Ingegneria del software

Ahmad Shatti

2020-2021

Indice

Chapter 1

Introduzione

1.1 Ingegneria del software

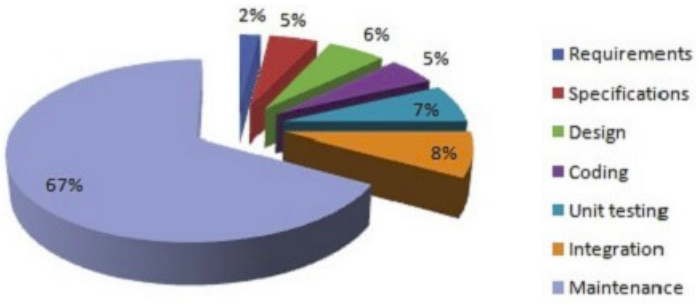
Ingegneria del software è una disciplina che ha lo scopo di produrre *fault-free* software, cioè software corretti, ma deve tenere conto anche degli aspetti economici ossia il software deve essere consegnato nei tempi previsti e che rispetti il budget iniziale. Infine deve soddisfare anche la necessità del committente ed deve essere facile da modificare.

1.2 Processo software

Con processo software si indica il percorso da svolgere per ottenere risultati di alta qualità e in tempi prefissati nello sviluppo di un prodotto. Incomincia da quando si inizia ad esplorare il problema e finisce quando il prodotto viene ritirato dal mercato. Le fasi del processo sofware sono:

- **Analisi dei requisiti**, cioè cercare di capire cosa ci viene richiesto, le funzionalità che il sistema deve offrire per garantire che il sistema risponda alle esigenze del committente. Se si introduce un errore durante l'analisi dei requisiti, l'errore apparirà anche nella specifica, nella progettazione e nel codice con un aumento considerevole dei costi;
- **Specifica**, cioè definire le funzionalità in modo più formale ma sempre ad un alto livello di astrazione;
- **Progettazione** (o design), cioè mettere insieme le varie parti;
- **Implementazione**, cioè la parte di coding, la traduzione da modello a codice;
- **Integrazione** dei vari moduli;
- **Mantenimento**, cioè una volta consegnato il sistema software al committente inizia il mantenimento che è la fase più importante, più costosa e più lunga. Si divide in *manutenzione correttiva* dove si rimuovono gli errori lasciando invariata la specifica e *manutenzione migliorativa* che consiste in cambiamenti nella specifica e nella implementazione, e può essere:
  - *perfettiva* (60%): modifiche per migliorare la qualità software;
  - *adattiva* (20%): modifiche a seguito di cambiamenti nell'ambiente hadware, nel sistema operativo, ecc...
- **Ritiro**, cioè quando viene ritirato dal mercato.

Il processo software include anche gli strumenti e le tecniche per svilupparlo e i professionisti del software coinvolti.



1.3 Progetti falliti

I problemi non sono sempre dovuti al software difettoso. Alcuni esempi tristemente famosi:

- *Aereoporto di Denver*, è stato progettato nel 1995 e chiuso nel 2005, riguardava un sistema di smistamento bagagli con un investimento di $193 000 000. L'inaugurazione dell'aereoporto è stata ritardata di 16 mesi con un costo pari a 1 milione di dollari per ogni giorno di ritardo con conseguente sforamento di 3,2 miliardi di dollari rispetto ai preventivi. La progettazione era difettosa perché i carrelli con all'interno le valigie spesso si scontravano e fatto più importante non vi era **fault tolerance** cioè il sistema non si riprendeva dopo un guasto, come il salto della corrente, e perdeva traccia di quali carrelli fossero pieni e quali vuoti;
- *Il caso Therac*, era un apparecchio medico che serviva per mandare delle radiazioni che ha causato il decesso di 3 persone per sovradosaggio. Il problema era causato da un editing troppo veloce dell'operatore e mancanza di controlli sui valori immessi. Il sistema era caratterizzato da **poca robustezza** e da **difetto latente**, cioè il problema si presentava solo in certe condizioni;
- *Il sistema antimissili Patriot*, per un difetto nel sistema di tracciamento missili una caserma in Arabia Saudita è stata colpita provocando la morte di 28 soldati americani. Il sistema era stato concepito per funzionare ininterrottamente per un massimo di 14h ma fu usato per 100h causando dei problemi nell'orologio interno del sistema a tal punto da rendere inservibile il sistema di tracciamento dei missili da abbattere. Anche in questo sistema è presente la poca robustezza;
- *London ambulance service*, era un sistema semiautomatico di gestione del sevizio ambulanze con ottimizzazione dei percorsi. Il sistema era progettato in modo che la prima ambulanza che arrivava sul luogo del sinistro, doveva avvisare le altre ambulanze collegandosi al sistema. Questa procedura non veniva quasi mai effettuata per la gravità dell'incidente, facendo in modo che il sistema, per via che nessun mezzo aveva risposto alla chiamata, mandasse un'altra ambulanza sullo stesso posto. Il sistema fu abbandonato dopo 3 giorni d'uso con un costo totale di €11 000 000. I problemi erano diversi come interfaccia utente inadeguata, poco addestramento utenti, nessuna procedura di backup e **scarsa verifica del sistema**;
- *Ariane 5*, il sistema, progettato per l'Ariane 4, tenta di convertire la velocità laterale del missile dal formato a 64 bit al formato a 16 bit, ma l'Ariane 5 vola molto più velocemente dell'Ariane 4, per cui il valore della velocità laterale è più elevato di quanto possa essere gestito dalla routine di conversione e esplode dopo 41 secondi dal lancio. Il problema è stato causato dall'overflow perché il sistema è stato **testato con dati vecchi**;
- *Il caso Toyota* è stato il primo caso dove fu condannata una azienda automobilistica per cattiva pratica di software engineering. L'acceleratore della macchina era composto da un sensore, collegato a un sistema software, che decideva quanto accelerare in base a quanto si premeva sul pedale. Quando si tentava di accelerare e in seguito frenare, la macchina prendeva ancora più velocità causando diversi incidenti.

Un esempio di successo è la *metropolitana di Parigi*: la linea 14 della metropolitana è stata la prima integralmente automatizzata ed è stata testata con metodi statici

1.4 Aspetti storici

Negli anni '60 i software venivano sviluppati prevalentemente per svolgere dei conti matematici, in seguito arrivano i grandi sistemi commerciali e con queste anche le piccole aziende incominciano ad usare i computer per gestire tutte le informazioni aziendali. Dalla programmazione individuale si passa alla programmazione di squadra. Dopo qualche anno ci si accorge che lo sviluppo software senza seguire delle indicazioni generali porta a software scadenti e diversi fallimenti di progetti, allora nel 1968 un gruppo di scienziati della NATO, nella conferenza di Garmish, istituiscono delle buone pratiche di progettazione. La produzione di software deve usare tecniche e paradigmi consolidate. Da delle analisi dello Standish Group nel 1994 indica che i progetti software completati in tempo erano solo il 16,2%, mentre quelli in ritardo erano 52,7% per difficoltà nelle fasi iniziali del progetto, cambi di piattaforma e tecnologia e difetti nel prodotto finale. I progetti abbandonati erano 31,1% per requisiti incompleti, scarso coinvolgimento degli utenti, incapacità di raggiungere gli obiettivi e per esaurimento fondi.

1.5 Specificità del software

Il software è diverso da altri prodotti di ingegneria perché non è vincolato da materiali, nè governato da leggi fisiche. Il software si "sviluppa" non si "fabbrica" nel senso tradizionale, non si "consuma" ma si "deteriora" cioè non è

più adatto a risolvere un certo problema, e infine spesso si "assembla" ma molte volte si realizza *ad hoc*. Un'altra differenza quando un edificio crolla parzialmente, si demolisce, e si costruisce da capo, quando un sistema operativo *crasha* lo si fa ripartire e inoltre il sistema operativo è progettato per minimizzare l'effetto del fallimento, cioè non si perdono i documenti su cui si stava lavorando (fault tolerance). La manutenzione di un edificio in genere si restringe a ripitturarlo o sistemare le crepe, mentre in un sistema software può invece essere modificato per passare ad una nuova macchina con caratteristiche hardware completamente diverse. Anche negli aspetti economici vi sono delle differenze: quando una ditta software scopre una nuova tecnica che permetterebbe di velocizzare la scrittura del codice non è detto che venga subito adattata per via del costo dell'introduzione della tecnologia, il costo del training del personale e il costo della manutenzione.

1.6 Lavoro in team

La maggior parte del software è oggi prodotto da team di programmatori. Il lavoro in team pone dei problemi di comunicazione tra i membri del gruppo e molto tempo deve essere dedicato alle riunioni. L'ingegnere del software deve essere capace di gestire i rapporti umani e organizzare un team, e amministrare gli aspetti economici e legali.

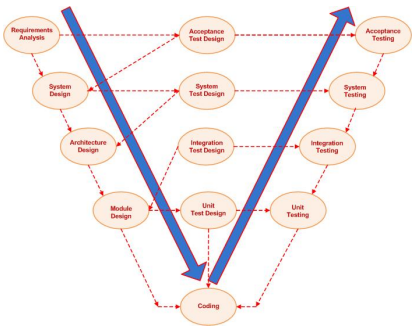
1.7 Modelli di cicli di vita del software

Modellare il processo software significa suddividerlo in attività, ordinare le attività, e per ogni attività chiedersi *cosa fa l'attività? cosa deve produrre l'attività? quando si passa da un'attività ad un'altra?*
Prima di iniziare lo sviluppo di un progetto si deve scegliere un modello e ve ne sono diversi:

- **Build-and-Fix: un non modello**
  Il prodotto è sviluppato senza specifica e senza un tentativo di progettazione. Lo sviluppatore scrive un programma e lo modifica più volte finché non soddisfa il committente. Questo metodo veniva usato nei primi programmi matematici per aiutare la computazione, cioè erano programmi costruiti ad hoc. Build-and-Fix viene utilizzato per progetti con poche righe di codice perché è improponibile per prodotti di grandi dimensioni a causa che i progetti grandi sono progettati in gruppo, cioè hanno una specifica condivisa. Lo sviluppatore deve programmare in modo redditizio cioè modificare il codice tutte le volte che non è funzionante e riscriverlo senza una struttura richiede tempi di sviluppo più lunghi. Un altro svantaggio è che la manutenzione di un prodotto senza specifica o documentazione che ne spieghi la progettazione è estremamente difficile;

- **Modello a cascata**
  Si decide di strutturare il progetto con le diverse fasi (analisi dei requisiti, specifica, ...) e al fine di ogni fase vi è uno stadio di verifica. Nel passaggio da una fase alla successiva viene prodotto un documento che relazioni tutto quello che è stato fatto nella fase. Questo documento deve essere studiato e approvato dal gruppo SQA (Software Quality Assurance Group) che è un gruppo esterno a quello degli sviluppatori, affinché si possa passare alla fase successiva. Nella fase di specifica viene chiesto l'approvazione del committente e prima della fase di design viene fatto un piano dei tempi e dei costi da far approvare al cliente. Il punto debole è una eccessiva produzione di documenti e manca l'interazione col cliente che dall'approvazione della specifica alla consegna del prodotto passa molto tempo è quindi ci può essere una sostanziale differenza tra come il cliente immagina il prodotto (descritto dal documento) e come il prodotto sarà alla fine;

- **Modello a V**
  É un processo sequenziale in cui la rappresentazione del modello invece di discendere lungo una linea retta, dopo la fase di coding risale a forma della lettera V. Durante l'esecuzione di ogni fase si preparano i test, prima della fase di codifica, che verranno controllati una volta scritto il codice. Il fatto di pianificare i test già all'inizio del progetto permette di abbattere i tempi. Le frecce blu rappresentano il tempo, quelle tratteggiate le dipendenze causali;

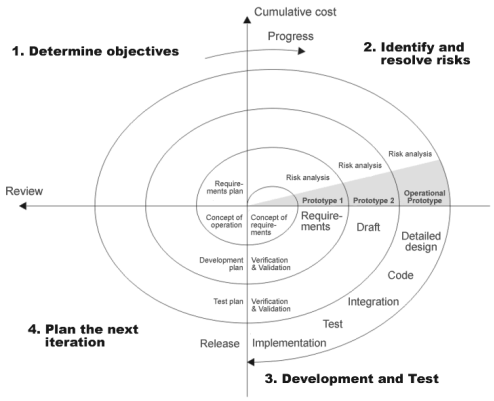


- **Rapid prototyping**
  É utile quando i requisiti del cliente non sono chiari. Si sostituisce la prima fase del modello a cascata, cioè l'analisi dei requisiti, con la costruzione di un prototipo per permettere al committente di sperimentarlo. In questo modo il cliente non approva semplicemente un documento ma un prodotto più fedele a quello che sarà

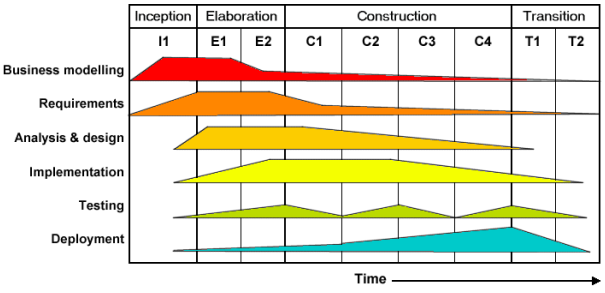
il risultato finale. Una volta che il committente ha approvato il prototipo si passa alla fase di specifica. Questo processo evita di ritornare alla fase dei requisiti tutte le volte che c'è qualcosa che non soddisfi il cliente. Il prototipo può essere *usa e getta*, o si continua a svilupparlo fino al raggiungimento dell'obiettivo.

I modelli a cascata, a V e rapid prototyping sono poco iterativi, cioè tornano alle fasi precedenti solo se esistono degli errori. Nell'età moderna si usano più i modelli iterativi, cioè si può ripartire dall'inizio e si rifanno tutte le fasi:

- **Modello incrementale**
  I requisiti e il progetto sono definiti inizialmente, poi il prodotto viene integrato e testato aggiungendo passo dopo passo funzionalità più dettagliate. I passaggi sono incrementali perché ogni volta che si inserisce una funzionalità si devono ricontrollare tutte le fasi. Sono molti i vantaggi come il fatto che ad ogni nuova funzionalità il cliente si rende subito conto di come sta andando il progetto e può chiedere una modifica, o che la manutenzione diventa un passaggio come gli altri. Se non è progettato bene, cioè per introdurre una nuova funzionalità, si devono modificare le vecchie, allora il modello diventa un Build-and-Fix;
- **Modello a spirale**
  Questo modello è ispirato all'economia, ogni iterazione è organizzata in 4 fasi: si analizzano i rischi cioè il costo del progetto e quanto è alto il rischio di fallimento; sviluppo e validazione del prototipo; la pianificazione del nuovo ciclo e infine si controlla se gli obiettivi sono stati raggiunti. Prima di passare all'iterazione successiva, cioè quella in cui si possono aggiungere funzionalità si riparte dall'analisi del rischio. Il modello a spirale ha il vantaggio che è molto attento al lato economico perché è basato sull'analisi del rischio e prevede maggior comunicazione e confronto con il committente;



- **Unified process**
  Si vuole cercare di eliminare i documenti, e mettere al centro i casi d'uso, cioè si cerca di capire le funzionalità del sistema, e l'analisi dei rischi. Unified process è un modello iterativo incrementale e consiste in una serie di cicli, ogni ciclo ha diverse iterazioni e si concludo con una release del sistema. Le iterazioni sono raggruppate in 4 fasi, ogni fase finisce con la decisione del manager di continuare o terminare il progetto. Le fasi si differenziano per importanza, cioè nella fase di elaborazione, i requisiti sono particolarmente importanti mentre nella fase di costruzione non sono importanti i requisiti ma l'implementazione. Le fasi sono *Avvio*: analisi dei rischi, fattibilità, eventuale prototipo; *Elaborazione*: analisi dei requisiti e dei rischi, sviluppo di un'architettura di base; *Costruzione*: analisi, implementazione, testing; *Transizione*: beta testing, aggiustamento delle prestazioni, attività di formazione.



- **Metodo agile**
  Si intende un particolare metodo per lo sviluppo software che coinvolge quanto più possibile il committente. Adatti a progetti con meno di 50 persone. Una metodologia agile si basa sui principi del **Manifesto di Snowbird**:

  – *Comunicazione*, le persone e le interazioni sono più importanti di processi e strumenti. Tutti possono parlare con tutti, persino l'ultimo dei programmatori può parlare con il cliente. La collaborazione diretta offre risultati migliori dei rapporti contrattuali;
  – *Semplicità*, descrizione formale il più semplice e chiara possibile, è più importante avere software funzionante che documentazione. Il codice deve essere semplice e avanzato tecnicamente riducendo la documentazione al minimo indispensabile;
  – *Feedback*, rilasciare nuove versioni del software ad intervalli frequenti e sin dal primo giorno si testa il codice;
  – *Coraggio*, dare in uso il sistema il prima possibile e implementare i cambiamenti richiesti man mano.

  La metodologia agile si divide in più modelli:

- **eXtreme programming**: adatto a piccoli gruppi che devono operare con requisiti vaghi e che possono cambiare nel tempo. Si basa su due prassi, *pianificazione flessibile* basata su scenari proposti dagli utenti che coinvolge tutti i programmatori, e *rilasci frequenti* ogni due/quattro settimane. Dopo ogni rilascio si inizia una nuova pianificazione. I progetti sono semplici e comprensibili, i test si producono prima del codice, meeting giornalieri e il cliente è sempre a disposizione. Un'altra caratteristica dell'eXtreme programming è che il lavoro è effettuato a coppie dove il *driver* scrive il codice mentre il *navigatore* controlla il lavoro del compagno in maniera attiva. Non vi è il lavoro straordinario, il codice è semplice e collettivo, nel senso che qualsiasi persona può accedere al codice degli altri in modo da poterlo controllare e eventualmente modificare o integrare. Si usa molto spesso il *code refactoring*, cioè una tecnica per modificare la struttura di porzioni di codice senza modificarne il comportamento, applicata per migliorare la leggibilità, la manutenibilità, la riusabilità e la riduzione della sua complessità.
- **Scrum**: è un processo iterativo e incrementale in cui un insieme di persone si muovono all'unisono per raggiungere un obiettivo predeterminato, tale obiettivo garantisce la soddisfazione delle ambizioni di squadra e personali. Fornisce alla fine di ogni iterazione un set di funzionalità potenzialmente rilasciabili. Scrum è composto in 3 fasi:
  * *Planning*, include la definizione del sistema che deve essere sviluppato e *architecture*, viene pianificato un design di alto livello del sistema;
  * *Development*, il sistema viene sviluppato attraverso una serie di sprint, cioè cicli iterativi nei quali vengono sviluppate o migliorate una serie di funzionalità. Ciascun sprint include le tradizionali fasi di sviluppo software e si svolge in un intervallo di tempo che va da una settimana ad un mese, con team meeting quotidiani;
  * *Post-game* che contiene la chiusura definitiva della release.

  I programmatori possono avere 3 ruoli:
  * *Product owner* è il riferimento per tutti i soggetti, cliente incluso, interessati al progetto. Gestisce l'intero processo secondo la pianificazione iniziale ed è una figura di raccordo in grado di effettuare stime e aggiustare processi che presentano difetti. Il product owner ha il potere di accettare o rigettare i risultati di un lavoro e terminare uno sprint se necessario;
  * *Team member* decidono cosa fare in ciascuno sprint. I team sono indipendenti e sono formati da 7 persone. Ogni membro realizza una cosa alla volta, cioè non vi è il multitasking, ed è presente la *cross-functional* cioè ogni componente sa fare più di un compito per essere al servizio di un altro team magari carico di lavoro;
  * *Scrum master* è il motivatore, si occupa di supportare il team garantendo le condizioni ambientali e le motivazioni necessario ad eseguire al meglio il lavoro. Non ha autorità sul team.

Chapter 2

Analisi dei requisiti

2.1 Studio di fattibilità

Prima di incominciare qualsiasi operazione, si deve effettuare uno studio di fattibilità cioè una descrizione sommaria del sistema software, e in base a ciò si effettua un'analisi per capire se ha senso realizzare il prodotto, sia perché sul mercato ci possono essere già delle soluzioni e quindi non converrebbe ricreare qualcosa già esistente con il rischio di realizzarla uguale o peggio, sia per un fatto economico, ovvero chiedersi se il prodotto avrà successo. Vi deve essere anche una analisi tecnica per capire se il prodotto è realizzabile.

2.2 Attività di analisi dei requisiti

Nell'attività di analisi dei requisiti si studia e si definisce il problema da risolvere per capire perfettamente *cosa* (non come) deve essere realizzato. Si deve documentare l'analisi dei requisiti per dare un'idea del risultato di questa fase, ed è importante per ricordarsi cosa implementare, per negoziare con il committente, e deve essere di descrizione del dominio e dei requisiti:

- **Dominio** è il campo di applicazione del prodotto. Prima di incontrare il committente, il team di analisti deve approfondire le conoscenze sul dominio di applicazione per comprendere meglio le richieste del cliente e per poter porre le domande giuste. Per definire il dominio, si costruisce poco alla volta un **glossario** cioè una lista di termini tecnici con la loro spiegazione;
- **Requisito** è una condizione che deve essere soddisfatta dal prodotto per accontentare la necessità di un utente.

In questa fase si può anche produrre opzionalmente un manuale utente e casi di test che spesso sono svolti in parallelo. Per acquisire conoscenza su dominio e requisiti si effettuano delle interviste, cioè i membri del team incontrano il committente e si procede con delle domande che possono essere:

- **Strutturate**, cioè domande precise con una specie di questionario a cui i clienti devono rispondere in maniera libera. Lo svantaggio è che ci possono essere delle direzioni suggerite dal cliente a cui il team di analisti non aveva pensato;
- **Non strutturate**, cioè quando il committente parla liberamente e poi si effettuano le domande sulla base di quello che dice. Lo svantaggio è che il cliente potrebbe non essere coinciso e non dica niente di particolarmente interessante dal punto di vista informatico.

In entrambi i casi con pochi incontri è difficile condurre una buona intervista per capire e identificare le vere esigenze del committente. Oltre a fare le interviste, si può osservare come il committente lavora, o costruire dei prototipi per verificare se le richieste del committente sono state comprese. Un modo per documentare e acquisire la conoscenza sono i **casi d'uso**, cioè consiste nel valutare ogni requisito focalizzandosi sugli utenti che interagiscono con il software. I casi d'uso devono includere non solo la sequenza di eventi corretta ma anche comportamenti inattesi: le eccezioni.

2.3 Categorie di requisiti

I requisiti si dividono in due categorie:

- **Requisiti funzionali**: sono quelli più importanti e descrivono le funzionalità che il sistema deve realizzare in termini di azioni che il sistema deve compiere, come il sistema software reagisce a specifici tipi di input e come si comporta in situazioni particolari;
- **Requisiti non funzionali**: descrivono quali proprietà il sistema deve avere in relazione a determinati servizi o funzioni. Questi requisiti possono essere di qualità come l'efficienza, affidabilità, usabilità o per caratteristiche esterne come l'interazione con sistemi di altre organizzazioni. Un esempio di requisito non funzionale può essere "il tempo di risposta del sistema di inserimento della password utente deve essere inferiore a 10 sec".

A volte si possono trovare dei requisiti sia funzionali che non funzionali, ma è bene tenerli separati. Per esempio "il sistema deve validare il pin inserito dal cliente entro 3 secondi" deve essere separato in requisito funzionale: "Il sistema deve validare il pin inserito dal cliente", e da requisito non funzionale: "La validazione deve essere completata entro 3 secondi".

2.4 Documento dei requisiti

Il documento dei requisiti è in genere riservato, ed è un contratto tra lo sviluppatore e il committente. Specifica cosa il prodotto deve fare e quali sono i vincoli che deve soddisfare e in genere vi è anche un *deadline* per la consegna del progetto. Il documento dei requisiti è svolto in 5 passi:

- **Acquisizione**: interviste strutturate o non strutturate, osservare come l'utente lavora, produzione prototipi e casi d'uso;
- **Elaborazione**: i requisiti vengono espansi e raffinati. La definizione del documento dei requisiti è basato sull'uso del linguaggio naturale e sulla descrizione del dominio. La struttura del documento è di circa 30 - 70 pagine e comprende una introduzione, un glossario, la definizione dei requisiti funzionali e non funzionali, l'architettura cioè la strutturazione in sottoinsiemi a cui si riferiscono i requisiti, specifica dettagliata dei requisiti funzionali, modelli astratti del sistema, l'evoluzione del sistema cioè la previsione di successivi cambiamenti, delle appendici e infine un lemmario con la lista dei termini con i puntatori ai requisiti che li usano. Il lemmario facilita la ricerca di inconsistenze, sinonimi e omonimi, e ridondanze. Il glossario oltre a definire i termini chiave del dominio serve anche per la validazione dei requisiti, e può essere fatto in due modi:
  - *Walkthrough*: una lettura sequenziale dei documenti;
  - *Ispezione*: una lettura strutturata dei documenti partendo dal lemmario.

  I difetti da evitare sono mancata presenza di un requisito, contraddizioni, ambiguità, presenza di sinonimi e omonimi, e la ridondanza nel senso che può esserci ma solo tra sezioni diverse. Per evitare di commettere degli errori esistono delle **tecniche di Noam Chomsky** cioè sono dei processi mentali in cui si ragiona su:
  - *Generalizzazione*: è il processo attraverso cui le persone partendo da una esperienza specifica, la decontestualizzano traendone un significato universale, questo perché utilizzano quantificatori universali: sempre, tutto, mai, ... e operatori modali: devo, posso, voglio... Ricercare generalizzazioni: sul serio tutti? davvero mai?
  - *Cancellazione*: è un processo di selezione dell'esperienza. Le persone prestano attenzione solo ad alcuni pezzi del proprio vissuto escludendone altri. Ricercare cancellazioni: chi? quando?
- **Negoziazione**: una volta scritti i requisiti e controllati, vengono sottoposti al committente. I requisiti possono essere modificati e sono divisi in classi: requisiti obbligatori, requisiti desiderabili, requisiti opzionali e requisiti postponibili. Ogni requisito oltre ad appartenere a una categoria, conterrà anche:
  - *identificatore:* unico, un numero sequenziale basato sulla struttura del documento;
  - *attributi:* con stato proposto, approvato, rifiutato, incorporato. Una pianificazione degli sforzi di ogni sviluppatore, del rischio e stabilità.

  Il documento dei requisiti e la sua negoziazione precede la stipulazione del contratto.
- **Convalida**;
- **Gestione**.

Chapter 3

UML - linguaggio di modellazione

3.1 Modello

Un modello è una tecnica di astrazione che cattura aspetti importanti dell'oggetto da modellare, e semplifica o omette il resto. Ovviamente in questo modo si ha solamente una visione dell'oggetto, mentre la visione completa dell'oggetto lo si ha solo con un insieme di tutte le astrazioni. Un modello è espresso con un formalismo che rende facile usarlo e comprenderlo, ed è uno strumento di documentazione, comunicazione e discussione. Ci si pone queste domande:

- **Come si modella un sistema?**
  La prima distinzione da fare è chiedersi se si vuole fornire un modello statico o dinamico:
  - **Modello statico**, rappresenta la struttura dell'oggetto o del sistema senza descrivere come interagiscono le varie parti del sistema fra di loro;
  - **Modello dinamico**, corrisponde ad un automa a stati finiti che rappresenta il comportamento degli oggetti del sistema.

  In entrambi i casi si decide a che livello di astrazione porsi, cioè il livello dei dettagli.
- **Come si rappresenta un modello?**
  Per rappresentare un modello si ha bisogno di un linguaggio comune a tutti, semi-formale perché l'esigenza è quella di discutere anche con persone magari non esperte in informatica, e adatto a descrivere aspetti diversi del progetto, cioè è flessibile.
- **Come si usa un modello?**
  Un progetto può essere realizzato come uno **sketch** dove il modello non è completo, oppure come un **progetto dettagliato** che contiene informazioni e dettagli utili anche per gli sviluppatori, o un **eseguibile** come il linguaggio UML che è talmente completo e preciso che permette di generare il codice automaticamente dal modello.

3.2 UML

UML è un linguaggio di modellazione unificato che permette di specificare molte parti di un progetto. È una famiglia di notazioni grafiche, cioè diagrammi, che consentono di visualizzare, descrivere e discutere diversi modelli da diversi punti di vista che sono generalmente facilmente comprensibili agli utenti con un minimo di conoscenze nel campo. UML è indipendente dal linguaggio di sviluppo, dal modello di ciclo di vita ed è applicabile a più tipi di progetti e domini per modellare ogni fase del processo di sviluppo. Il vantaggio di utilizzare UML rispetto, per esempio, al documento dei requisiti è la possibilità di essere più formale, cioè è meno ambiguo. Vengono considerati due aspetti fondamentali del sistema:

- **Modello statico**: entità e relazioni per descrivere concetti del dominio, componenti e classi di realizzazione;
- **Modello dinamico**: modella il comportamento delle entità descritte nel modello statico.

UML è composto da 14 diagrammi, e ogni diagramma dà una vista della parte di realtà descritta da un modello. Ogni diagramma possiede una rappresentazione grafica di un insieme di elementi del modello che corrisponde a un

grafo con i nodi che sono gli elementi del modello e gli archi che sono le relazioni fra gli elementi. Un insieme di diagrammi descrive un modello. Per mezzo di un diagramma si modellano le funzionalità, i processi e le architetture software di un progetto.

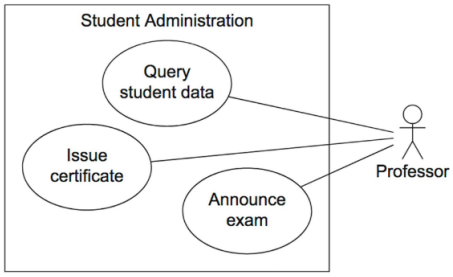
3.3 Diagramma dei casi d'uso

Il diagramma dei casi d'uso modella i requisiti, cioè è in alternativa o in supporto al documento dei requisiti, e fa parte sia del modello statico che dinamico. Il diagramma dei casi d'uso descrive i requisiti funzionali del sistema, cioè individua le funzionalità che un sistema deve offrire quando è visto dall'esterno: *quali funzionalità deve offrire a chi lo usa?* Si può pensarlo come i compiti che un utente vuole voler fare con l'aiuto del sistema, quindi l'utente viene visto come un **attore**, cioè una entità esterna al sistema, che interagisce direttamente con esso in un determinato ruolo. Un attore può essere un utente, un altro sistema esterno, o il tempo. Un caso d'uso è una funzionalità o un servizio offerto dal sistema a uno o più attori, e formalmente un compito che un attore può svolgere con l'aiuto del sistema.

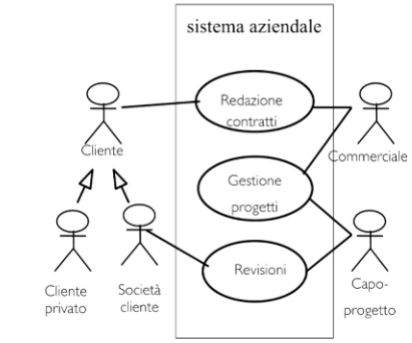
3.3.1 Sintassi

La modellazione dei requisiti usando il diagramma dei casi d'uso prevede individuare:

- **Confine del sistema**, cioè quale è la parte che spetta lavorare;
- **Attori**, cioè chi interagisce con il sistema;
- **Casi d'uso**, cioè quello che gli attori possono fare con il sistema;
- **Relazioni attore-caso d'uso**, cioè come gli attori interagiscono con le funzionalità.



Nel modello statico si disegna sempre un rettangolo che rappresenta i confini del sistema, gli viene assegnato un nome e in seguito si cerca di capire quale funzionalità offrire all'interno del sistema. In figura vi è un sistema che permette di gestire gli studenti, quindi l'attore che ha a che fare con questo sistema può essere un professore. Le funzionalità che il sistema deve avere sono visualizzare i dati degli studenti, creare certificati e annunciare esami. Tutti i nomi assegnati devono essere chiari e non ambigui. Nel diagramma dei casi d'uso si può specificare che un attore può essere l'unione di ruoli diversi, per esempio in un sistema aziendale i clienti possono essere privati o di società. Clienti e commerciale partecipano alla redazione dei contratti, società e capoprogetto partecipano alle revisioni, non tutti i clienti partecipano alle revisioni per questo motivo è convenuto spezzare clienti in due ruoli, e infine commerciale e capoprogetto gestiscono i progetti. Se ci fosse stata una freccia da cliente a revisioni al posto di società cliente non sarebbe stato sbagliato ma sarebbe stato ambiguo perché ora anche i clienti privati possono partecipare alle revisioni.



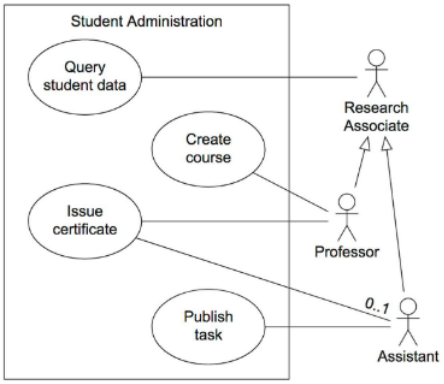
L'associazione attori - casi è molti a molti : un attore può essere associato a più casi d'uso e viceversa. Un caso d'uso è iniziato solo da un attore che viene chiamato *attore principale*. Il modello statico indica quali sono le funzionalità ma non sta mostrando il *come* queste funzionalità funzionano. Per ogni caso d'uso viene specificato a parte una **descrizione narrativa** che mostra:

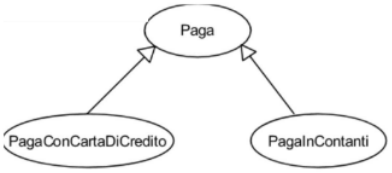
- **Nome del caso d'uso**;
- **Breve descrizione riassuntiva**;
- **Attori primari**, cioè quelli che avviano il caso d'uso;
- **Attori secondari** cioè quelli che interagiscono con il caso d'uso;
- **Precondizione**, cioè delle condizioni che devono valere prima dell'esecuzione del caso d'uso, per esempio se si deve specificare la narrativa di un prelievo in una banca si può usare come precondizioni che l'utente deve avere un conto presso l'istituto bancario;

- **Sequenza degli eventi principali**, nell'esempio della banca si può usare "l'utente digita il pin della carta, il sistema verifica che il codice sia corretto, ...";
- **Postcondizione** indica quali condizioni valgono dopo la sequenza degli eventi principali se tutto è andato a buon fine;
- **Sequenze alternative degli eventi** cioè degli eventi che non portano alla postcondizione come degli errori della sequenza principale, per esempio, l'utente non possiede la cifra che desidera prelevare;

Si devono elencare tutte le sequenze di eventi principali che portano dalla precondizione alla postcondizione in maniera corretta e prevedere degli eventi per cui non si può arrivare alla postcondizione, tra questi eventi eccezionali si può decidere se meritano a loro volta di essere descritte da un'altra narrativa. Ad esempio se una di queste situazioni richiede una funzionalità della macchina, allora questo evento potrebbe volere una narrativa che lo descrive.

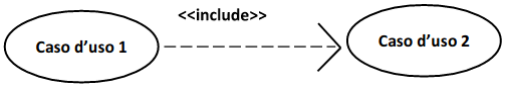
Si suppone di voler automatizzare l'amministrazione degli studenti. Un attore può essere anche **abstract**, cioè non è una persona fisica, ma è l'unione di due sottoclassi, come in questo esempio. Per il primo caso d'uso, cioè *query student data*, la può attivare sia un professore che un assistente. Il compito di creare un corso può essere svolto solamente da un professore, mentre pubblicare un task può essere fatto solo da un assistente. Per generare un certificato occorre la presenza di un professore, e ci può partecipare o meno un assistente. Quando due attori, come in questo esempio, partecipano ad un caso d'uso, uno sarà l'attore primario, mentre l'altro sarà l'attore secondario però da questo schema non si può vedere ma sarà specificato nella descrizione narrativa. Esiste la generalizzazione anche sui casi d'uso per esempio "*paga*" può essere la generalizzazione di "*paga con carta di credito*" e "*paga in contanti*. La generalizzazione va utilizzata facendo attenzione che il classificatore specializzato eredita tutte le relazioni del classificatore padre (**Liskov**).





3.3.2 Inclusione di un caso d'uso

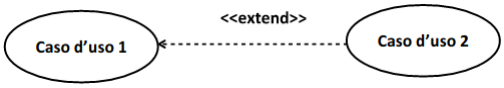
Quando un caso d'uso ne richiede un altro, si parla di inclusione di caso d'uso. Per esempio in un biblioteca per prendere in prestito un libro, prima si deve verificare che il libro appartenga alla biblioteca. Il caso d'uso di verificare che il libro appartenga alla biblioteca può essere anche indipendente, ma sicuramente sarà sempre chiamato quando viene preso in prestito un libro. I due casi d'uso si mettono in relazione con una linea tratteggiata e con la freccia verso il caso d'uso incluso, all'interno della descrizione narrativa verrà scritto *include* e il caso d'uso a cui si riferisce. Per ogni caso d'uso si ha bisogno di un attore principale, ma questo non vale nella possibilità in cui il caso d'uso sia incluso, l'attore inizia il caso d'uso 1, che chiamerà il caso d'uso 2 che quindi non avrà necessariamente bisogno di attore principale.



Nella sequenza degli eventi principali il caso d'uso incluso verrà chiamato come una sottoprocedura e nella narrativa ci sarà la descrizione di come funziona il caso d'uso incluso che può essere specificato a parte, cioè può non rientrare direttamente nella narrativa nel caso che lo include.

3.3.3 Estensione di un caso d'uso

Un caso d'uso può essere anche esteso, la freccia tratteggiata punta al caso d'uso che viene esteso. La differenza tra inclusione ed estensione è sottile: nel caso dell'inclusione tutte le volte che viene chiamato il caso d'uso principale si ha bisogno nel caso d'uso incluso, cioè è praticamente una chiamata di procedura necessaria in un certo momento, mentre, l'estensione è subordinata al verificarsi di una certa condizione, ovvero *se* succede qualcosa il caso d'uso viene chiamato, altrimenti non viene chiamato. Un esempio può essere quando si crea nuovo corso *si può* riservare un'aula ma *non è obbligatorio*. Nella narrativa verrà scritto *extend*, e si troverà sotto un *if - else*



Chapter 4

Diagramma delle classi e degli oggetti

4.1 Classi e oggetti

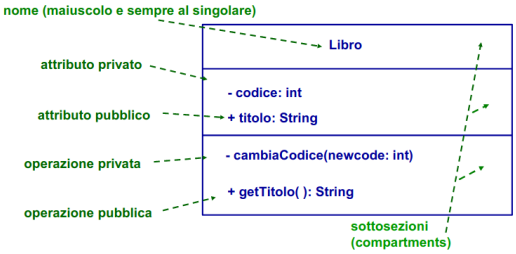
Un oggetto è un entità caratterizzata da un'identità, uno stato e un comportamento, mentre una classe descrive un insieme di oggetti con caratteristiche simili, cioè oggetti che hanno lo stesso tipo. Le classi sono classificatori e gli oggetti sono istanze, modellare a livello dei classificatori significa vincolare i modelli a livello di istanza.

4.2 Diagramma delle classi

Il diagramma delle classi è uno dei più utilizzati e serve soprattutto per descrivere gli elementi del dominio. Una classe cattura un concetto nel dominio del problema o della realizzazione. Il diagramma delle classi descrive il tipo degli oggetti che fanno parte di un sistema software o del suo dominio, e le relazioni statiche che uniscono le classi, cioè gli elementi e le relazioni che non cambiano nel tempo. I diagrammi delle classi mostrano anche le proprietà e le operazioni di una classe.

4.2.1 Sintassi

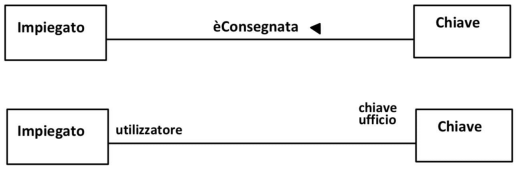
Nella classe vi è il titolo, cioè il nome della classe, in seguito si possono avere delle sottosezioni che contengono gli attributi, che definiscono lo stato dell'oggetto, e delle operazioni, che definiscono il suo comportamento. È una specifica molto simile a come poi verrà implementato in un linguaggio ad oggetti. Questo tipo di schema delle classi può essere usato a diversi livello di dettaglio e in diverse fasi del progetto fino alla generazione del codice. Non è obbligatorio indicare gli attributi e le operazioni, dipende in quale fase ci si trova, per esempio nella fase iniziale possono essere omessi. Se si decide di aggiungere un attributo li si deve dare un nome e opzionalmente mettere la visibilità, il tipo, indicare il valore iniziale e qual è la proprietà. La visibilità può essere **public** **+** accessibile ad ogni elemento che può vedere e usare la classe, **protected #** accessibile ad ogni elemento discendente, **private -** solo le operazioni della classe possono vedere e usare l'elemento in questione, **package** $\sim$ accessibile solo agli elementi dichiarati nello stesso package. Se si vogliono aggiungere le operazioni, come gli attributi, è necessario specificare un nome e opzionalmente si può aggiungere una lista di parametri, un tipo di ritorno e specificare la visibilità.



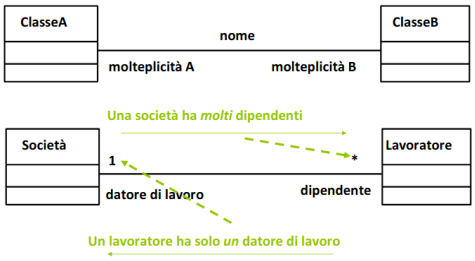
4.2.2 Cosa va modellato con una classe

Con una classe si modellano tutte le cose del dominio che sono ritenute importanti, se sono presenti degli attributi o dei modelli statici vanno sottolineati. Un altro strumento utile sono le **enumerazioni** che vengono create con lo stereotipo *enumeration*, e servono per specificare un insieme di valori prefissati che non hanno altre proprietà oltre al loro valore simbolico. Una relazione rappresenta un legame tra due o più oggetti:

- **Associazione,** si indica con una linea retta che significa che classeA e classeB hanno una certa relazione. Talvolta a livello di implementazione una freccia specifica la navigabilità. Il nome denota il senso della relazione, e il verso indica in quale verso leggere la dipendenza. In alternativa al nome si può denotare il ruolo della classe nella associazione. Il nome dell'associazione è normalmente un verbo, mentre il ruolo è generalmente un sostantivo. Figura: nell'implementazione quando si porterà il progetto fino al codice, dentro la classe impiegato ci sarà un attributo *chiave ufficio* e/o dentro la classe chiave ci sarà un attributo del tipo *utilizzatore*.

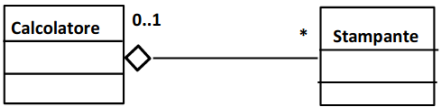


Interessa sapere anche la **molteplicità**, cioè sapere quante istanze di una classe sono in relazione con una istanza dell'altra. Le molteplicità si possono definire indicando gli estremi inferiori e superiori di un intervallo. L'estremo inferiore può essere 0 o un numero positivo, mentre quello superiore un numero positivo o * (indefinito). L'intervallo n...n equivale a n, mentre 0...* corrisponde a *. Il numero 1 è di default e si può omettere.



un oggetto società può essere in relazione con nessuno o con molti oggetti lavoratore. Un oggetto lavoratore può essere in relazione con un solo oggetto società. Nelle **associazioni riflessive** è importante aggiungere il ruolo della classe nella associazione;

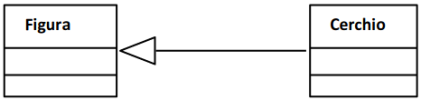
- **Aggregazione e Composizione**, sono tipi particolari di associazione (un raffinamento), entrambe specificano che un oggetto di una classe è *una parte di* un oggetto di un'altra classe. Una aggregazione è una relazione tra oggetti poco forte e si indica con un rombo vuoto:



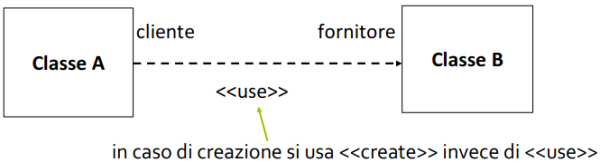
la stampante nel tempo può essere collegata a calcolatori diversi ed esiste anche senza calcolatore. Se il calcolatore viene distrutto la stampante esiste comunque. Una composizione è una forma di associazione più forte dell'aggregazione e si indica con un rombo pieno:

le parti non hanno senso senza il tutto, se il mouse viene distrutto anche i suoi pulsanti lo saranno;

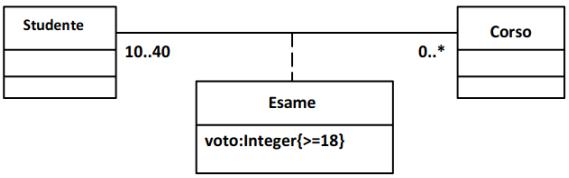
- **Generalizzazione**, è una relazione tra un elemento generico e uno più specializzato. L'elemento più specializzato è completamente consistente con quello più generico ma contiene più informazione. Vale il principio di Liskov: l'elemento specializzato può essere usato al posto dell'elemento generico.



Le sottoclassi ereditano tutte le caratteristiche della superclasse: attributi, operazioni, relazioni e vincoli. Le sottoclassi possono aggiungere caratteristiche e ridefinire le operazioni. Un'altra informazione che si può specificare in un diagramma delle classi è quella delle **dipendenze**, cioè indicare nello schema quali sono le dipendenza di uso:



il cliente dipende dal fornitore e una modifica nel fornitore può influenzare il cliente. Lo stereotipo è *use* nel caso in cui un parametro di un'operazione di A è di tipo B, e un'operazione di A restituisce un oggetto di tipo B. Nel caso in cui un'operazione di A crea dinamicamente un oggetto di tipo B si può essere più specifici utilizzando lo stereotipo *create*, o altrimenti è corretto anche usare *use*. Un'associazione può avere attributi propri, rappresentati con una **classe associazione**.



Le istanze sono collegamenti con attributi propri. Voto non è attributo né di Corso né di studente. Per ogni coppia di oggetti collegati tra loro può esistere un unico oggetto della classe associazione, se si vuole tenere traccia dei voti negativi non si possono usare le classi associazione.

4.2.3 Individuare le classi di analisi

Si devono individuare i concetti concreti del dominio, per esempio tutto ciò che viene descritto nel glossario. Ciascuna classe di analisi sarà raffinata in una o più classi di progettazione. Si deve cercare di astrarre uno specifico elemento del dominio con un numero ridotto di funzionalità, cioè evitare le classi "onnipotenti". Si deve evitare anche funzioni travestite da classi e gerarchie di ereditarietà profonda (≥ 3), e infine ogni classe non deve essere troppo dipendente dalle altre classi. Si aggiungono le operazioni e gli attributi solo quando sono veramente utili e limitare la specifica di tipi, valori, ... Nelle prime fasi di sviluppo si possono avere due approcci:

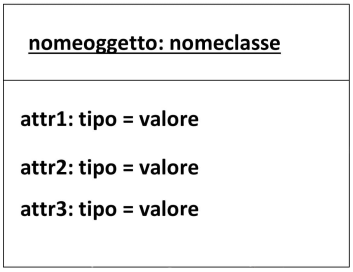
- **Data driven**, si identificano tutti i dati del sistema e si dividono in classi;
- **Responsibility driven**, si identificano le responsabilità e si dividono in classi;

Per individuare le classi non esiste un metodo automatico che le estrae da un documento, ma si può usare la **tecnica del Nomi - Verbi:**

- I sostantivi sono classi o attributi, mentre i verbi sono le responsabilità o le operazioni;
- I passi da compiere sono:
  - individuazione delle classi;
  - assegnazione di attributi e responsabilità alle classi;
  - individuazione di relazione tra le classi.
- Tagliare le classi inutili e individuare le classi nascoste cioè le classi implicite del dominio del problema che possono anche non essere mai menzionate esplicitamente.

4.3 Diagramma degli oggetti

Il diagramma degli oggetti viene chiamato anche diagramma delle istanze e può essere utile quando le connessioni tra gli oggetti sono complicate. Nel diagramma degli oggetti il nome della classe viene sottolineato e la lista degli attributi, come nel diagramma delle classi, è opzionale. Il tipo dell'attributo è ridondante ed è consigliabile ometterlo, mentre il valore è la parte interessante. Un collegamento è una istanza di una associazione, e collega due (o più) oggetti. Un collegamento non ha un nome e non ha molteplicità, cioè è sempre 1 a 1, e se è utile si possono indicare i ruoli.



Chapter 5

Diagramma delle attività

5.1 Scopo

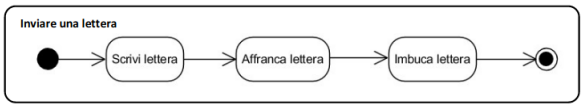
Il diagramma delle attività serve per descrivere delle mansioni che devono essere effettuate all'interno del progetto, delinea il comportamento dinamico e illustra cosa deve essere fatto sequenzialmente o in ordine sparso. Si può applicare a diversi momenti del progetto: quando uno o più attori interagiscono con il sistema e si vuole documentare l'attività, quando si vuole descrivere un'operazione di classe o, quando una o più classi collaborano in una attività comune.

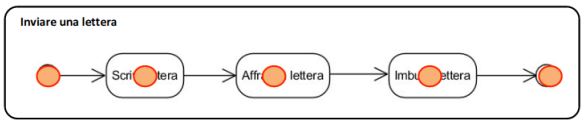
5.2 Sintassi

Il diagramma delle attività si esprime con un rettangolo in cui all'interno si descrivono alcune attività che vengono connesse per far intendere qual è il flusso delle informazioni. Il contenuto di un'attività è un grafo diretto i cui:

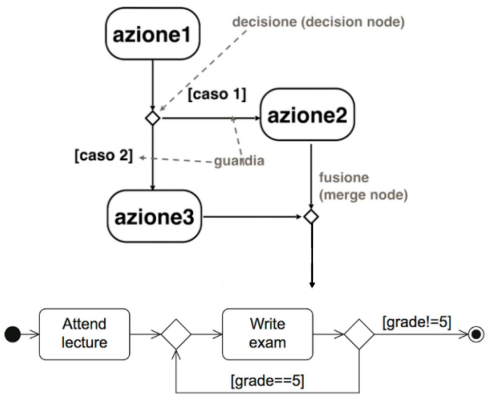
- **Nodi** rappresentano le componenti dell'attività come le **azioni** o i **nodi di controllo** (nodo inizio, nodo fine, ...);
- **Archi** rappresentano il control flow, cioè i possibili path eseguibili per le attività.

Le azioni sono rappresentate anche esse da rettangoli con angoli smussati che godono di un nome significativo che suggerisce cosa svolge e tipicamente è un verbo. Le azioni sono intese come atomiche, cioè non possono essere svolte parzialmente, e hanno una sola freccia entrante e una uscente, dove la freccia di uscita è presa appena è terminata l'azione. La semantica è descritta con il **token game**: l'azione può essere eseguita quando riceve il token, quando un'azione ha terminato il proprio lavoro scatta una **transizione automatica** in uscita dell'azione che porta all'azione successiva.

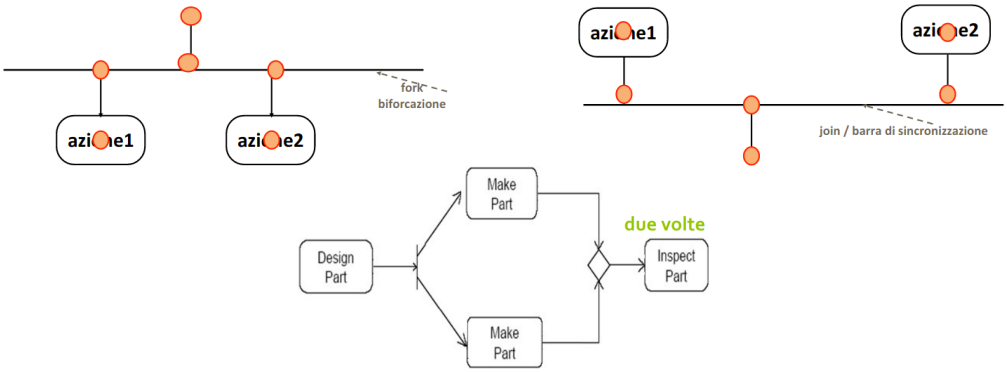




Questo meccanismo di passaggio del token viene alterato da una **choice** che permette di strutturare l'attività in modo da poter scegliere in base a qualche condizione, cioè in altre parole data una azione 1 si sceglie se andare verso l'azione 2 oppure 3 in base a una certa circostanza. I vari casi sono chiamate **guardie**, devono stare tra parentesi quadre e devono coprire tutti i casi, cioè il token non può bloccarsi. Nel caso in cui non si riesce a coprire tutte le possibilità, in una guardia si può scrivere **[else]**. È bene ma non è necessario che caso 1 e caso 2 siano *mutuamente esclusivi*, ovvero se x = 1 e caso 1 indica $x \geq 1$ mentre caso 2 $x \leq 1$ si potrebbe andare in entrambi i casi ma questo comportamento non è un problema perché non blocca il token. Il nodo choice è composto da un rombo con guardie sulle frecce uscenti. Alla fine dopo che il token è passato dall'azione 2 o 3 passa in un **merge node** che è composto da un rombo con frecce entranti quante sono le azioni. Dato un nodo decisione non è obbligatorio un nodo fusione corrispondente. All'interno del diagramma si possono trovare dei **loop**. Con il diagramma delle attività si cerca di mettere in evidenza quali sono le azioni che si devono compiere in



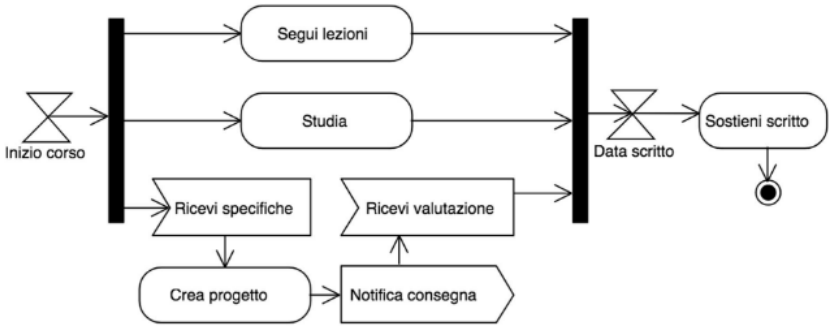
maniera obbligatoria sequenzialmente e quali azioni non interessa l'ordine. Nel caso in cui la sequenza non è importante si usa la **fork** che moltiplica i token, cioè dato un token in ingresso, ne produce uno per ogni freccia uscente e non è presente la sincronizzazione tra i diversi token. L'opposto del nodo fork è il nodo **join** dove tutti i token si consumano e ne esce solo uno e in questo caso è presente la sincronizzazione dei token. Non è necessaria una join per ogni fork.



Quando il token parte da Design Part, il nodo fork lo moltiplica in due, il nodo fusione fa passare i due token in ordine casuale ovvero l'azione *Inspect Part* viene eseguita due volte: dal token che arriva per primo e in seguito dal secondo token. Se un token raggiunge un **nodo di fine attività**, l'intera esecuzione è terminata, cioè il primo token che arriva termina l'attività. Il **nodo di fine flusso** non termina tutta l'attività ma solo l'execution path.

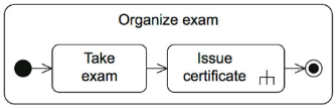
5.2.1 Segnali ed Eventi

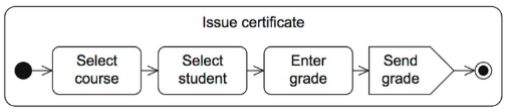
Con il diagramma delle attività si possono esprimere altri avvenimenti come il fatto che una attività si blocca aspettando l'**accettazione di evento esterno**, oppure l'**invio di un segnale**. Un altro segnale è l'**accettazione di evento temporale**, raffigurato con una clessidra, che indica che a un certo punto può succedere un evento. Se l'avvenimento è assoluto significa che in un determinato momento parte un token, oppure se il tempo è relativo si può indicare che, per esempio, dopo dieci minuti parte un token. Se la clessidra è assoluta non necessariamente deve possedere un arco entrante, cioè non aspetta un token per farne partire uno, solitamente è la clessidra relativa che fruisce di un arco entrante. Si studia un caso dove si modellano le azioni che devono essere compiute per superare un corso: è presente una clessidra iniziale che indica che tutte le attività possono incominciare solamente quando inizia il corso, a questo punto concorrentemente si devono seguire le lezioni, studiare e infine si deve sviluppare un progetto, cioè ricevere le specifiche dal professore, creare il progetto, notificare al professore che si è finito, e in seguito il professore valuterà il progetto. La clessidra della data dello scritto possiede un arco entrante perché le azioni di studiare, seguire le lezioni e fare il progetto devono essere tutte eseguite per sostenere lo scritto. Il token si ferma durante nella seconda clessidra perché aspetta la data dello scritto, successivamente si sostiene l'esame e infine si arriva al nodo di fine attività. Le azioni sono modellate dal punto di vita dello studente, quindi il fatto di ricevere notifiche e ricevere valutazione sono eventi esterni perché l'insegnante in questo diagramma non è modellato.



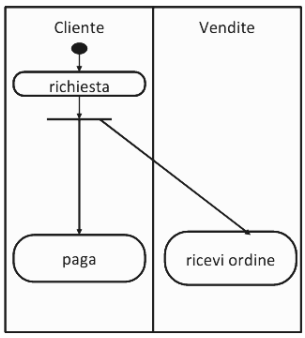
5.2.2 Sotto-attività e partizioni

Per migliorare il riuso e la leggibilità, a volte al posto di aggiungere una azione nel diagramma delle attività, si inserisce una **sotto-attività**, cioè un'azione che include (chiama) un'altra attività secondaria. Si usa il "rastrello" come simbolo per indicare che l'azione include una sotto-attività e si descrive quest'ultima in un diagramma a parte.





L'ultima caratteristica che si può esprimere con un diagramma delle attività sono le **partizioni** che sono uno strumento utile per dividere le azioni in gruppi. Spesso corrisponde alla divisione in unità operative in un modello business e permettono di assegnare la responsabilità delle azioni. In genere il diagramma viene diviso verticalmente e può voler esprimere, per esempio, quali sono le azioni a carico del cliente e quali a carico del sistema.



Chapter 6

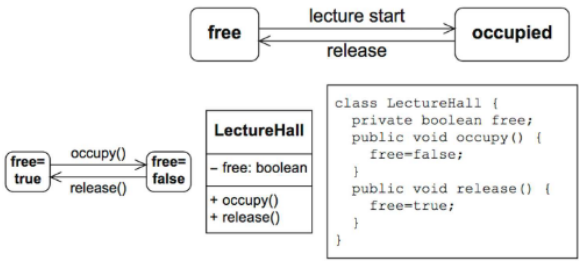
Diagramma di macchine a stati

6.1 Scopo

Una macchina a stati descrive il comportamento dinamico delle istanze di un classificatore. Per costruire una macchina a stati si deve individuare gli stati significativi in cui si può trovare un oggetto durante la sua vita e descrivere come da ciascuno di questi stati l'oggetto può **transitare** in un altro. Le transizioni avvengono in risposta al verificarsi di un **evento**. Gli eventi sono tipicamente: *messaggi inviati da altri oggetti* o *eventi generati internamente*. Una macchina a stati è rappresentata con un grafo di stati e transizioni, associata a un classificatore.

6.2 Stato

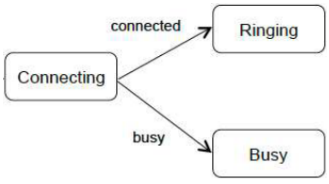
Lo stato di un istanza sono un insieme di valori che caratterizza un oggetto. Lo stato può essere composito ed ha un nome unico. Si immagina di voler modellare la vita di un'aula dove si può fare lezione: i due stati principali sono "occupata" o "libera", e si transita da uno stato all'altro nel momento in cui inizia o finisce una lezione. All'interno della classe che rappresenta l'aula ci sarà una variabile che indica se la stanza è libera o occupata, lo stato è una variabile e le transizioni sono dei metodi che una volta chiamati fanno cambiare il valore della stato.



6.3 Sintassi

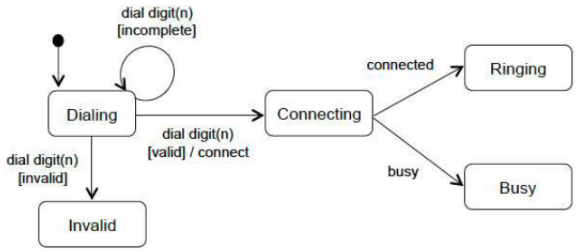
Gli stati sono rappresentati con dei rettangoli arrotondati. Un cerchio nero indica l'inizio dell'esecuzione e non è uno stato ma semplicemente segnala qual è lo stato iniziale. Il disco nero bordato indica il nodo finale cioè che l'esecuzione è terminata. Una transizione collega tra loro due stati, ed è rappresentata con una freccia. L'uscita da uno stato definisce la risposta dell'oggetto all'occorrenza di un evento, viene presa solo se la condizione è vera, e comporta l'esecuzione delle azioni specificate.

Un evento è l'occorrenza di un fenomeno collocato nel tempo e nello spazio, e si suppone che l'evento occorre istantaneamente. Si modella come evento quello che permette di passare da uno stato all'altro, cioè se ha delle conseguenze. Gli eventi, anche se non tutti, corrispondono alle operazioni della classe. È ammesso il non-determinismo: cioè si potrebbe avere un evento, che porta in due stati diversi dallo stesso stato, in questo caso viene scelto uno dei due stati. Per esempio in una chiamata al telefono:



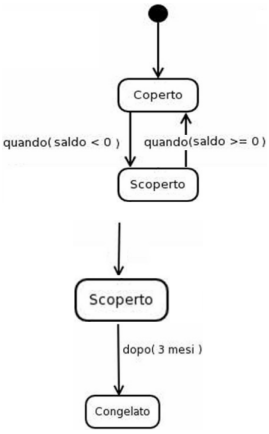
6.3.1 Tipi di evento

Gli eventi trattati fino ad ora sono del tipo **operazione o segnale** *op(a:T)* dove la transizione è abilitata quando l'oggetto riceve una chiamata di metodo / segnale con parametri (a) e tipo (T) (i parametri sono opzionali). Sempre nell'esempio della chiamata al telefono si parte in uno stato in cui si digita il numero: fino a che il numero non è completo. Quando il numero è completo, può succedere che il numero è invalido e si arriva in uno stato invalid, oppure il numero è valido e il telefono si connette. A questo punto, il cellulare fa l'azione di connettersi e può succedere che il telefono squilla o altrimenti è già occupato.



Esiste un modo più ricco di determinare gli eventi:

- **Evento di variazione** *quando(exp)*, la transizione è abilitata appena l'espressione diventa vera. L'espressione può indicare un tempo assoluto o una condizione su variabili. Si immagina di voler modellare la vita di un conto corrente, si passa dall'evento di essere un conto coperto a scoperto quando il saldo è minore di zero, ma può ritornare ad essere coperto nel momento in cui il saldo è maggiore o uguale di zero;
- **Evento temporale** *dopo(time)*, la transizione è abilitata dopo che l'oggetto è stato fermo "time" in quello stato. Continuando con l'esempio del conto corrente, si può immaginare che se per tre mesi il conto corrente è scoperto allora diventa congelato.

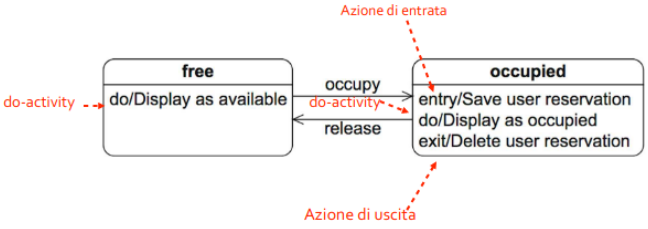


6.3.2 Transizioni e attività interne

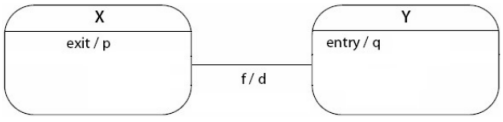
Oltre a specificare delle transizione sulle frecce, si possono avere delle azioni all'interno dello stato in risposta ad un evento, e possono essere:

- **Azione di entrata**, eseguita all'ingresso in uno stato. Keyword: *entry*;
- **Azione di uscita**, eseguita all'uscita di uno stato. Keyword: *exit*;
- **Transizione interna**, risposta ad un evento. Keyword: *evento-name*.

In aggiunta esiste una azione particolare denominata **do-activity** che viene eseguita in modo continuato mentre l'oggetto si trova in quello stato. Al contrario di tutte le altre azioni che sono atomiche: consuma del tempo, può essere interrotta, e la sua keyword è *do*. Nell'esempio dell'aula, oltre a modellare quando la stanza è libera o occupata, si può aggiungere che fino a che la classe è libera si mostra in un display che è disponibile. Quando viene occupata, si può decidere di salvare l'utente che l'ha riservata, di mostrare sul display che la stanza è occupata e quando si esce dall'aula si cancella la prenotazione dell'utente che l'aveva salvata.



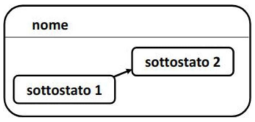
Le transizioni e le attività interne possono essere combinate come nell'immagine dove non si esce dallo stato X fino a che f non è vero: quando f si verifica, viene eseguito p, poi l'azione d, poi si entra in uno stato Y e quindi si esegue q.



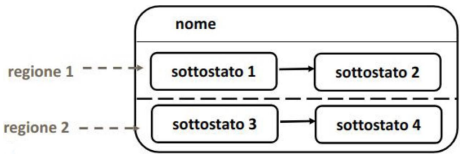
6.3.3 Stati compositi

Gli stati possono essere anche compositi, ovvero:

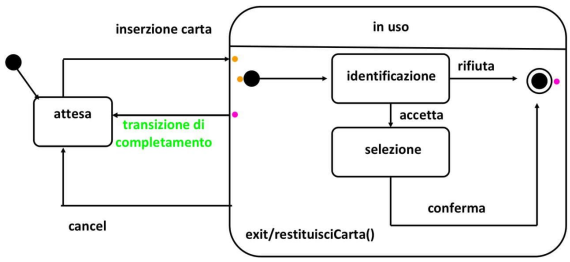
- **Composito sequenziale**, un sottostato attivo in ogni istante;



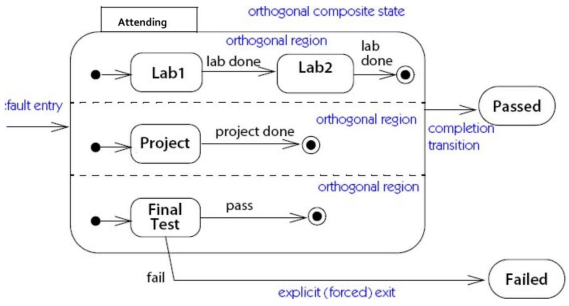
- **Composito parallelo**, sottostati attivi contemporaneamente, uno per regione.



Un esempio di stato composito sequenziale è la modellazione di una carta di credito, sono presenti due stati: attesa e in uso. Quando si inserisce la carta e si va nello stato in uso si identifica la carta, e ora si possono avere due eventi: la carta viene rifiutata e quindi si va in uno stato finale, altrimenti la carta viene accettata, si effettua la selezione dell'attività che si vuole compiere e quindi si va nello stato finale. Come azione finale nello stato in uso si restituisce la carta. In qualsiasi stato all'interno dello stato composito, se si verifica l'evento *cancel*, cioè l'utente vuole annullare l'inserzione della carta, si esegue la transizione di uscita e si esce.



Un esempio di stato composito parallelo è la modellazione di un corso di laboratorio: per seguire tutto il corso, bisogna seguire due laboratori in sequenza, contemporaneamente fare un progetto e un test finale. Quando si entra nello stato *attending* si prosegue in tutti e 3 gli stati iniziali in parallelo. Per quanto riguarda l'esame finale se si fallisce, non vi sono altre possibilità di riprovare il corso. Per passare il corso si deve aver raggiunto tutti e 3 gli stati finali.



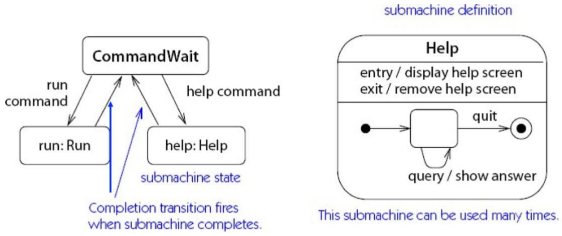
6.3.4 Transizioni di completamento

Le **transizioni di completamento** hanno priorità sugli eventi normali, e ci possono essere quando non è presente un evento, ed hanno senso:

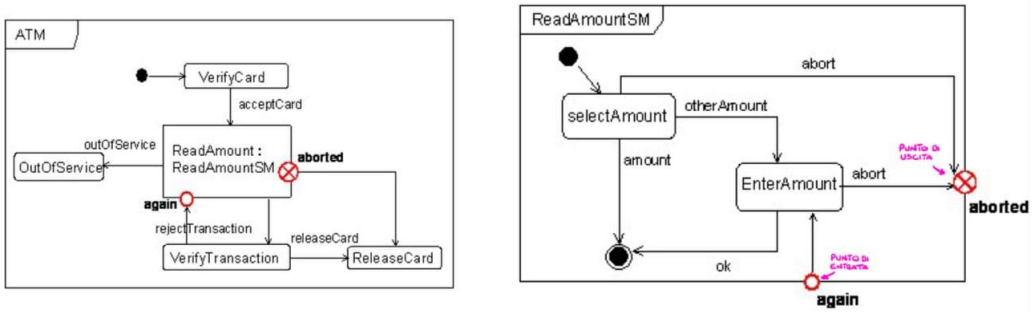
- quando è presente uno stato composito, scattano al raggiungimento della terminazione di un'attività composita;
- alla terminazione di un entry e/o di un do-activity.

6.3.5 Sottomacchine

Si usa quando si vuole descrivere uno stato composito in un diagramma a parte, per leggibilità o per definirlo una volta per tutte e riusarlo in più contesti. La sottomacchina ha un nome (tipo), le istanze di uso si indicano con nomeIstanza: Tipo.



Una sottomacchina può definire entry e exit points che servono per collegare le transizioni della macchina principale. Per esempio:

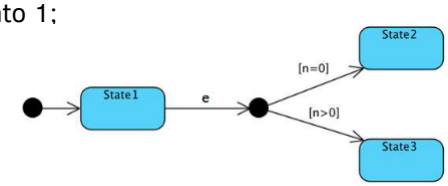


Se la transazione viene rigettata, si entra nella macchina, ma non si riparte dal *SelectAmount* ma da *EnterAmount*.

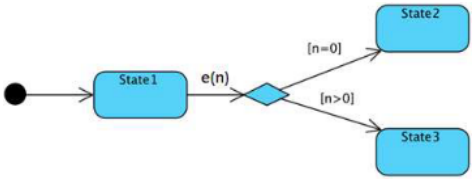
6.3.6 Altri tipi di stato

Ci possono essere altri tipi di stati:

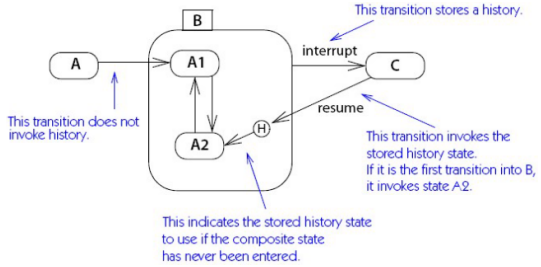
- **Giunzione**, eventuali condizioni sono valutabili in modo statico (prima dell'evento e). Se $n < 0$ l'evento "e" viene ignorato e si rimane nello stato 1;



- **Choice**, condizioni valutate dinamicamente. La disgiunzione delle guardie deve essere true, ed è ammesso il non determinismo. Le guardie sono valutate dopo e(n), in questo esempio occorre avere garanzia che n sia maggiore o uguale a zero;



- **History** servono in presenza di una transizione esterna da uno stato composito per ricordarsi quale era l'ultimo stato attivo, in modo da poter rientrare esattamente in quello stato;



- **Fork/Join**.

Chapter 7

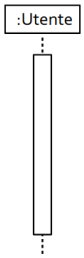
Diagrammi di sequenza

7.1 Scopo

Si usano per descrivere le **interazioni**, cioè lo scambio di messaggi e dati tra oggetti, per esempio un attore e il sistema per la realizzazione di un caso d'uso, oppure, in fase di progettazione i messaggi scambiati tra sottosistemi. Interessa la sequenza temporale dei messaggi.

7.2 Elementi del diagramma

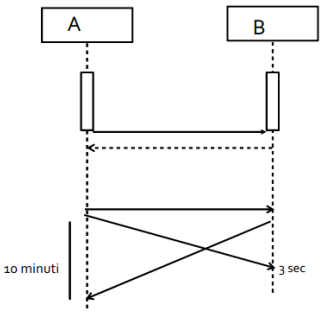
Se si vuole modellare l'interazione tra un attore e il sistema, servono due oggetti:



- **Rettangolo**, che indica il ruolo nell'interazione e/o tipo dell'oggetto;
- **Linea verticale**, che indica la linea di vita dell'oggetto:
  - la linea è tratteggiata quando l'oggetto è inattivo;
  - continua e doppia quando l'oggetto è attivo. Gli oggetti sempre attivi, come per esempio gli attori, hanno l'intera linea di vita continua e doppia.

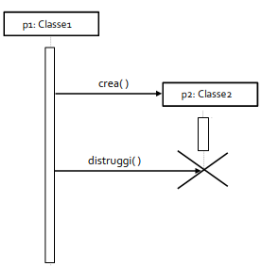
I diagrammi di sequenza si esprimono con un rettangolo, denominato con un nome, che racchiude l'interazione che si sta modellando. All'interno del rettangolo ci sono i partecipanti dell'interazione. I messaggi, rappresentati come frecce, possono essere,

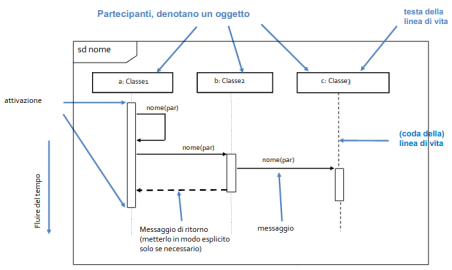
- **Continue e orientate verso un altro oggetto**, e sono del tipo:
  - **Sincrone**, dopo che ha calcolato un qualcosa, comunica con con un altro oggetto passandogli i parametri di cosa ha calcolato;
  - **Asincrona**, a differenza delle altre frecce si rappresenta con una punta non piena, e si può indicare con un esplicito consumo di tempo. Per esempio A manda un messaggio asincrono a B che può rispondere entro 10 minuti. Un'ultima cosa che si può specificare è che un messaggio può avere una certa durata per essere ricevuto e si rappresenta con una freccia in diagonale.



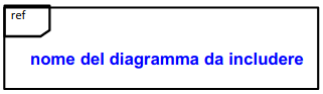
- **Tratteggiate**, indicano il messaggio di ritorno per messaggi sincroni, si suppone che ci sia sempre e viene rappresentato solo se necessario.

Il messaggio ha un nome, e opzionalmente può avere dei parametri e un valore di ritorno. Alcuni partecipanti possono essere aggiunti o cancellati dinamicamente all'interazione e per fare ciò si rappresentano come in figura.





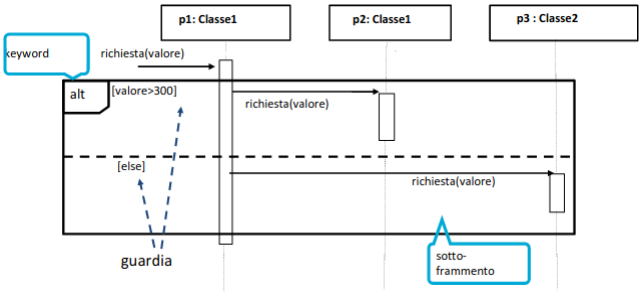
Nel caso in cui il diagramma è molto complicato, si può includere una interazione definita altrove e si usa la keyword *ref*.



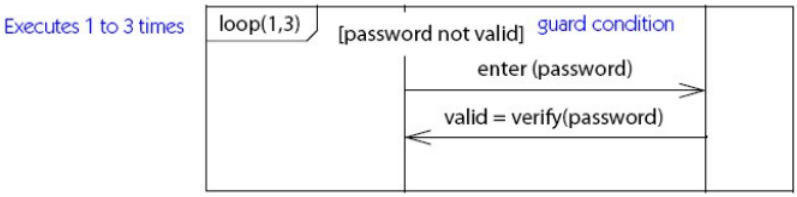
7.2.1 Tipi di frame

I frame possono essere di 4 tipi:

- **Frame condizionale**, i partecipanti possono scambiarsi determinati messaggi in risposta a qualche condizione. Se non è presente nessuna guardia è come se fosse [*true*], se sono presenti più guardie vere si esegue una scelta non deterministica, e se infine tutte le guardie sono *false* allora il frame viene saltato. Il frame condizionale ha come keyword *alt*;



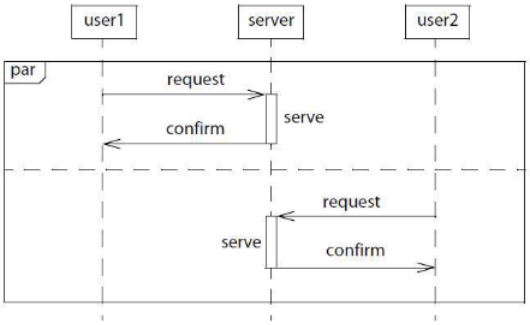
- **Frame opzionale**, si ha la possibilità di fare un if senza l'else. Le interazioni contenute nel frame vengono eseguite solo se la guardia è vera, altrimenti si salta il frame;
- **Frame iterativo**, ha come keyword *loop* e deve avere un *min* e un *max*, che indicano il numero minimo e massimo di volte in cui si esegue l'iterazione. Il numero di volte in cui itera il ciclo dipende dalla condizione. Nell'esempio il frame deve essere eseguito almeno una volta, si controlla la guardia, dopo tre iterazioni si esce in ogni caso.



I vari tipi di loop si modellano in questo modo:

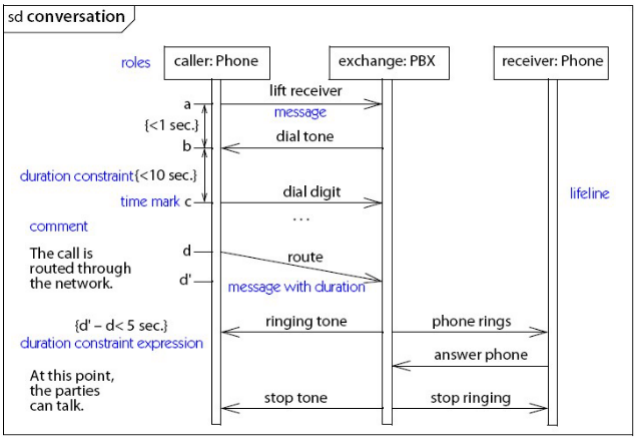
  - **while**, *loop(0,*) [guardia]* oppure *loop [guardia]*;
  - **do - while**, *loop(1, *) [guardia]*;
  - **for**, *loop(n,n)* si scrive senza guardia perché essa è sempre true e il minimo di volte in cui si esegue è n, quindi attenzione a non scrivere loop(0, n). Si può abbreviare con *loop(n)*.

- **Frame parallelo**, si possono avere frame in parallelo dove le interazioni contenute nei due sotto-frammenti sono eseguite in parallelo. Si usa la keyword *par*.



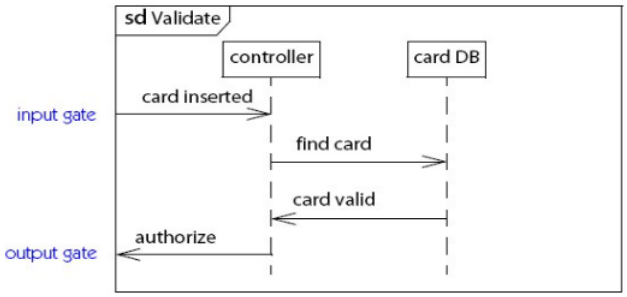
7.2.2 Vincoli di durata

Si possono indicare vincoli temporali, ad esempio quanto tempo passa da quando si alza la cornetta al segnale della centrale.



7.2.3 Gates

Ci possono essere delle situazioni in cui le frecce, al posto di partire da oggetti del modello, partono da oggetti che sono esterni al diagramma. Un gate è un punto sul bordo del diagramma a cui è collegato un messaggio, in ingresso o in uscita. Il nome del gate è quello del messaggio, ed è utile quando si riferisce a ref di altri programmi



Chapter 8

Architetture software

8.1 Progettazione

Nella fase di progettazione è facile correggere i possibili errori nel progetto a differenza di quando si produce, quindi è molto importante questa fase per valutare tutti i possibili problemi. La progettazione è composta da due stadi:

- **Progettazione di alto livello**, scompone il problema in sottosistemi. La scomposizione aiuta ad affrontare la complessità del problema e a identificare le connessioni tra i diversi sistemi;
- **Progettazione di dettaglio**, decide su come la specifica di ogni parte sarà realizzata.

La progettazione costituisce la fase ponte fra la specifica e la codifica, la fase in cui si decide come passare da *che cosa* deve essere fatto a *come* deve essere fatto. Il prodotto di questa fase si chiama **architettura software**. L'architettura di un sistema software è la struttura del sistema costituita dai suoi componenti, dalle relazioni tra i componenti e dalle loro proprietà visibili.

8.2 Viste

Una vista su una architettura software è una particolare astrazione secondo un aspetto di interesse. Ci sono 3 punti di vista simultanei sul sistema:

- **Vista comportamentale (componenti e connettori)**, la struttura come insieme d'unità con comportamenti e interazioni a tempo d'esecuzione;
- **Vista strutturale**, la struttura come insieme di unità di codice, cioè sapere com'è scritto;
- **Vista logistica (problemi di allocazione)**, i componenti individuati con la vista comportamentale vanno allocate.

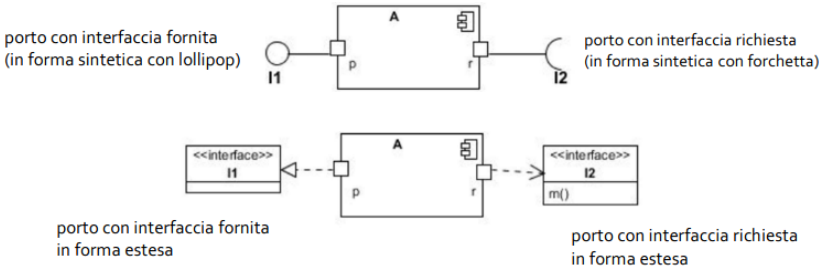
8.2.1 Viste comportamentale

Un **sistema software** è una composizione di componenti software basata sulla connessione di più componenti, realizzata sulla base delle interfacce e mediante l'ausilio di connettori. La vista comportamentale descrive un sistema software specificando i componenti con le loro interfacce, descrivendo le caratteristiche dei connettori e la struttura del sistema in esecuzione, infine la vista comportamentale è utile per l'analisi delle caratteristiche di qualità a tempo d'esecuzione. In questo tipo di diagramma si vogliono individuare:

- **Componenti**, sono delle unità concettuali di decomposizione di un sistema a tempo d'esecuzione. Per esempio: processi, oggetti, serventi, depositi di dati, ... Il componente software è in grado di:
  - incapsulare un insieme di funzionalità e/o dati di un sistema;
  - restringere l'accesso a quell'insieme di funzionalità e/o dati tramite delle interfacce definite;
  - avere un proprio contesto di esecuzione;
  - poter essere distribuito e installato in modo, possibilmente, indipendente da altri componenti.

  Dal punto di vista della notazione in UML, si può rappresentare in uno di questi modi, tutti equivalenti fra loro:

I **porti** identificano i punti di interazione di un componente. I componenti possono avere più porti, uno per ogni tipo di connessione con altri componenti. Un porto fornisce o richiede una o più interfacce. Un porto è rappresentato con un quadratino, mentre esistono due notazioni diverse per fornire/ricevere interfacce:

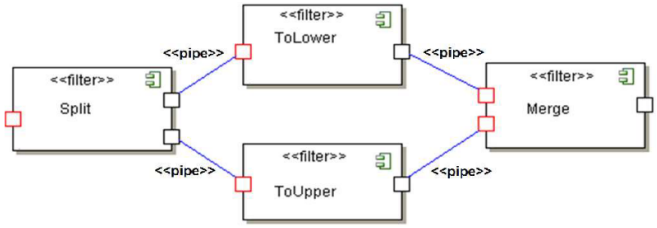


- **Connettori**, sono canali di interazione tra componenti che collegano i porti, per esempio protocolli, flussi d'informazione, accessi ai depositi,... Un connettore non ha un descrittore specifico ma si usa una linea che collega tra loro due porti. Per documentare la linea si usa uno stereotipo in base al tipo di scambio:
  - <<clientServer>>
  - <<dataAccess>>
  - <<pipe>>
  - <<peer2peer>>
  - <<publish-subscribe>>

  Oppure, come nel caso del client - server, si indicano i ruoli dei componenti.

Uno **stile** è una proprietà di una architettura, e caratterizza una famiglia di architetture con caratteristiche comuni:

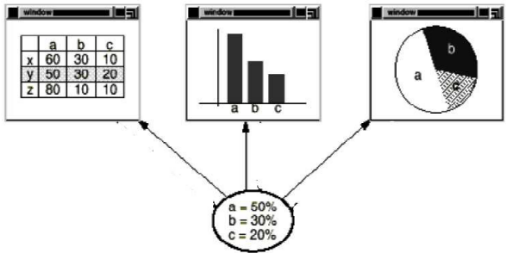
- **Condotte e filtri (pipe & filters)**, si usano connettori con keyword *pipe* quando i componenti hanno un flusso di dati in entrata e producono un flusso di dati in uscita **filtrando** qualche caratteristica. I connettori possiedono un canale di comunicazione unidirezionale bufferizzato che preserva l'ordine dei dati dal ruolo d'ingresso a quello d'uscita. Si usa questo stile in pre-elaborazione in sistemi di elaborazione di segnali e analisi dei flussi dei dati;



- **Client - server**, il sistema è formato da due componenti: il client e il server. Il server svolge le operazioni necessarie per realizzare un servizio e aspetta le richieste dei client in un porto. Il client invia al server le richieste ed attende una risposta. Un caso particolare del client - server è lo stile comportamentale **master - slave** ma risponde a esigenze diverse: la componente server serve un solo master (client). L'architettura master - slave è usata per esempio nella replica di database, dove il database master è considerato come fonte autorevole e i database slave sono sincronizzati con esso;
- **Peer2peer**, è un caso particolare del client - server, dove i client e i server si scambiano i ruoli continuamente, quindi vi è uno scambio di servizi alla pari;

- **Pubblish - subscribe**, le componenti interagiscono in questo modo: una componente si abbona a un servizio (*subscriber*), quindi qualcuno produce dei messaggi *(pubblisher)*, e quest'ultime vengono recapitate a chi si è abbonato. Chi produce i messaggi non è collegato direttamente con gli abbonati, ma passa attraverso una componente intermedia denominata **dispatcher**. Il meccanismo di sottoscrizione consente ai subscriber di precisare a quali messaggi sono interessati. Questo schema implica che ai publisher non sia noto quanti e quali sono i subscriber e viceversa. Il dispatcher ha due modi per mandare i messaggi: con una **push** del messaggio a tutti gli iscritti, oppure avere un sistema di **pull**, cioè quando i subscriber si connettono al canale controllano se ci sono messaggi disponibili.

- **Model - view - controller (MVC)** è uno stile in cui si isola la logica di business dal controllo sull'input e dalla presentazione, consentendo sviluppo indipendente, test e manutenzione di ciascuno:
  - *Modello*, è la rappresentazione del dominio dei dati su cui opera l'applicazione. Quando un modello cambia il suo stato, notifica le sue viste associate in modo che si possono aggiornare;
  - *Vista*, rende il modello in una forma adatta all'interazione, in genere un elemento dell'interfaccia utente. Ci possono essere più viste per un singolo modello, per scopi diversi;
  - *Controllore*, è un qualcosa che mette in relazione la vista con il modello, riceve l'input e effettua chiamate agli oggetti del modello.



  Il modello è unico ma le viste possono essere molteplici: tabelle, istogrammi, diagrammi a torta, ... Nello stile MVC si slega il modello dalla vista, e l'utente non interagisce direttamente sul modello, ma sulle viste. Quando qualcosa cambia nel modello, in risposta a qualche azione, il modello notifica alla vista che il suo stato è cambiato. Il controller riceve le azioni dell'utente e le interpreta, in seguito chiede al modello di cambiare il suo stato;

- **Coordinatore di processi**, il coordinatore conosce la sequenza di passi necessari per realizzare un processo. Riceve la richiesta di una determinata funzione, chiama i server secondo l'ordine prefissato e fornisce una risposta. Normalmente è usato per realizzare processi complessi. I server non conoscono il loro ruolo nel processo complessivo né l'ordine dei passi del processo, ogni server semplicemente definisce un servizio. La comunicazione può essere sincrona o asincrona.

8.2.2 Viste strutturali

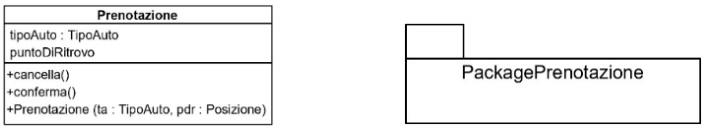
La vista strutturale indica come sono descritte le classi fisicamente e serve a:

- **Costruzione**, capire com'è costruito il codice;
- **Analisi**, aiuta la tracciabilità dei requisiti e l'impatto di eventuale modifiche;
- **Comunicazione**, se la vista è gerarchica, offre una presentazione top - down della suddivisione delle responsabilità nel sistema ai novizi.

La vista strutturale non è utile per delle analisi dinamiche fatte invece con viste comportamentali e logistiche. Nella vista strutturale, gli elementi sono:

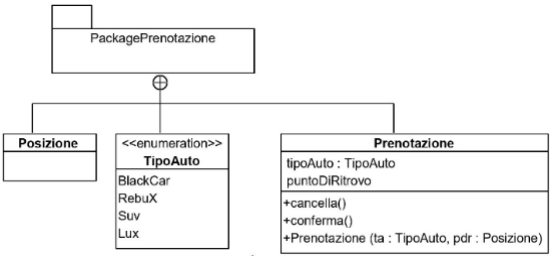
- **Moduli**, sono le unità software cioè il codice che deve essere organizzato con un insieme coerente di responsabilità. Esempio: classi, collezioni di classi, livelli;
- **Relazioni tra elementi**, le relazioni che si possono specificare tra i moduli sono diversi: *parte di, eredita da, dipende da, può usare*.

Notazione UML per documentare una vista strutturale: **classi** hanno lo stesso tipo dei diagrammi precedenti ma a questo livello ha più senso specificare le variabili e i metodi perché sono vicini al codice; **packages** sono simili alle classi ma la notazione grafica è leggermente diversa.

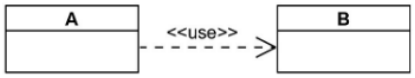


Ci sono varie viste strutturali:

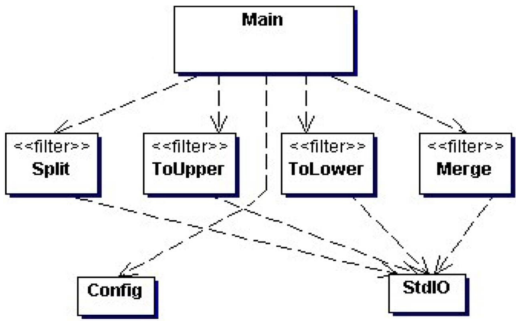
- **Decomposizione**, la relazione *parte di* corrisponde a "una classe fa parte di un package, un package fa parte di un altro package più grande". I criteri per raggruppare sono: incapsulamento per modificabilità, supporto alle scelte costruisci/compra e moduli comuni in linea di prodotto. La vista serve per l'apprendimento del sistema e come punto di partenza per l'allocazione del lavoro. La relazione *parte di* si indica con un cerchio con un più all'interno;



- **D'uso**, la relazione *usa* corrisponde a "il modello A usa il modulo B se dipende dalla presenza di B (funzionante correttamente) per soddisfare i suoi requisiti". A differenza della precedente vista che interessa la posizione, in questa vista interessa l'uso. La vista serve a pianificare lo sviluppo incrementale, e i test d'unità e di integrazione. La notazione UML per la dipendenza d'uso è semplicemente una freccia tratteggiata con lo stereotipo *use*;



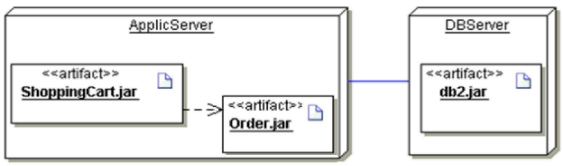
- **A strati**, è un caso particolare della relazione *usa*. Questa vista, chiamata anche macchine virtuali, è composta da strati dove uno strato è un insieme coeso di moduli. Offre una interfaccia pubblica per i suoi servizi. La relazione è antisimmetrica e non implicitamente dichiarativa. Una visione di questo tipo serve per fare delle analisi di efficienza perché la stratificazione permette la portabilità, la modificabilità e il controllo della complessità;



- **Generalizzazione**, gli elementi sono le classi o i packages e questa vista serve a rappresentare la relazione tipo - sottotipo (tra classi) o a rappresentare la relazione tra un framework (collezioni di classi, anche astratte, con relazioni d'uso tra loro) e una sua specializzazione.

8.2.3 Viste logistica di dislocazione

I componenti che sono stati individuati con la vista comportamentale vanno allocati nell'hardware. Gli elementi di questa vista sono software, quindi artefatti dove con artefatti si intende l'eseguibile legato al codice. Una volta individuata la dislocazione sulle macchine, a livello di reti, si può incominciare ad analizzare le prestazioni più in dettaglio delle varie connessioni tra i vari componenti. In questo tipo di diagramma un nodo rappresenta un nodo hardware e le relazioni tra i nodi sono connessioni fisiche o protocolli. Esistono le viste ibride dove due o più viste possono stare in un diagramma per collegare gli elementi.



Chapter 9

Principi di progettazione e qualità di un progetto

9.1 Manutenzione e riuso

Le buone pratiche e le tecniche di progettazione mirano a produrre un sistema, non solo che realizza i requisiti funzionali, ma anche che facilita i lavori di **manutenzione**, e **riusabilità**, cioè deve essere facile riusare parti del progetto in altri sistemi futuri.

9.2 Principi e pattern di progettazione

Le unità di progettazione si distinguono in due tipologie:

- **Componenti**, cioè elementi a *run time*. Un componente di un sistema ha un'interfaccia ben definita verso gli altri componenti, e la progettazione dei componenti dovrebbe facilitare il riuso e la manutenzione;
- **Moduli**, cioè elementi a *design time*. I moduli sono anche:
  - *unità di incapsulamento*, permettono di separare l'interfaccia dal corpo;
  - *unità di compilazione* quando possono essere compilati separatamente.

9.2.1 Information hiding

L'incapsulamento indica la proprietà degli oggetti di mantenere al loro interno (incapsulare) sia gli attributi che i metodi, cioè stato e comportamento dell'oggetto. L'**information hiding** permette, a differenza dell'incapsulamento, che alcuni attributi e metodi possono essere nascosti all'interno dell'oggetto, cioè sono resi invisibili agli altri oggetti. Attributi e metodi si possono "nascondere" dichiarandoli privati. Nell'information hiding interessa separare l'interfaccia visibile dall'implementazione, cioè i componenti e i moduli si possono chiamare solamente attraverso dei metodi pubblici senza sapere come funziona all'interno perché gli algoritmo usati e le strutture interne sono mantenute nascoste, quindi per l'utente la classe è solo un interfaccia. L'interfaccia di una unità rende disponibile solo gli elementi che l'interfaccia mette visibile. I vantaggi dell'information hiding sono:

- **Comprensibilità**, nessuna necessità di comprendere i dettagli implementativi di un'unità per usarla;
- **Manutenibilità** si può cambiare l'implementazione di una unità senza dover modificare le altre;
- **Lavoro in team** la separazione corpo - interfaccia facilita lo sviluppo da parte di persone che lavorano in modo indipendente, il riuso, le riparazioni e le riconfigurazioni
- **Sicurezza** i dati di una unità possono essere modificati solo da funzioni interne alla stessa, e non dall'esterno.

La modalità standard per accedere agli attributi di una classe sono con dei particolari metodi:

- **get()**, leggere un valore in un oggetto;
- **set()**, assegnare un valore in un oggetto.

Molti editori permettono di generare automaticamente set e get per ogni attributo che viene aggiunto a una classe, e ci sono almeno due svantaggi:

- potrebbero non essere necessari. Una volta introdotti altri moduli potrebbero usarle e a quel punto devono essere mantenuti;
- si potrebbe non voler permettere un accesso.

9.2.2 Astrazione sul controllo e sui dati

Il concetto di modulo è identificato con il concetto di subroutine o procedura. Una procedura può effettivamente nascondere una scelta di progetto riguardante l'algoritmo utilizzato. Per esempio gli algoritmo di ordinamento. Le procedure in quanto astrazioni sul controllo sono utilizzate come parti di alcune classi di moduli, che prendono il nome di **librerie**. Un'astrazione di dato è un modo di incapsulare un dato in una rappresentazione tale da regolamentarne l'accesso e la modifica, l'interfaccia rimane stabile anche in presenza di modifiche alla struttura dati.

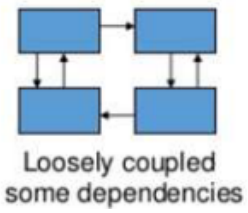
9.2.3 Coesione

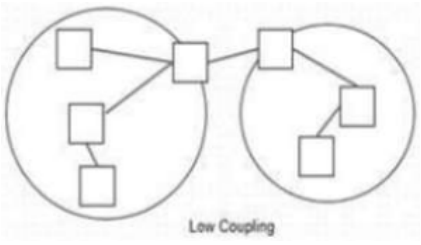
La coesione è una proprietà di una unità di realizzazione (o sottosistema) dove il sottosistema è coeso per realizzare un sola funzionalità, un concetto. Le funzionalità che riguardano lo stesso concetto devono stare nella stessa unità. L'obiettivo del progettista è creare sistemi coesi, in cui tutti gli elementi di ogni unità di progettazione siano strettamente collegati tra loro. I tipi di coesione sono:

- **Coesione funzionale**, raggruppa parti che collaborano per realizzare una funzionalità. È la situazione ideale;
- **Coesione comunicativa**, tra elementi che operano sugli stessi dati di input o contribuiscono agli stessi dati di output. Non è un buon modo di raggruppare, non supporta il riuso;
- **Coesione procedurale**, tra elementi che realizzano i passi di una procedura. Le azioni sono debolmente connesse e difficilmente riutilizzabili;
- **Coesione temporale**, tra azioni che devono essere fatte in uno stesso arco di tempo. Le azioni sono debolmente connesse e difficilmente riutilizzabili, una soluzione preferibile: dividere le azioni in diverse unità, avere una routine che manda a tutte loro un evento di avvio;
- **Coesione logica**, tra elementi che sono logicamente correlati e non funzionalmente. Le operazioni sono debolmente connesse e difficilmente riutilizzabili;
- **Coesione accidentale**, tra elementi non correlati ma piazzati assieme. È la peggiore forma di coesione.

9.2.4 Disaccoppiamento

La coesione è importante perché favorisce la manutenzione e il riuso, ma anche perché è legata all'**accoppiamento**. Si vuole valutare, nella divisione in unità, come le unità sono connesse tra di loro. Si vorrebbe che le unità siano il più slegate possibile perché se sono legate significa che ci sono delle dipendenze, scambio di messaggi e quindi essere dei colli di bottiglia. L'obiettivo del progettista è creare sistemi disaccopiati, in cui le unità di progettazione non sono strettamente legate tra loro.





I vantaggi con sistemi che esibiscono:

- un altro grado di coesione;
- un basso grado di accoppiamento.

sono il maggior riuso, migliore manutenibilità, migliore compressione del sistema e maggiore efficienza.

9.3 SOLID

Cinque principi di base di progettazione e programmazione object-oriented, dove SOLID è l'acronimo di:

- **Single responsibility principle**, una classe o un metodo dovrebbe avere un solo motivo per cambiare. La responsabilità è identificata come un motivo per cambiare. Questo principio afferma che se ci sono 2 motivi per cambiare una classe, allora significa che la classe aveva due funzionalità e quindi si deve dividere in due classi, questo perché:
  - se in futuro, si dovesse fare un cambiamento lo si fa nella classe che realizza la funzionalità;
  - se si dovesse fare un cambiamento in una classe che ha più responsabilità, le modifiche potrebbero influenzare altre funzionalità della classe e a cascata tutti i moduli che le usano.

  Fondamentalmente si sta parlando della coesione. Non si vuole fare una classe per ogni metodo, ma si vuole cercare di capire *come* e *quando* vanno divise le responsabilità. Ci possono essere degli elementi che non vanno divise perché magari non c'è la possibilità di cambiare e quindi si deve cercare di immaginare quali possono essere le modifiche future: se non ci può essere nessuna modifica allora non ha senso dividere le due classi. Ci deve essere una sorta di compromesso tra il *non dividere troppo* e il *mettere insieme elementi che non hanno nulla in comune*;
- **Open closed principle**, estendere una classe non dovrebbe comportare modifiche alla stessa. L'entità software devono essere aperte per estensione ma chiuse per modifiche, cioè l'entità si può estendere per dei cambiamenti futuri ma non si deve cambiare il codice già scritto. Le estensioni si devono effettuare con interfacce e classi astratte;
- **Liskov substitution principle**, istanze di classi derivate possono essere usate al posto di istanze della classe base. Il comportamento di un programma che usa un oggetto di tipo T deve rimanere immutato quando al posto di T si sostituisce un oggetto di tipo S che è un sottotipo di T. Le classi derivate devono potersi sostituire alle classi base;
- **Interface segregation principle**, fate interfacce a grana fine e specifiche per ogni cliente. Questo principio dice che occorre prestare attenzione al modo in cui si scrivono le interfacce, cioè mettere solo i metodi necessari e ogni metodo che si aggiunge, anche se inutile, deve essere implementato e mantenuto ;
- **Dependecy inversion principle**, si deve programmare per l'interfaccia e non per l'implementazione. I moduli di alto livello non devono dipendere da quelli di basso livello ma entrambi devono dipendere da astrazioni. Anche i dettagli dipendono dalle astrazioni. In sostanza non ci si deve mai basare su implementazioni concrete di alcuna classe ma solo su astrazioni

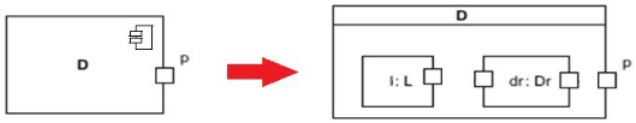
Chapter 10

Progettazione di dettaglio

10.1 Diagramma di struttura composita

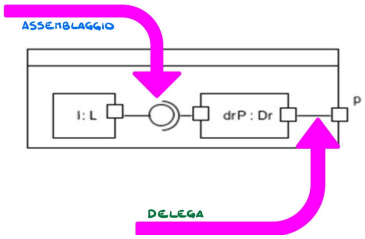
La progettazione di dettaglio consiste nella definizione di unità di realizzazione. Un sistema va definito fino a quando dettagliare ulteriormente avrebbe un costo ingiustificato o porterebbe a un'inutile esposizione ai dettagli. Il diagramma di struttura composita mostra la struttura di dettaglio di un:

- **Classificatore strutturato**, definisce l'implementazione di un classificatore data in termini di parti, porti e connettori. All'interno di un componente vi sono delle **parti** e ogni parte ha un nome, un tipo e una molteplicità *nomeParte: Tipo [molt]* ma sono tutti facoltativi. Una parte p:T descrive il ruolo che una istanza di T gioca all'interno dell'istanza del classificatore la cui struttura contiene p. La molteplicità indica quante istanze possono esserci in quel ruolo. Un **porto** è un insieme di interfacce omogenee, come nei diagrammi di componenti.



  Un **connettore** può essere di due tipi:

  - *di assemblaggio*, esprime un legame che permette una comunicazione tra due istanze dei ruoli specificati dalla struttura. Si usa la notazione lollipop;
  - *di delega*, identifica l'istanza che realizza le comunicazioni attribuite a un porto.



- **Collaborazione**, per specificarla meglio.

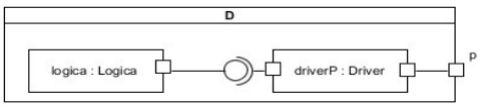
10.2 Pattern generale di strutturazione

Un modo conveniente di strutturare una componente prevede di separare gli aspetti di comunicazione da quelli di realizzazione delle funzionalità richieste. Si suppone di avere una componente D con un porto p, quindi la struttura di D dovrà avere almeno due parti:

- driverP, che realizza la parte di comunicazione richiesta per implementare il porto;
- logica, che realizza la funzionalità richiesta alla componente.

e due connettori:

- di delega tra driverP e P;
- di assemblaggio tra driverP e logica. Il verso del lollipop dipende dalla funzionalità.



La parte che realizza la funzionalità si chiama **logica**, mentre quella che realizza il porto si chiama **driver**. Una componente può avere n porti e la sua struttura minima deve prevedere:

- una parte che realizza la logica delle componente, collegata a tutti i driver con connettori di assemblaggio.
- n parti col ruolo di driver, collegate ognuno a un porto con un connettore di delega;

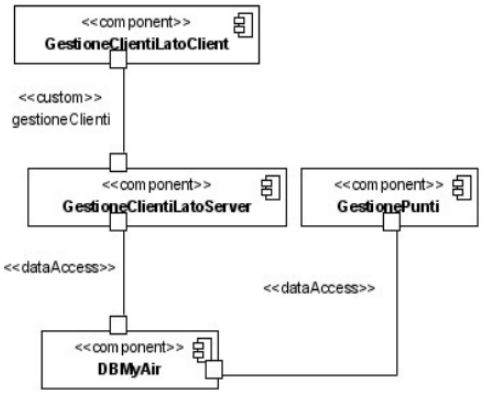
È ovviamente possibile, e normalmente necessario, dettagliare maggiormente la struttura di una componente raffinando la logica in un insieme di parti interconnesse con connettori di assemblaggio, oppure, introducendo dei **proxi** per realizzare comunicazioni con sistemi remoti o chiamate al sistema operativo. Per esempio si assume che la componente C abbia due porti verso le componenti C1 e C2, e comunichi con un sistema esterno S. La struttura C dovrà avere almeno 4 parti:

- DriverC1 e DriverC2 che realizzano la comunicazione con C1 e C2 rispettivamente;
- Logica che realizza la funzionalità richiesta alla componente;
- ProxiS che realizza la comunicazione con S.

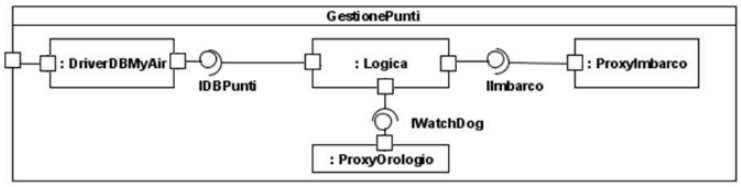
e un certo numero di connettori:

- 2 di delega tra i driver e i porti che realizzano;
- 3 di assemblaggio tra driver/proxi e logica

Si analizza un altro esempio con una vista componenti - connettori:



si suppone di dover progettare il diagramma di struttura composita del componente *GestionePunti* sapendo che la componente acquisisce informazione da un'altra componente esterna che non è presente nello schema. Vi è una connessione con *DBMyAir* quindi sicuramente ci sarà un driver e una logica, e la comunicazione con la componente esterna suggerisce che ci sarà un proxi:



- *DriverDBMyAir*, realizza l'interfaccia iDBPunti che permette a Logica di accedere tramite il solo porto della componente al DBMyAir;
- *ProxiImbarco*, realizza la connessione con il sistema imbarco, passando alla logica la lista degli imbarchi;
- *ProxiOrologio*, sveglia la logica alla data e ora richiesta tramite lWatchDog;
- *Logica*, realizza i casi d'uso sfruttando le interfacce introdotte.

Chapter 11

Introduzione alla fase e ai concetti di verifica e validazione

11.1 Problema della terminazione

Il problema della verifica di correttezza, cioè come il programma si dovrebbe comportare, è generalmente difficile, cioè non esiste un algoritmo che li risolva per tutti i possibili input. Nel 1937 Alan Turing ha dimostrato che alcuni problemi non possono essere risolti da un algoritmo, tali problemi sono quelli che coinvolgono il **problema della terminazione**: *esiste un algoritmo/programma che preso in ingresso un qualsiasi altro programma e un altro input, stabilisca se il programma su tale input termina o no?* Si conclude che non esiste un programma P, che per ogni programma Q e input D, dice se il programma Q sull'input D termina o no (**Halting problem**). Purtroppo non è solo un risultato teorico perché quasi tutte le proprietà interessanti dei programmi incorporano l'halting problem, quindi non esiste alcun programma che prende in input altri programmi e *per ognuno* di questi decide in tempo finito se è corretto o meno: esistono programmi che è possibile dimostrare corretti in tempo finito e ne esistono altri per cui ciò non è possibile.

11.2 Attività di verifica e di validazione

La verifica richiede che il sistema sviluppato sia coerente con la specifica, mentre la validazione richiede che il prodotto progettato sia quello che desiderava il cliente all'inizio del progetto Il software ha alcune caratteristiche che rendono particolarmente difficile l'attività di verifica e validazione come:

- requisiti di qualità diversi;
- il software è sempre in evoluzione;
- distribuzione irregolare dei guasti;
- non linearità sia per il tempo, sia per la grandezza dell'input.

Spesso la verifica dipende dal linguaggio usato, per esempio in un linguaggio concorrente ci possono essere problemi di *deadlock* o *race conditions* che in un programma sequenziale non persistono. Vi sono una varietà di approcci e chi progetta la fase di verifica deve scegliere e programmare la giusta combinazione di tecniche per raggiungere il livello richiesto di qualità entro i limiti di costo e di tempo. Si deve progettare una soluzione specifica che si adatta al problema, ai requisiti e all'ambiente di sviluppo. Non vi sono "ricette" fisse ma ci si deve far guidare da queste cinque domande:

- *Quando iniziare la verifica e convalida? Quando sono complete?*
  L'attività di verifica e validazione inizia non appena si decide di creare un prodotto software (o anche prima) e dura molto oltre la consegna del prodotto per far fronte alle evoluzioni e agli adattamenti alle nuove condizioni. La verifica e validazione incomincia dallo studio di fattibilità che deve tener conto delle qualità richieste e dall'impatto sul costo complessivo, e le attività correlate alla qualità comprendono:
  - analisi del rischio;
  - valutazione dell'impatto di nuove funzionalità;

  - valutazione economica sui costi e tempi di sviluppo.

  Il testing, cioè provare il codice su determinati input, non è una fase finale dello sviluppo software ed è solo una piccola parte del processo di verifica e convalida. Dopo il rilascio, le attività di manutenzione comprendono:

  - analisi delle modifiche ed estensioni;
  - generazione di nuovi test per le funzionalità aggiuntive;
  - riesecuzione dei test per verificare la non regressione delle funzionalità del software dopo le modifiche e le estensioni;
  - rilevamento e analisi dei guasti.

- *Quali tecniche applicare?*
  Nessuna singola tecnica è sufficiente per tutti gli scopi e le principali ragioni per combinare diverse tecniche sono:

  - efficacia per diverse classi di errori;
  - alcune di queste tecniche sono applicabili prima di fare il codice eseguibile, mentre altre non lo sono;
  - differenza negli scopi;
  - compromessi in termini di costo e affidabilità.

- *Come si può valutare se un prodotto è pronto per essere rilasciato?*
  Per rispondere alla domanda, possono essere di aiuto delle misure oggettive come:

  - **Disponibilità**, misura la qualità di un sistema in termini di tempo di esecuzione quando è disponibile. Per esempio se si decide che il sistema progettato deve essere disponibile il 90% delle volte, allora lo si rilascia solamente quando si riesce ad ottenere un sistema che funzioni correttamente in questi termini;
  - **Tempo medio tra i guasti**, misura la qualità di un sistema in termini di tempo tra un guasto e il successivo;
  - **Affidabilità**, indica la percentuale di operazioni che terminano con successo.

  Un altro modo per capire se il prodotto è pronto per essere rilasciato sono:

  - **Alfa test**, test eseguiti dagli sviluppatori o dagli utenti in ambiente controllato, osservati dal team di sviluppo;
  - **Beta test**, test eseguiti da utenti reali nel loro ambiente, eseguendo attività reali senza interferenze o monitoraggio ravvicinato.

- *Come si può controllare la qualità delle release successive?*
  L'attività dopo la consegna riguarda:

  - test e analisi del codice nuovo e modificato;
  - riesecuzione dei test di sistema;
  - memorizzazione di tutti i bug trovati;
  - test di regressione;
  - distinzione tra *major* e *minor* revision, per esempio 2.0 vs 1.4.

- *Come può essere migliorato il processo di sviluppo?*
  Spesso si incontrano gli stessi difetti progetto dopo progetto, quindi si cerca di identificare e rimuovere i punti deboli nel processo di sviluppo, come per esempio cattive pratiche di programmazione.

11.3 Terminologia IEEE

- **Malfunzionamento**, quando il sistema software non si comporta secondo le aspettative o le specifiche. Un malfunzionamento ha una natura dinamica: accade in un certo istante di tempo causato da uno o più difetti, e può essere osservato solo mediante esecuzione;
- **Difetto**, appartiene alla struttura statica del codice e non sempre manifesta un malfunzionamento che in questo caso si dice *latente*. Per esempio quando un difetto è contenuto in un cammino che non viene praticamente mai eseguito o quando un difetto è coperto da un altro difetto causando un effetto totale nullo;
- **Errore**, incomprensione umana nel tentativo di comprendere o risolvere un problema, o nell'uso di strumenti.

11.4 Limiti del testing

Per **optimistic inaccuracy** si intende quando in fase di testing si parte con la supposizione che il software sia corretto e si testa alcuni casi, per sostenere questa convinzione, pensando che tutto quello che si testa sia corretto. Il testing non è una attività esaustiva perché eseguire e provare ogni possibile input del programma richiederebbe un tempo infinito se gli input sono infiniti, o un tempo troppo lungo per domini di input finiti ma molto grandi. **Tesi di dijkstra**: *il test di un programma può rilevare la presenza di difetti, ma non dimostrarne l'assenza*.

11.5 Verifica statica

Oltre al testing, si esegue anche una verifica statica, cioè si analizza come è scritto il codice del programma senza eseguirlo. La verifica statica consiste nel avere dei metodi manuali, basati sulla lettura del codice e dei metodi formali basati sulla interpretazione astratta e sul model checking. Dal documento dei requisiti in poi, si può sempre avere un team, diverso da chi ha scritto il codice, che controlla quello che è stato scritto. Questo tipo di verifica manuale oltre ad essere economica, è utile per la praticità e intuitività. I metodi per questo tipo di verifica sono **inspection** e **walkthrough**, sono metodi pratici basati sulla lettura del codice, sono molto dipendenti dall'esperienza dei verificatori e sono complementari tra loro perché entrambi controllano il codice ma in due modi diversi:

Inspection:

- **Obiettivo**, rilevare la presenza di difetti eseguendo una lettura mirata del codice, guidata da una lista di controllo che viene aggiornata via via. La lista contiene solo aspetti che non possono essere controllate in maniera automatica e sono frutto dell'esperienza degli ispettori;
- **Strategia**, focalizzare la ricerca su aspetti ben definiti;
- **Agenti**, verificatori diversi dai programmatori.

Walkthrough:

- **Obiettivo**, rilevare la presenza di difetti eseguendo una lettura critica del codice;
- **Strategia**, percorrere il programma simulando l'esecuzione;
- **Agenti**, gruppi misti ispettori e sviluppatori.

Chapter 12

Progettazione delle prove

12.1 Testing

Il testing è una verifica dinamica che si compone di più fasi:

- **Progettare**, i casi di test individuando input, output atteso, ecc..
- **Definire l'ambiente di test**;
- **Esecuzione del codice**;
- **Analisi dei risultati**, cioè il confronto con l'output ottenuto con l'esecuzione e output atteso;
- **Debugging**.

Nel testing vi sono dei vantaggi come:

- **Ripetibilità**, se eseguendo un test il programma restituisce un errore, si può riproporre il test e riottenere lo stesso errore, cioè non è un evento unico;
- **Verifica di componenti vs Verifica di sistema**, il testing si può fare in livelli diversi: classe, modulo, integrazione cioè vedere se tutti i componenti hanno un comportamento atteso;
- **Test di accettazione**, cioè il test che controlla se tutto il sistema funziona correttamente come si voleva.

12.2 Caso di prova

Un caso di prova è una tripla $<$input, output, ambiente$>$ dove l'output è quello atteso mentre per ambiente si intende se l'esecuzione dipende da alcune caratteristiche su dove è eseguito il test. L'insieme di triple deve essere progettato in modo intelligente sapendo che se anche il sistema riuscisse a superare un insieme di test non è detto che sia corretto. Esistono dei criteri di *inadeguatezza*, cioè dei casi in cui i test non sono corretti:

- se il programma ha due casi, e i test si limitano a testare solamente un caso;
- se nel codice del programma ci sono n istruzioni e i casi di test ne testano solo k istruzioni con k $<$ n.

12.2.1 Test obligation

Un test obligation è una descrizione sommaria che indica quali proprietà devono avere i test. Le test obligation possono essere definiti in base a dei criteri:

- **Funzionalità (*black box*)**, basati sulla conoscenza delle funzionalità e mirati a evidenziare malfunzionamenti relativi a questi. Per esempio in una funzione che fa la somma di due numeri si possono progettare i test prima ancora prima di vedere il codice perché già si sa come funziona la somma tra due numeri;
- **Struttura (*white box*)**, basati sulla conoscenza del codice e mirati a esercitare il codice indipendentemente dalle funzionalità. Per esempio si vuole testare che si passa da ogni loop del programma almeno una volta;
- **Modello del programma**, modelli utilizzati nella specifica o nella progettazione, o derivati dal codice.

- **Fault ipotetici**, cercano bug comuni;

Un criterio di adeguatezza diventa un insieme di test obligations e una raccolta di test soddisfa un criterio di adeguatezza se:

- tutti i test hanno successo;
- ogni test obligation è soddisfatta da almeno un caso di test.

12.3 Batteria e procedura

Una batteria di prove (o *test suite*) è un insieme¡ di casi di prova. Si avrà bisogno anche di procedure automatiche per eseguire, registrare, analizzare e valutare i risultati di una batteria di prove. Una prova si conduce in questo modo:

- **Definizione dell'obiettivo della prova**;
- **Progettazione della prova**, consiste nella scelta e nella definizione dei casi di prova;
- **Realizzazione dell'ambiente di prova**, driver e stub da realizzare, ambienti da controllare, strumenti per la registrazione dei dati da realizzare. Si prova a testare il programma non per intero, ma le funzionalità specifiche.
- **Esecuzione della prova**;
- **Analisi dei risultati**, alla ricerca di malfunzionamenti;
- **Valutazione della prova**.

Tutto il codice che si deve scrivere per fare le prove si chiama **test scaffolding**, cioè del codice aggiuntivo necessario per eseguire un test. Lo scaffolding può includere:

- **Driver di test**, porzioni di codice che sostituiscono il programma principale;
- **Stub**, porzioni di codice che sostituiscono la funzione chiamata dal programma;
- **Tool per gestire l'esecuzione del test**;
- **Tool per registrare i risultati**.

12.4 Criteri funzionali

Si vogliono definire dei test obligations che si basano sulla funzionalità, e non su come è fatto il codice. La strategia è separare le funzionalità da testare e derivare un insieme di casi di test per ogni funzionalità:

- Per ogni tipo di parametro di input si individuano dei valori da testare e per questo si usano alcuni metodi:
  - **Metodo random**, generare in modo automatico un insieme grande a piacere di valori. La generazione costa zero ma gli svantaggi sono che il test non è ripetibile (se non sono memorizzati) e può essere difficile trovare l'output atteso e considerare i casi limite;
  - **Metodo statistico**, i casi di test sono selezionati in base alla distribuzione di probabilità dei dati di ingresso del programma. Il test è quindi progettato per esercitare il programma sui valori di ingresso più probabili per il suo utilizzo a regime. Il vantaggio è che, nota la distribuzione di probabilità, la generazione dei dati di test è facilmente automatizzabile ma è oneroso calcolare il risultato atteso;
  - **Partizione dei dati d'ingresso**, il dominio dei dati di ingresso è ripartito in classi di equivalenza. In questo modo se con un certo input si ottiene lo stesso risultato non ha senso testare tutti i valori che stanno nella stessa classe, ma se ne testano solo alcuni, perché si suppone che gli altri avranno lo stesso risultato;
  - **Valori di frontiera** ha la stessa idea del metodo precedente, cioè si basa su una partizione dei dati di ingresso ma oltre a testare dei valori interi, si testano anche valori estremi di ogni classe di equivalenza;
  - **Casi non validi**, per ogni input si definiscono anche i casi non validi che devono generare un errore
  - **Test basato su catalogo**, nel tempo un organizzazione può essersi costruita un'esperienza nel definire tests. Collezionare questa esperienza in un catalogo può rendere più veloce il processo e automatizzare alcune decisione riducendo l'errore umano.
- Per l'insieme dei parametri si usano tecniche che vanno sotto il nome di testing combinatorio per ridurre le combinazioni.

12.4.1 Testing combinatorio

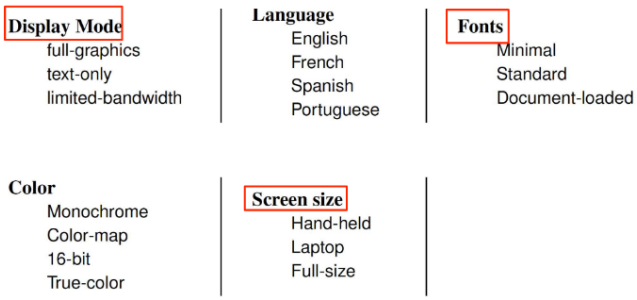
In presenza di più dati di input, se si prende il prodotto cartesiano dei casi di test individuati, facilmente si ottengono numeri non gestibili anche per tempo e costi. Occorrono strategie per generare casi di test significativi in modo sistematico, cioè tecniche per ridurre l'**esplosione combinatoria**:

- **Vincoli**, servono per ridurre le possibili combinazioni e sono di tre tipi:
  - **Di errore**, si immagina di avere un metodo con 5 parametri in input $<x_1, x_2, x_3, x_4, x_5>$
    * per x_1 e x_2 il dominio è ripartibile in 8 classi di cui una di valori non validi;
    * per x_3 e x_5 il dominio è ripartibile in 4 classi di cui una di valori non validi;
    * per x_4 il dominio è ripartibile in 7 classi di cui una di valori non validi;

    Un rappresentante per classe è $8 \cdot 8 \cdot 4 \cdot 4 \cdot 7 = 7168$ casi di test. Si cerca di diminuire i casi di test eliminando i casi di errore: viene preso un solo caso per ogni posizione (5) con input non valido quindi $5 + 7 \cdot 7 \cdot 3 \cdot 3 \cdot 6 = 2651$;
  - **Di proprietà**, si cerca di abbattere il $7 \cdot 7$ della moltiplicazione precedente, allora per x_1:
    * classe1, classe2, classe3, classe4 *[valori negativi]*
    * classe5, classe6, classe7 *[valori positivi]*
    * ~~classe8 *[error]*~~

    per x_2:
    * classe1, classe3, classe5, classe7 *[if negativi]*
    * classe2, classe4, classe6 *[if positivi]*
    * ~~classe8 *[error]*~~

    dove le classi dispari di x_2 si vanno a provare solo se in x_1 vi è un input negativo, cioè classe1, classe2, classe3, o classe4. Viceversa per le classi positive di x_2 che si provano solo con un input positivo di x_1. In questo modo la moltiplicazione si riduce in $5 + (4 \cdot 4 + 3 \cdot 3) \cdot 3 \cdot 3 \cdot 6 = 1355$;
  - **Singoletti**, per uno o più parametri si può decidere di testare un solo valore. Per esempio x_4 *[single]* la moltiplicazione diventa $5 + (4 \cdot 4 + 3 \cdot 3) \cdot 3 \cdot 3 \cdot 1 = 230$.
- **Pairwise testing** nel caso in cui il dominio non contenga in sè i vincoli precedenti è preferibile optare per la generazione di tutte le combinazioni per solo k variabili con $k < n$. Si studia l'esempio:



Se si volesse generare tutte le combinazioni per *Display mode*, *Screen size* e *Fonts* si avrebbe $3^3 = 27$. Se invece si fissano tutte le coppie tra *Display mode* e *Screen size* e poi si sistema il terzo parametro in modo da coprire tutte le combinazioni di *Fonts* · *Screen size* e *Font* · *Display mode* si avrebbe $3^2 = 9$.

Display mode × *Screen size*		*Fonts*
Full-graphics	Hand-held	Minimal
Full-graphics	Laptop	Standard
Full-graphics	Full-size	Document-loaded
Text-only	Hand-held	Standard
Text-only	Laptop	Document-loaded
Text-only	Full-size	Minimal
Limited-bandwidth	Hand-held	Document-loaded
Limited-bandwidth	Laptop	Minimal
Limited-bandwidth	Full-size	Standard

L'idea è generare tutte le coppie tra *Display mode*, *Screen size*, e poi aggiungere il terzo elemento in modo efficiente. Nella prima riga si aggiunge *minimal* perché in questo modo si hanno le coppie *(full-graphics · minimal)* e *(hand-held · minimal)*. Stesso procedimento con le altre coppie. In generale si fissano quelli con hanno cardinalità più alta e poi si sistemano gli altri elementi in maniera efficiente per coprire tutte le combinazioni. È impossibile da fare a mano per molti parametri con molti valori ma può essere fatta con euristiche.

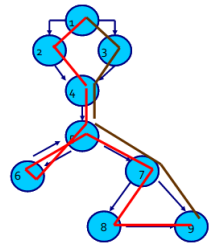
12.5 Criteri strutturali

I criteri strutturali individuano dei casi di input che si basano sulla struttura del codice (*white box*), a differenza dei criteri funzionali che si basano sulle funzionalità. I criteri strutturali aggiungono altri test e rispondono alla domanda "*Quali altri casi si devono aggiungere per fare emergere malfunzionamenti che non sono apparsi con il blackbox testing?*"

12.5.1 Copertura comandi

Si potrebbe dire che un programma non è testato adeguatamente se alcuni suoi elementi non vengono mai esercitati dai test. Dato un codice, si disegna un **grafo di flusso**, che definisce la struttura del codice identificandone le parti e i collegamenti tra loro. Nell'esempio in figura: dall'istruzione 1 si può andare all'istruzione 2 o 3; l'istruzione 4 viene fatto in ogni caso, e così via. Il flusso mostra tutti possibili cammini del programma, ma in realtà ne mostra anche di più, per esempio, non si può passare da 1 - 3 - 4 e poi passare da 7 - 8 - 9. Una volta disegnato il flusso, si vogliono coprire un certo numero di comandi, cioè avere dei casi di test. Dopo aver deciso quali test utilizzare si potrà avere una misura della copertura di comandi: $\frac{numero-comandi-esercitati}{numero-comandi-totali}$

```
double eleva(int x, int y) {
   1. if (y<0)
   2.     pow = 0-y;
   3.     else pow = y;
   4. z = 1.0;
   5. while (pow!=0)
   6.     { z = z*x; pow = pow-1 }
   7. if (y<0)
   8.     z = 1.0 / z;
   9. return(z);
}
```



- (x = 2, y = -2) si ha una copertura di 8/9 = 89% (percorso rosso);
- (x = 2, y = 0) si ha una copertura di 6/9 = 66% (percorso marrone);
- (x = 2, y = -2) e (x = 2, y = 0) si ha una copertura di 9/9 = 100%.

Esiste una copertura totale che si ottiene con solo due casi di test uno con y $<$ 0 e uno y $\geq$ 0. La copertura non è monotona rispetto alla dimensione dell'insieme di test e non sempre vale la pena cercare a tutti i costi una copertura minimale che dia copertura al 100%.

12.5.2 Copertura decisioni

La copertura delle decisioni testa tutte le condizioni del programma sia per *true* che per *false*. Si suppone di esserci dimenticati di scrivere l'else del programma dell'esempio: facendo la copertura dei comandi con (x = 2, y = -1) potremmo non accorgerci del problema perché con questi valori di x e y si esercitano tutti i comandi. Si arriva alla conclusione che testare i comandi non è sufficiente ma si devono testare anche le decisioni. Per avere una copertura delle decisioni ci devono essere almeno due casi di test uno y $<$ 0 e uno y $\geq$ 0. La misura di copertura si calcola come: $\frac{numero-archi-esercitati}{numero-archi-totali}$. Nel caso di una **condizione composta** come nel caso:

```
if (x > 1 || y == 0) { comando 1 }
      else { comando 2 }
```

si potrebbe testare il *true* e il *false* di questa condizione senza mai testare le singole condizioni. Per esempio:

- (x = 0, y = 0) e (x = 0, y = 1) garantisce la piena copertura delle decisioni, ma non esercita tutti i valori di verità della prima condizione;
- (x = 2, y = 2) e (x = 0, y = 0) esercita i valori di verità delle due condizioni ma non tutte le decisioni;
- (x = 2, y = 1), (x = 0, y = 0) e (x = 1, y = 1) esercita tutti i valori di verità delle due condizioni e tutte le decisioni.

Un insieme di test T per un programma P copre tutte le condizioni semplici di P, se, per ogni condizione semplice (*basic condition)* CS in P, T contiene un test in cui CS vale *true* e un test in cui CS vale *false*. La copertura delle basic condition è $\frac{numero-valoriverita-basicCondition}{2*n-basicConditions}$.

```
if (x > 1 && y == 0 && z > 3) { comando 1 }
      else { comando 2 }
```

La **multiple condition coverage** richiede di testare tutte le possibili combinazioni: 2^3, cioè:

- vero, vero, vero ;
- vero, vero, falso ;
- vero, falso, - ;
- falso, -, - .

12.5.3 Copertura dei cammini

Richiede di percorrere tutti i cammini. In presenza di cicli il numero di cammini è potenzialmente infinito. Per limitare il numero di cammini da attraversare si richiedono casi di test che esercitano il ciclo: zero volte, esattamente una volta o più di una volta.

12.5.4 Fault based testing

Si suppone di scrivere la funzione:

```
int foo(int x, int y) {
   if(x < y) return x + y;
   else return x*y; }
```

e la seguente batteria di test: { <(0, 0), 0>, <(2, 3), 5>, <(4, 3), 12> } che copre:

- **Criteri funzionali**, le classi di equivalenza e la frontiera;
- **Criteri strutturali**, tutte le decisioni, tutte le istruzioni;

L'idea del fault based testing è, dopo aver individuato una batteria di test, di introdurre dei difetti, dette **mutazioni**, se la test suite rileva il difetto significa che è buona. Se la batteria di test non è in grado di rilevare l'errore significa che non è valido e vanno aggiunti altri test. Se la funziona la si modifica con questo errore:

```
int foo(int x, int y) {
   if(x < y) return 3;
   else return x*y; }
```

e la batteria di test **non uccide il mutante**, cioè non rileva l'errore, significa che la batteria di test individuata precedentemente non è adeguata. In questo caso la batteria di test va bene perché con x = 2 e y = 3, si aspetta come risultato 5 e non 3. La tecnica del fault based testing si applica in congiunzione con altri criteri di test e nella sua formulazione è prevista l'esistenza, oltre del programma da controllare, anche di un insieme di test già realizzati. Esempi di mutazioni sono:

- la sostituzione di costante per costante: da (x $<$ 5) a (x $<$ 12);
- la sostituzione dell'operatore relazionale: da (x $\leq$ 5) a (x $<$ 5);
- eliminazione dell'inizializzazione di una variabile: da (int x = 5) a (int x);
- sostituzione di un operatore logico: da && a $\|$;
- inserimento di un valore assoluto: da x a $|x|$;
- e altri...

Un mutante è invalido se non è sintatticamente corretto, perché deve essere un piccolo difetto del programma ma deve consentire di passare la compilazione. Un mutante è utile se è valido e se non è facile distinguerlo dal programma originale, cioè se esiste solo un piccolo sottoinsieme di test che permette di distinguerlo dal programma originale. Trovare mutazioni che producono mutanti validi e utili non è facile, e dipende dal linguaggio. Se un mutante non viene ucciso significa che il programma è equivalente al programma originale, per esempio (x = y) con (x = y + 1 - 1), oppure la suite di test è inadeguata.

12.5.5 Individuazione degli output attesi

Trovare l'output atteso non è sempre semplice, per esempio, è inutile produrre automaticamente 10.000 casi di input se l'output atteso deve essere calcolato a mano. Ci sono tante metodologie, che dipendono dal problema, e si possono elencare come:

- Risultati ricavati dalle specifiche: formali ed eseguibili;
- Se la funzione è invertibile si può usare la funzione inversa, cioè partire dall'output per trovare l'input. Per esempio per testare un algoritmo di ordinamento prendere un array ordinato (output atteso) e rimescolarlo per ottenere un input;
- Versioni precedenti dello stesso codice;
- Invece di usare degli algoritmi complessi, si potrebbero utilizzare degli algoritmi meno efficienti e più banali solamente per calcolare il risultato;
- Semplificare i dati d'ingresso.