuove caratteristiche in una componente distributore-fornitore possono essere valutate implementando un prototipo verticale per le funzionalità maggiormente utili alla caratteristica.

A questo punto è lecito chiedersi: Quanto sforzo si può spendere per il system design?
In realtà, è bene non preoccuparsene, infatti la decomposizione e le interfacce dei sottosistemi quasi certamente cambieranno durante l'implementazione. Appena nuove informazioni sulle tecnologie di implementazione saranno disponibili, gli sviluppatori avranno una chiara comprensione del sistema, e saranno scoperte alternative di design. A tal proposito, gli sviluppatori dovrebbero anticipare i cambiamenti e riservare un po' di tempo all'update dell'SDD prima dell'integrazione di sistema.

Iterazione nel system design

Come nel caso dei requisiti, il system design nasce attraverso successive iterazioni e cambiamenti. I cambiamenti, comunque, dovrebbero essere controllati per prevenire il caos, specialmente in progetti complessi che includono molti partecipanti. Distinguiamo tra tre tipi di iterazioni durante il system design:

- Decisioni importanti all'inizio del system design incidono sulla decomposizione in sottosistemi appena ognuna delle differenti attività del system design ha inizio.
- Le revisioni delle interfacce dei sottosistemi avvengono quando i prototipi di valutazione sono creati per valutare specifici problemi.
- Errori e sviste che sono scoperti tardi, introducono cambiamenti alle interfacce dei sottosistemi e, a volte, alla stessa decomposizione del sistema.

In particolare:

1. Iterazione delle decisioni importanti

Sono meglio gestite nelle sessioni di brainstorming (sia faccia a faccia che elettroniche).
In questo scenario, le definizioni sono ancora in evoluzione, gli sviluppatori non hanno ancora padronanza dell'intero sistema e la comunicazione dovrebbe essere massimizzata a costo della formalità o delle procedure. Inoltre la decomposizione iniziale del sistema permette di assegnare le responsabilità dei differenti sottosistemi a differenti team.
Cambiamenti ed indagini dovrebbero essere incoraggiati, anche solo per allargare le conoscenze condivise dagli sviluppatori o per dare luogo a dimostrazioni di supporto al corrente design.
Per queste ragioni, un processo di cambiamento formale e burocratico non dovrebbe essere utilizzato durante questa fase.

2. Iterazione di revisione alle interfacce dei sottosistemi

Mirano a risolvere problemi difficili e centrali, come le scelte di uno specifico distributore o tecnologia. La decomposizione del sistema è stabile (idealmente, dovrebbe essere indipendente dal distributore e dalla tecnologia), e molte di queste indagini mirano ad identificare se uno specifico package è appropriato per il sistema o meno. Durante questo periodo, gli sviluppatori possono anche creare un prototipo verticale per un caso d'uso critico, per testare l'appropriatezza della decomposizione. Questo permette di individuare e trattare in un momento antecedente le problematiche riguardanti il flusso di controllo.
Di nuovo, un processo di cambiamento formale non è necessario. Invece, un elenco di problemi in attesa di decisione e dei loro stati può aiutare gli sviluppatori a propagare velocemente il risultato di un'analisi sulla tecnologia.

3. Iterazione per gli errori e le sviste

Rimediano a problemi di design individuati tardi nel processo. Sebbene gli sviluppatori dovrebbero piuttosto evitare queste iterazioni, perché tendono ad essere costose e ad introdurre molti nuovi bug nel sistema, potrebbero anticipare cambiamenti successivi durante sviluppo.

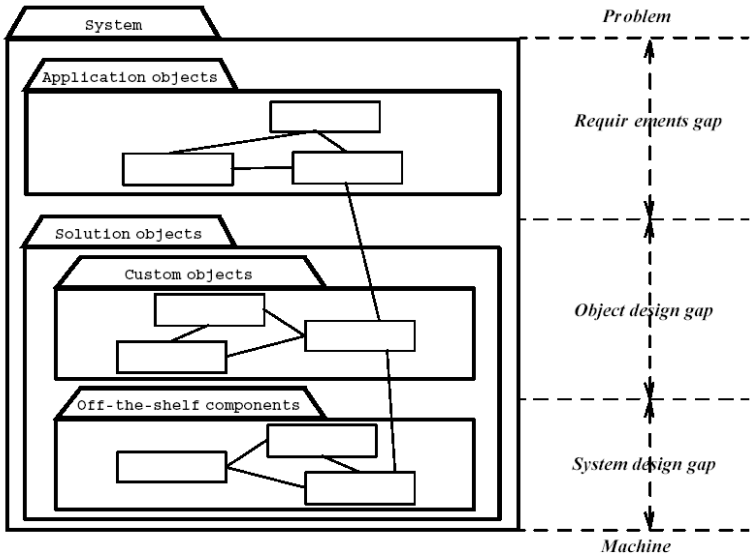
Anticipare le iterazioni successive include documentare le dipendenze tra i sottosistemi, il design ragionevole delle interfacce dei sottosistemi, ed alcuni lavoretti che molto probabilmente falliranno in caso di cambiamenti.
I cambiamenti dovrebbero essere gestiti attentamente, ed un processo di cambiamento simile ad un cambiamento nel tracciabile requisito dovrebbe essere messo a posto.

Design Window

Possiamo raggiungere la progressiva stabilizzazione della decomposizione in sottosistemi usando il concetto di *design windows*.
In altre parole, per incoraggiare i cambiamenti, ma in maniera controllata, problemi critici vengono lasciati aperti solo per un periodo di tempo specifico. Ad esempio, la piattaforma hardware/software su cui il sistema dovrà girare dovrebbe essere scelta inizialmente nel progetto, in modo che le decisioni riguardanti l'acquisto dell'hardware possano essere prese in tempo per lo sviluppo.
Le questioni in merito a strutture dati interne e algoritmi, comunque, possono essere lasciate aperte fino a dopo l'integrazione, in modo da permettere agli sviluppatori di revisionarle in base a test di performance.
Una volta che la finestra di design è stata chiusa, i problemi devono essere risolti e possono essere riaperti solo per un'iterazione successiva.
Visto lo stimolante passo dell'innovazione tecnologica, molti cambiamenti possono essere anticipati se c'è una parte dedicata dell'organizzazione come responsabile della gestione tecnologica. In tal caso, i manager tecnologici esaminano le nuove tecnologie, le valutano, e accumulano conoscenze che vengono usate durante la scelta delle componenti.
Si pensi che, spesso, i cambiamenti avvengono così velocemente che le compagnie non sono nemmeno consapevoli delle tecnologie che forniscono.

12.1 Overview

L'**Object Design** è il processo che si occupa di aggiungere dettagli all'analisi dei requisiti e di prendere decisioni di implementazione. L'object designer, quindi, deve scegliere tra diversi modi di implementare il modello di analisi, con l'obiettivo di minimizzare il tempo di esecuzione, la memoria e altre misure di costo.
Nell'analisi dei requisiti, i casi d'uso, il modello funzionale e quello dinamico definiscono le operazioni per il modello a oggetti. Nell'object design iteriamo nel processo di assegnazione di queste operazioni al modello a oggetti. In altre parole, l'object design serve come base per l'implementazione e durante il suo sviluppo chiude il *gap* tra gli oggetti di applicazione e le componenti off-the-shelf, identificando oggetti soluzione e raffinando gli oggetti esistenti.

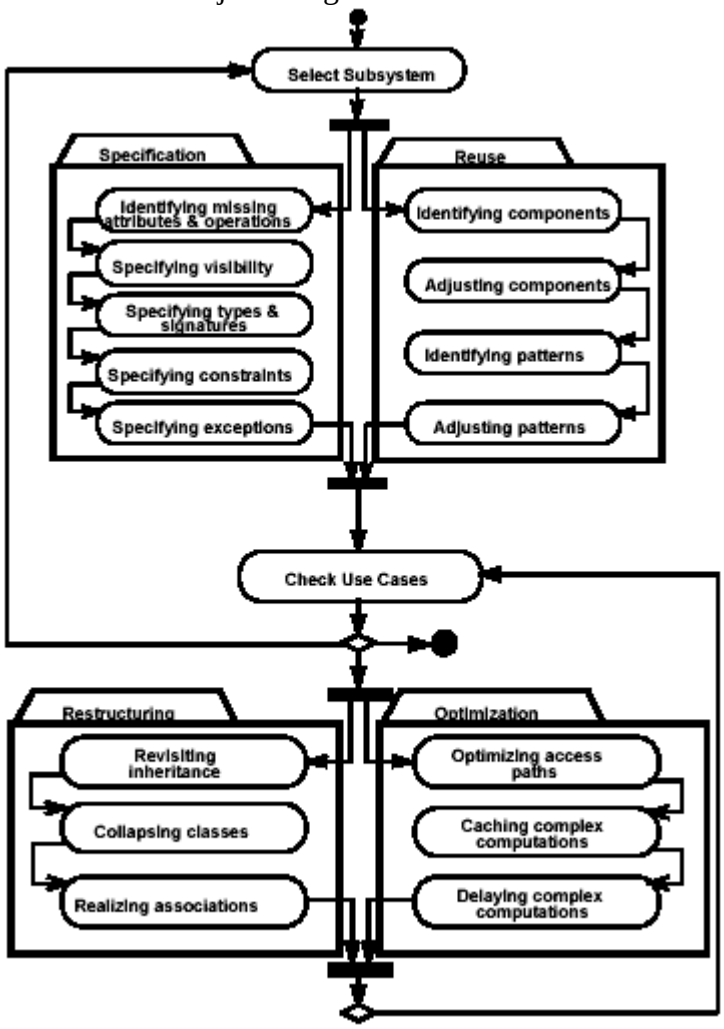


Esempi di attività di object design sono: identificazione delle componenti esistenti, definizione completa delle relazioni, definizione completa delle classi (System Design => Servizi, Object Design => API), specifica del contratto per ogni componente, scelta degli algoritmi e delle strutture dati, identificazione delle possibilità di riuso, identificazione delle classi del dominio delle soluzioni, ottimizzazione, incremento dell'ereditarietà, decisioni sul controllo, packaging.

Un po' di terminologia relativa a queste attività:

- Metodologie object-oriented
  System Design: decomposizione in sottosistemi.
  Object Design: scelta del linguaggio di implementazione, delle strutture dai e degli algoritmi.
- SA/SD usa una terminologia differente
  Design Preliminare: decomposizione in sottosistemi e scelta delle strutture dati.
  Design Dettagliato: si scelgono degli algoritmi, le strutture dati vengono raffinate, viene scelto il linguaggio di implementazione, il tutto tipicamente in parallelo al design preliminare, non in fase separata.

Vista più dettagliata delle attività di object design:



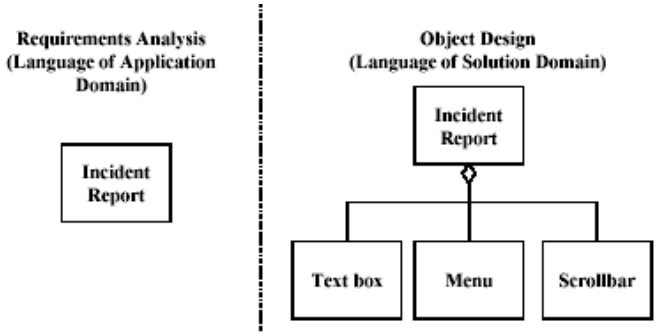
Quindi, le principali attività dell'Object Design sono: *Riuso*, *Specifica dei Servizi*, *Ristrutturazione* del modello a oggetti e *Ottimizzazione* del modello a oggetti. Queste attività di solito non sono sequenziali, ma realizzate in maniera concorrente. Ci potrebbero però essere delle dipendenze, pertanto, di solito vengono realizzare prima le attività di riuso e di specifica delle interfacce, ottenendo un modello ad oggetti di design, che viene verificato rispetto ai corrispondenti casi d'uso. Una volta che il modello si è stabilizzato, vengono svolte le attività di ristrutturazione e ottimizzazione.

12.2 Application Object e Solution Object

Le attività più difficili nello sviluppo di un sistema sono l'identificazione degli oggetti e la decomposizione del sistema in oggetti. L'analisi dei requisiti è concentrata sul dominio applicativo, per ottenere l'identificazione degli oggetti, il system design, invece, è indirizzato sia al dominio applicativo che implementativo, per ottenere l'identificazione dei sottosistemi. Infine, l'object design focalizza sul dominio implementativo: identificazione di più oggetti.
Durante l'analisi dei requisiti, le tecniche utilizzate per trovare gli oggetti si basano sui seguenti passi: si parte con i casi d'uso e si identificano gli oggetti partecipanti, si effettua un'analisi testuale del flusso di eventi (alla ricerca di nomi, verbi, etc…), si estraggono gli oggetti del dominio applicativo, intervistando i clienti, e si trovano gli oggetti usando conoscenze generali.

Durante il system design, le tecniche per trovare gli oggetti si basano sulla decomposizione in sottosistemi e sul tentativo di identificare gli strati e le partizioni.
Durante l'object design, invece, le tecniche per trovare gli oggetti sono incentrate sulla ricerca degli oggetti addizionali, applicando conoscenze del dominio implementativo.

Gli **application object**, anche detti *domain object*, rappresentano concetti del dominio che sono rilevanti per il sistema. Vengono identificati da specialisti del dominio applicativo e dagli utenti finali.
I **solution object**, invece, rappresentano concetti che non hanno una controparte nel dominio applicativo. Questi vengono identificati dagli sviluppatori. Ad esempio, memorie di dati persistenti, oggetti dell'interfaccia utente, middleware.



Ricapitolando, durante l'analisi identifichiamo gli entity object e le loro relazioni, attributi e operazioni. Molti di questi oggetti sono application object indipendenti da uno specifico sistema.
Identifichiamo, inoltre, anche solution object visibili all'utente, come i boundary e i control object.
Durante il system design, invece, identifichiamo altri solution object in termini di piattaforme hardware e software.
Durante l'object design, raffiniamo e dettagliamo tutti questi application e solution object, e identifichiamo ulteriori solution object per il chiudere il gap di object design.

12.3 Design Pattern: overview

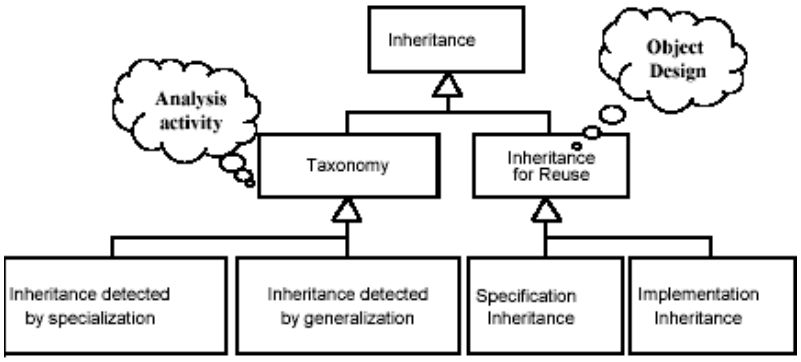
Un'ulteriore sorgente per trovare gli oggetti sono i **Design Pattern**. Nascono dall'osservazione (*Gamma et al '95*): "la modellazione rigida del mondo reale conduce a sistemi che riflettono l'odierna realtà, ma non necessariamente quella futura".
C'è bisogno, pertanto, di progettazioni riusabili e flessibili. Le conoscenze di design completano quelle sul dominio applicativo e sul dominio implementativo. Ma cosa sono i Design Pattern?
Un **Design Pattern** descrive un problema ricorrente nel nostro ambiente, e il cuore della soluzione a tale problema, in modo tale da poter usare questa soluzione ancora un milione di volte, senza mai doverla ricercare.
In pratica si tratta di template di soluzioni che gli sviluppatori hanno raffinato nel tempo per risolvere un insieme di problemi ricorrenti.
Un design pattern ha quattro elementi: un *nome* che lo identifica, una *descrizione del problema* che descrive le situazioni in cui il pattern può essere usato, una *soluzione* presentata come un insieme di classi e interfacce, un insieme di *conseguenze* che descrivono i trade-off e le alternative che devono essere considerate rispetto ai design goal fissati.

12.4 Ereditarietà e Delegazione

Prima di proseguire nella trattazione dei pattern, è bene rivedere alcuni termini.

L'**ereditarietà** è usata per raggiungere due differenti obiettivi: descrivere le tassonomie e specificare le interfacce. L'identificazione delle tassonomie è utilizzata durante l'analisi dei requisiti, in particolare sottoforma di identificazione degli oggetti del dominio applicativo che sono gerarchicamente correlati. In quel caso, l'obiettivo è rendere più comprensibile il modello di analisi. La specifica dei servizi è utilizzata durante l'object design, con l'obiettivo di aumentare la riusabilità e migliorare la modificabilità e l'estendibiltà.

Meta-modello per l'ereditarietà:



Durante l'analisi, l'ereditarietà è ottenuta sia con la *specializzazione* che con la *generalizzazione*.
Nel caso dell'object design, e in particolare nel caso in si vuole aumentare il **riuso**, l'obiettivo principale diventa: riusare le conoscenze assunte in esperienze precedenti per il problema corrente e riusare funzionalità già disponibili. Ciò lo si può ottenere mediante:

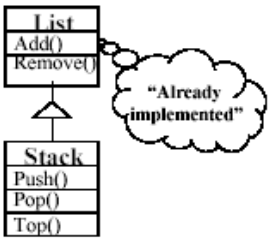
- *Composizione* (anche detta *Black Box Reuse*): la nuova funzionalità è ottenuta mediante l'aggrezione. Il nuovo oggetto con più funzionalità è un aggregato di componenti esistenti.
- *Ereditarietà* (anche detta *White Box Reuse*): la nuova funzionalità è ottenuta mediante l'ereditarietà.

In particolare, esistono tre modi per ottenere le nuove funzionalità: *ereditarietà implementativa*, *ereditarietà di interfaccia*, *delegazione*.

Con l'**ereditarietà di interfaccia**, anche detta **subtyping**, si ereditano da una classe tutte le operazioni specificate, ma non ancora implementate. E' usata allo scopo di classificare concetti in tipi di gerarchia.

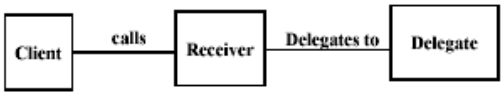
Invece, l'**ereditarietà implementativa**, anche detta **ereditarietà di classe**, ha come obiettivo quello di estendere le funzionalità dell'applicazione mediante il riuso di funzionalità nella classe padre. Si ereditano da una classe esistente alcune o tutte le operazioni già implementate. E' usata allo scopo di riutilizzare il codice.
Siamo nella situazione in cui una classe molto simile, e già implementata, ha funzionalità molto simili a quelle desiderate per la classe da implementare.

Ad esempio, supponiamo di avere una classe *List* e di aver bisogno di una classe *Stack*. Come "sottoclassare" la classe *Stack* dalla classe *List* e fornire i tre metodi, *Push()*, *Pop()* e *Top()*?

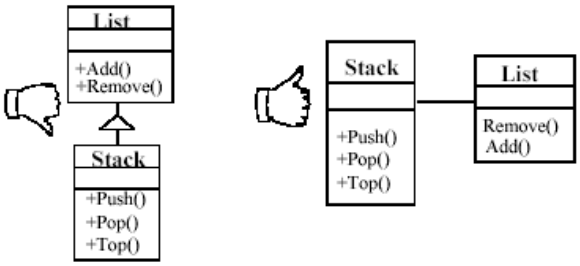


Il problema con questo tipo di ereditarietà è che alcune delle operazioni ereditate potrebbero esibire comportamenti non voluti: cosa accade se gli utenti della classe *Stack* usano *Remove()* invece che *Pop()*?

La **Delegazione**, infine, è un modo di fare composizione (ad esempio aggregazione) in modo potente per il riuso, similmente all'ereditarietà. E' l'alternativa all'ereditarietà di implementazione e rende esplicite le dipendenze tra la classe riutilizzata e la nuova classe.
Con la delegazione sono coinvolti due oggetti nella gestione di una richiesta: un oggetto ricevente delega operazioni al suo delegato. Lo sviluppatore può essere sicuro che l'oggetto ricevente non permette al client di usare impropriamente l'oggetto delegato.



Quindi, è chiaro ora, che ereditare significa estendere una classe base con una nuova operazione o sovrascrivendone una, mentre delegare significa prendere un'operazione e spedirla a un altro oggetto. Nasce spontanea la domanda: Quale dei due modelli è migliore per implementare uno stack?



Alcuni design pattern usano una combinazione di ereditarietà e delegazione.

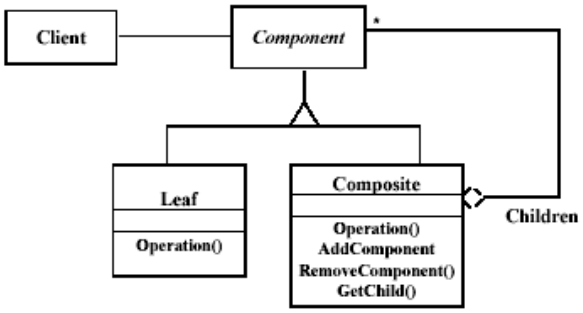
La Delegazione ha come vantaggio la flessibilità, infatti, ogni oggetto può essere rimpiazzato a run time da un altro (purché sia dello stesso tipo). Tuttavia, ha come contro l'inefficienza: gli oggetti sono incapsulati.
L'Ereditarietà, invece, ha come vantaggi la chiarezza nell'uso, è supportata da molti linguaggi di programmazione ed è semplice per implementare nuove funzionalità. D'altra parte, ha come contro il fatto che espone una sottoclasse ai dettagli della sua classe padre e, inoltre, ogni cambiamento nell'implementazione della classe padre forza la sottoclasse a cambiare (il che richiede la ricompilazione di entrambe).
Non è sempre chiaro, tuttavia, quando usare la delegazione o quando usare l'ereditarietà: è richiesta esperienza da parte dello sviluppatore. Le migliori euristiche di design, indicano le seguenti regole:

- Mai usare l'ereditarietà implementativa, ma usare sempre quella di interfaccia.
- Una sottoclasse non dovrebbe mai nascondere le operazioni implementate in una superclasse.
- Se si è tentati dall'utilizzo dell'ereditarietà implementativa, ricorrere piuttosto alla delegazione.

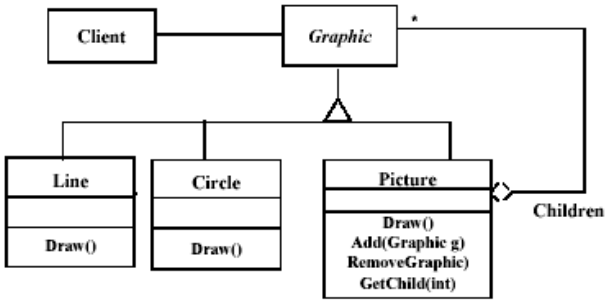
In generale, comunque, possono aiutare i design pattern.

12.5 Composite Pattern

Compone gli oggetti in una struttura ad albero, per rappresentare intere parti di gerarchie con profondità e altezza arbitrarie. Permette ai client di trattare oggetti individuali e composizioni di questi in maniera uniforme.

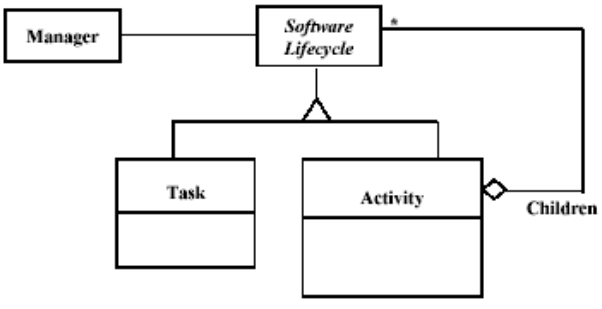


Le applicazioni grafiche, ad esempio, usano i Composite Pattern:

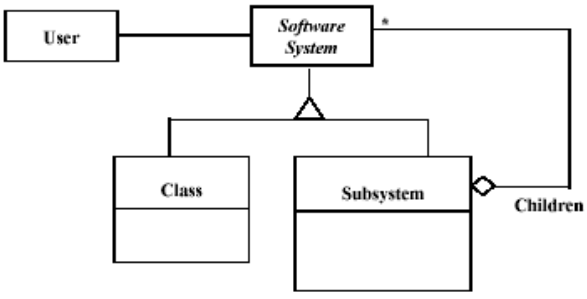


La classe *Graphic* rappresenta sia le primitive (*Line*, *Circle*) che i loro contenitori (*Picture*).

Con i Composite Pattern si può modellare anche lo sviluppo software. In particolare, si può modellare il *Ciclo di Vita del Software* oppure un *Sistema Software*. Secondo questa tecnica, il Ciclo di Vita del Software consiste di un insieme di attività di sviluppo che sono o attività o collezioni di task. In particolare, queste attività sono composite, cioè il ciclo di vita del software consiste di attività, che consistono di attività, che consistono di attività, che… Le foglie di questa struttura sono i task.
Un Sistema Software, invece, viene visto come un insieme di sottosistemi che sono o sottosistemi o collezioni di classi. Allo stesso modo del CVS, i sottosistemi sono compositi, cioè un Sistema Software consiste di sottosistemi, che consistono di sottosistemi, che consistono di sottosistemi, che… Le foglie di questa struttura sono le classi.



Modellazione di un CVS



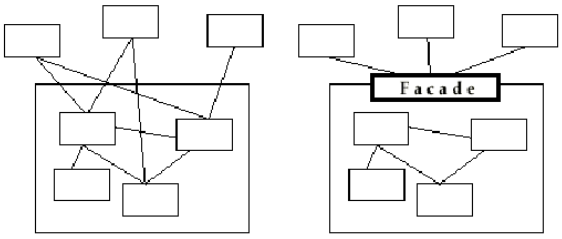
Modellazione di un Sistema Software

12.6 Struttura ideale di un sottosistema: Facade, Adapter, Bridge

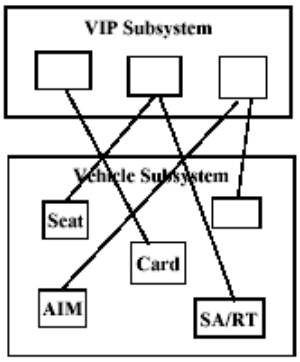
Un sottosistema consiste di un oggetto interfaccia, di un insieme di oggetti del dominio applicativo (entity object) che modellano entità reali o sistemi esistenti (alcuni di questi oggetti sono interfacce di sistemi esistenti), uno o più control object. In questa situazione, la realizzazione delle interfacce avviene mediante i **Facade**, che forniscono le interfacce per i sottosistemi, e la realizzazione dei sistemi esistenti avviene mediante gli **Adapter** o i **Bridge**, che forniscono interfacce ai sistemi esistenti. Chiaramente, i sistemi esistenti non sono necessariamente object-oriented!

Facade Pattern

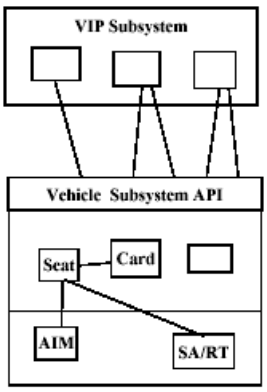
Forniscono un'interfaccia unificata per un insieme di oggetti in un sottosistema. Un facade definisce un'interfaccia di livello più alto che rende il sottosistema più semplice da usare e permette di costruire architetture chiuse.



In un'architettura aperta, ogni client ha visione all'interno del sottosistema *Vehicle* e può chiamare tutte le operazioni delle componenti o delle classi che vuole. Il sistema è buono perché efficiente, d'altra parte, non ci si può aspettare che il chiamante sappia come lavora il sottosistema o che conosca le complesse relazioni tra i sottosistemi. Possiamo essere certi che i sottosistemi saranno utilizzati impropriamente, dando luogo a codice non portabile.



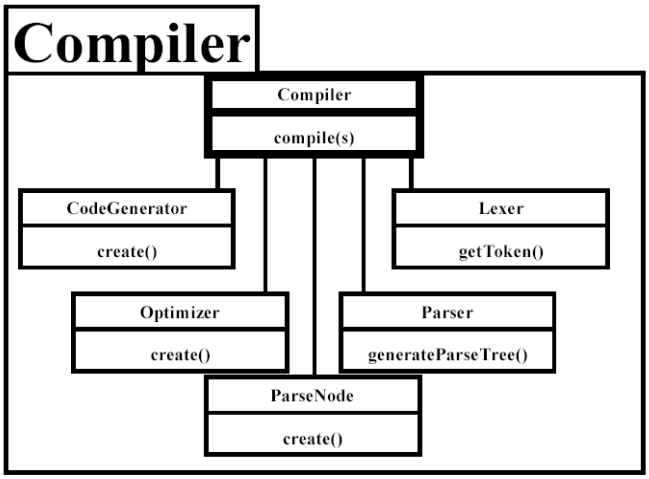
Si può realizzare, però, un'architettura chiusa con un Facade:



Il sottosistema, in questo caso, decide esattamente come essere acceduto.
Non c'è bisogno di preoccuparsi di usi impropri da parte dei chiamanti. Se viene usato un facade, il sottosistema può essere usato in un più semplice test di integrazione: c'è solo bisogno di scrivere un driver.

Con un Facade Pattern si può realizzare un compilatore. A tale scopo, è bene ricordare la notazione UML per i sottosistemi: **Package**. I Package sono collezioni di classi raggruppate insieme, e

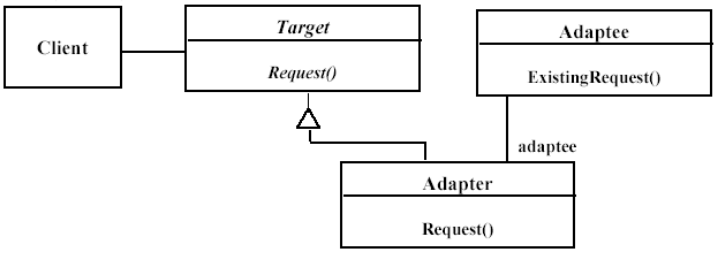
vengono spesso usati per modellare sottosistemi. La notazione è relativamente semplice: un box con una linguetta che contiene il nome del package.



In *Together-J*, ogni classe è assegnata di default a un package. Quando si crea una classe, questa è assegnata direttamente al package di default che contiene il class diagram. Si possono creare altri package, ma non si può cancellare il package di default.

Adapter Pattern

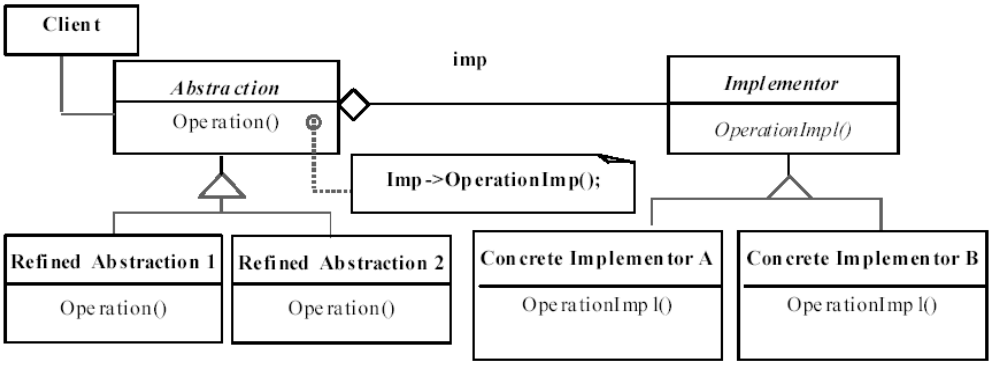
Converte le interfacce di una classe in altre interfacce, che i client si aspettano. **Adapter** permette di far lavorare insieme classi che altrimenti non potrebbero, a causa di incompatibilità di interfacce.
Viene usato per fornire una nuova interfaccia a componenti legacy esistenti (Interface enigeering, reengineering) ed è noto anche come **Wrapper**. Possiamo distinguere tra due adapter pattern: *Class adapter*, che usa ereditarietà multipla per adattare un'interfaccia ad un'altra e *Object adapter*, che usa ereditarietà singola e delegazione.
Gli Object adapter sono molto più frequenti, ne useremo molti e li chiameremo *simply adapter*.



Viene usata la delegazione per legare un *Adapter* a un *Adaptee* (classe esistente) e viene usata l'eridarietà dell'interfaccia per specificare l'interfaccia della classe *Adapter*. *Target* e *Adaptee* (solitamente chiamata legacy system) esistono prima di *Adapter* e *Target* (l'interfaccia per il client). Quest'ultima potrebbe essere realizzata come un'interfaccia in Java.

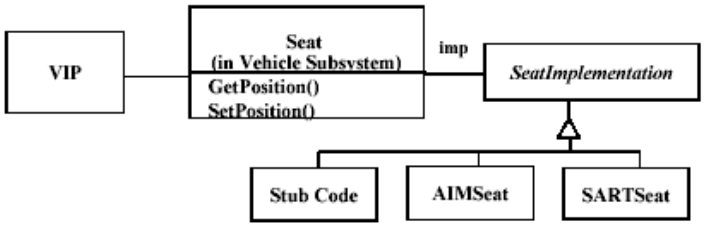
Bridge Pattern

Si usa un **Bridge** per separare un'astrazione dalla sua implementazione, in modo che le due possano variare indipendentemente (Da [*Gamma et al 1995*]). Il **Bridge Pattern**, anche noto come **Handle/Body Pattern**, permette a differenti implementazioni di un'interfaccia di essere decise dinamicamente. E' usato per fornire implementazioni multiple sotto la stessa interfaccia (ad esempio, interfaccia a un componente incompleto, non ancora noto o non disponibile durante il testing).



Esempio: Progetto JAMES

Se è richiesta la lettura dei dati del sedile (seat), ma il sedile ancora non è stato implementato, non è ancora noto o è solo disponibile per una simulazione, si fornisce un bridge:



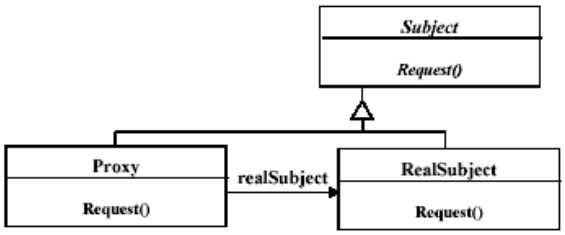
Implementazione del seat:

```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class Stubcode implements SeatImplementation {
  public int GetPosition() {
    // stub code for GetPosition
  }
  ...
}
public class AimSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the AIM simulation system
  }
  ...
}
public class SARTSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the SART seat simulator
  }
}
```

In conclusione, entrambi Adapter e Bridge vengono usati per nascondere i dettagli dell'implementazione sottostante. Differiscono, però, perché l'adapter pattern è attrezzato per la creazione di componenti non correlate che lavorano insieme, per questo è applicato ai sistemi dopo che sono stati progettati (reengineering, interface engineering). Un bridge, invece, viene usato prima del design per permettere di poter variare astrazioni e implementazioni indipendentemente. Usato in greenfield engineering di sistemi estendibili. Chiaramente, *è possibile combinare i due pattern*.

12.7 Proxy Pattern

La creazione e l'inizializzazione di oggetti sono due attività molto costose e con un pattern del genere, si rimandano al momento in cui se ne ha bisogno. Il **Proxy Pattern**, infatti, riduce il costo dell'accesso agli oggetti, usando un altro oggetto (il **proxy**) che fa le veci dell'oggetto reale. Il proxy crea l'oggetto reale solo quando l'utente lo richiede.

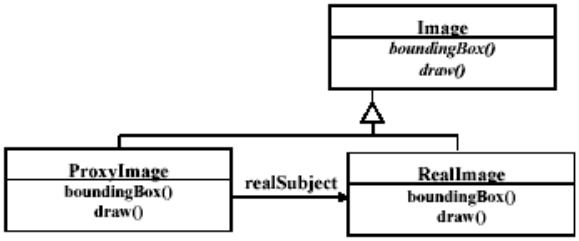


Viene usata l'ereditarietà di interfaccia per specificare l'interfaccia condivisa tra le classi *Proxy* e *RealSubject* e, viene usata la delegazione per catturare e rimandare ogni accesso al *RealSubject* (se desiderato). I proxy pattern possono essere usati per computazioni lente e per invocazioni remote, e possono essere implementati con una interfaccia Java. Inoltre possono essere *facilmente combinati con i Bridge*.

In generale, i proxy possono essere usati in tre maniere differenti:

- *Proxy Remoti*: i proxy sono rappresentativi in locale di oggetti in differenti spazi. Viene usato il *caching delle informazioni*, che chiaramente è utile se queste non cambiano molto spesso.
- *Proxy Virtuali*: gli oggetti sono troppo costosi da creare o da scaricare, un proxy ne fa le veci.
- *Proxy di Protezione*: i proxy forniscono accessi controllati a oggetti reali, e sono utili quando differenti oggetti dovrebbero avere accessi differenti e diritti di visione di uno stesso documento.

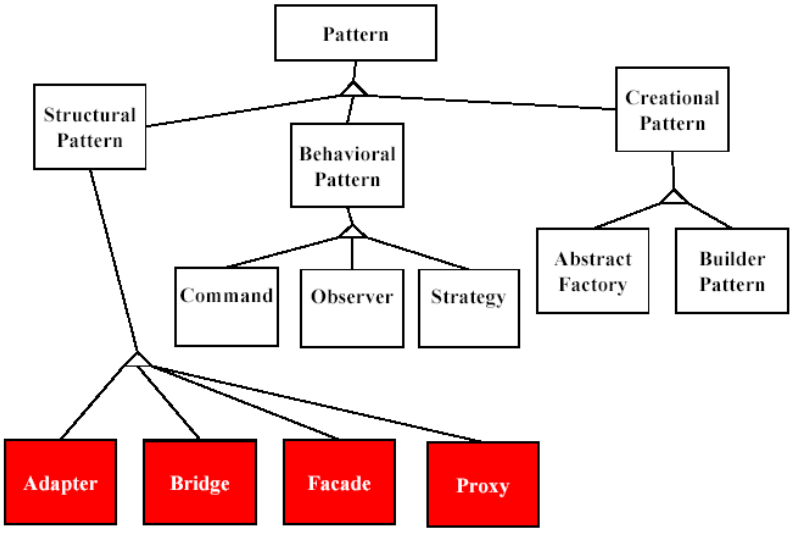
Esempio di Proxy Virtuale:



Le immagini sono memorizzate e caricate separatamente dal testo. Se una *RealImage* non è caricata, un *ProxyImage* mostra un rettangolo grigio al posto dell'immagine. Il client non può sapere se sta trattando con un *ProxyImage* o con una *RealImage*.

12.8 Tassonomia di Pattern

In generale, esistono molti pattern differenti, che possono essere messi in relazione gerarchica tra loro. In realtà ne esiste proprio una tassonomia:



Adapter, Bridge, Facade e Proxy (*pattern strutturali*) sono variazioni di un singolo tema: riducono l'accoppiamento tra due o più classi, introducono una classe astratta per permettere estensioni future e incapsulano strutture complesse.

12.9 Object Design: Attività di Riuso

Vengono cercate classi esistenti in librerie di classi, utili a scegliere le strutture dati appropriate agli algoritmi: classi contenitore, array, liste, code, stack, insiemi, alberi,…
Potrebbe essere necessario definire nuove classi interne ed operazioni. Le operazioni complesse definite in termini di operazioni di più basso livello potrebbero richiedere nuove classi e operazioni.
Probabilmente, bisogna implementare delle classi del dominio applicativo. Infatti, spesso, durante l'object design, sono richiesti nuovi oggetti: l'uso di design pattern introduce nuove classi, l'implementazione di algoritmi può necessitare di oggetti per memorizzare valori, nuove operazioni di basso livello possono essere richieste durante la decomposizione di operazioni di alto livello.
Ad esempio, l'operazione *EraseArea()* in un programma di disegno, concettualmente è molto semplice. Dal punto di vista implementativo, un'area è rappresentata da pixel, *Repair()* riordina oggetti parzialmente coperti dall'area cancellata, *Redraw()* disegna oggetti scoperti dalla cancellazione, *Draw()* cancella pixel con il colore dello sfondo non coperti da altri oggetti.

Bisogna, inoltre, scegliere delle componenti. In particolare, scegliere librerie di classi off-the-shelf esistenti, framework esistenti o componenti esistenti. Adattare le librerie di classi, i framework o le componenti: cambiare l'API se si ha il codice sorgente e usare l'adapter o il bridge pattern se non si ha l'accesso. Si effettua, quindi, un'*Architecture Driven Design*.

Ricordiamo che un'application **framework** è un'applicazione riusabile che può essere specializzata per ottenere applicazioni custom. Il suo riutilizzo evita la ricreazione e la rivalidazione di una soluzione ricorrente.

15. INTRODUZIONE AL TESTING

15.1 Overview

Il **Testing** consiste nel trovare le differenze tra il comportamento atteso, specificato attraverso il modello del sistema, e il comportamento osservato del sistema implementato. Dal punto di vista della modellazione, con il testing si cerca di mostrare che l'implementazione del sistema è inconsistente con il modello del sistema. Quindi, l'obiettivo è quello di progettare test per provare il sistema e rilevare problemi, cercando di massimizzare il numero di errori scoperti, che verranno poi corretti dagli sviluppatori.
Questa attività, ovviamente, va in contrasto con tutte quelle svolte precedentemente: analisi, design e implementazione sono attività "costruttive". Il testing, invece, tenta di "rompere" il sistema. Per questo motivo, dovrebbe essere realizzato da sviluppatori che non sono stati coinvolti nelle attività di costruzione del sistema stesso.

I termini maggiormente utilizzati in questo contesto sono:

- *Affidabilità*: la misura di successo con cui il comportamento osservato di un sistema è conforme ad alcune specifiche del suo comportamento.
- *Failure*: qualsiasi deviazione del comportamento osservato dal comportamento specificato.
- *Stato di Errore*: il sistema è in uno stato tale che ogni ulteriore elaborazione, da parte del sistema stesso, porterà ad un failure.
- *Fault (Bug)*: la causa meccanica o algoritmica di uno stato di errore.

Ci sono molti tipi differenti di errori e molti modi per far fronte a questi. Chiaramente, prima di poter affermare che un comportamento "strano" sia un fault, un errore o un difetto, bisognerebbe prima specificare il comportamento desiderato!

Esempi di Fault ed Errori:

- Fault nella specifica dell'interfaccia: discordanze tra i bisogni del client e ciò che offre il server, discordanze tra requisiti e implementazione, ...
- Fault algoritmici: inizializzazioni mancanti, errori di ramificazione (troppo presto, troppo tardi), test mancanti per null (zero).
- Fault meccanici (molto difficili da trovare): la documentazione non corrisponde alle reali condizioni o procedure operative.
- Errori: errori dovuti a stress o sovraccarico, errori di capacità o limiti, errori di temporizzazione, errori legati alle performance o al throughput.

Ma in che modo bisognerebbe trattare gli errori e i fault? Si potrebbe usare la *Verifica*, che assume ambienti ipotetici che non corrispondono con l'ambiente reale. La dimostrazione, però, potrebbe introdurre bug (perché omette vincoli importanti, perciò è semplicemente sbagliato). Si potrebbe usare *Ridondanza modulare*, ma è costosa, o addirittura *dichiarare il Bug come una caratteristica*, ma non è una buona pratica. Si potrebbero anche usare tecniche di *Patching* (mettere una toppa), ma rallenterebbero le prestazioni.
In definitiva, per affrontare questi errori, adotteremo il **Testing**, anche se non è mai buono abbastanza!

In generale, per trattare gli errori si utilizzano i seguenti approcci:

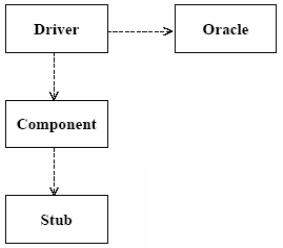
- **Error prevention** (prima di rilasciare il sistema): usare buone metodologie di programmazione per ridurre la complessità, usare versioni di controllo per prevenire inconsistenze di sistema, applicare verifiche per prevenire bug negli algoritmi.
- **Error detection** (mentre il sistema è in esecuzione): mediante Testing (creare failure in maniera pianificata), Debugging (iniziare con failure non pianificati), Monitoraggio (distribuire informazioni sullo stato e cercare bug nelle performance).
- **Error recovery** (recuperare da un failure una volta che il sistema è stato rilasciato): usare transazioni atomiche in sistemi di database, ridondanza modulare e recupero dai blocchi.

Diamo ora delle definizioni di base per il Testing:

- *Errori*: le persone commettono errori.
- *Fault*: un fault è il risultato di un errore nella documentazione software, nel codice, etc…
- *Failure*: un failure avviene quando si esegue un fault.
- *Incident* (episodio): conseguenza di failure (l'occorrenza di una failure può o meno essere visibile dall'utente).
- *Testing*: usare il software con casi di test allo scopo di trovare fault o di acquisire confidenza col sistema.
- *Caso di Test* (Test Case): insieme di dati di input e di risultati attesi (oracle) che usa un componente con l'obiettivo di causare failure.
- *Test suite* (o test): insieme di casi di test.

15.2 Concetti di Testing

Una **Componente** è una parte del sistema che può essere isolata per essere testata: può essere un gruppo di oggetti, o uno o più sottosistemi. Eseguire test case su una componente o su una combinazione di componenti richiede che queste siano isolate dal resto del sistema. Se però, ci sono parti mancanti nel sistema, per eseguire i test si utilizzano i *Test Driver* e i *Test Stub*.
Un **Test Stub** è un'implementazione parziale di una componente da cui dipende la componente testata. In pratica simula le componenti chiamate dalla componente testata.
Un **Test Driver** è un'implementazione parziale di una componente che dipende dalla componente testata. In pratica simula il chiamante della componente testata.
Test stub e Test driver abilitano le componenti ad essere isolate dal resto del sistema per poter essere testate. Possono essere interattivi o automatici: driver e stub automatici e ben progettati permetto di ridurre il tempo di esecuzione del testing. Chiaramente driver e stub ben progettati richiedono uno sforzo maggiore.
Il bisogno di driver e stub, dipende dalla posizione della componente (da testare) nel sistema.

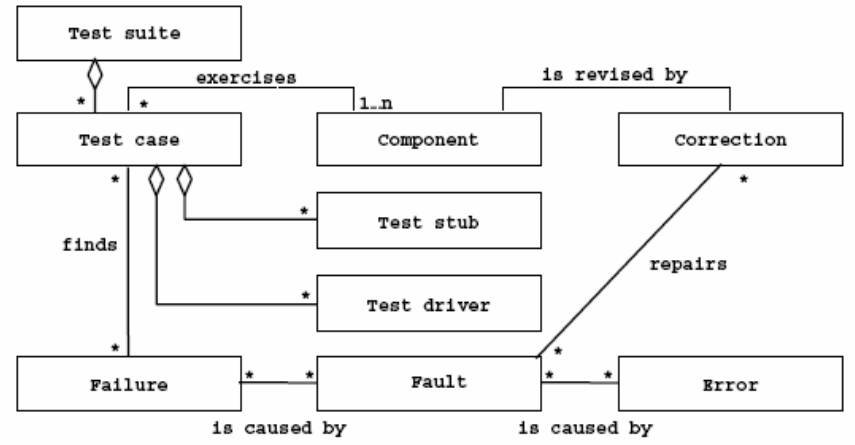


L'oracolo conosce l'output atteso, per un input alla componente, da confrontare con l'output reale per identificare le failure.

Una volta che i test sono stati eseguiti e i fallimenti sono stati rilevati, gli sviluppatori cambiano la componente per il eliminare il fallimento sospettato. Una **correzione** è un cambiamento apportato a

una componente con lo scopo di riparare a un fault. Chiaramente, una correzione potrebbe introdurre nuovi fault, e per questo si utilizzano delle tecniche per gestire questi difetti, come il **Testing di Regressione**: riesecuzione di tutti i test effettuati precedentemente, dopo un cambiamento (richiede di mantenere dei Test Suite).

Sommario delle definizioni:



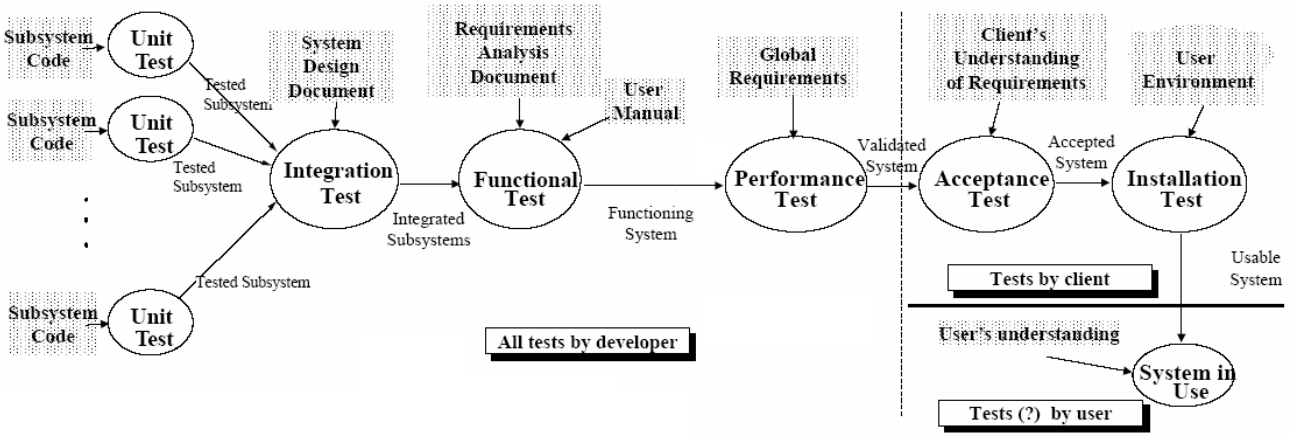
Osservazioni

E' impossibile testare completamente ogni modulo o sistema non banale, a causa di limitazioni teoriche (problemi di arresto) e pratiche (tempi e costi proibitivi). Secondo Dijkstra, il Testing può solamente mostrare la presenza di bug, non la loro assenza, pertanto, un test ha successo se causa una failure. Nella pratica, quindi, il Testing dovrebbe essere integrato con altre attività di verifica, come ad esempio le *ispezioni*.

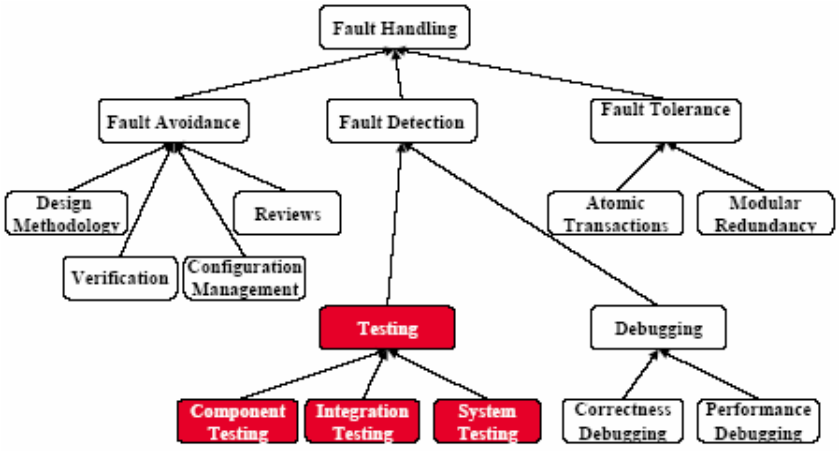
Inoltre, il Testing è spesso visto come uno sporco lavoro. Infatti, per sviluppare un test effettivo, si devono avere conoscenze dettagliate del sistema, conoscenze delle tecniche di testing, skill per applicare queste tecniche in maniera effettiva ed efficiente.

Come è facile intuire, il Testing è fatto meglio da tester indipendenti, perché spesso sviluppiamo una certa attitudine mentale, secondo la quale il programma sarebbe in una certa maniera, mentre in effetti non lo è. I programmatori spesso si attengono a insiemi di dati che consento ai loro programmi di lavorare correttamente, mentre invece un programma non lavora se provato da qualcun altro.

15.3 Attività del Testing



Tecniche di Gestione dei Fault:



Il *test delle componenti* può essere effettuato mediante i seguenti:

- **Unit Testing**: eseguito dagli sviluppatori su sottosistemi individuali, con l'obiettivo di confermare che il sottosistema è codificato correttamente e che esegue le funzionalità intese.
- **Integration Testing**: eseguito dagli sviluppatori su gruppi di sottosistemi (collezioni di classi) ed eventualmente sull'intero sistema, con l'obiettivo di testare le interfacce tra i sottosistemi.

Il test del sistema, invece, viene effettuato mediante i seguenti (più avanti li vedremo tutti):

- **System Testing**: eseguito dagli sviluppatori sull'intero sistema, con l'obiettivo di determinare se il sistema rispecchia i requisiti (funzionali e globali).
- **Acceptance Testing**: valuta il sistema fornito dagli sviluppatori e viene eseguito dai clienti. Può coinvolgere l'esecuzione di transazioni tipiche su basi di prova. Ha l'obiettivo di dimostrare che il sistema rispecchia i requisiti del cliente e che è pronto all'uso.

Implementazione e Testing vanno di pari passo.

15.4 Documentare il Testing

Per minimizzare le risorse necessarie, il Testing deve essere gestito. Alla fine, gli sviluppatori dovrebbero rilevare e riparare un numero sufficiente di bug tale che il sistema soddisfi i requisiti funzionali e non funzionali entro un limite accettabile da parte del cliente. Per questi motivi, bisogna **pianificare le attività**. Si utilizzano a tale scopo i seguenti documenti:

Test Plan: focalizza sugli aspetti manageriali del testing. In particolare, documenta gli scopi, gli approcci, le risorse, lo schedule delle attività di testing. In questo documento sono identificati i requisiti e le componenti che devono essere testati.

1. Introduction
2. Relationship to other documents
3. System overview
4. Features to be tested/not to be tested
5. Pass/Fail criteria
6. Approach
7. Suspension and Resumption
8. Testing materials
9. Test Cases
10. Testing Schedule

Test Case Specification: documenta ogni test, in particolare, contiene gli input, i driver, gli stub, e gli output attesi dai test, così come i task che devono essere eseguiti.

1. Test case specification identifier
2. Test items
3. Input Specifications
4. Output Specifications
5. Environmental needs
6. Special procedural requirements
7. Intercase dependencies

Test Incident Report: documenta ogni esecuzione, in particolare, i risultati reali dei test e le differenze dagli output attesi.

Test Summary Report: elenca tutte le failure, rilevate durante i test, che devono essere investigate. Da questo documento, gli sviluppatori analizzano e assegnano priorità a ogni failure e pianificano i cambiamenti da apportare al sistema e ai modelli. Questi cambiamenti posso introdurre nuovi test case e nuove esecuzioni di test.

16.1 Overview

L'**Unit Testing** può essere eseguito in maniera *informale*, mediante la codifica incrementale, oppure mediante analisi statiche o dinamiche. L'*analisi statica* avviene mediante: esecuzione manuale, leggendo il codice sorgente; walk-through (presentazione informale ad altri); ispezione del codice (presentazione formale ad altri); tool automatici per controllare errori sintattici e semantici, e divergenze dagli standard di codifica. L'*analisi dinamica*, invece, avviene mediante: **Black-box testing** (per testare i comportamenti di input/output), **White-box testing** (per testare la logica interna del sottosistema o dell'oggetto), **Testing basato sulle strutture dati** (i tipi di dati determinano i casi di test).
In generale, il Testing di Unità nasce da tre motivazioni: si riduce la complessità concentrandosi su una sola unit del sistema per volta, è più facile correggere i bug perché poche componenti sono coinvolte, è possibile testare più unità in parallelo. Le unità candidate per il test sono prese dal modello a oggetti e dalla decomposizione in sottosistemi (i sottosistemi devono essere testati dopo che ogni classe e oggetto del sottosistema è stato testato individualmente).

16.2 Black-Box Testing

Si focalizza sul comportamento di I/O, senza preoccuparsi della struttura interna della componente. Se per ogni dato input, siamo in grado di prevedere l'output, allora il modulo supera il test. Tuttavia, è quasi sempre impossibile generare tutti i possibili input, cioè i test case. L'obiettivo diventa, quindi, quello di ridurre il numero di casi di test effettuando un partizionamento: si dividono le condizioni di input in **classi di equivalenza** e si scelgono i test case per ogni classe di equivalenza. Ad esempio, se si suppone che un oggetto accetta un numero negativo, testare un numero negativo è sufficiente.
Per selezionare le classi di equivalenza, non ci sono regole, ma solo due linee guida:

- se l'input è valido per un range di valori, si scelgono i test case da 3 classi di equivalenza: sotto il range, dentro il range, sopra il range;
- se l'input è valido solo se appartiene ad un insieme discreto, si scelgono i test case da 2 classi di equivalenza: valore discreto valido, valore discreto non valido.

Un'altra soluzione per scegliere solo un numero limitato di test case è giungere a conoscenza dei comportamenti interni delle unità da testare (*White-Box Testing*).

16.3 Equivalence Class Testing

Nasce dalla motivazione di cercare di avere un testing completo e con la speranza di evitare le ridondanze. Le classi di equivalenza sono partizioni dell'input set. L'intero input set viene coperto (ottenendo la *completezza*) e le classi sono disgiunte (*evitando ridondanza*). I test case sono elementi di ogni classe di equivalenza. Il problema è che le classi di equivalenza devono essere scelte saggiamente, indovinando i più probabili comportamenti sottostanti.

Siano A, B e C tre variabili del dominio di input, definite come segue:

$$A = A1 \cup A2 \cup A3 \text{ where } a_n \in An$$
$$B = B1 \cup B2 \cup B3 \cup B4 \text{ where } b_n \in Bn$$
$$C = C1 \cup C2 \text{ where } c_n \in Cn$$

Il **Weak Equivalence Class Testing** sceglie un variabile valore da ogni classe di equivalenza e lo **Strong Equivalence Class Testing**, basato sul prodotto cartesiano degli insiemi partizione (AxBxC), testa tutte le interazioni tra le classi. Se le condizioni di errore hanno alta priorità, dovremmo estendere lo Strong Equivalence Class Testing per includere le classi non valide.
L'Equivalence Class Testing è appropriato quando i dati in input sono definiti in termini di range e insiemi di valori discreti, mentre lo Strong Equivalence Class Testing parte dall'assunzione che le variabili sono indipendenti (le dipendenze generano "error" test case).

16.4 Boundary Value Testing

Abbiamo partizionato i domini di input in classi convenienti, basandosi sull'assunzione che il comportamento del programma sia simile. Alcuni tipici errori di programmazione, però, avvengono ai confini tra classi differenti. Su questo concetto focalizza l'attenzione il **Boundary Value Testing**, più semplice, ma complementare alle precedenti tecniche di testing viste.

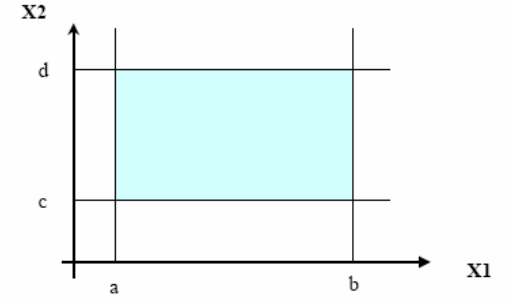
Sia *F* una funzione di due variabili x_1 e x_2. I limiti (possibilmente non fissati) sono:

$$a \le x_1 \le b \text{ e } c \le x_2 \le d.$$

In alcuni linguaggi di programmazione, la forte specializzazione permette di specificare tali intervalli. Per identificare i test case, si focalizza sui limiti dello spazio input, proprio perché gli errori tendono ad avvenire in prossimità dei valori estremi delle variabili input.

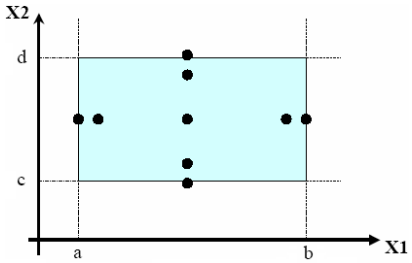
I valori che possono assumere le variabili input sono il minimo, un po' più del minimo, un valore nominale, un po' meno del massimo, il massimo. Per convenzione, denotiamo questi valori nella maniera seguente: min, min+, nom, max-, max.

L'input domain della funzione *F* è:

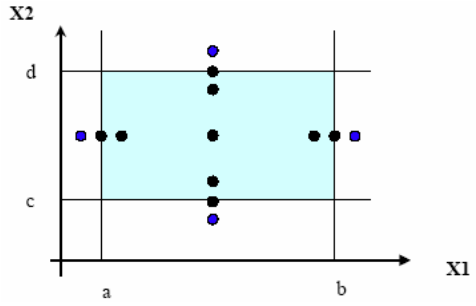


Analizziamo i Boundary Test Case. Il Test set si ottiene fissando una variabile al suo valore nominale e facendo variare le altre (in questo caso una) per tutti i possibili valori. Questo, per ogni variabile (nel nostro caso due):

$$\text{Test set} = \{<x1_{nom}, x2_{min}>, <x1_{nom}, x2_{min+}>, <x1_{nom}, x2_{nom}>, <x1_{nom}, x2_{max-}>, <x1_{nom}, x2_{max}>, <x1_{min}, x2_{nom},>, <x1_{min+}, x2_{nom},>, <x1_{max-}, x2_{nom}>, <x1_{max}, x2_{nom}>\}$$



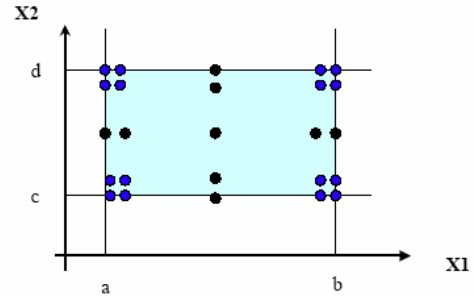
E' chiaro, quindi, che una funzione di *n* variabili richiede *4n +1* test case. Questo tipo di testing lavora bene con variabili che rappresentano quantità fisicamente limitate, ma non c'è nessuna considerazione sulla natura della funzione e sul significato delle variabili. Inoltre è una tecnica rudimentale, sensibile al **Robustness Testing**:



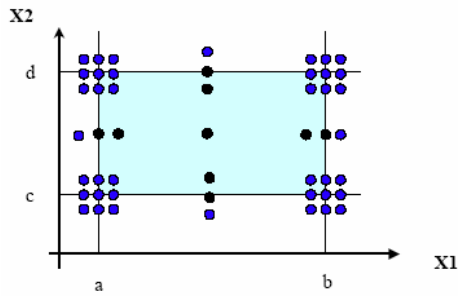
16.5 Worst Case Testing

I valori limite (il testing precedente) partono dalla comune assunzione che le failure, per la maggioranza delle volte, sono originate da un fault. Ma cosa accade se più di una variabile assume un valore estremo? L'idea nasce dall'elettronica nell'analisi dei circuiti: prodotto cartesiano di {min, min+, nom, max-, max}. Chiaramente è una tecnica più completa dell'analisi dei Boundary Value, ma molto più pesante: 5^n casi di test.
E' una buona strategia se le variabili fisiche hanno numerose interazioni, e dove le failure sono costose. Ancora meglio è il **Robust Worst Case Testing**.

Worst Case Testing per due variabili



Robust Worst Case Testing per due variabili



16.6 White-Box Testing

Si focalizza sulla completezza (copertura). Ogni statement nella componente è eseguita almeno una volta. Indipendentemente dall'input, ogni stato nel modello dinamico dell'oggetto e ogni interazione tra gli oggetti viene testata.

Esistono quattro tipi di White-box Testing:

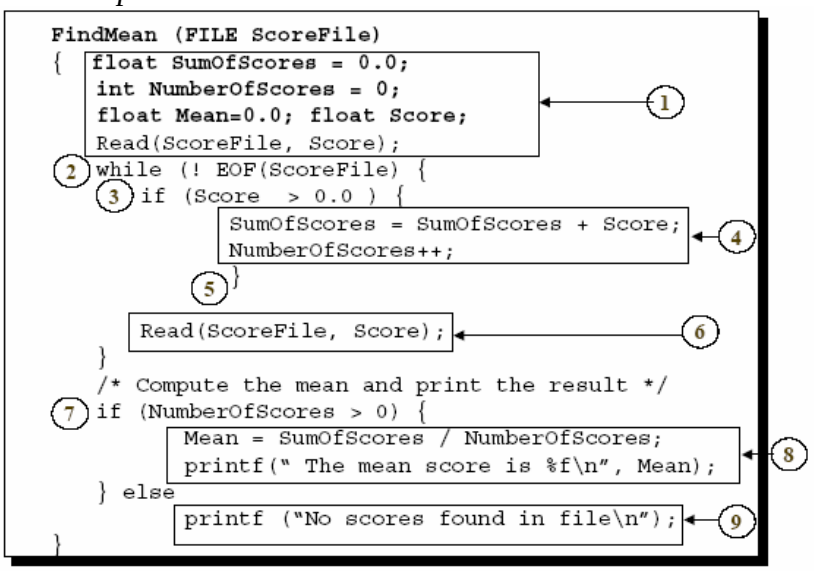
- **Statement Testing** (testing algebrico): si testano i singoli statement (scelta di operatori polinomiali, etc.).
- **Loop Testing**: provoca l'esecuzione del loop che deve essere saltato completamente (eccezione: ripetere i cicli). I loop possono venire eseguiti esattamente una volta o più di una volta.
- **Path Testing**: assicura che tutti i path nel programma siano eseguiti.
- **Branch Testing** (testing condizionale): assicura che ogni possibile uscita da una condizione sia testata almeno una volta.

```
if ( i =TRUE) printf("YES\n");else printf("NO\n");
Test cases: 1) i = TRUE; 2) i = FALSE
```

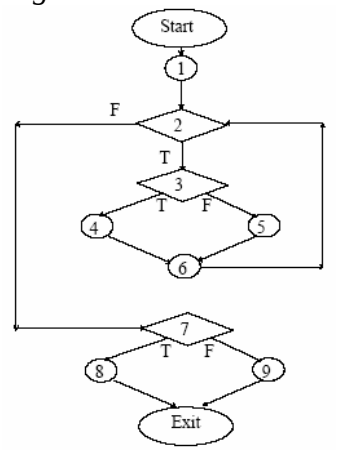
Esempio di White-box Testing

```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
              if ( Score > 0.0 ) {
               SumOfScores = SumOfScores + Score;
              NumberOfScores++;
             }
 Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
             Mean = SumOfScores/NumberOfScores;
             printf("The mean score is %f \n",  Mean);
 } else
             printf("No scores found in file\n");
}
```

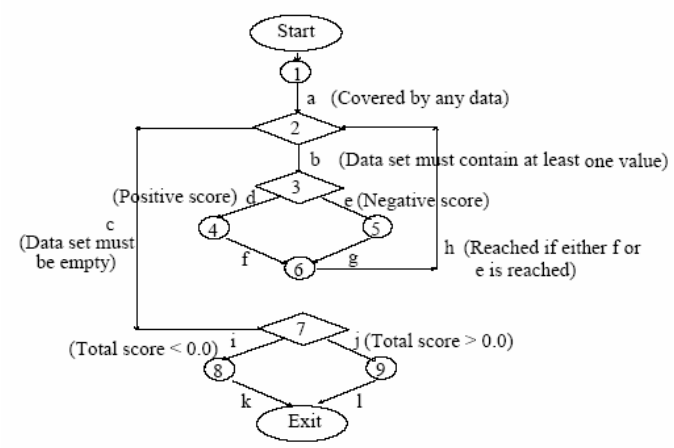
Determinazione dei path:



Costruzione del Logic Flow Diagram:



Trovare i Test Case:

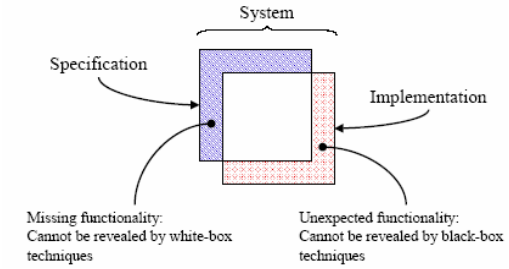


Test Case:

Test case 1 : ? (eseguire il loop esattamente una volta)
Test case 2 : ? (saltare il corpo del loop)
Test case 3 : ? (eseguire il loop più di una volta)
Questi 3 test case coprono tutti i cammini del flusso di controllo.

16.7 Confronto tra White-Box e Black-Box Testing

Potenzialmente un numero infinito di cammini dovrebbero essere testati. Il White-box Testing spesso testa ciò che è stato fatto, invece di ciò che dovrebbe essere fatto. Tuttavia, non individua i casi d'uso mancanti. Il Black-box Testing, invece, ha la peculiarità di essere una potenziale esplosione combinatoria di casi di test (dati validi e non). Spesso, però, non è chiaro se i casi di test selezionati scoprono particolari errori o meno e non scopre casi d'uso estranei.



In generale, entrambi i tipi di testing sono necessari e rappresentano il culmine di una serie ininterrotta di testing. Ogni scelta di un caso di test si trova (e dipende) tra i seguenti: numero di possibili cammini logici, natura dei dati in input, quantità di computazione, complessità degli algoritmi e delle strutture dati.

16.8 Consigli sul Testing di Unità

Per organizzare il testing, dovrebbero bastare i seguenti quattro passi:

1. Selezionare cosa deve essere misurato: completezza dei requisiti, codice testato per l'affidabilità, design testato per la coesione.
2. Decidere come deve essere fatto il testing: ispezione del codice, dimostrazioni, black-box, white-box, selezionare la strategia per il testing di integrazione (big bang, bottom up, top down, sandwich).
3. Sviluppare i test case: un test case è un insieme di dati di test o di situazioni di test che saranno usati per usare le unità (codice, modulo, sistema) da testare o gli attributi da misurare.
4. Creare il test oracle: un oracolo contiene il risultato previsto per un insieme di test case, e deve essere scritto prima che l'attuale testing abbia luogo.

Per scegliere i Test Case, si può fare riferimento alle seguenti linee guida:

- Usare conoscenze di analisi a riguardo dei requisiti funzionali (black-box): casi d'uso, dati attesi in input, dati in input non validi.
- Usare conoscenze di design a riguardo della struttura del sistema, degli algoritmi, delle strutture dati (white-box): strutture di controllo (rami di test, loop,…), strutture dati (campi record di test, array,…), …
- Usare conoscenze di implementazione a riguardo degli algoritmi: forzare la divisione per zero, usare sequenze di casi di test per interrompere l'handler, …

Euristiche di Unit Testing

1. Creare test di unità appena l'object design è completato: Black-box test (testa i casi d'uso e il modello funzionale), White-box test (testa il modello dinamico), Data-structure test (testa il modello a oggetti).
2. Sviluppare i casi di test: l'obiettivo è trovare il minimo numero di casi di test che coprono il maggior numero di cammini possibile.
3. Controllare i test case per eliminare i duplicati (senza sprecare tempo).
4. Provare il codice sorgente (riduce il tempo di testing).
5. Creare un test da sfruttare: i test driver e i test stub sono richiesti anche per il test di integrazione.
6. Descrivere il test oracle: spesso il risultato del primo test eseguito con successo.

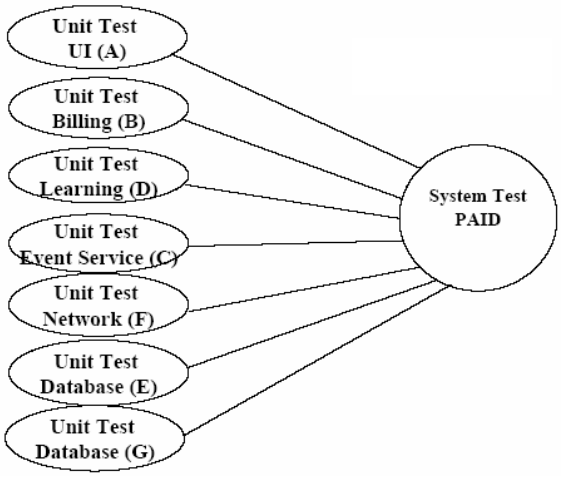
7. Eseguire i test case (non dimenticare il test di regressione e rieseguire i test case ogni volta che avviene un cambiamento).
8. Confrontare i risultati del test con il test oracle (automatizzando il processo il più possibile).

17.1 Overview

Quando i bug in ogni componente sono stati rilevati e riparati, le componenti sono pronte per essere integrate in sottosistemi più grandi. Il **Test di Integrazione** rileva bug che non sono stati determinati durante il Test di Unità, focalizzando l'attenzione su un insieme di componenti che vengono integrate. Due o più componenti vengono integrate e analizzate, e quando dei bug sono rilevati, possono essere aggiunte nuove componenti per correggerli.
Siamo, quindi, nella situazione in cui l'intero sistema è visto come una collezione di sottosistemi (insiemi di classi) determinati durante il system e l'object design. L'ordine in cui i sottosistemi vengono selezionati per il testing e per l'integrazione determina le **strategie di testing**: *Big bang integration*, *Bottom up integration*, *Top down integration*, *Sandwich testing*, *Variazioni* delle precedenti.
Per la scelta si usa la decomposizione del sistema effettuata durante il System Design.

17.2 Approccio Big-Bang

Le componenti vengono prima testate individualmente e poi insieme, come un unico sistema.

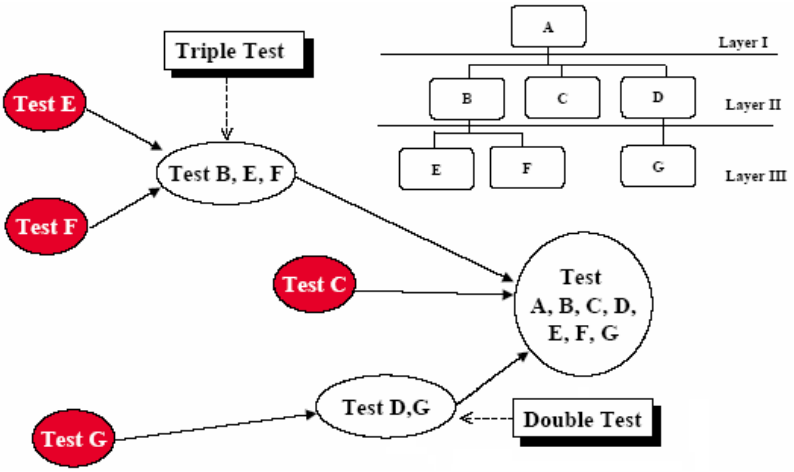


Sebbene sia semplice, è costoso: se un test scopre una failure, è impossibile stabilire se è nell'interfaccia o all'interno della componente.

17.3 Strategia di Testing Bottom-up

I sottosistemi al livello più basso della gerarchia sono testati individualmente. I successivi sottosistemi ad essere testati sono quelli che chiamano i sottosistemi testati in precedenza. Si ripete quest'ultimo passo finché tutti i sottosistemi non sono stati testati.

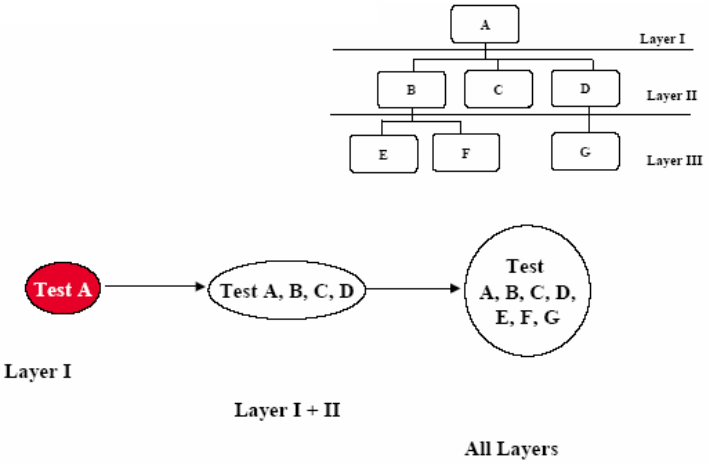
Per il testing vengono utilizzati i Test Driver (che simulano le componenti dei livelli più alti, che non sono state ancora integrate).



Questo approccio non è buono per sistemi decomposti funzionalmente, perché testa i sottosistemi più importanti solo alla fine, ma è utile per integrare sistemi object-oriented, real-time, con rigide richieste di performance.

17.4 Strategia di Testing Top-down

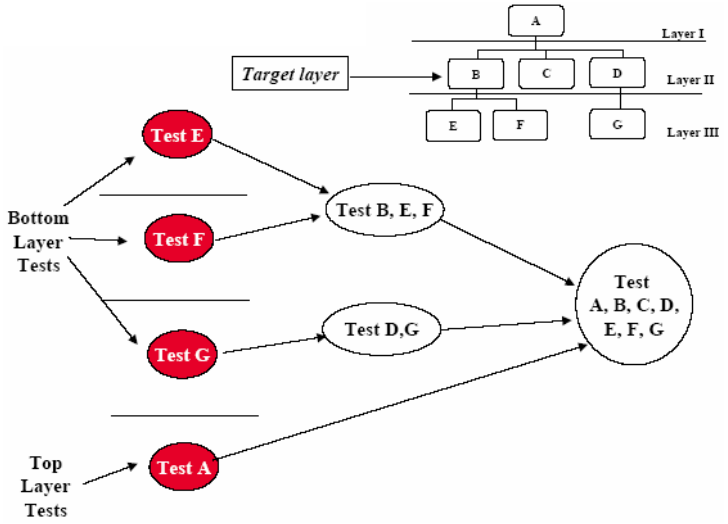
Testa prima i livelli al top o i sottosistemi di controllo e, successivamente, combina tutti i sottosistemi che sono chiamati da quelli già testati e testa la collezione risultante di sottosistemi. Ripete il tutto fino a quando tutti i sottosistemi non sono incorporati nel test.
Per il testing vengono utilizzati i Test Stub (che simulano le componenti dei livelli più bassi, che non sono state ancora integrate).



Il vantaggio con questo approccio è che i test case possono essere definiti in termini delle funzionalità del sistema (requisiti funzionali), tuttavia, scrivere gli stub può essere difficile: devono consentire tutte le possibili condizioni da testare. Inoltre, è possibile che un grande numero di stub sia richiesto, specialmente se il livello più in basso nel sistema contiene molti metodi.

17.5 Strategia di Testing Sandwich

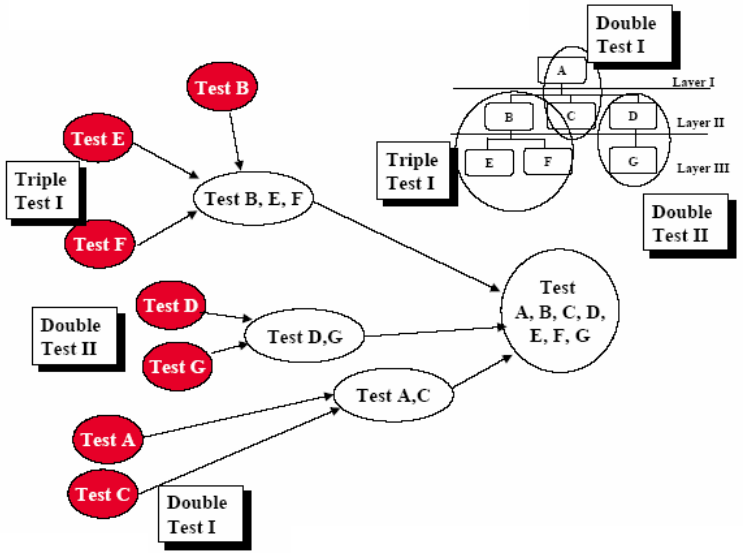
Combina l'uso di strategie top-down e bottom-up. Il sistema è visto come se avesse 3 strati: un livello target nel mezzo, uno sopra il target, uno sotto il target.
Chiaramente, il testing converge al target. Ma come selezionare il layer target se ci sono più di 3 layer? Si può provare a minimizzare il numero di stub e di driver.



Il vantaggio principale, nell'utilizzare questo approccio, è che i test dei livelli in alto e in basso possono essere eseguiti in parallelo. Il problema è che non vengono testati sottosistemi individuali del livello target completamente, prima dell'integrazione. Soluzione: *Strategia di Testing Sandwich Modificata*.

17.6 Strategia di Testing Sandwich Modificata

Testa, in parallelo, il livello al top con stub per il target, il livello nel mezzo con driver e stub per i livelli al top e in basso rispettivamente, il livello in basso con driver per il target. Inoltre, testa in parallelo il livello in alto che accede a quello nel mezzo (il livello al top rimpiazza i driver) e il livello in basso acceduto da quello nel mezzo (il livello in basso rimpiazza gli stub).



17.7 Testing di Integrazione: Conclusioni

I passi da seguire per effettuare un Testing basato su Componenti sono:

1. Basandosi sulla strategia di integrazione, selezionare una componente da testare e fare il Test di Unità di tutte le classi nella componente.
2. Mettere insieme le componenti selezionate e apportare ogni aggiustamento preliminare necessario a rendere operativo il test di integrazione (driver, stub).
3. Eseguire un testing funzionale: definire casi di test che usano tutti i casi d'uso con le componenti selezionate.
4. Eseguire un testing strutturale: definire casi di test che usano le componenti selezionate.
5. Eseguire un test delle performance.
6. Tenere traccia dei casi di test e delle attività di testing.
7. Ripetere i passi da 1. a 7. fino a che tutto il sistema è stato testato.

L'obiettivo primario del testing di integrazione è quello di identificare errori nella configurazione della componente (corrente).
Il problema è: quale strategia di integrazione usare? I fattori da considerare sono: quantità di test da sfruttare, locazione delle parti critiche nel sistema, disponibilità dell'hardware e delle componenti, interessi di scheduling.
L'approccio bottom up è buono per metodologie di design orientate agli oggetti, ma ha comunque dei problemi: i test driver delle interfacce devono corrispondere con le componenti interfacce; le componenti di alto livello sono di solito importanti e non possono essere trascurate fino alla fine del testing; l'individuazione di errori di design è posticipata fino alla fine del testing.
L'approccio top down, invece, impone che i test case siano definiti in termini di funzioni esaminate e richiede di mantenere la correttezza dei test stub, la cui scrittura può però essere difficoltosa.

17.8 System Testing

Unit Testing e Integration Testing focalizzano l'attenzione sulla ricerca di bug nelle componenti individuali e nelle interfacce tra le componenti. Il **System Testing**, assicura che il sistema completo è conforme ai requisiti funzionali e non. Le attività per questo testing sono: *Structure Testing*, *Functional Testing*, *Pilot Testing*, *Performance Testing*, *Acceptance Testing*, *Installation Testing*.
L'impatto dei requisiti sul testing di sistema è pesante: più espliciti sono i requisiti, più facili sono da testare; la qualità degli use case determina la facilità del functional testing; la qualità della decomposizione in sottosistemi determina la facilità dello structure testing, la qualità dei requisiti non funzionali e dei vincoli determina la facilità del performance testing.

Structure Testing
Essenzialmente è lo stesso di un White-Box Testing. L'obiettivo è coprire tutti i cammini nel system design: usa tutti i parametri di input e output per ogni componente, usa tutte le componenti e tutti i chiamanti (ogni componente è chiamato almeno una volta e tutte le componenti sono chiamate da tutti i possibili chiamanti), usa testing delle condizioni e delle iterazioni come il testing di unità.

Functional Testing

Essenzialmente è lo stesso di un Black-Box Testing. L'obiettivo è testare le funzionalità del sistema, che viene trattato come una black box. I test case sono progettati a partire dal RAD (meglio dal manuale utente) e incentrati attorno ai requisiti e alle funzioni chiave (casi d'uso): istanziare casi d'uso per derivare casi di test (le istanze dei casi d'uso sono i scenari,…), differenti scenari possono essere ottenuti applicando tecniche black box ai dati in input. I casi di test di unità possono essere riusati, ma nei nuovi casi di test orientati agli utenti finali dovrebbero essere sviluppati al meglio.

Performance Testing

Si vuole spingere il sistema (integrato) verso i suoi limiti, con l'obiettivo di provare a "romperlo". Bisogna testare come il sistema si comporta quando è sovraccarico: possono essere identificati colli di bottiglia? (I primi candidati ad essere riprogettati nella prossima iterazione). Provare ordini di esecuzione non usuali (chiamare receive() prima di send(), ad esempio). Controllare le risposte del sistema a grandi volumi di dati: se si suppone che il sistema possa gestire 1000 item, provarlo con 1001.

Qual è la quantità di tempo spesa nei differenti casi d'uso? I casi tipici vengono eseguiti in un tempo fattibile?

I Test effettuati per testare le performance sono:

- *Stress Testing*: stressa i limiti del sistema (massimo numero di utenti, picchi di richieste, operazioni estese).
- *Volume Testing*: test cosa accade se vengono gestite grosse quantità di dati.
- *Configuration Testing*: testa le varie configurazioni hardware e software.
- *Compatibility Test*: testa la compatibilità all'indietro con sistemi esistenti.
- *Security Testing*: prova a violare i requisiti di sicurezza.
- *Timing Testing*: valuta i tempi di risposta e il tempo per eseguire una funzione.
- *Environmental Test*: testa la tolleranza al caldo, all'umidità, al movimento, alla portabilità.
- *Quality Testing*: testa l'affidabilità, la mantenibilità e la disponibilità del sistema.
- *Recovery Testing*: testa la risposta del sistema in presenza di errori o di perdite di dati.
- *Human factor Testing*: testa le interfacce utente con l'utente.

Acceptance Test

L'obiettivo è dimostrare che il sistema è pronto per l'uso operativo. A tale scopo:

- la scelta dei test è fatta dai clienti/sponsor;
- molti test possono essere presi dal testing di integrazione;
- il test di accettazione è eseguito dal cliente, non dagli sviluppatori.

La maggioranza di tutti i bug nel software è tipicamente individuata dal cliente, dopo che il sistema è in uso, non dagli sviluppatori o dai tester. Perciò vengono introdotti due test addizionali:

- **Alpha Test**: gli sponsor usano il software negli ambienti degli sviluppatori. Il software è usato con un settaggio controllato, con gli sviluppatori sempre pronti a correggere bug.

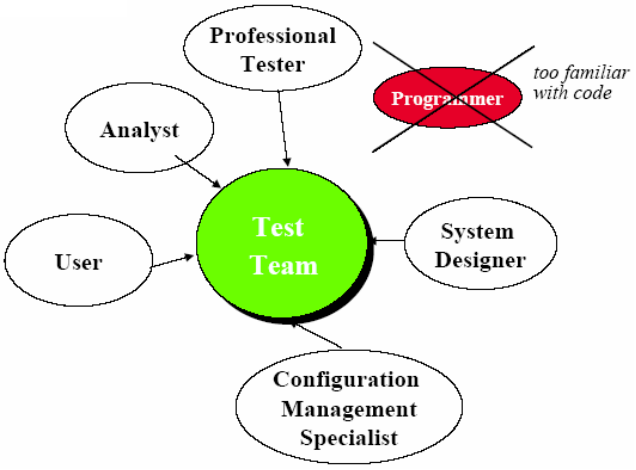
- **Beta Test**: è condotto negli ambienti degli sponsor, senza la presenza degli sviluppatori. Il software inizia a lavorare realisticamente nell'ambiente per il quale è stato progettato. Clienti potenziali potrebbero essere scoraggiati.

17.9 Considerazioni conclusive

Il Testing è ancora un'arte oscura, ma sono comunque disponibili molte regole ed euristiche. Questa attività consiste di Component Testing (Unit Testing e Integration Testing) e System Testing e l'intero processo, sembra avere un suo **Ciclo di Vita**:

Stabilire gli obiettivi dei test
Progettare i test case
Scrivere i test case
Testare i test case
Eseguire i test case
Valutare i risultati dei test
Cambiare il sistema
Eseguire il testing di regressione

Di seguito è riportata la struttura di un **Test Team**:



18. TESTING PER SOFTWARE OBJECT ORIENTED

18.1 Overview

Il testing di sistemi software object oriented deve *sempre* tener conto dell'astrazione sui dati (stato e information hiding), dell'ereditarietà, del polimorfismo, del binding dinamico, e deve tener *spesso* conto della genericità, della gestione delle eccezioni, della concorrenza.

Pertanto, nel caso in cui il software da trattare sia object oriented, l'impatto sul Test è considerevole. Bisogna, infatti, definire nuovi livelli di test, perché il concetto di classe (vista come dati + operazioni) modifica il concetto di unità e il test di integrazione di oggetti è diverso dal test di integrazione tradizionale. Bisogna, inoltre, costruire una nuova infrastruttura (driver e stub devono considerare lo stato) e nuovi oracoli (lo stato non può essere ispezionato con tecniche tradizionali). Bisogna, infine, introdurre nuove tecniche di generazione dei casi di test.

Nei linguaggi procedurali standard, la componente di base è la procedura ed è questa che viene testata con metodi basati su input/output. Nei linguaggi object oriented, la componente di base è la classe (strutture dati + insieme di operazioni), e le istanze di classe sono oggetti. Quindi, la correttezza non è legata solo all'output, ma anche allo stato definito dalla struttura dati, e lo stato "privato" può essere osservato solo utilizzando metodi pubblici della classe (e quindi affidandosi a codice sotto test).

18.2 Test dei Metodi (Test di Unità??)

Il test del singolo metodo può essere fatto con tecniche tradizionali (metodo=procedura), ma i matodi sono più semplici delle procedure e lo scaffolding più complicato. Prestare attenzione all'ereditarietà, perché lo stesso metodo può venire usato più volte in contesti diversi.
Può convenire fare il test dei metodi durante il test delle classi, ma non si può ignorare il test del singolo metodo!
L'infrastruttura deve settare opportunamente lo stato per poter eseguire i test (driver) ed esaminare lo stato per poter stabilire l'esito dei test (oracoli), ma lo stato è "privato"… Si usano allora approcci intrusivi, per modificare il codice sorgente (e aggiungere un metodo *testdriver* alla classe) e per usare costrutti del linguaggio (ad esempio il costrutto *friend*).
La tecnica è quella di costruire una *macchina a stati finiti*, dove gli stati sono l'insieme di stati della classe e le transizioni sono le invocazioni dei metodi.
Per derivare i casi di test, bisogna percorrere la macchina a stati finiti. Si usano due approcci: basato su specifiche e basato su codice.
I cammini di una macchina a stati finiti corrispondono a sequenze di esecuzione, quindi, un insieme di test ragionevole può essere ottenuto selezionando tutti i cammini senza cicli. In tal caso, i casi di test sono i cammini della macchina a stati finiti.
Chiaramente diverse tecniche (per generare macchine a stati finiti e limitare il numero di cammini) corrispondono a diversi insiemi di test generati.

La macchina a stati finiti può essere utilizzata per identificare *criteri di copertura strutturale*:

- generare sequenze di test a partire da specifiche;
- generare la macchina a stati finiti a partire da codice;
- la qualità del test generato (o del software) può essere misurata dal grado di copertura dei cammini della macchina a stati finiti.

I problemi che si riscontrano nell'utilizzo di questa tecnica, sono dovuti al fatto che esistono cammini che non corrispondono a test funzionali, a causa di:

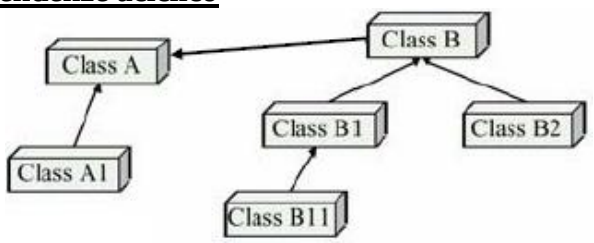
- cattiva definizione dei test funzionali (sono stati dimenticati casi significativi);
- cattiva specifica (mancano casi significativi);
- cattiva implementazione (casi non richiesti);

Ed inoltre, esistono test funzionali che non corrispondono a cammini a causa di una cattiva implementazione (missing paths).

18.3 Testing di Integrazione per Software OO

L'approccio *Big-bang* è in generale poco adatto. Per quanto riguarda le strategie *Top-down* e *bottom-up*, cambia il tipo di dipendenze tra "moduli". Le dipendenze riguardano l'uso di classi e l'ereditarietà: se A usa B allora A dipende da B, se A eredita da B allora A dipende da B. E' preferibile una strategia bottom-up (testare prima le classi indipendenti), ma gli Stub sono troppo difficili da costruire. Si potrebbe anche usare un approccio basato su Thread.
In generale, le dipendenze tra le classi possono essere espresse su un **grafo delle dipendenze**. Se il grafo è aciclico esiste un ordinamento parziale sui suoi elementi: è possibile definire un ordinamento totale topologico e privilegiare le dipendenze di specializzazione. L'ordine d'integrazione viene definito in base a tale ordinamento.
Se esistono dipendenze cicliche tra le classi, è impossibile definire un ordinamento parziale, ma ogni grafo orientato ciclico è riducibile a un grafo aciclico, collassando i sottografi massimi fortemente connessi.

Esempio: Grafo delle dipendenze aciclico



L'ordinamento parziale è: A < A1, A < B, B < B1 < B11, B < B2.
Un possibile ordinamento totale: A < A1 < B < B2 < B1 < B11.

Una volta definito l'ordine di integrazione, si aggiungono le classi incrementalmente, esercitandone le interazioni. I possibili problemi che si incontrano, sono dovuti al fatto che l'ereditarietà implica problemi in caso di modifiche di superclassi e il polimorfismo comporta problemi legati al binding dinamico.

Testing di integrazione basato su Thread

La generazione dei casi di test può essere effettuata a partire dai *diagrammi di interazione* (specifiche). E' opportuno costruire threads anche dal codice e verificare la corrispondenza con le specifiche. I problemi sono sempre dovuti a ereditarietà, polimorfismo e binding dinamico.
I diagrammi di interazione indicano possibili sequenze di messaggi e dovrebbero indicare i casi frequenti e quelli particolari. Si può effettuare una selezione immediata, secondo cui bisogna generare un test per ogni diagramma di interazione, oppure una selezione più accurata, per la quale, per ogni diagramma, si devono individuare le possibili alternative e per ogni alternativa, selezionare un ulteriore insieme di casi di test.

17.2 Problemi di Integrazione

Ereditarietà

Nei linguaggi procedurali classici il codice è strutturato in procedure (che possono essere contenute in moduli). Una volta eseguito il test di una procedura di un modulo non è necessario rieseguirlo (salvo modifiche). Nei linguaggi orientati agli oggetti il codice è strutturato in classi, l'ereditarietà è una relazione fondamentale tra queste e nelle relazioni di ereditarietà alcune operazioni restano invariate nella sottoclasse, altre sono ridefinite, altre aggiunte (o eliminate).
Il problema è: possiamo "fidarci" delle proprietà ereditate? È necessario identificare le proprietà che devono essere ritestate: operazioni aggiunte, operazioni ridefinite, operazioni invariate ma influenzate dal nuovo contesto. Può, inoltre, essere necessario verificare la compatibilità del comportamento tra metodi omonimi in una relazione classe-sottoclasse (mediante riuso dei test e test specifici).

Polimorfismo e binding dinamico

Nei linguaggi procedurali classici le chiamate a procedura sono associate staticamente al codice corrispondente. Nei linguaggi orientati agli oggetti un riferimento (variabile) può denotare oggetti appartenenti a diverse classi in relazione *tipo-sottotipo (polimorfism*o), ovvero il tipo dinamico e il tipo statico dell'oggetto possono essere differenti. Inoltre, si possono avere più implementazioni di una stessa operazione e il codice effettivamente eseguito è identificato a *run-tim*e, in base alla classe di appartenenza dell'oggetto *(binding dinamic*o).
In questo scenario, il test strutturale può diventare non praticabile: Come definiamo la copertura in un'invocazione su un oggetto polimorfo? Come creiamo test per "coprire" tutte le possibili chiamate di un metodo in presenza di binding dinamico? Come gestiamo i parametri polimorfi?
Anche il test esaustivo può diventare impraticabile, a causa dell'esplosione combinatoria dei possibili casi. Abbiamo, inoltre, bisogno di definire un criterio di selezione specifico: selezione casuale o basata sull'analisi del codice (individuazioni di combinazioni critiche, analisi del flusso di dati, …).

Genericità

Le classi parametriche devono essere istanziate per poter essere testate. Che ipotesi possiamo e dobbiamo fare sui parametri? Sicuramente, servono classi "fidate" da utilizzare come parametri (un tipo di *stub* particolare). E, quale metodo dobbiamo seguire quando facciamo il test di un componente generico che riusiamo?

Gestione delle eccezioni

Le eccezioni modificano il flusso di controllo senza la presenza di un esplicito costrutto di tipo *test and branch*. Ci sono problemi nel calcolare gli indici di copertura della parte di codice relativa alle eccezioni:

- *Copertura ottimale*: sollevare tutte le possibili eccezioni in tutti i punti del codice in cui è possibile farlo (può non essere praticabile).
- *Copertura minima*: sollevare almeno una volta ogni eccezione.

Concorrenza

Il problema principale è il non-determinismo: risultati non-deterministici o esecuzione non-deterministica. Inoltre, i casi di test composti da valori di *input* e *output* sono poco significativi.
Si potrebbero usare, allora, casi di test composti da valori di *input*/*output* e da una sequenza di eventi di sincronizzazione, occorre, però, forzare lo scheduler a seguire una data sequenza.

19. SOFTWARE CONFIGURATION MANAGEMENT

19.1 Overview

Nella maggioranza dei casi, molte persone devono lavorare su software in evoluzione e devono essere di supporto a più di una sua versione: sistemi rilasciati, sistemi configurati per i clienti (differenti funzionalità), sistemi in sviluppo. Inoltre, questi software devono essere eseguibili su differenti macchine e sistemi operativi. In questo scenario, c'è bisogno di coordinazione: nasce, così, il **Software Configuration Management**, per gestire l'evoluzione dei sistemi software e controllare i costi coinvolti nei cambiamenti da apportare ai sistemi.
Per definizione, si tratta di un insieme di discipline di gestione all'interno del processo di ingegneria del software, per sviluppare una *baseline* (verrà introdotta più avanti).
Il Software Configuration Management include le discipline e le tecniche di inizializzazione, valutazione e controllo dei cambiamenti da apportare al prodotto software durante e dopo il processo di ingegneria del software.
Gli standard approvati dall'ANSI e che trattano l'argomento, sono:

- IEEE 828: *Software Configuration Management Plans*
- IEEE 1042: *Guide to Software Configuration Management*

Il Software Configuration Management è una funzione del progetto (come definito nel software project management plan) con l'obiettivo di rendere le attività tecniche e manageriali più efficaci. Può essere gestito in diversi modi:

- un singolo team esegue tutte le attività di gestione per l'intera organizzazione;
- un *configuration management team* separato, viene organizzato per ogni progetto;
- tutte le attività di gestione sono eseguite dagli sviluppatori stessi;
- un misto delle precedenti.

In generale, le attività di Configuration Management sono

- *Configuration item identification*: modellazione del sistema come un insieme di componenti in evoluzione.
- *Promotion management*: creazione di versioni per altri sviluppatori.
- *Release management*: creazione di versioni per i clienti e gli utenti.
- *Change management*: gestione, approvazione e annotazione delle richieste di cambiamenti.
- *Branch management*: gestione degli sforzi di sviluppo concorrenti.
- *Variant management*: gestione di versioni intese a coesistere.

Non ci sono, inoltre, regole fissate: le attività sono solitamente eseguite in modi differenti (formalmente, informalmente) e a seconda del tipo di progetto e della fase del ciclo di vita (ricerca, sviluppo, mantenimento).

19.2 Terminologia

Definiamo, ora, alcuni termini che ci serviranno per comprendere il processo di gestione: configuration item, baseline, SCM directories, version, revision, release. La definizione di questi termini segue gli standard IEEE, e chiaramente, differenti sistemi di gestione della configurazione possono usare termini differenti.

Configuration Item

"Un'aggregazione di hardware, software, o entrambi, che è progettata per il configuration management e che è trattata come una singola entità nel processo di configuration management". Gli item di configurazione del software non sono solo segmenti di codice, ma tutti i tipi di documenti riguardanti lo sviluppo, ad esempio: tutti i tipi di file codice, i driver per i test, i documenti di analisi o design, i manuali utente o di sviluppo, le configurazioni del sistema.

In alcuni sistemi, esistono configuration item non solo software, ma anche hardware (le CPU, le frequenze di bus, …)! Ogni prodotto commerciale usato nel sistema può essere un configuration item. Vediamo, ora, come trovarli.

I grandi progetti tipicamente producono migliaia di entità (file, documenti, dati, …) che devono essere univocamente identificati. Ogni entità gestita nel processo di ingegneria del software può potenzialmente essere portata sotto il controllo del configuration management. Ma non tutte le entità necessitano di essere sotto tale controllo per tutto il tempo.

Abbiamo, quindi, due problemi: selezionare i configuration item (*Cosa* dovrebbe stare sotto il controllo della configurazione?) e decidere *quando* iniziare a mettere le entità sotto il controllo della configurazione.

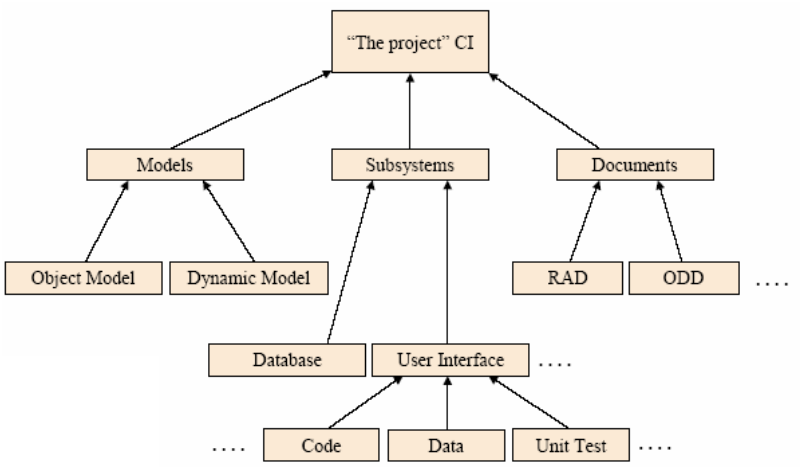
I problemi del project manager sono dovuti al fatto che iniziare con i configuration item troppo presto introduce troppa burocrazia, ma iniziare troppo tardi introduce il caos. Inoltre, alcuni item devono essere mantenuti per tutto il ciclo di vita del software.

Potrebbe essere definito un *entity naming scheme* in modo che i documenti correlati abbiano nomi correlati. Selezionare i giusti configuration item, comunque, è un compito che richiede molta pratica: è molto simile alla modellazione a oggetti. Usiamo, allora, tecniche simili alla modellazione a oggetti per trovare i configuration item e le relazioni tra questi.

Esempio di una possibile selezione di configuration item:

- Problem Statement
- Software Project Management Plan (SPMP)
- Requirements Analysis Document (RAD)
- System Design Document (SDD)
- Project Agreement
- Object Design Document (ODD)
- Dynamic Model
- Object model
- Functional Model
- Unit tests
- Integration test strategy
- Source code
- API Specification
- Input data and data bases
- Test plan
- Test data
- Support software (part of the product)
- Support software (not part of the product)
- User manual
- Administrator manual

Una volta che gli item sono stati selezionati, di solito vengono organizzati in strutture ad albero:



Baseline

"Una specifica o un prodotto che è stato formalmente revisionato e concordato dai responsabili del management, e che in seguito serve da base per i successivi sviluppi. Può esse cambiata solo mediante formali procedure di cambiamento".

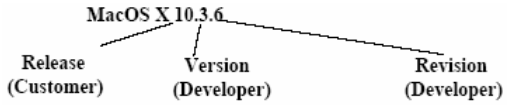
Esempi:

Baseline A: tutte le API sono state completamente definite; i body dei metodi sono vuoti.
Baseline B: tutti i metodi di accesso ai dati sono implementati e testati.
Baseline C: le GUI sono implementate.

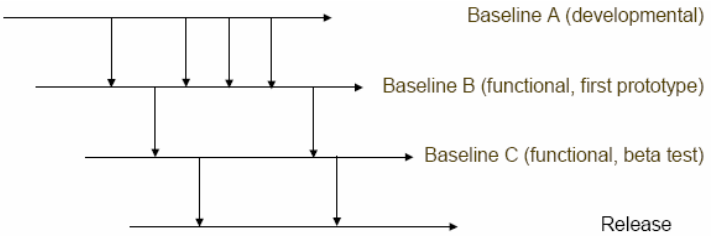
Insieme allo sviluppo dei sistemi, vengono sviluppate una serie di baseline, di solito dopo una revisione (revisioni di analisi, di design, del codice, testing del sistema, accettazione, …):

- *Development baseline*: l'obiettivo è coordinare le attività di ingegneria, dove i configuration item sono il RAD, l'SDD, il test di integrazione, …
- *Functional baseline*: l'obiettivo è fornire una prima esperienza ai clienti sul sistema funzionale, dove i configuration item sono il primo prototipo, la release alpha, la release beta.
- *Product baseline*: l'obiettivo è coordinare le vendite e il supporto ai clienti, dove il configuration item è il prodotto.

Esistono molti schemi di nomi per le baseline (1.0, 3.14159, 6.01a, …). E' molto comune uno schema a tre cifre:



Il problema è come gestire i cambiamenti nelle baseline:



Una richiesta di cambiamento conduce alla creazione di una nuova release. Il processo generale di cambiamento è il seguente:

- il cambiamento viene richiesto (da chiunque, incluso utenti e sviluppatori);
- la richiesta di cambiamento è valutata in base agli obiettivi del progetto;

- in seguito alla valutazione, il cambiamento è accettato o rifiutato;
- se è accettato, il cambiamento è assegnato ad uno sviluppatore ed implementato;
- il cambiamento implementato viene controllato.

La complessità di questo processo varia a seconda del progetto. Piccoli progetti possono eseguire richieste di cambiamento informalmente e velocemente, mentre progetti complessi richiedono forme di richieste di cambiamento dettagliate e l'approvazione ufficiale da parte di uno o più manager.

Esistono due modalità per controllare i cambiamenti:
- **Promotion**: è cambiato lo stato di sviluppo interno di un software.
- **Release**: un cambiamento al sistema software è reso visibile all'esterno dell'organizzazione.

Le politiche di cambiamento possono essere informali (buone per promotion e ambienti di ricerca) o formali (buone per configuration item sviluppati esternamente e per release).

SCM Directories

Le directory usate per il Software Configuration Management sono: le directory del programmatore, le master directory e le software repository.
La **Directory del programmatore** (IEEE: Dynamic Library) è una libreria che mantiene le entità software create o modificate. Lo spazio di lavoro del programmatore è controllato solo dal programmatore stesso.
Una **Master Directory** (IEEE: Controlled Library) è una directory centrale che mantiene tutte le promotion. Gestisce le baseline correnti e controlla i cambiamenti apportati a queste. Le entrate sono controllate, di solito con verifiche e i cambiamenti devono essere autorizzati.
Un **Software Repository** (IEEE: Static Library) è un archivio che contiene le varie baseline (esternamente) rilasciate per usi generali (e quindi contiene le release). Le copie di queste baseline possono essere rese disponibili a organizzazioni che lo richiedono.

Version

Il rilascio o il ri-rilascio iniziale di un configuration item associato a una compilazione completa o a una ri-compilazione dell'item. Differenti versioni hanno differenti funzionalità.
Una Version rappresenta lo stato di un configuration item o di un aggregato di questi in un determinato istante di tempo.

Revision

Cambiamento di una versione che corregge solo errori nel design/codice, ma non influenza le funzionalità.

Release

Una versione che è stata resa disponibile esternamente, la distribuzione formale di una versione approvata.

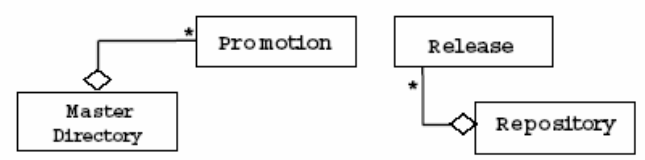
Politiche di Cambiamento

Ogni volta che viene realizzata una promotion o una release, una o più politiche vengono applicate. Lo scopo di queste politiche è quello di garantire che ogni versione, revisione o release sia conforme ai criteri comunemente accettati.

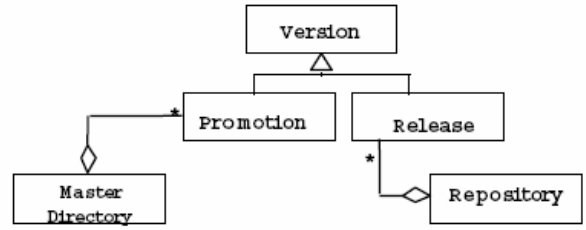
Esempi di politiche di cambiamento: "A nessuno sviluppatore è permesso di promuovere codice sorgente che non è stato compilato senza errori e warning", "Nessuna baseline può essere rilasciata senza essere stata beta-testata da almeno 500 persone esterne".

19.3 Creazione di un Modello per il Configuration Management

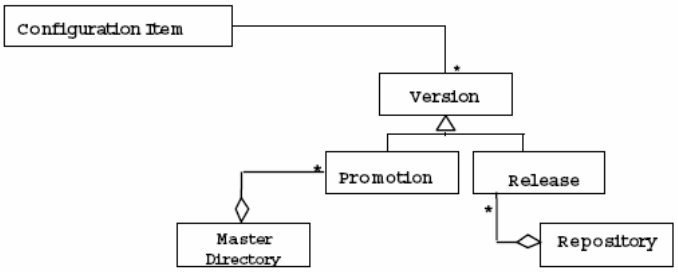
Abbiamo appena visto che le promotion sono memorizzate nella master directory e che le release sono memorizzate nel repository. Il problema è che possono esserci molte promotion e molte release. La soluzione, è l'uso della molteplicità:



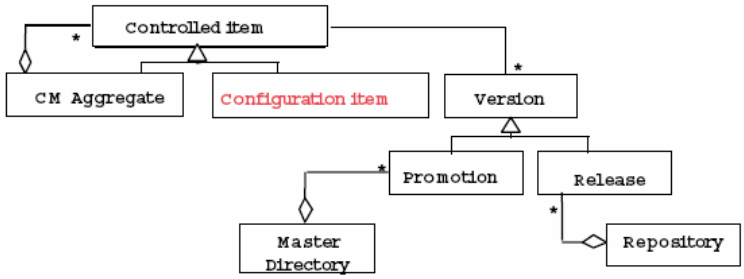
Inoltre, osserviamo che promotion e release sono entrambe delle versioni: usiamo l'ereditarietà.



Il problema è che un configuration item ha molte versioni. Creiamo allora un'associazione 1-a-molti tra il Configuration Item e la Versione:



Osserviamo, che i configuration item possono essere raggruppati: usiamo un Composite Design Pattern.



Quest'ultimo diagramma è il nostro **Configuration Item Model**, che è un Class Diagram UML.

19.4 Pianificazione del Software Configuration Management

Questa pianificazione inizia durante le fase iniziali di un progetto e dà luogo al **Software Configuration Management Plan** (**SCMP**) che potrebbe essere esteso o rivisitato durante il resto del progetto. L'SCMP può seguire sia uno standard pubblico come l'IEEE 828 che uno standard interno (ad una compagnia specifica). Questo documento, in generale, definisce:

- i tipi di documenti da gestire e un documento contenente il naming scheme;
- chi mantiene la responsabilità delle procedure di Configuration Management e di creazione delle baseline;
- le politiche di controllo dei cambiamenti e la gestione delle versioni;
- i tool che dovrebbero essere usati per assistere il processo di configuration management e ogni limitazione sul loro utilizzo;
- il database di configuration management usato per registrare le informazioni di configurazione.

Struttura di un Software Configuration Management Plan (IEEE 828-1990)

1. *Introduzione*: descrive gli scopi, il fine dell'applicazione, termini chiave e riferimenti.
2. *Management* (CHI?): identifica le responsabilità e le autorità per la realizzazione delle attività di configuration management pianificate.
3. *Attività* (COSA?): identifica le attività da eseguire nell'attuazione del progetto.
4. *Schedule* (QUANDO?): stabilisce la sequenza e la coordinazione delle attività del SCM con milestone di progetto.
5. *Risorse* (COME?): identifica i tool e le tecniche richieste per l'implementazione dell'SCMP.
6. *Maintenance*: identifica le attività e le responsabilità sulla base delle quali il documento dovrebbe essere tenuto durante il ciclo di vita del progetto.

La *Sezione 1* (Introduzione), si dirama nelle seguenti:

1.1 Panoramica semplificata delle attività di configuration management.
1.2 Scope: panoramica descrittiva del progetto e identificazione dei configuration item a cui il software configuration management sarà applicato.
1.3 Identificazione di altri software da includere come parti dell'SCMP (software di supporto e software di test).
1.4 Relazioni di software configuration management sull'hardware delle attività di system configuration management.
1.5 Livello di formalità e profondità del controllo per applicare il software configuration management al progetto.
1.6 Limitazioni e vincoli temporali sull'applicazione del software configuration management al progetto.
1.7 Assunzioni che dovrebbero avere impatto su costo, schedule e capacità di eseguire le attività di software configuration management definite.

La *Sezione 2* (Management), si dirama nelle seguenti:

2.1 Organizzazione: contesto dell'organizzazione (tecnico e manageriale) all'interno del quale le attività di software configuration management vengono implementate. Identifica tutte le unità dell'organizzazione (clienti, sviluppatori, manager) che partecipano in un'attività di software

configuration management, i ruoli funzionali di queste persone all'interno del progetto e le relazioni tra le unità.

2.2 Responsabilità: per ogni attività di software configuration management elenca il nome o il titolo del lavoro da eseguire, e per ogni attività elenca scopi e obiettivi, membri e affiliati, periodo di efficacia e fini di autorità, procedure operazionali.

2.3 Politiche applicabili: vincoli esterni posti sul software configuration management.

La *Sezione 3* (Attività), si dirama nelle seguenti:

3.1 Configuration identification.

3.2 Configuration control.

3.2.1 Come identificare la necessità di un cambiamento (il layout di un form di una richiesta di cambiamento).

3.2.2 Analisi e valutazione di una richiesta di cambiamento.

3.2.3 Approvazione o disapprovazione di una richiesta..

3.2.4 Verifica, implementazione e rilascio di un cambiamento.

3.3 Configuration status accountig.

3.4 Configuration audits e review.

In particolare:

- La *Sezione 3.2.1* (Richiesta di Cambiamento), specifica le procedure per richiedere un cambiamento alla baseline di un configuration item e le informazioni che devono essere documentate: nomi e versioni dei configuration item dove appare il problema, nome e indirizzo del richiedente, data della richiesta, indicazione sull'urgenza, le necessità per il cambiamento, la descrizione della richiesta di cambiamento.
- La *Sezione 3.2.2* (Valutazione di un cambiamento), specifica l'analisi richiesta per determinare l'impatto dovuto al cambiamento proposto e specifica la procedura per revisionare i risultati dell'analisi.
- La *Sezione 3.2.3* (Approvazione o Disapprovazione del Cambiamento), descrive l'organizzazione del **Configuration Control Board** (**CCB**), che può essere un individuo o un gruppo. Sono possibili anche livelli multipli di CCB, a seconda della complessità del progetto. In tal caso deve essere specificato (se lo sforzo di sviluppo è modesto, basta un solo livello). Questa sezione indica anche il livello di autorità del CCB e le sue responsabilità, in particolare, si deve specificare quando il CCB viene invocato.
- La *Sezione 3.2.4* (Implementazione dei cambiamenti), specifica le attività per verificare e implementare un cambiamento approvato. Una richiesta di cambiamento completa deve contenere le originali richieste di cambiamento, i nome e le versioni dei configuration item coinvolti, la data di verifica e le parti responsabili, l'identificatore della nuova versione, la data di rilascio o di installazione e le parti responsabili. In questa sezione devono essere specificate anche le attività per archiviare le richieste di cambiamento completate, per pianificare e controllare le release, per coordinare cambiamenti multipli, per aggiungere nuovi configuration item alla configurazione, per individuare una nuova baseline.

Inoltre:

- La *Sezione 3.3* (Configuration Status Accounting), deve contenere le sezioni rispondenti alle seguenti domande: Di quanti elementi bisogna tenere traccia e riportare per le baseline e i cambiamenti? Quanti tipi di accounting report stanno per essere generati? Con che frequenza? Come è l'informazione da collezionare, memorizzare e riportare? Com'è controllato l'accesso alle informazioni sullo stato del configuration management?
- La *Sezione 3.4* (Configuration Audits e Review): identifica i controlli (audit) e le revisioni del progetto (un **audit** determina per ogni configuration item, se ha un richiesta caratteristica fisica e funzionale, mentre una **review** è uno strumento di gestione per stabilire una baseline). Per

ogni audit o review il piano deve definire obiettivi, configuration item sotto review, schedule per la review, procedure per condurre una review, partecipanti al lavoro, documentazione richiesta, procedure per registrare deficienze e per come correggerle, criteri di approvazione.

Ruoli di configuration management

Il **Configuration Manager** è responsabile di identificare i configuration item. Può anche essere responsabile della definizione delle procedure per creare promotion e release. In generale, i suoi task sono: definire i configuration item, definire politiche per promote/release, definire le attività e le responsabilità, impostare il sistema di configuration management.
Il **Change control board member**, invece, è responsabile di approvare o rigettare richieste di cambiamento.
Il **Developer** crea promotion introdotte da richieste di cambiamento o da normali attività di sviluppo, e controlla i cambiamenti e risolve i conflitti.
L'**Auditor**, infine, è il responsabile della selezione e della valutazione di promotion per il rilascio ed è responsabile di assicurare la consistenza e la completezza di questa release.

Preparare l'SCMP

Lo standard IEEE consente una certa flessibilità nella preparazione (*tailoring*) di un SCMP. Questo può essere:

- *tailored upward*: per aggiungere informazioni e usare uno specifico formato;
- *tailored downward*: alcuni componenti dell'SCMP potrebbero non essere applicabili a particolari progetti, e invece di omettere le sezioni associate, menziona le loro applicabilità; le informazioni che non sono state decise nel momento in cui l'SCMP è stato approvato potrebbero essere considerate come "da determinare".

In generale lo standard può essere confezionato per un particolare progetto: progetti grandi necessitano di piani dettagliati per avere successo e progetti piccoli non dovrebbero essere esasperati dalla burocrazia di piani dettagliati.

Conformità allo Standard IEEE 828-1990

- *Formato di presentazione e informazioni minime*: un documento separato e una sezione unificata in un altro documento intitolato "Software Configuration Management Plan". Ha 6 sezioni: introduzione, management, attività, schedule, risorse e piano di mantenimento.
- *Criteri di consistenza*: tutte le attività definite nell'SMCP (dalla sezione 3.1 alla 3.6) sono assegnate a un'unità o a persone dell'organizzazione e sono associate alle risorse per portare a termine le attività. Tutti i configuration item definiti nella sezione 2.1 definiscono processi per stabilire baseline e controlli di cambiamento (sezione 3.2).

Se vengono rispettati questi criteri, l'SCMP può includere la seguente frase: "*Questo SCMP è conforme ai requisiti dello Std IEEE 828-1990*".
Da notare che i criteri di consistenza possono essere anche usati in un meeting di revisione dell'SCMP.

Tool per il Software Configuration Management

Il Software Configuration Management può essere supportato da tool, che variano da quelli semplici, per la memorizzazione delle versioni, a sistemi sofisticati con procedure automatizzate per politiche di controllo e supporto alla creazione dei documenti di management. Ad esempio:

- *RCS*: datato ma ancora in uso, ha solo una versione di controllo del sistema.

- *CVS* (*Concurrent Version Control*): basato su RCS, permette di lavorare concorrentemente senza locking. E' possibile reperirlo su http://www.cvshome.org/.
- *CVSWeb*: è il frontend web al CVS:
- *Perforce*: è un repository server che tiene traccia delle attività degli sviluppatori. E' reperibile su http://www.perforce.com.
- *ClearCase*: server multipli, processi di modellazione, meccanismi di politiche di controllo. E' reperibile su http://www.rational.com/products/clearcase/.