Dario Olianas A.A. 2014/2015

Appunti di Ingegneria del Software

(IS)

1. Introduzione

L'ingegneria del software si rivolge solo alla programmazione in the large, cioè quei progetti con periodo di sviluppo più lungo di 6 mesi, progetto seguito da team di molte persone, con milioni di linee di codice. Lo sviluppo di un progetto del genere è estremamente complesso. Ci concentreremo soprattutto sullo sviluppo di sistemi informativi/gestionali (molto richiesti dalle aziende) ma vedremo anche sistemi real-time.
Il fatto che lo sviluppo in the large sia complesso lo dicono sia i testi di IS, sia i più famosi disastri software dovuti a cattiva progettazione. Lo Standish Group è un azienda di professionisti IT specializzata nello spiegare perché un progetto software è fallito.
I problemi più riscontrati nello sviluppo software sono

- progetti cancellati
- sforamento del budget
- ritardi nella consegna
- bug
- insoddisfazione del cliente perché il prodotto non rispondeva alle specifiche

I problemi dello sviluppo non sono causati tanto dalla complessità intrinseca del problema da risolvere o da dettagli tecnici come la scelta di linguaggi e applicativi, quanto da problemi di natura umana, solitamente problemi di comunicazione tra i vari stakeholder[1] coinvolti nel progetto: il linguaggio di un manager sarà diverso da quello di uno sviluppatore, che a sua volta potrà non conoscere i dettagli del problema che deve risolvere (es. software di gestione acquedotto, centrale elettrica ecc.). Ci sono inoltre problemi organizzativi di gestione del personale (gente che cambia lavoro, va in maternità, altri problemi di gestione del personale). Un'altra difficoltà è inoltre la compresione del dominio del business (ad esempio, capire qual'è la relazione tra ordine e fattura in un sistema di fatturazione, se non so nulla di contabilità).

Tra i motivi per cui conviene seguire precise procedure per lo sviluppo di un progetto software c'è quello di evitare la nascita di **software monster:** un software che inizialmente era semplice, veloce e facilmente estendibile, a cui man mano vengono aggiunte nuove funzionalità. Ma aggiungendo funzionalità si introducono anche nuovi bug, le cui correzioni potrebbero interferire con altre parti del software, il codice diventa sempre meno leggibile e modificabile finché il prodotto non diventa un software monster: un mastodontico programma impossibile da capire e da modificare.

Disastro dell'Ariane 5: razzo esploso a 37 secondi dal lancio perché il software di controllo memorizzava un float a 64 bit in un intero a 16 bit. È successo perché l'if che controllava se il numero era troppo grande era stato rimosso per efficienza (codice nelle slides, scritto in Ada). Sebbene il disastro sia stato causato da un bug software, è stato un problema di gestione del progetto: questo codice era stato copiato pari pari dal sistema di controllo dell'Ariane 4, in cui il numero non poteva crescere troppo perché i motori erano molto meno potenti(in questo caso quindi il controllo dell'overflow era inutile).

[1] **Stakeholder**: portatore di interesse, l'insieme dei soggetti coinvolti dal progetto (lato committente)

Un altro disastro software è stato il Therac-25: macchina per radioterapia che a causa di un errore somministrò dosi letali di radiazioni ad alcuni pazienti: questo perché ci si è dimenticati di mettere nel sistema di controllo un vincolo che impedisse di somministrare la radioterapia senza applicare il filtro.

Storia dell'ingegneria del software

- **pre-1965:** il software era sviluppato in maniera estremamente artigianale: si studiavano solo algoritmi e strutture dati e spesso utente e programmatore coincidevano
- **1965-1985**: crisi del software: aumentando il numero di utenti aumentano anche i problemi; inizio della programmazione strutturata. Alla **Conferenza di Garmisch (1968),** una conferenza NATO, viene ammessa la crisi del software e viene coniato il termine software engineering per indicare la disciplina che si occupa di risolvere questi problemi
- **1985-1990**: utilizzo massivo del software da parte di industria e attività commerciali. Fred Brooks prevede che non ci sarà, nei prossimi 10 anni, nessuna silver bullet, ovvero soluzione che migliorerà notevolmente lo sviluppo di software, Ci ha visto giusto dato che di anni ne sono passati 20 e questa soluzione non è ancora arrivata. Si pensava che nuovi linguaggi più complessi avrebbero potuto migliorare il processo di sviluppo, ma questi linguaggi si sono rivelati troppo complessi per il programmatore medio.
- **1991-1999** web engineering: la nascita del web introduce nuovi modelli di sviluppo
- Dal **2000** ad oggi: metodi agili e sviluppo model driven.

1.1 Fasi del processo di sviluppo

Definizione di IS: L'ingegneria del software è una disciplina ingegneristica che si occupa di tutti gli aspetti relativi allo sviluppo di un progetto software, dalla progettazione alla dismissione, passando per la manutenzione. È un insieme di teorie, metodi, tecniche e tool che ci permettono di sviluppare del buon software. L'idea di base è che per lo sviluppo software vada usato lo stesso approccio che si usa in ingegneria civile ed industriale.
È una disciplina che dovrebbe evitarci stress, frustrazione, perdite di tempo e soldi. Per conseguire questi obbiettivi, occorre puntare sulla qualità del processo di sviluppo: spostare l'attenzione dai dettagli pratici dell'implementazione del prodotto al processo di sviluppo in generale.

L'ingegneria del software si pone come obiettivo definire metodi, strumenti e procedure per ottenere sistemi software di grandi dimensioni, di alta qualità, a basso costo, ed in breve tempo. Per raggiungerlo è necessario ottimizzare il **processo**[2] **di sviluppo** del software, seguendo il principio che con un processo produttivo di migliore qualità avremo un prodotto di migliore qualità.

Deliverable: prodotto intermedio di un processo; il deliverable di una fase è l'input della fase successiva. Ci sono diversi modelli di processo, tra cui waterfall (a cascata).

[2] **processo**: sequenza di attività da seguire per raggiungere un obbiettivo.

1.1.1 Raccolta, analisi e specifica dei requisiti

Attraverso interviste al committente, svolte da un **analista**, cerchiamo di definire bene che cosa dovrà fare il nostro sistema. Il deliverable di questa fase è la **definizione dei requisiti**: un documento, in linguaggio naturale o testo strutturato, che definisce informalmente cosa dovremo andare a sviluppare. La **specifica dei requisiti** è invece la descrizione in un linguaggio formale (come Z) del progetto che stiamo andando a sviluppare. Per aiutare l'utente a capire cosa andremo a realizzare e chiarire subito le prime incomprensioni possiamo aiutarci con degli **screen mockup**: prototipi dell'interfaccia grafica.

1.1.2 Design

Nella fase precedente si è ragionato sul cosa, in questa sul come: corrisponde al lavoro di un architetto prima della costruzione di un edificio, si va a vedere come andrà implementato quello che abbiamo in mente di fare. Può essere **high level design (HLD)** o **low level design (LLD)**: nell'HLD definiamo la struttura del sistema in termini di componenti e relazioni tra di essi, nell'LLD si definisce la struttura interna di ciascun componente. Il design dovrebbe essere il più possibile platform independent, indipendente dalla piattaforma specifica che andremo ad utilizzare, del linguaggio o degli algoritmi che useremo. Deliverable di questa fase è la specifica del design, in linguaggio UML.
La **documentazione** del nostro progetto sarà la documentazione dei requisiti, specifica di HLD e LLD. Si realizza prima l'HLD e poi LLD.

1.1.3 Implementazione

Avendo in mano i deliverable delle fasi precedenti, si procde all'implementazione in codice del progetto. Se abbiamo ricevuto un design molto dettagliato, l'implementazione sarà una fase estremamente meccanica, addirittura totalmente assente nel model driven development, in cui a partire dal modello il codice è generato in automatico. Altrimenti, diventa un'attività creativa in cui è lasciata molta libertà al programmatore (metodi agili).

1.1.4 Testing

Si esegue il software per valutare se il suo comportamento risponde ai requisiti. Si divide in unit testing (test dei singoli componenti), integration testing e system testing (test dell'intero sistema software). Al termine si stende il test report, per descrivere i risultati.

1.1.5 Delivery e manutenzione

Consegna del sistema al cliente, che comprende anche il training: insegnare all'utente ad utilizzare il software. La manutenzione è l'attività per mantenere operativo il sistema dopo la consegna all'utente tra cui correggere gli errori e introdurre nuove funzionalità.

I ruoli tecnici di ingegneria del software sono:

- analista
- designer
- sviluppatore
- tester
- trainer

Al di sopra di questi vi è il project manager, che si occupa valutazione pianificazione e controllo del progetto.

Nelle slide, grafico della ripartizione dei costi delle varie fasi dello sviluppo software.
Per giudicare la qualità di un software abbiamo due visioni possibili: quella dell'utente e quella dello sviluppatore (nelle slide i dettagli)
Nell'utilizzo dei metodi agili si da più libertà allo sviluppatore: l'importante è che il software sia funzionante, la documentazione è di importanza secondaria.

Model Driven Development
Si prepare un modello che, dato in pasto ad un software, viene tradotto in codice.

2. Modello di processo

Modellare il processo di sviluppo software è importante perché da ordine, stabilità e controllo: se abbiamo predicibilità e standardizzazione delle procedure sarà più semplice organizzarsi. E con un processo ben organizzato avremo sviluppatori più produttivi e software di migliore qualità: migliorare la qualità del processo migliora la qualità del prodotto. Questo principio viene dall'industria manifatturiera, ed è stato messo in discussione ultimamente da chi ha proposto i metodi agili, perché il software è un prodotto particolare, diverso dagli altri manufatti: è un elemento immateriale, di cui è difficile visualizzare i concetti e stabilire la qualità.

Parliamo di modello di processo e non semplicemente di processo perché ci riferiamo ad una visione astratta del processo. Ci sono modelli più astratti, che determinano solo l'ordine delle attività da svolgere, e modelli più prescrittivi che definiscono il processo con molte piu regole su cose da fare, documentazione da produrre ecc.
La terminologia dell'IS non sempre è precisa: spesso vedremo lo stesso modello chiamato in modo diverso, o spiegato in una maniera differente,

2.1 Evoluzione dei modelli di processo

Nel periodo pre-1965 della produzione di software il modello seguito era il code & fix (che non era neanche un vero modello di processo). Dopo la crisi, nel 1970 viene proposto il modello a cascata (waterfall), nel 1975 i modelli evolutivi. Nell'86 arriva il V-model, nell'88 il modello a spirale.
Nel 1999 arriva la prima rottura: si capisce che forse avere modelli troppo prescrittivi non è una buona cosa, e nascono i metodi agili tra cui l'**Extreme Programming (XP)**. Nel 2001 arriva il model driven development

2.1.1 Code & fix

Si arriva per tentativi al codice finale: si butta giù il codice senza fasi di analisi e progettazione, si esegue, se i test vanno male si fixano i bug e così via finché non abbiamo il prodotto che vogliamo. Può essere usato solo per progetti molto piccoli (<1500 LOC[3]), è totalmente inadatto su progetti grandi. Non è un vero modello di processo.

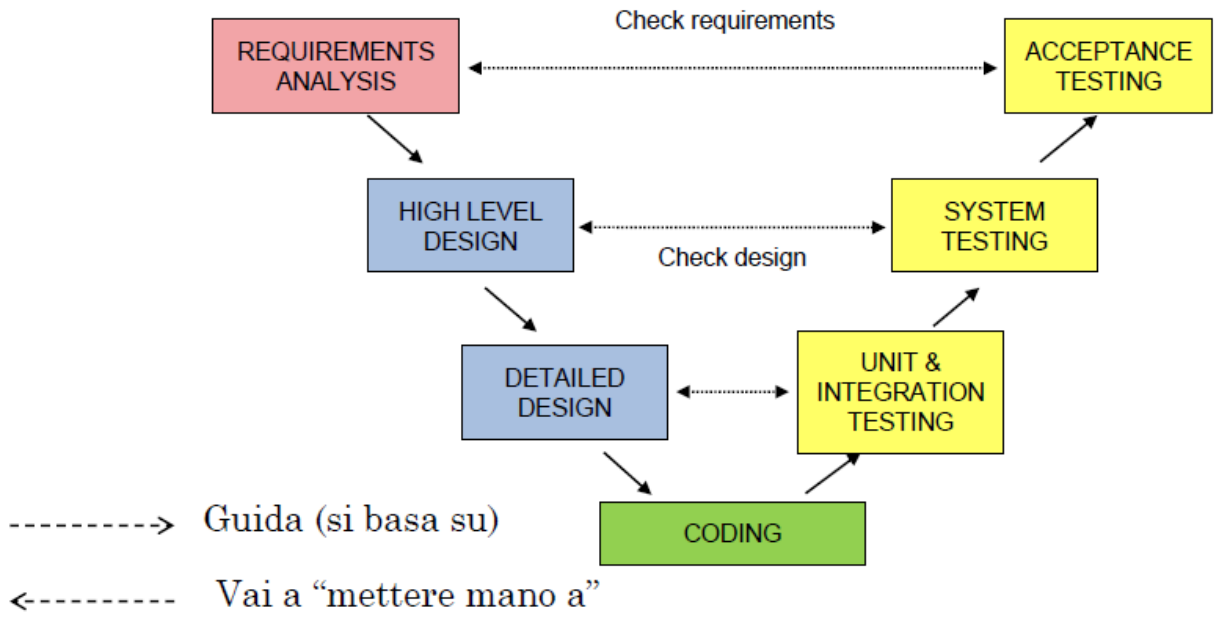
[3] LOC: lines of code

2.1.2 Waterfall

Il primo modello di processo: formalizzato nel 70, ma in precedenza era già usato dalla difesa USA (anni 50, prima presentazione del 56). è il modello che abbiamo visto la scorsa lezione, in cui il deliverable di una fase è l'input dell'altra. È un vecchio modello, con molti difetti, ed è stato abbandonato. I pregi del modello waterfall sono l'enfasi sulle fasi precedenti alla stesura del codice: analisi dei requisiti e progettazione, rimandando l'implementazione fino a quando non si sono capiti a fondo i bisogni del clienti, e il fatto che introduce disciplina e pianificazione nel processo di sviluppo. Ma ha anche molti difetti: è un modello molto rigido, che non ha feedback tra le fasi e non permette di tornare indietro, se ad esempio ad un certo punto del processo i requisiti cambiano. Questo problema è reso più grave dal fatto di avere solo una data di consegna: se il processo dura anni è molto facile che nel frattempo i requisiti cambino. Inoltre il modello waterfall crea tantissima documentazione, che può creare confusione.
Per migliorare il modello senza abbandonarlo del tutto sono state introdotte le varianti **modello waterfall con prototipazione**, in cui prima di iniziare lo sviluppo si crea un prototipo del prodotto per avere un primo feedback dall'utente, e il **modello waterfall con feedback e iterazioni**, in cui si può tornare alla fase precedente.

2.1.3 V-model

È un'altra variante del waterfall, utilizzata ancora al giorno d'oggi (la Piaggio Aero utilizza il V-Model per lo sviluppo di un software montato a bordo del suo aereo P180). Esplicita alcune iterazioni che nel modello waterfall classico sono nascoste. V-Model perché il grafo del processo è a forma di V: sulla sinistra abbiamo le fasi di progettazione e design, sulla destra le fasi di testing. Si scende tutto il ramo della progettazione, si arriva alla punta della V, che è l'implementazione, e si risale il ramo dei test. Se un test fallisce, ci rimanda alla corrispondente fase sul ramo della progettazione.



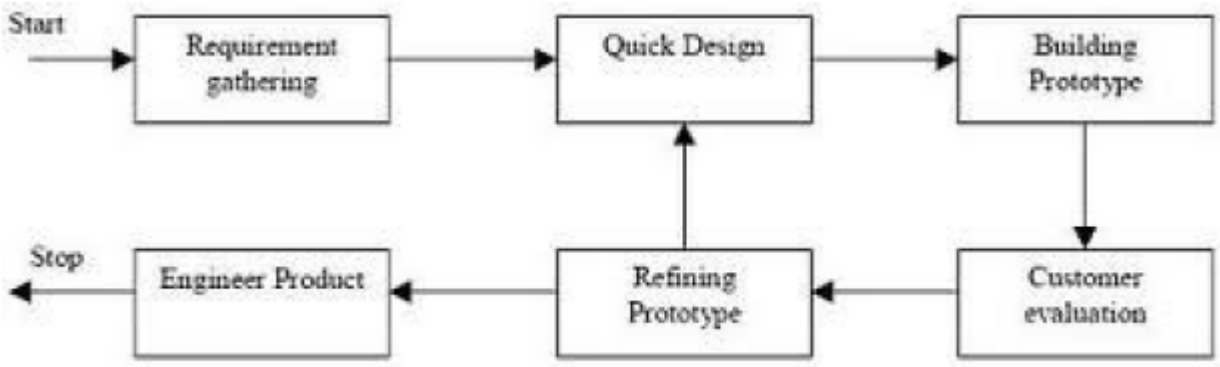
Grafo del V-Model

2.1.4 Modelli evolutivi

Con i modelli evolutivi, invece di rilasciare il software al cliente in un'unica soluzione, abbiamo un meccanismo di **release**.
Nello sviluppo software, i cambaimenti sono inevitabili: i requisiti cambiano continuamente, così come le opportunità di business, la disponibilità di nuove tecnologie, nuove idee ecc. Con i modelli evolutivi sviluppiamo un'implementazione iniziale, la rilasciamo agli utenti e la raffiniamo nelle release successive con il feedback che ci danno gli utenti, introducendo le nuove idee che abbiamo avuto ecc.
I modelli evolutivi sono i modelli a prototyping e i modelli iterativi/incrementali

2.1.4.1 Modelli a prototyping

Si sviluppa un prototipo, solitamente usa e getta, con meno funzionalità o meno efficiente di una release definitiva, giusto per avere un primo feedback del cliente. A volte il prototipo evolve nel prodotto vero e proprio, ma di solito no: è una scelta abbastanza rischiosa. Spesso il prototipo non è neanche realizzato nello stesso linguaggio che useremo per il progetto, ma in un linguaggio di alto/altissimo livello per accorciare i tempi di realizzazione.
E' conveniente sviluppare prototipi delle funzionalità su cui il cliente non sembra avere le idee chiare, di cui i requisiti sono vaghi.
Vantaggi del modello a prototipi sono la rilevazione precoce di errori nell'interpretazione delle richieste del cliente e il fatto che permette di raffinare requisiti definiti in maniera vaga. I difetti sono che comporta uno "spreco" di tempo e risorse perché il prototipo spesso è usa e getta.



Grafo del ciclo di prototipazione

2.1.4.2 Modelli iterativi-incrementali

Si ha lo sviluppo di varie release, di cui solo l'ultima è definitiva: ogni release è proposta al cliente in modo da avere un feedback da cui partire per lo sviluppo della release successiva. Supponiamo di dover sviluppare un word processor, con le funzionalità di creare testo, formattare testo e copia & incolla. Adottando un **modello incrementale**, queste funzionalità verranno implementate un po' alla volta: avremo una release che può solo creare testo, una release che può anche formattarlo e infine la release definitiva con tutte e tre le funzionalità. Se abbiamo scelto un **modello iterativo** (o di raffinamento) invece, implementiamo da subito tutte le funzionalità ma in forma ridotta. In questo modello le fasi dello sviluppo vengono ripetute più volte: nel nostro esempio potremmo avere nella prima release una semplice interfaccia a comandi, nella seconda avere un'interfaccia grafica ma basse performance e alla terza release il sistema finale. Probabilmente, nella pratica un modello iterativo puro non è mai utilizzato, sia per la difficoltà di avere un sistema completo in prima battuta sia perché non posso presentare al cliente un prodotto imperfetto, rischierei di perderlo.

Spesso viene usato un modello misto tra iterativo e incrementale, non si sceglie mai uno solo dei due.
I vantaggi dei modelli iterativi/incrementali sono l'importanza data al concetto di feedback, che aiuta gli sviluppatori ma anche gli utenti, li aiuta a capire di cosa hanno bisogno. Semplifica le varie fasi perché lo sviluppo di una release successiva è più facile avendo come base la release precedente. Si ha inoltre una risposta rapida dal mercato, senza attendere anni per terminare lo sviluppo come in waterfall e V-model.

2.1.5 Modello a spirale

È un modello risk-driven: ogni scelta è basata sui risultati dell'analisi dei rischi. È attualmente il modello più realistico per progetti di grandi dimensioni. Non è un modello a se stante, ma un meta-modello che si può applicare agli altri modelli di processo.
Rischio: circostanza potenzialmente avversa in grado di pregiudicare lo sviluppo e la qualità del software. Ogni scelta comporta un rischio, che si misura nella gravità delle conseguenze e nella probabilità che si verifichi.
Si distinguono quattro fasi:

- **planning:** si determinano gli obiettivi, le eventuali alternative (ad es. sviluppare una componente oppure comprarla), ed i vincoli, basandosi sui desideri del cliente.
- **risk analysis**: si valutano le alternative per ogni scelta identificando ogni rischio che essa comporta, e si prendono le precauzioni per ridurli il più possibile (ad esempio, se il rischio è di sviluppare un prodotto non corrispondente ai desideri del cliente perché le specifiche sono poco chiare, la soluzione sarà sviluppare un prototipo per chiarire i requisiti ambigui). La fase di risk analysis si conclude con una **go-no go decision**, uno studio di fattibilità in cui scegliamo se proseguire il progetto o abbandonarlo per i troppi rischi.
- **engineering**: lo sviluppo vero e proprio, svolto utilizzando uno degli altri modelli di processo.
- **customer evaluation**: il cliente ci da il suo feedback, valutando il prodotto finale.

Tra i vantaggi del modello a spirale c'è il fatto che è adatto allo sviluppo di sistemi complessi e che è il primo modello a considerare il rischio (risk driven). Gli svantaggi sono che la valutazione dei rischi richiede competenze di alto livello, e che è sempre possibile che un rischio non venga rilevato. Non è la panacea di tutti i mali dello sviluppo software

2.1.6 Unified Process

Nasce dall'unione di tre metodi di sviluppo proposti da Booch, Rumbaugh e Jacobson (gli stessi che hanno ideato UML). È specifico per sistemi sviluppati ad oggetti, utilizza UML ed è guidato dagli use cases, che vedremo in seguito. Comprende molti concetti del modello a spirale, come l'analisi del rischio e il fatto di essere un metamodello da integrare con altri modelli. È supportato in ogni fase da più tool grafici ed è un modello estremamente prescrittivo (secondo Ricca nesuna azienda lo segue completamente). Ogni fase dell'UP viene reiterata molte volte. Abbiamo quattro fasi: inception, elaboration, construction e transition, ognuna suddivisa in raccolta dei requisiti, analisi, design, codifica e test. Ogni fasi si conclude con una **milestone**, un prodotto intermedio.

- **Inception:** studio di fattibilità, raccolta dei requisiti essenziali per il sistema, analisi del rischio
- **Elaboration:** sviluppa la comprensione del dominio del problema, gli use cases e si definisce l'architettura generale del sistema

- **Construction:** design, codifica e testing delle release da rilasciare
- **Transition:** si mette in funzione la release nel suo ambiente, facendola testare ad utenti fidati.

2.1.7 Sviluppo basato sulle componenti (CBSE)

Primo cambio di paradigma dell'ingegneria del software, che vede lo sviluppo di un prodotto software come la compisizone di varie componenti, che possono essere già in possesso dell'azienda o acquistate da terze parti (COTS: Commercial Off The Shelf) Molto orientato al riutilizzo del software. Lo sviluppo inizia con l'analisi dei requisiti per l'identificazione dei componenti che dobbiamo usare. Potrebbe richiedere una modifica dei requisiti per poter usare componenti che abbiamo già: è un difetto perché dovremo far cambiare idea al cliente.
Tra i vantaggi ha di ridurre la quantità di software da sviluppare, riduce costi e rischi e velocizza le consegne; tra gli svantaggi ci sono la necessità di scendere a compromessi tra quello che vuole il cliente e quello che possiamo realizzare con le componenti a nostra disposizione.

2.1.8 Metodi agili

I metodi visti finora sono plan driven (o prescrittivi): vincolano a processi ben definiti in cui non c'è molta libertà, e producono molta documentazione. I metodi agili invece abbandonano la visione della programmazione come processo industriale, ma la vedono più come arte: sono pensati per fornire la flessibilità e velocità richieste dalla internet economy, in cui è fondamentale saper rispondere ai cambiamenti richiesti dal mercato in modo veloce. Da statistiche Standish Group risulta che i progetti sviluppati con metodi agili hanno più probabilità di successo.
The Agile Manifesto: nel 2001 a Salt Lake City viene scritto il manifesto degli intenti dello sviluppo agile, in cui si mettono gli individui e l'interazione al di sopra di processi e tool, si mette l'avere un software funzionante prima della produzione di documentazione, mette la collaborazione col cliente davanti alla negoziazione di contratti, si mette la risposta veloce ai cambiamenti davanti allo stendere piani.
I metodi agili non sono un ritorno all'anarchia della programmazione code & fix, ma sono anch'essi processi ben organizzati e documentati. Ne esistono molti, ma il più usato è senz'altro Extreme Programming. Dal punto di vista manageriale, tutti i metodi agili hanno queste caratteristiche:

- iterazioni rapide e consegne frequenti
- stand up meeting giornalieri: meeting di riflessione, rigorosamente in piedi, in cui si organizza la suddivisione dei compiti
- tacit/implicit knowledge: la maggior parte della conoscenza è nella testa degli sviluppatori, non nella documentaizone
- co-location (customer on site): il team di sviluppo lavora insieme all'utente, che deve essere sempre disponibile a dare chiarimenti
- la comunicazione come aspetto fondamentale

Dal punto di vista ingegneristico invece, le tecniche adottate sono

- test driven development: si sviluppano prima i test, poi l'applicazione che dovranno testare
- KISS (Keep It Short and Simple): si implementano solo le funzionalità richieste, e nel modo più semplice possibile
- refactoring continui
- pair programming: in due ad una postazione, uno scrive il codice e l'altro revisiona

- code conventions stabilite e seguite sempre da tutti, in modo che non ci siano difficoltà a leggere il codice degli altri
- utilizzo di sistemi di controllo di versione (Subversion)

I problemi dei metodi agili sono principalmente gestionali: si produce meno documentazione quindi la maggior parte della conoscenza è nella testa degli sviluppatori, ma se questi cambiano spesso (in azienda c'è un alto turnover) questo può creare problemi. È inoltre poco adatta a sviluppatori junior, che lavorano meglio con metodi più prescrittivi, e richiede clienti molto dipsonibili a darci continui chiarimenti. Ci sono inoltre problemi contrattuali, dato che i requisiti da soddisfare non sono definiti in maniera formale.
Approcci plan driven e approcci agili si trovano meglio se giocano in casa: per sistemi grandi e complessi e safety critical (centrali nucleari, aerei ecc.), con requisiti stabili e da usare in un ambiente predicibile useremo metodi plan driven, se invece progettiamo un sistema piccolo, da usare in un ambiente con requisiti volatili, abbiamo team di molta esperienza e tempi di consegna brevi ci converrà usare metodi agili. Per i casi dubbi, un ottimo riferimento è il libro **Balancing agility and discipline (B. Boehm-R.Turner).**

3. Analisi dei requisiti

Un requisito è una descrizione di qualcosa che il sistema dovrà fare, o di un vincolo che il sistema dovrà rispettare durante l'esecuzione dei suoi compiti, e che può essere data su diversi livelli di astrazione.
Quella della specifica dei requisiti è una delle fasi più critiche dello sviluppo: secondo un sondaggio Standish Group del 1995, tra i motivi del fallimento di un progetto software la maggior parte sono relativi ai requisiti: circa il 44% dei progetti software falliti ha avuto problemi nella fase di specifica dei requisiti. Per Fred Brooks (premio Turing 1999) "*La parte più complessa nella realizzazione di un sistema è decidere cosa realizzare: nessuna parte del lavoro pregiudica maggiormente il risultato se viene eseguita in modo errato e nessuna altra parte è più difficile da correggere successivamente*".
Nelle slides possiamo vedere come aumenta il costo di rimuovere un difetto man mano che si va avanti nello sviluppo di un progetto: corregere un difetto in fase di specifica dei requisiti costa 1, in fase di desgin 5, in codifica 10 e ben 200 se dobbiamo correggere un bug dopo la consegna al cliente.

3.1 Suddivisione dei requisiti: utente e sistema, funzionali e non funzionali

I requisiti si possono dividere per livelli di dettaglio (suddivisione proposta da Alan Mark Davis): si distingue tra i **requisiti utente**, una specifica in linguaggio naturale dei bisogni dell'utente, e i **requisiti di sistema**, una specifica molto dettagliata, scritta per lo sviluppatore, di cosa il sistema dovrà fare. Potranno essere scritti in linguaggio naturale (normale o strutturato attraverso form e template) ma anche attraverso linguaggi di specifica algebrici come il linguaggio Z. I requisiti utente spesso sono usati come forma di contratto tra cliente e software house. Naturalmente, dev'esserci consistenza tra requisiti di sistema e requisiti utente (in un documento potranno anche esserci riferimenti all'altro).

E’ importante che i requisiti utente non contengano termini tecnici, ma termini del dominio applicativo per cui stiamo sviluppando: dev’essere comprensibile al cliente.
Nelle slides esempio di requisito utente e di sistema che dice come deve svolgersi l'aggiunta di un nodo in un editor grafico per modelli di design.

Requisiti funzionali/non funzionali
I requisiti funzionali sono quelli che hanno a che fare con le funzionalità del sistema che andremo a implementare, ad es. per un sistema bancario, *il sistema dovrà permettere la consultazione del saldo* è un requisito funzionale. I requisiti non funzionali invece non sono direttamente collegati con le funzionalità da implementare, ma si riferiscono a modalità operative e vincoli, come ad esempio tempi di risposta, piattaforme supportate, scelta dei linguaggi, risorse richieste, strumenti e tecniche di implementazione varie. I requisiti non funzionali possono essere ulteriormente suddivisi tra loro in requisiti di prodotto (efficienza, usabilità, portabilità, affidabilità), requisiti di organizzazione (tempi di consegna, standard da adottare) e requisiti esterni (etici, come ad esempio la scelta di usare solo software libero, legali o di interoperabilità).

3.2 Requirements engineering
Il requirements engineering è il termine che definisce le attività necessarie per documentare, raccogliere e tenere aggiornato l'insieme dei requisiti di un sistema software: i requisiti infatti non vengono buttati via una volta finita la prima fase di progettazione, ma vengono tenuti e aggiornati per esigenze future. Scopo primario del RE è la produzione e il mantenimento di un requirement document.
Il procedimento con cui si genera il requirement document è iterativo: inizia con la requirements elicitation (raccolta dei requisiti), seguita dall'analisi dei requisiti, specifica dei requisiti e validazione dei requisiti, solitamente compiuta da persone esterne con un meccanismo di peer review:

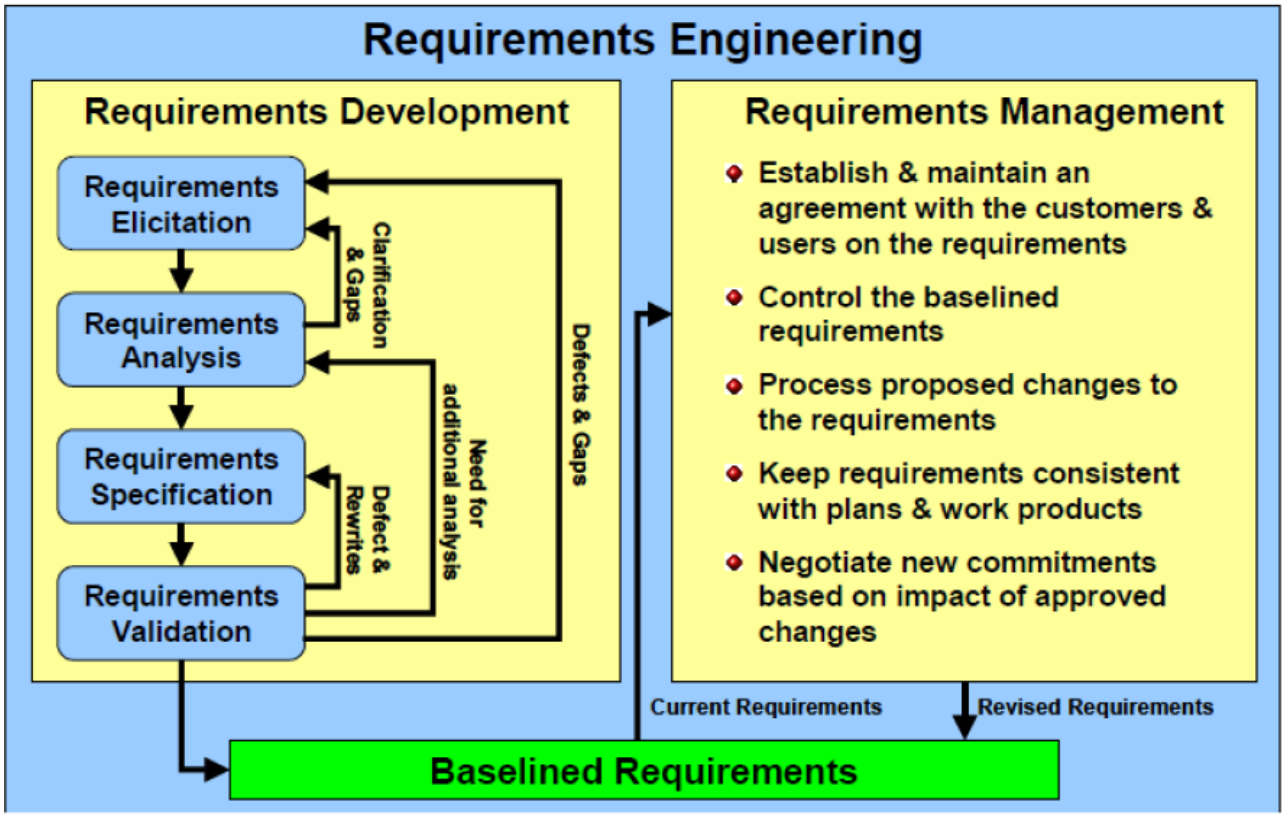


Grafico delle iterazioni richieste per generare il documento dei requisiti

Requirements elicitation
L'elicitation è la fase in cui si "tirano fuori" i requisiti dal cliente e da altri stakeholders. Il primo passo di questa fase è stabilire chi sono gli stakeholders, ovvero tutti quelli che hanno interessi nel sistema che andiamo a sviluppare: una volta fatto, l'analista andrà a intervistarli per determinarne i bisogni. Altri metodi integrativi alle interviste sono l'osservazione diretta sul luogo di lavoro, l'analisi dei prodotti dei competitors e workshop (brainstorming) per cercare di chiarire i requisiti. Spesso questa fase prevede anche l'analisi di leggi e regolamenti dell'ambiente in cui il nostro sistema andrà ad operare, report degli help desk e change requests per prodotti simili.

Analisi dei requisiti
I bisogni degli stakeholders, raccolti nella fase precedente, vengono raffinati e analizzati per capire se sono tutti fattibili e se c'è qualche requisito necessario che non è stato specificato nelle fasi precedenti (**missing requirements**). In questa fase si identificano anche i conflitti tra i requisiti, e se ne stabilisce la priorità.
Un esempio di requisiti in conflitto, nel caso del progetto di un generico sistema hardware potrebbero essere questi:

- al fine di minimizzare il consumo di energia si deve minimizzare il numero di chip preferendo quelli a basso consumo
- si devono garantire tempi di risposta molto rapidi

Questi due requisiti saranno quasi sicuramente stati proposti da due stakeholder diversi, ad esempio il primo da un ingegnere che deve implementare il sistema e il secondo da un manager. Il conflitto si risolve mediando tra gli stakeholder al fine di arrivare ad una soluzione che tenti, per quanto possibile, di soddisfarli entrambi. Se non tutti i requisiti sono possibili da soddisfare entra in gioco la **priorità dei requisiti**, per sapere quali risolvere per primi e quali è possibile tagliare. Se nelle fasi successive adotteremo un metodo di sviluppo incrementale, inoltre, la priorità dei requisiti ci dice in che ordine implementare le funzionalità.
Ci sono due strategie per organizzare la priorità dei requisiti: scala numerica e scala MoSCoW
La scala MoSCoW suddivide i requisiti in

- **MUST**: da soddisfare assolutamente
- **SHOULD**: andrebbero soddisfatti, ma se questo causerebbe altri problemi, o siamo in ritardo con la consegna ecc. possiamo trascurarli
- **COULD**: da implementare solo se rimangono tempo e risorse dopo aver risolto tutto il resto -> quasi mai implementati

Definizione e specifica dei requisiti
Stesura vera e propria del documento, suddiviso in **definizione** dei requisiti (per il cliente, requisiti utente) e **specifica** dei requisiti (per lo sviluppatore e, se serve, per i tecnici del cliente, requisiti di sistema). In fase di **validazione** invece i requisiti vengono analizzati minuziosamente per vedere se sono ben scritti, completi e se ci sono ambiguità. Generalmente svolta da professionisti non coinvolti nella stesura originale dei requisiti, con un meccanismo di **peer review**, può anche essere svolta con lo sviluppo di prototipi.

Requirements management
Il requirements management è la gestione dei requisiti nelle fasi successive dello sviluppo: potrebbe infatti essere necessario aggiornarli, e per ogni aggiornamento è necessaria l'approvazione a seguito di una impact analysis, per vedere quanto la modifica richiesta impatterà sul resto del progetto.

Proprietà dei (buoni) requisiti

- **validità e correttezza:** dobbiamo chiederci se i requisiti che abbiamo descritto sono veramente conformi ai bisogni dell'utente e se ce ne sono di troppo.
- **consistenza:** non ci sono requisiti incompatibili tra di loro, il documento è scritto con chiarezza (ad esempio: ogni volta che utilizziamo una parola ha lo stesso significato, non ci sono ambiguità).
- **completezza:** tutto ciò che vuole il cliente è coperto dai requisiti (questo avviene molto raramente).
- **realismo:** tutto ciò che è richiesto è implementabile (qui viene in aiuto l'informatica teorica che ci dice cosa possiamo/non possiamo calcolare).
- **inequivocabilità:** una sola interpretazione possibile
- **verificabilità:** dev'essere sempre possibile eseguire un test per verificare i requisiti (ad es. un vincolo di tempo sarà non "la risposta deve essere immediata" ma "la risposta deve avvenire in X millisecondi").
- **tracciabilità:** dev'essere possibile risalire da una funzionalità al requisito che l'ha generata in modo semplice (nelle slide un esempio di matrice di tracciabilità).

Analista software
La figura professionale dell'analista (generalmente è un quadro, ci si arriva dopo essere stati senior developer). deve avere competenze non solo tecniche ma anche psicologiche ed economiche: è quello che deve negoziare col cliente. La comunicazione è estremamente importante perché spesso si trova a dover relazionare realtà molto diverse: il linguaggio utilizzato da un informatico è molto diverso da quello utilizzato da un manager o da un tecnico di un altro settore. In particolare, è necessario prestare molta attenzione a termini con molto significati, significati sottintesi, termini tecnici (da evitare per quanto possibile) ecc.

I problemi principali di un documento di requisiti sono la mancanza di chiarezza, la verbosità,i termini tecnici (da evitare), ambiguità ed hidden assumptions (affermazioni date per scontate ma che potrebbero non esserlo per chi viene da un contesto diverso).

Tra le buone pratiche per produrre dei buoni requisiti c'è il riuso dei requisiti (quando abbiamo requisiti già stesi adatti alla nostra situazione), l'utilizzo di un glossario comune tra clienti, utenti e analisti, l'utilizzo di template e form e l'utilizzo di software per la gestione dei requisiti.

Esistono diversi modi di rappresentare i requisiti, che vanno dal non scriverli per nulla all'utilizzo di linguaggi formali. Nei metodi agili si utilizzano le **user stories**, semplicissime descrizioni di quello che vogliono gli utenti (stanno su un post-it).

Per condurre una buona intervista è necessario fare molte domande al cliente, in modo da non lasciare ambiguità, in particolare, ogni volta che il cliente utilizza un quantificatore universale (tutti, nessuno, ogni ecc.) è necessario farselo chiarire. Ad es., per un sistema di una biblioteca, se

lo stakeholder dice "tutti devono avere una tessera per prendere in prestito i libri" bisogna subito chiedergli "non esistono utenti che possono operare senza tessera?"

Prima di scrivere i requisiti, conviene rappresentarli sotto forma di mind map, un buon tramite tra le idee astratte dell'utente a un documento strutturato dei requisiti.

3.3 Use cases

Proposti da Ivar Jacobson (uno dei tre di UP e UML) nel 1992, i casi d'uso sono una tecnica per esprimere i requisiti funzionali di un sistema (tutti i requisiti non funzionali non possono essere espressi con use cases). Descrivono i requisiti dal punto di vista dell'utente del sistema, senza preoccuparsi di come sarà implementato: il sistema è visto come una black box, qualcosa che esegue un compito ma non ci interessa come: viene espressa solamente l'interazione tra l'utente e il sistema.
Sono slegati dal mondo object oriented e da UML: volendo si può passare direttamente dagli use cases alla codifica. È una tecnica non visuale: gli use cases sono espressi come testo strutturato, anche se poi tutti gli use cases relativi ad un certo progetto vengono rappresentati su un diagramma.
Sono utilizzati come base per la fase di design ma anche per il testing (testiamo l'applicazione nel modo in cui la userà l'utente).

3.3.1 Componenti del modello dei casi d'uso

Nel modello ci sono quattro componenti:

- **attori**: persone e cose che utilizzeranno il sistema. Rappresentano un ruolo nel nostro sistema e si suddividono in **primari** e **secondari**: gli attori primari sono quelli che guadagnano qualcosa dall'utilizzo del sistema (nell'esempio delle slides, sia il cliente che l'agente sono attori primari). Gli attori secondari sono invece quelli che vengono utilizzati dal nostro sistema, chi produce qualcosa od offre un servizio che verrà usato da un sistema: ad esempio, per un sito di e-commerce, PayPal è un attore secondario.
- **casi d'uso**: quello che gli attori possono fare col sistema (rappresentato con una frase in camel case racchiusa in un ovale)
- **relazioni** tra gli attori e i casi d'uso
- **confini del sistema:** rettangolo che mostra cosa sta all'interno del sistema (e quindi dovremo svilupparlo noi) e cosa sta fuori (e quindi esiste già). Quando vogliamo costruire un sistema software la fase di definizione dei confini è fondamentale, perché avrà poi un grosso impatto sulla fase dei requisiti: molti fallimenti di progetti software derivano da un cattivo posizionamento dei confini del sistema.

Questi elementi vengono rapprsentati su un grafico (esempio su slides) a cui segue una descrizione testuale dei casi d'uso.

3.3.2 Scenari

Uno scenario è una sequenza ordinata di interazioni tra il sistema ed un insieme di attori. Ad esempio, nella progettazione di un bancomat lo scenario d'uso sarà quello mostrato nelle slides: da il benvenuto al cliente, chiede il numero di conto, il cliente inserisce il numero di conto, chiede il PIN, il cliente inserisce il PIN ecc. Lo scenario rappresenta come vanno le cose in un caso specifico, non ci son diramazioni (ad esempio, se il cliente inserisce un PIN sbagliato siamo in uno scenario diverso rispetto a quello in cui il PIN è corretto).
Rappresentano quindi un'istanza di uno use case, un singolo cammino attraverso di esso, e per ogni use case andranno analizzati tutti gli scenari possibili perché è su questi che si baseranno i progettisti.
Se uno scenario è un'istanza di un caso d'uso, allora un caso d'uso rappresenta un'insieme di scenari che hanno in comune lo scopo finale dell'utente: un caso d'uso può essere visto come un'albero, con diramazioni ogni volta che le cose possono andare diversamente, e lo scenario sarà un singolo cammino attraverso il caso d'uso.

3.3.3 Descrizione dei casi d'uso

Poiché stiamo rappresentando requisiti utente, dovranno essere descritti in un linguaggio comprensibile all'utente, utilizzando il vocabolario del dominio dell'applicazione, evitando termini tecnici poiché in questa fase siamo ancora indipendenti dalle tecnologie che useremo.
Descriveremo uno scenario principale, quello in cui tutto va come previsto, ed una serie di scenari secondari che descrivono cosa può succedere di diverso e in che modo reagirà il sistema a queste situazioni.
Esistono Use Case Template, pattern che le descrizioni degli use cases dovranno seguire.

Di solito specifichiamo

Nome dello use case: quello che il caso d'uso dovrà fare, scritto in UpperCamelCase
Identificatore: numero progressivo
Livello di astrazione: può essere

- **Summary (o kite, aquilone) level Use case:** descrive gli scopi principali del sistema, nel modo più generale possibile, ad esempio GestioneLibri, e vengono suddivisi poi in user-goal Use case
- **User-goal (sea) Use case:** descrivono l'obbiettivo (goal) che un attore primario ha sul sistema, come AggiungiLubro
- **Subfunction (fish) Use case:** forniscono supporto operativo agli user-goal

Breve descrizione
Attori primari
Attori secondari
Precondizioni: eventuali vincoli sullo stato del sistema
Sequenza eventi principale: un elenco di passi concisi, numerati e ordinati temporalmente. Oltre ai passi possiamo avere anche deviazioni e ripetizioni: le deviazioni possono essere semplici (if) o complesse, che portano a descrivere un altro scenario: in genere rappresentano errori o casi particolari che non riportano sullo scenario principale.
Per eventi come errori o annullamento dell'utente le best practices prescrivono di passare ad un altro use case/sequenza eventi alternativa, se invece è un cambio piccolo conviene usare se/altrimenti.
Sequenza eventi alternativa
Postcondizioni

È possibile, nel diagramma degli use cases, inserire dei ruoli astratti con specializzazioni. Come nel paradigma OO, le specializzazioni di un ruolo astratto ereditano le relazioni del genitore.
È possibile anche introdurre relazioni tra i casi d'uso, tra cui una relazione molto importante è l'**include**: serve a decomporre in parti più piccole un caso d'uso che altrimenti sarebbe molto complesso. L'**extend** invece mostra un comportamento opzionale di uno use case, ad esempio per lo use case RestituisciLibro di una biblioteca potrebbe esserci l'extend AddebitaMulta se la restituzione è in ritardo.
Esistono anche gerarchie di generalizzazione/specializzazione.

L'include è simile alla chiamata a funzione in un programma: un caso d'uso principale ad un certo punto passa il controllo ad un caso d'uso incluso, che ritornerà al princiaple una volta terminato. Il caso d'uso principale sarebbe incompleto senza il caso d'uso incluso, e si indica con un passo che dice include(CasoDUsoIncluso).

Nell'extend invece, il caso d'uso base è completo anche senza l'extend, mentre il caso d'uso esteso da solo è incompleto: ad un certo punto della sua esecuzione fornisce un punto di estensione al caso d'uso esteso, su cui però il caso d'uso principale non sa niente. A seconda dei casi, può essere indifferente utilizzare un extend o una sequenza eventi alternativa (in un contesto reale ad esempio, l'esempio della multa per il ritardo nella consegna di un libro probabilmente verrebbe rappresentato con una sequenza eventi alternativa). Sono cose che dipendono dalle scelte dell'analista.

Le gerarchie di generalizzazione/specializzazione rappresentano dei casi d'uso più specifici di un caso d'uso più generale, ad esempio TrovaCD può essere una specializzazione di TrovaProdotto.

Per rappresentare nella sequenza eventi i passi modificati, aggiunti o ereditati esistono convenzioni, vedi slides.

Ma se abbiamo detto che i casi d'uso devono essere comprensibili all'utente, perché ci sono anche questi costrutti non proprio immediati da capire? Infatti, le best practices dicono di usarli il meno possibile, in particolare extend e generalizzazioni (più complessi dell'include). Starà poi all'analista decidere se e quando è indispensabile usarli.

Screen mockups
Per migliorare la compresione dei casi d'uso si possono mostrare degli screen mockup all'utente, schizzi di come sarà l'interfaccia del prodotto ad un certo passo del caso d'uso. Sono utili per il cliente ma anche per lo sviluppatore, aiutano a migliorare la compresione dei bisogni del cliente. Possono essere fatti su carta o usando tools.

Come scrivere buoni use cases

- mantenerli brevi e semplici: la sequenza eventi principale non dovrebbe superare una pagina
- extend e generalizzazioni solo quando semplificano il modello
- non confondere il cosa col come: evitare di includere dettagli tecnici o scelte di design (ad esempio, scrivere "L'utente cinferma la scelta" e non "L'utente clicca OK")

- evitare l'eccessiva scomposizione funzionale

Gli use cases sono un mezzo potente per esprimere i requisiti funzionali, aiutano le fasi di elicitation e testing, molto usati in pratica per sistemi informativi e gestionali e fanno parte del modello di sviluppo UP. Esistono svariati template di use cases, più o meno formali.
Ci sono anche dei contro nell'utilizzo degli use cases: la cosa che più viene contestata agli use cases è che rischiano di portare a costruire un sistema non object-oriented, sono più vicino al modo di ragionare procedurale. Sono inoltre poco adatti a sistemi dominati da requisiti non funzionali, con poca interazione con l'utente e molti algoritmi complessi.

4. Design

Il design è il processo che trasforma un problema in una soluzione. Nella fase precedente ci siamo preoccupati di definire in maniera esaustiva, chiara e non ambigua i requisiti: ma i requisiti definiscono solo il problema, ci dicono che cosa andremo a risolvere. Il design invece propone una soluzione, definisce come andremo a risolvere il problema specificato dai requisiti. A differenza dell'implementazione, in cui la soluzione viene attuata, il design si occupa solamente di definirne la struttura: facendo il paragone con la costruzione di una casa, il design è il lavoro dell'architetto, l'implementazione è l'esecuzione dei lavori.
Anche il design si sviluppa per raffinamento: abbiamo una fase di **design architetturale** (detto anche **system** o **high-level** design) che definisce la struttura generale del sistema, mappando i requisiti su componenti e sottosistemi, ed una fase di **component design** (detto anche **subsystem** o **low-level** design) che specifica la struttura dei singoli componenti, definendone meglio i dettagli.
Il design può essere indipendente dalla piattaforma (platform independent) o legato alla piattaforma (platform specific). Il design platform independent ha il vantaggio di essere riusabile e lo svantaggio di essere più astratto: lascia più lavoro allo sviluppatore.

La maggior parte del lavoro di design è un lavoro di routine, in cui si cerca di riadattare per il nostro contesto soluzioni a problemi simili. Ci sono tre livelli di riuso: la clonazione, in cui si copia integralmente design e codice di un'altra soluzione introducendo solo piccoli aggiustamenti; il design pattern, in cui si usa una soluzione generica per un problema comune; e gli stili architetturali.
In fase di design è molto importante l'esperienza: un buon designer deve aver visto svariate soluzioni ed essere sempre informato sulle nuove tecnologie in modo che, di fronte ad un problema, possa sempre ricondursi ad una soluzione che conosce.

4.1 Design architetturale

Il design architetturale è la fase del processo di design che identifica le macrocomponenti di un sistema, l'eventuale framework di controllo e definisce come le varie componenti comunicheranno tra loro. Deliverable di questo processo è l'architettura software.

L'avere un'architettura software porta diversi vantaggi:

- guida e semplifica lo sviluppo poichè è più facile sviluppare componenti distinti piuttosto che un unico macrosistema.
- aiuta le decisioni manageriali, perché avremo il personale suddiviso in team più piccoli.
- semplifica l'analisi di alcune proprietà, come la manutenibilità.
- aumenta anche le possibilità di riuso, perché possiamo riutilizzare una stessa architettura in sistemi simili
- previene lo creazione di software monsters
- permette di capire le dipendenze tra le componenti.

I componenti di un'architettura software si dividono in sottosistemi e moduli: un **sottosistema** è una parte del sistema che può essere eseguita e utilizzata anche da sola, in grado di svolgere le sue operazioni indipendentemente dagli altri sottostitemi, e generalmente è suddiviso in moduli. Un **modulo** è la parte di un sistema che viene utilzzato dalle altre componenti, ma che non può essere eseguita da solo. Spesso però questi due termini vengono utilizzati come sinonimi.
L'architettura software normalmente si esprime con un **diagramma a blocchi**, dove i blocchi rappresentano le componenti e i connettori le relazioni tra essi. Esistono anche linguaggi formali per rappresentare le architetture software, molto poco utilizzati. Il linguaggio UML fornisce il **diagramma delle componenti**, che ha un po' di sintassi in più rispetto ad un semplice diagramma a blocchi.
L'architettura software ha un grosso impatto su alcune proprietà del sistema: a seconda di quale scegliamo privilegeremo o penalizzeremo alcune proprietà, quindi baseremo la scelta della nostra architettura su quali proprietà ci servono di più. Se ad esempio avremo bisogno di performance elevate, privilegeremo componenti large-grain (poche e grandi) piuttosto che fine grain (molte e piccole), per ridurre al minimo le comunicazioni tra componenti (che costano in termini di tempo). Se invece puntiamo più sulla maintainability ci conviene organizzarci con componenti fine-grain, più facilmente sostituibili. Queste scelte sono quasi sempre dei trade-off[4]: migliorando una caratteristica andiamo a peggiorarne un'altra. Bisogna quindi fare molta attenzione e sopratuttto avere ben chiaro quali solo le caratteristiche prioritarie per il sistema che andiamo a sviluppare.

4.1.1 Stili architetturali

Un'architettura software può conformarsi ad uno o più stili architetturali: conoscere questi stili ci semplifica il design dell'architettura perché potremo adattarci a modelli esistenti invece di progettare da zero. Uno stile architetturale è definito dal tipo delle componenti architetturali di base (procedure, filtri, livelli, modalità di servizio, package) e dal tipo dei connettori, che possono essere invocazioni di procedure/metodi, pipes, eventi, e tutto quello che mette in connessione diverse componenti architetturali di base.

Pipe and filters
Stile in cui le componenti sono dei filtri, elementi che prendono un input e lo trasformano per produrne un output; i connettori sono pipe che prendono l'output di un filtro per darlo in input a un altro. Il più famoso esempio di design pipe&filters è la shell UNIX.
I vantaggi di questo stile sono che supporta il riuso delle trasformazioni, è molto intuitivo da capire, semplifica l'aggiunta di nuove trasformazioni e soprattutto supporta l'esecuzione concorrente. Ha lo svantaggio però di essere inadatto per sistemi interattivi, si adatta meglio a

[4] **Trade-off:** situazione in cui la scelta di avvantaggiare un'opzione ne penalizza un'altra. Termina che deriva dall'economia.

sistemi batch: inoltre ogni filtro deve fare il parsing dell'output del filtro precedente e questo può essere inefficiente.

Object Oriented Organization
Stile architetturale in cui le componenti sono oggetti e i connettori sono invocazioni a metodi. Gli oggetti si mostrano all'esterno tramite interfacce, nascondendo le implementazioni (information hiding). Questo porta il vantaggio di poter modificare le implementazioni senza che il mondo esterno alla classe modificata se ne accorga, e inoltre semplifica il lavoro di design perché gli oggetti posso riflettere entità del mondo reale: modelliamo la soluzione utilizzando gli oggetti che compongono il problema, senza troppe astrazioni.
Tra gli svantaggi abbiamo che per interagire tra loro gli oggetti devono conoscere le rispettive interfacce: la modifica di un interfaccia comporta la modifica di tutti gli oggetti che la usano. Inoltre, le entità complesse sono difficili da rappresentare come oggetti: è una granularità troppo fine.

Stile architetturale stratificato (layered)
Questo stile organizza il sistema su vari livelli, ognuno dei quali fornisce una serie di servizi, e ogni livello si appoggia esclusivamente sul livello direttamente inferiore (questo in teoria: nella pratica spesso e volentieri vengono fornite shortcuts ai livelli inferiori per migliorare l'efficienza). È molto usato per la progettazione di sistemi operativi. Il suo principale svantaggio è l'efficienza: passare attraverso tutti gli strati riduce le performance del sistema. Per questo spesso vengono fornite shortcut per saltare qualche livello.
Ha l'enorme vantaggio di semplificare lo sviluppo, perché ci permette di non interessarci di tutti i livelli di astrazione, ma solo di quello su cui stiamo sviluppando e quello immediatamente inferiore: chi sviluppa un'applicazione non deve preoccuparsi dell'interazione con l'hardware. Inoltre supporta il riuso: diverse implementazioni di uno stesso livello possono essere usate in modo intercambiabile, e i cambiamenti apportati su un livello influenzano solo il livello immediatamente superiore.

Repository model
In questo stile architetturale ci sono vari sottosistemi che condividono i dati su un database centrale o repository (detto anche blackboard). Molto usato nei CASE[5] tool, come Visual Paradigm: hanno un repository centrale in cui risiede il progetto e vari tool come design analyser, design editor, code generator etc. che ci lavorano sopra. È un modo efficiente di condividere grandi quantità di dati, semplificandone gestione, backup e sicurezza, ma ha lo svantaggio di avere un'unica rappresentazione dei dati per tutte le componenti che li usano: questo ci porta ad avere dei compromessi perché difficilmente una sola rappresentazione è ottimale per tutti i sottosistemi interessati. Ci sono inoltre problemi di gestione della concorrenza.

Modello client-server
Modello distribuito in cui un server fornisce i servizi che i client andranno ad utilizzare. Il protocollo di comunicazione sarà di tipo request-reply (come HTTP).
Un sistema client-server può essere a due o tre livelll (two tier o three tier): nel two tier abbiamo tre componenti software (interfaccai utente, gestione processi e logica [business logic], gestione database) da distribuire su due livelli (client e server). Un approccio fat client metterà la logica

[5] **CASE:** Computer Aided Software Engineering

applicativa sul client e il database sul server, un approccio thin client mette anche la logica applicativa sul server lasciando sul client solo l'interfaccia utente.
Un sistema three tier invece avrà sempre un thin client che ha solo l'interfaccia utente, uno strato server che gestisce la logica applicativa e uno strato server che gestisce il database.

Architettura P2P
Modello client-server in cui ogni nodo agisce sia da client sia da server. Ogni peer è distinto dagli altri non dal ruolo, che non è definito a priori come nel modello client-server classico, ma dai dati che possiede: quando un peer ha bisogno di dati che non ha li chiede al peer che li ha.
I sistemi P2P scalano bene e hanno molta tolleranza agli errori, poiché sono decentralizzati: mentre in un sistema client-server classico la caduta del server provoca la sospensione del servizio, in un sistema P2P la caduta di un peer ha l'unico effetto di rendere inaccessibili i dati in possesso di quel peer: ma poiché solitamente i dati sono replicati anche su altri nodi in genere la caduta di un peer non ha effetti. Esempio di architettura P2P, oltre ai software di file sharing, è anche Skype.

4.1.2 Stili di controllo
Finora abbiamo parlato solo di componenti software e connessioni tra essi, ma è molto importante anche lo stile di controllo: decide chi manda in esecuzione cosa, quali eventi provocano quali azioni ecc. Esistono due stili di controllo: il **controllo centralizzato**, in cui un sottosistema ha la responsabilità di controllare gli altri sottosistemi, avviandoli e fermandoli quando necessario, e il **controllo a eventi** in cui non esiste un controllore, ogni sottosistema risponde agli eventi in maniera indipendente. Esempi di controllo centralizzato sono il **call-return model** e il **manager model**, esempio di controllo basato su eventi è il **broadcast model**.

Il **call-return model** (chiamato anche **main program-subroutine**) è quello adottato dai programmi C (e simili), in cui c'è un sottosistema di controllo (il main) che ha la responsabilità di gestire l'esecuzione degli altri sottosistemi, decidendo l'ordine delle chiamate a funzione. E' un modello top-down in cui il controllo parte dalla radice e si sposta verso il basso, ed è applicabile solo a sistemi sequenziali.

Per il controllo centralizzato di sistemi paralleli/concorrenti c'è il **manager model**, in cui un sottosistema centrale detto manager controlla tutti gli altri, interrogando le varie componenti per sapere se hanno cambiato stato.

Uno stile di controllo basato su eventi invece è il **broadcast model**: al posto di invocare direttamente una procedura lancia un evento che tutte le altre componenti sentiranno: a seconda del tipo di evento lanciato alcuni sottosistemi si attiveranno e altri no. Seguito generalmente da sistemi militari e di sorveglianza.
Di solito un sistema non adotta un solo stile architetturale, ma più stili combinati tra loro (sistemi eterogenei).

4.2 Design delle componenti

Nella fase precedente abbiamo visto come progettare l'architettura dell'inteo sistema a partire dal documento dei requisiti. Adesso, dopo aver definito la struttura generale del sistema, dovremo progettarne le singole componenti, e per prima cosa bisognerà definirne le interfacce: questo può essere fatto anche mediante design by contract. Dopo aver definito le interfacce, andremo a definire l'interno delle componenti (component design): è da qui che si inizia a parlare di **low level design**. Se abbiamo optato per un design OO, è in questa fase che andremo a definire le classi. Successivamente vengono decise le strutture dati da utilizzare, e infine la scelta/progettazione degli algoritmi.

4.2.1 Design by contract

Il design by contract è un metodo di design secondo cui ogni classe deve fornire un contratto, cioè una specifica precisa e verificabile dell'interfaccia. Il contratto prevede

- **precondizione**: un'asserzione booleana che deve essere vera prima che venga eseguita l'operazione (ad esempio, per un metodo che esegue una radice quadrata, la precondizione sarà x >= 0)
- **postcondizione**: asserzione booleana che deve essere vera dopo l'esecuzione dell'operazione (per sqrt, sarà x = risultato*risultato)
- **invariante di classe**: condizione che ogni oggetto della classe deve soddisfare quando è in equilibrio, ovvero sia prima che dopo l'esecuzione di un'operazione, ma non durante. Deve essere vera in ogni momento in cui può iniziare l'esecuzione di un'operazione.

Le precondizioni sono importanti perché definiscono chi è responsabile dei controlli: se qualcosa è indicato come precondizione, è compito del chiamante controllare che sia vera prima di utilizzare l'oggetto a cui si riferisce il contratto. Altrimenti potrebbero verificarsi duplicazioni di codice per controlli eseguiti inutilmente sia fuori che dentro il metodo, oppure che entrambe le parti pensino che i controlli vengano svolti dall'altro e che quindi alla fine non li svolga nessuno. Definendo le precondizioni invece si specificano chiaramente quali controlli vanno svolti dal chiamante.
Permette inoltre di capire dove stanno gli errori: infatti se precondizioni e invariante sono valide, ma si ottiene un risultato errato, l'errore sarà responsabilità del fornitore: se invece precondizioni e invariante non sono valide il fornitore non da nessuna garanzia sul funzionamento del modulo.
Servono inoltre da documentazione, da guida in fase di sviluppo e soprattutto semplificano notevolmente i test: conoscendo precondizioni, invariante e postcondizioni posso automatizzare i test.

4.2.2 Progettazione degli algoritmi

È la parte del design più vicina alla codifica, infatti spesso viene lasciata direttamente agli sviluppatori (viene quindi esclusa dal design e considerata già parte della fase di implementazione). Per prima cosa si analizza la descrizione di design del componente su cui stiamo lavorando, e verifichiamo se si possono adottare algoritmi esistenti. Se invece dobbiamo definirli from scratch, dobbiamo prima di tutto scegliere una notazione. Si definisce poi l'algoritmo con un processo di **stepwise refinement**, e dopo averlo definito, se è il caso (ad es. sistemy safety critical) se ne prova la correttezza con metodi formali.
Per rappresentare l'algoritmo esistono diverse notazioni, visuali (activity diagram UML) o testuali (pseudocodice). Uno pseudocodice (PDL, program design language) è un linguaggio semplificato basato su un linguaggio di programmazione esistente, ma con parti di narrativa al suo interno che semplificano lo stepwise refinement.

Lo stepwise refinement è un modo di definire algoritmi partendo da un livello molto astratto e scendendo passo passo sempre più nel dettaglio: serve infatti a passare dal linguaggio naturale ad un linguaggio di programmazione. Nelle slide esempio di stepwise refinement su un semplice algoritmo.

4.2.3 Principi di progettazione

La progettazione di software non è un'attività completamente creativa, ma va fatta seguendo principi di buona progettazione che portano ad avere software migliore. Ne esistono veramente tanti: in un articolo del 1995 Davis ne presenta 201. Sono principi che valgono e si possono applicare per tutti i sistemi, non solo quelli object oriented, anche se l'OO semplifica notevolmente l'applicazione di alcuni di questi (uno su tutti l'information hiding).

Un modulo è un'entità software con nome che contiene e fornisce servizi: contiene dati e istruzioni, può essere incluso in un altro modulo e può usare a sua volta altri moduli.

Principio di astrazione

Permette di nascondere i dettagli non necessari al livello in cui stiamo lavorando, lasciandoci concentrare meglio sul problema. Ci dice come dare le specifiche ad ogni livello: se stiamo facendo high level design non dovremo dare lo pseudocodice degli algoritmi, se stiamo dando le specifiche allo sviluppatore si. Esistono tre tipi di forme di astrazione: funzionale, di dati e di controllo

- **astrazione funzionale:** si definisce una funzionalità senza definirne l'algoritmo che la implementa
- **astrazione di dati**: si definisce un tipo di dato in base alle operazioni che ci posso fare, senza darne la struttura concreta
- **astrazione di controllo**: si definisce una meccanismo di controllo, come un semaforo, senza specificarne i dettagli interni

Principio di decomposizione

Risolvere un problema in un colpo solo è più difficile che scomporlo in sottoproblemi e risolverlo un pezzo alla volta (alcune equazioni che lo dimostrano nelle slides). Ma applicando questo principio alla lettera, ne concluderemmo che scomponendo un problema infinite volte, lo sforzo (effort) per svilupparlo sarebbe nullo. Va quindi considerato anche l'effort di integrazione: una volta suddiviso il software in moduli vanno anche integrati tra loro in modo che risolvano il problema originale. Bisogna quindi trovare un equilibrio tra eccessiva scomposizione ed eccessivo raggruppamento.

Principio di modularità (separation of concerns)

Deriva direttamente dal principio precedente: un software deve essere al più possibile modulare, con moduli e interfacce ben definite. Ogni modulo dovrebbe avere soltanto un compito, e bisogna cercare di separare tutto ciò che non è logicamente collegato ad un certo compito (separation of concerns).
È lo stesso principio che porta a sviluppare le applicazioni su tre livelli: presentazione (interfaccia utente), logica applicativa, dati e risorse. Mantenere separati i compiti permette di evitare i software monsters: un tipico errore di progettazione è quando ci si trova ad avere una "god class", una sola classe che fa praticamente tutto. Per non incappare in queste situazioni un'ottima regola è massimizzare la coesione tra i moduli e minimizzare l'accoppiamento. Massimizzare la coesione

vuol dire cercare di fare in modo che ogni modulo abbia un singolo compito, minimizzare l'accoppiamento vuol dire cercare di evitare le interdipendenze tra moduli. Facendo questo aumentiamo la comprensibilità del codice, la riusabilità, semplifichiamo le modifiche future e il testing.
Di solito per coesione si intende coesione funzionale: tutti gli elementi del moduli contribuiscono ad una singola operazione ben definita (ad esempio tokenizing di una stringa), ma esistono anche altri tipi di coesione, come ad esempio la coesione di utilità: quella per cui in Java abbiamo una classe Math che fornisce varie funzioni e costanti matematiche.
L'accoppiamento va ridotto al minimo perché le dipendenze tra moduli complicano notevolmente la manutenibilità del software: modifiche in un modulo comportano modifiche anche nell'altro, e viene limitato anche il riuso perché per riutilizzare il modulo dovrò portarmi dietro tutte le sue dipendenze. Con molte dipendenze, inoltre, è più difficile capire cosa fa una componente perché bisognerà capire anche cosa fanno tutte le componenti da cui dipende. Non può essere evitato completamente, altrimenti avremmo moduli completamente scollegati tra loro, ma esistono coupling "buoni" e "cattivi": un esempio di coupling buono è la chiamata di un metodo di un altro modulo, l'utilizzo di un tipo di dato definito in un altro modulo o l'inclusione/importazione di Java e C++. Un coupling cattivo è quello in cui un modulo va a modificare il valore di una variabile in un altro modulo, quello che salta da un modulo all'altro o quello che altera uno statement in un altro modulo. L'utilizzo di un linguaggio OO aiuta ad evitare queste situazioni perché non viene fornito il GOTO e viene facilitato l'incapsulamento delle variabili di istanza.

Principio di information hiding
Prescrive di non appoggiarsi alle implementazioni ma alle interfacce, in modo che la modifica di un'implementazione non comporti la modifica di tutti i moduli che la usano: aiuta quindi a mantenere debole l'accoppiamento tra moduli.

Alto fan-in e basso fan-out
Quando rappresentiamo le dipendenze su un grafo, in cui i moduli sono nodi e le dipendenze sono archi, il fan-in è dato dal numero di archi entranti e il fan-out il numero di archi uscenti. Un alto fan-in indica un buon riuso, un alto fan-out indica che un modulo fa troppo (rischia di diventare una god class) e va decomposto in più moduli.

Principio di generalità
Permette di migliorare il riuso di un modulo: si cerca di rendelo il più generale possibile in modo che possa essere usato anche in altri contesti

Semplicità
Applicare il rasoio di Occam: fare il design nel modo più semplice possibile, senza complicare ciò che essere risolto in modo più semplice. Principio KISS: Keep It Short and Simple.

Progettazione dell'interfaccia utente
La progettazione di interfacce utente richiede competenze non solo tecniche (come la conoscenza delle librerie grafiche) ma anche competenze trasversali e conoscenza dei fattori umani: anche un sistema tecnicamente eccellente può non avere successo a causa di un'interfaccia mal progettata. Per questo si parla di User Experience Architect e non di User Interface Architect, perché durante la progettazione bisognerà sempre avere come obbiettivo migliorare l'esperienza dell'utente: ad esempio fornendo messaggi di errore chiari e non troppo tecnici e non visualizzando troppe

informazioni in una volta (le persone sono in grado di ricordare istantaneamente solo 7 informazioni per volta). Altro errore tipico è pensare che gli utenti abbiano le stesse esigenze degli sviluppatori. Alcuni principi di progettazione UI orientati alla user experience sono:

- **User familiarity**: l'interfaccia deve usare termini e concetti che siano familiari agli utenti piuttosto che agli sviluppatori.
- **Consistency**: seguire le stesse convenzioni in tutta l'interfaccia: comandi e menu devono avere lo stesso formato da tutte le parti.
- **Minimal surprise**: gli utenti devono essere in grado di predire il funzionamento dei comandi, riconoscendo la somiglianza con comandi che già conoscono: non dovrebbero mai essere sorpresi dal funzionamento del sistema.
- **Recoverability**: bisogna fornire meccanismi con cui l'utente possa rimediare velocemente ad un errore: esempio tipico è l'undo.
- **User guidance**: l'interfaccia deve fornire strumenti di aiuto (come guide sensibili al contesto, guide in linea ecc.) e di segnalazione errori.
- **User diversity**: l'interfaccia deve presentare diversi strumenti di interazione a seconda delle esigenze di diversi utenti.

Metriche
Alla fine del design, come valutiamo se abbiamo fatto le cose per bene? Esistono delle metriche che permettono di misurare parte dei principi che abbiamo visto, e dei tool che le implementano (noi vedremo Stan4J)

5. UML

UML (Unified Modeling Language) è una famiglia di notazioni grafiche utile a descrivere la progettazione di un sistema software (soprattutto object oriented), ma si adatta anche ad altri ambienti (hardware, dominio del business...). UML è uno standard OMG (Object Management Group) un consorzio no-profit di industrie con l'obiettivo di promuovere e far rispettare standard per il mondo IT. È detto unificato perché unisce e standardizza varie notazioni preesistenti.
UML è un linguaggio di modellazione, non una metodologia: fornisce solo delle notazioni, non dei metodi per la progettazione software. Ovviamente non è un linguaggio di programmazione (anche se versioni arricchite di UML vengono usate nel model driven development in cui il progetto viene implementato automaticamente). Non obbliga a seguire il modello UP, né nessun altro modello di processo in particolare: può essere usato in tutti i processi, sia plan-driven sia agili. Tuttavia UP prescrive l'utilizzo di UML.
Ci sono molti modi diversi di usare UML all'interno del processo di sviluppo: può essere usato come **abbozzo**, per facilitare la discussione del problema tra i vari sviluppatori (che rappresenteranno strutture dati e problemi in schizzi UML su lavagna o UML editor). All'opposto c'è il **blueprint**: rappresentare il progetto dettagliato in UML, attraverso un tool di UML modeling (come Visual Paradigm); questo trasformerà la codifica in un'attività meccanica di traduzione del modello UML. Estremizzazione dell'uso blueprint sono i modelli eseguibili: attraverso tool specifici

(e versioni arricchite del linguaggio rispetto all'UML standard) possiamo tradurre i modelli UML in codice eseguibile (tool come WebRatio generano codice Java).

Esistono due prospettive nell'utilizzo di UML: **prospettiva software** e **prospettiva concettuale**. Nella prospettiva software gli elementi del diagramma sono elementi di un sistema software, solitamente tradotti come classi: questa prospettiva serve come documentazione del sistema per le fasi successive di sviluppo. In prospettiva concettuale invece stiamo modellando il dominio del business e i processi: se in prospettiva software l'utente era una classe del nostro sistema, in prospettiva concettuale è la persona fisica che lo userà, e quindi non avrà senso, per esempio, parlare di classi e metodi. Può sembrare secondaria rispetto alla prospettiva software, ma è fondamentale per avere chiari i processi di business dell'azienda per cui stiamo progettando, che dobbiamo per forza conoscere perché il nostro sistema andrà a modificarli: di norma UML viene usato in prospettiva concettuale prima ancora di scrivere i requisiti, subito dopo lo studio di fattibilità.
Un **modello di dominio** è una rappresentazione visuale di classi concettuali o di oggetti del mondo reale di un dominio (definizione di Fowler). È un glossario visuale, comune tra i vari stakeholder, delle astrazioni significative per il nostro dominio: astrazioni perché prendiamo in considerazione solo le caratteristiche relative al livello che ci interessa. Può definire una classe concettuale (studente, corso) o una sua istanza: un oggetto del dominio(mario rossi, ingegneria del software).
Negli anni '90, quando ci si rende conto che i linguaggi di programmazione OO non bastano a gesitre tutte le fasi del processo di sviluppo, vengono proposti vari modelli (OMT, Booch 93, OOSE). Nel ‘95 Rational aveva al suo interno le tre persone più importanti nel campo della modellazione (i tre amigos, Rumbaugh, Booch e Jacobson), e le altre aziende, spaventate dal fatto che Rational potesse creare uno standard, chiamano in campo OMG che fa una call for proposal per la creazione di un linguaggio di modellazione. I tre amigos propongono UML 1.0, che otterrà lo status di standard OMG con l'edizione 1.1, nel 1997. Adesso la notazione ufficiale UML è la 2.x (ultima versione 2.5).

UML va studiato perché è una "lingua franca" di modellazione del software, ed è molto richiesto dall'industria (soprattutto a livello di abbozzo, non molte aziende lo usano come blueprint). Inoltre è molto comodo perché essendo grafico permette di spiegare molto meglio e in maniera più comprensibile concetti complessi come può essere un design pattern. È indispensabile per le figure di analista e progettista software.
Tuttavia non lo studieremo in maniera approfondita perché è estremamente grande e complesso, ma soprattutto perché l'approccio blueprint è molto poco usato: è infatti estremamente difficile produrre un design dettagliato e allo stesso tempo corretto, per il semplice motivo che un design non si può testare. Non esistono studi empirici che dimostrino che l'approccio blueprint sia migliore: anche Fowler sostiene che "i progetti dettagliati sono troppo complessi da realizzare e rallentano il processo di sviluppo."

La notazione di UML è la sintassi grafica del linguaggio, il meta-modello invece definisce i concetti del linguaggio come classi, attributi e operazioni. Se usiamo UML solo a livello di abbozzo, possiamo anche non conoscere il meta-modello. Nelle slides c'è un class diagram che rappresenta il meta-modello di UML.
In UML un sistema software viene descritto da diversi punti di vista: una **vista** è un particolare aspetto del nostro sistema, ad esempio gli use cases sono una vista (il sistema guardato dal punto di vista dell'utente). Un **diagramma** UML descrive una vista, un **modello** è un insieme di diagrammi

che descrivono diversi aspetti dello stesso sistema. Ci sono 13 tipi di diagrammi UML, di cui ne vedremo bene solo 6.

UML può essere utilizzato in fase di specifica dei requisiti, in fase di architettura e in fase di design. Dal punto di vista dei requisiti UML viene usato per gli use case diagrams: sempre nell'ambito degli use cases UML può essere usato anche per descrivere gli scenari, con **activity diagrams** e **sequence diagrams**. Quando si passa all'architettura invece UML ci fornisce il **component diagram**. Il **deployment diagram** invece ci mostra la relazione tra nodi computazionali (le macchine fisiche che andranno ad eseguire il nostro software) e le componenti software. In fase di design è importantissimo il **class diagram**, che per ogni classe delle nostre componenti ci dice membri e metodi. L'**object diagram**, molto meno utilizzato, rappresenta istanze delle classi del class diagram e può aiutare a capirlo meglio. Per rappresentare le operazioni svolte dai metodi, in alternativa allo pseudocodice, possiamo usare gli activity diagrams, e per descrivere oggetti che possono avere diversi stati abbiamo gli **state diagrams**.

Nonostante i 13 diagrammi, UML non è sufficiente a rappresentare ogni parte del processo di sviluppo: la soluzione può essere estenderlo tramite profili o utilizzare altri diagrammi che non fanno parte di UML. Un caso in cui può servire usare diagrammi non UML è per rappresentare le schermate: UML è sprovvisto di un diagramma per questo scopo.
L'alternativa è estenderlo con un profilo per rappresentare particolari domini applicativi o tipologie di applicazioni. Un profilo è composto da:

- stereotipi: elementi grafici aggiuntivi
- constraint che possono vietarmi di usare certi elementi
- informazioni semantiche sugli elementi aggiunti

La corrispondenza tra UML e codice è un argomento spinoso: non esiste infatti una corrispondenza univoca, perché UML si trova ad un livello di astrazione più alto.

Siccome lo scopo principale di UML è semplificare la comunicazione, si è sempre messo il farsi capire prima del rispetto dello standard: il risultato è che esistono numerose varianti di UML, introdotte da chi ha adottato particolari convenzioni per semplificarlo: quando si vuole usare una di queste notazioni, va aggiunto al grafico un rettangolino con scritto "non normativo".

Per evitare le incomprensioni conviene attenersi all'UML normativo e precisare sempre la prospettiva in cui siamo ed il livello di dettaglio (abbozzo, blueprint). Nel caso in cui si omettano delle informazioni, specificare quale informazione è soppressa (come ad esempio la visibilità degli attributi), ed esplicitare sempre anche quelle informazioni per cui esiste un valore di default (ad es. la molteplicità 1 nelle associazioni). Inoltre, siccome UML è grande e complesso, conviene utilizzare solo un sottoinsieme di diagrammi e costrutti, quelli più noti.

5.1 Class diagram

È il diagramma più diffuso, conosciuto e importante di UML (alcune aziende utilizzano solo quello). Definisce le classi, le loro feature (attributi e operazioni NB: operazione non è sinonimo di metodo) e le relazioni tra classi, che possono essere di diverso tipo. Definisce la parte statica del software che andiamo a modellare. Come per ogni diagramma UML, il significato di un class diagram cambia a seconda che lo guardiamo in prospettiva concettuale o software: se siamo in prospettiva

concettuale una classe del class diagram sarà un concetto del dominio, una classe di elementi del dominio con caratteristiche comuni (ad es. dipendente, utente), ed un'operazione sarà un'azione o responsabilità che avrà la nostra classe (ad es. dipendente preleva stipendio).
In prospettiva software invece un rettangolo del diagramma sarà una classe di un linguaggio OO, e le operazioni saranno implementate attraverso metodi.
Il concetto di classe in UML è lo stesso dei linguaggi OO: un'entità che accorpa caratteristiche comuni ad un gruppo di oggetti e viene usata per creare oggetti, che saranno istanze della classe. Gli attributi determinano lo stato, le operazioni determinano il comportamento. In UML viene rappresentata con un rettangolo con nome: sotto il nome vengono rappresentati gli attributi, e ancora più sotto le operazioni. Si possono aggiungere anche il tipo degli attributi e il tipi dei parametri delle operazioni. I vari livelli di dettaglio (solo nome, nome+attributi, nome+attributi+operazioni, nome+attributi tipati+operazioni(con tipo dei parametri)) sono tutti validi: sceglierò quale usare a seconda del livello di dettaglio che mi serve. Il simbolo di fianco ad attributi e operazioni ne indica la visibilità:

- **+** pubblico
- **-** privato
- **#** protetto
- **~** package

Quando rappresentiamo gli attributi, l'unica parte obbligatoria è il nome. Il tipo dell'attributo può essere un tipo primitivo o il nome di un'altra classe del class diagram. L'opzione default definisce il valore di default dell'attributo in un oggetto appena creato. Tra parentesi graffe possono venire indicate altre proprietà aggiuntive, come readOnly. Dato che le regole di visibilità di UML e dei vari linguaggi OO sono spesso differenti, Fowler consiglia di usare quelle del linguaggio con cui andremo ad implementare e di cercare di usare solo + e -.
La **molteplicità** indica il quantitativo degli attributi, ad esempio la dimensione se l'attributo è un array o una lista.
I valori possibili sono 1 (uno e uno solo), 0..1 (al più uno), * (un numero imprecisato), 1..* (imprecisato ma almeno uno). Gli attributi molteplici sono considerati insiemi, quindi non ordinati: se vogliamo un attributo molteplice ordinato dobbiamo indicarlo come ordered.
Anche delle operazioni possiamo specificare la visibilità, ed in più posso mettere lista di parametri e tipo di ritorno. I parametri sono scritti nella forma **direzione nome tipo = default**, dove direzione è in o out.
In genere, per mantenere il class diagram più semplice, getter e setter vengono omessi dal diagramma.

In UML esiste il concetto di datatype, che è diverso dal concetto di oggetto: gli oggetti infatti hanno una loro identità, due oggetti che si trovano nello stesso stato (stessi valori degli attributi) vengono considerati diversi. Le istanze di un datatype invece sono valori e non oggetti: esempi di datatype sono date e somme di denaro, due date uguali sono considerate una stessa data così come una somma di denaro. Un caso particolare di datatype usato molto spesso è l'enumeration: un datatype di cui definisco l'insieme di valori accettati.

Associazioni
Un'associazione rappresenta una relazione, fisica o concettuale, tra classi: ne può essere indicata anche la direzione, come proprietario possiede auto. Alternativamente si può usare un ruolo: rapprsento l'associazione senza nome tra persona e automobile e indico la persona con ruolo di

proprietario. Il **verso di navigazione** invece indica in quale direzione è possibile reperire le informazioni: se ad esempio persona possiede -> automobile data una persona posso sapere tutte le automobili che possiede, ma data un'automobile non ne posso reperire tutti i proprietari.
La molteplicità di un'associazione indica il numero di link (istanza dell'associazione) tra le classi: sempre per persona automobile se ho 1 dal lato persona e 1...* dal lato automobile vuol dire che ogni automobile ha un solo proprietario e che ogni persona può avere un numero imprecisato di automobili, ma deve averne almeno una. (occhio perché le molteplicità sono invertite rispetto ai diagrammi E-R!)
Attributi ed associazioni concettualmente indicano la stessa cosa, sono entrambi **proprietà** della classe: per convenzione di solito si usano attributi e datatype per le piccole cose (stringhe, date, booleani) e associazioni per le classi più significative. Posso avere anche associazioni riflessive: una classe che ha un'associazione con se stessa. Esistoni anche classi associative, per descrivere le proprietà che sono solo di un'associazione (se ad esempio ho due classi studente e corso, con un'associazione esame, voto sarà una proprietà di esame).Vanno usate con cautela: solo quando indispensabili perché difficili da implementare, non ne esiste un corrispondente nelle classi dei linguaggi di programmazione.
Posso avere anche operazioni ed attributi statici, che hanno lo stesso significato di metodi a campi static di Java.
Esistono anche associazioni bidirezionali, navigabili in entrambe le direzioni: nell'esempio di prima se l'associazione persona automobile fosse bidirezionale potrei sapere sia quante auto ha una persona sia chi sono i possessori di un'automobile. Sono complesse da implementare: un singolo attributo non basta, ce ne vuole almeno uno per classe, ed avrò il problema della sincronizzazione: dovrò mantenerli coerenti da entrambe le parti dell'associazione.
Questo ci fa vedere che, generalmente, più è libero il design più si complica la codifica.

Aggregazioni e composizione
Permettono di rappresentare una suddivisione di tutto-parti: ad esempio se ho le classi Nazione e Regione, Nazione sarà un'aggregazione di Regioni, che saranno una parte di Nazione. Fowler ne sconsiglia l'utilizzo.
L'aggregazione forte è espressa dalla **composizione**: abbiamo una classe che può essere parte di altre classi, ma ogni istanza può esserlo solo di una classe alla volta. Ad esempio potremmo avere una classe Point, che può essere sia parte di un rettangolo sia il centro di un cerchio: ogni istanza di Point però sarà o uno o l'altro, mai tutte e due insieme. Questa è la regola base della composizione, ciò che la distingue dall'aggregazione: un altro aspetto che di solito si considera implicitamente valido è che la cancellazione del poligono porta alla cancellazione automatica di tutti i punti che gli appartengono.

Generalizzazione
A livello concettuale è come nei diagrammi E-R: rappresenta la relazione "è un", posso avere la classe Persona che è una generalizzazione di Donna e Uomo. A livello implementativo rappresenta invece l'ereditarietà: dal lato superiore ho la superclasse e dal lato inferiore le sottoclassi, che erediteranno attributi, operazioni e associazioni delle superclassi.

operazioni e metodi non sono la stessa cosa: un'operazione viene invocata su un oggetto e corrisponde alla dichiarazione di una procedura, un metodo è invece il corpo di tale procedura, che ne specifica il comportamento. Il metodo è l'implementazione dell'operazione. Le operazioni e le classi astratte vengono indicate in corsivo (o con l'etichetta {abstract}).

Operazioni e attributi statici
Quando un attributo/operazione si riferisce alla classe e non agli oggetti della classe (esattamente come metodi/campi statici di Java) lo si indica sottolineandolo nel class diagram.

Dipendenze
Si ha dipendenza tra due classi se la modifica di una può comportare modifiche nell'altra. Si genera tipicamente quando in una classe se ne usa un'altra come tipo di una variabile locale. Per come le vediamo in UML le dipendenze non sono transitive. All'interno di un sistema software avremo tantissime dipendenze, quindi non vanno indicate tutte ma solo le più importanti. In fase di sviluppo e codifica bisogna cercare di minimizzare le dipendenze, in particolare vanno evitate le dipendenze cicliche (soprattutto tra package). Questo viene semplificato da tool come Stan4J.

Object diagram
L'object diagram è l'istanziazione di un class diagram, in cui rappresentiamo gli oggetti con Nome oggetto: nome classe (sottolineato, è la prima cosa per distinguere un class diagram da un object diagram), volendo posso rappresentare anche i valori degli attributi. Sono molto utili a chiarire diagrammi delle classi complessi.

Interfacce
Il termine interfaccia può significare l'insieme di operazioni visibili all'esterno degli oggetti istanza di una certa classe oppure un'entità simile ad una classe, ma senza implementazione e solo con operazioni pubbliche. In UML una notazione molto usata per rappresentarle è il lollipop, che rappresenta l'interfaccia esposta, racchiuso da un semicerchio (socket) che rappresenta l'interfaccia richiesta.

OCL
Per specificare i vincoli che non possono essere espressi attraverso associazioni, attributi e generalizzazioni abbiamo l'OCL (Object Constraint Language), basato sul calcolo dei predicati. Aggiungere un vincolo ad un class diagram significa, in pratica, ridurre il numero di combinazioni di oggetti ammissibili: limitiamo il numero di object diagrams che possiamo creare a partire da quel class diagram.
Esempio: classi Persona e Veicolo, Persona può essere proprietario di Veicolo: voglio mettere il vincolo che per essere proprietario la persona deve avere almeno 18 anni. Ragionerò in termini di context, precondizioni, postcondizioni ed invariante.

5.2 Sequence diagrams
I diagrammi UML possono essere suddivisi in due tipi: i diagrammi per la **modellazione statica**, o **strutturali**, sono quelli che descrivono appunto la struttura del sistema, occupandosi di aspetti come le entità del modello e le relazioni tra di essi. Esempi di diagrammi per la modellazione statica sono il class diagram e l'object diagram. I diagrammi per la **modellazione dinamica** o **comportamentali** invece si occupano delle interazioni tra gli oggetti e del loro comportamento durante l'utilizzo: un esempio che abbiamo già visto sono gli use cases.

I diagrammi di interazione sono un tipo di diagramma comportamentale che descrive la collaborazione di un gruppo di oggetti, la quale avviene tramite scambio di messaggi. I messaggi possono essere sincroni quando il mittente si pone in attesa della risposta, asincroni quando dopo l'invio procede normalmente con l'esecuzione (è questo il caso dei processi concorrenti). UML li rappresenta con due notazioni grafiche diverse.
I diagrammi di interazione possono essere diagrammi di sequenza o diagrammi di comunicazione: forniscono le stesse informazioni, ma mentre il diagramma di sequenza si concentra sulla successione temporale dei messaggi scambiati, il diagramma di comunicazione da enfasi alla comunicazione tra oggetti. Se stiamo usando UML come blueprint conviene usare un sequence diagram, se invece lo stiamo usando come sketch conviene un communication diagram.
I sequence diagram sono usati sia in fase di definizione dei requisiti sia in fase di design. In fase di definizione dei requisiti vengono usati nel modello dei casi d'uso per descrivere le interazioni tra gli attori e il sistema, con un diagramma molto meno dettagliato rispetto all'uso in fase di design, perché in fase di requisiti il sistema è visto ancora come una black box: stiamo definendo cosa dovrà fare ma non ci interessiamo ancora del come lo farà. Il sequence diagram usato in questa fase viene chiamato System Sequence Diagram (SSD), ed è molto utile averlo perché chiaririsce quali sono gli input/output del sistema, semplifica la fase di testing ed è un ottimo punto di partenza per definire i contratti delle operazioni.
Quando passiamo in fase di design invece dovremo anche andare a descrivere come il sistema svolge i suoi compiti, ed avremo quindi un sequence diagram molto più dettagliato.

La struttura non sto a scriverla qua che tanto è spiegata nelle slide e sul libro molto meglio.

I sequence diagram sono un ottimo strumento per visualizzare l'interazione tra oggetti ma, nonostante abbiano costrutti per rapprsentari cicli e if (frame d'interazione) non sono la scelta migliore per rappresentare la logica degli algoritmi, Fowler infatti consiglia di non usare questi costrutti: se abbiamo bisogno di rappresentare un ciclo o una scelta ci conviene passare allo pseudocodice.

5.3 State machine diagrams

Gl state machine diagrams sono diagrammi **behavioral** (o di **modellazione dinamica**) che descrivono il comportamento di un entità come variazione del suo stato interno quando sottoposto a sollecitazioni dal mondo esterno. L'entità può essere un sistema software o hardware, una classe OO particolarmente complessa o un'entità del mondo reale. A complicare il tutto abbiamo il fatto che il comportamento dell'entità può variare a seconda sia dell'input ricevuto sia dello stato interno. Lo **stato** rappresenta una situazione durante la vita di un'oggetto durante il quale delle condizioni sono verificate e delle attività possono essere eseguite: mentre uno sviluppatore li penserà come le combinazioni del valori assunti dai campi di un oggetto, in UML ha un significato più astratto: non basta che i valori degli attributi di un oggetto siano in una certa combinazione, ad ogni combinazione deve anche corrispondere un diverso comportamento dell'oggetto perchè si possa parlare di stato.
In UML i diagrammi di stato sono rapprsentati da un grafo, in cui i nodi sono gli stati e gli archi sono le transizioni, con qualche notazione semantica in più: la macchina a stati riceve eventi, accodati in una coda FIFO ed estratti uno alla volta solo quando l'evento precedente è stato processato. Nel caso in cui ci fossero più transizioni eseguibili in un dato momento ne viene eseguita solo una, scelta in modo non deterministico: è quindi una situazione da evitare.

Le condizioni che abilitano il passaggio da uno stato all'altro sono dette **guardie** e vengono indicate tra parentesi quadre sull'arco.
Gli stati iniziale e finale (che non sono stati veri e propri, ma pseudostati) vengono indicati rispettivamente con un cerchio nero e con un cerchio nero bordato di bianco: mentre di stato iniziale ne possiamo avere solo uno, di stati finali possono essercene di più.
Le transizioni sono gli archi del grafo e sono contrassegnate da **evento [guardia] attività**: l'evento è il trigger che attiva il passaggio di stato, la guardia è una condizione che deve essere vera perché la transizione possa verificarsi, mentre l'attività è un'azione svolta durante la transizione, può essere una qualsiasi espressione che rappresenti un comportamento.
L'evento è qualcosa che l'entità subisce, come la ricezione di un messaggio, mentre un'azione è qualcosa che l'entità esegue, e fa parte dell'attività.
Tutte e tre le componenti sono opzionali: se manca l'attività significa che durante la transizione non si farà niente (a parte ovviamente cambiare stato), se manca la guardia la transizione avverrà ogni volta che si verifica l'evento e se manca l'evento vuol dire che la transizione avviene immediatamente (caso che si verifica molto raramente, di solito in presenza di stati di attività).
L'attività può essere data sia in linguaggio naturale sia in pseudolinguaggio, con una serie di azioni separate da ;. Quando usiamo UML come linguaggio di programmazione le azioni vanno espresse in ALF (Action Language for Foundational UML).
Le azioni possono far riferimento solo a cose che l'oggetto conosce: attributi, operazioni e link dell'oggetto stesso più i parametri del messaggio ricevuto.
Quando ci troviamo in prospettiva software, lo state machine diagram sarà associato ad una classe.
Finora abbiamo visto l'evento solo come la ricezione di un messaggio ma ce ne sono altri:

- **evento di cambiamento**: si verifica quando una condizione passa da falsa a vera, indicato con when(condizione)
- **eventi temporali**: si verificano dopo un certo lasso di tempo o ad una certa data ora
- **eventi di segnale**: non ci interessano.

Uno stato può reagire ad un evento anche senza bisogno di passare ad un altro stato, eseguendo un'attività interna. Entry ed exit sono speciali attività interne che vengono eseguite ogni volta che si entra o esce dallo stato, e sono quelle che rendono diverse un'attività interna da una self transition: una self transition comporta l'esecuzione di exit ed entry.
Una do activity è un'altra speciale attività interna, che non ha durata istantanea ma rimane in esecuzione per un certo lasso di tempo, e può essere interrotta dal verificarsi di un evento.

Con gli **stati compositi** possiamo suddividere la complessità del modello: abbiamo uno stato che si suddivide in diversi sotto-stati, e a seconda del livello di astrazione che vogliamo usare possiamo considerarlo una black box, rappresentandolo come uno stato unico (indicando però con un'icona che si tratta di stato composito) o una white box, rappresentando lo stato esterno e i sottostati interni.
Anche se non molto spesso, gli stati compositi si usano anche per modellare la concorrenza: suddividiamo ortogonalmente con linea tratteggiata uno stato in due o più sottodiagrammi che indicano gli stati paralleli che vengono eseguiti in uno stesso momento. Per gli stati compositi si possono inoltre definire entry ed exit points, rispettivamente con un cerchietto vuoto ed un cerchietto con la x dentro sul confine dello stato.

Gli state machine diagram andrebbero utilizzati solo per le entità che hanno una logica interna complessa ed interessante per il nostro sistema: esempi sono oggetti e sistemi di controllo, distributori automatici, sistemi di gestione documentale (documenti che devono girare diversi uffici per essere firmati), GUI....

La maggior parte dei linguaggi non implementa nativamente le macchine a stati, che andranno quindi implementate a mano con

- **switch**: approccio più diretto; un case per ogni stato, ma difficile da modificare
- **design pattern State**: l'approccio più elegante, utilizza una classe per ogni stato e sfrutta il polimorfismo per modellare il comportamento
- **tabelle di stato** che verranno trasformate automaticamente, attraverso un tool model driven come SMC, in codice che implementa il design pattern State

5.4 Activity diagrams

L'activity diagram è un diagramma di modellazione dinamica (o comportamentale) che descrive come viene svolta una certa attività relativa a cose di tipo diverso: può descrivere le operazioni di una classe, di una componente, uno use case, una persona fisica o un processo di business. Sono simili ai flowchart, con in più il supporto all'esecuzione parallela/concorrente: descrivono il flusso di azioni necessarie per portare a termine l'attività. Vengono usati in diverse fasi dello sviluppo: in fase di **analisi** si usano sia per modellare i processi di business su cui andrà a lavorare il nostro software sia per modellare il flusso degli use cases. In fase di **design** si usano per modellare un algoritmo o il funzionamento dell'operazione di una classe. Se stiamo usando UML come linguaggio di prigrammazione, invece, verranno usati activity diagram di bassisimo livello ogni volta che ci sarà da modellare un comportamento complesso in un altro diagramma (come le azioni negli state machine).

Un'attività è costituita da un flusso di azioni, rappresentate come rettangoli con gli angoli smussati. È un grafo con nodi e archi: i nodi possono essere di **azione**, **oggetto** o **di controllo**: i nodi oggetto specificano oggetti usati come input/output dalle azioni, i nodi di controllo modificano il flusso dell'attività: possono biforcarlo per l'esecuzione di due flussi paralleli, rappresentare un punto di decisione, di inizio o di fine.
Sui flussi entranti e uscenti da un nodo transitano i **token**.
I nodi iniziali, rappresentati con un pallino pieno, generano un token su ogni flusso uscente dal nodo. I nodi finali di attività, rappresentati dal pallino pieno cerchiato, terminano tutta l'attività (distruggendo tutti i token) non appena un token li raggiunge, mentre i nodi finali di flusso (croce cerchiata) terminano solo il flusso a cui si riferiscono.
Ogni activity diagram dovrà avere almeno un nodo iniziale, ma può anche non avere nodi finali perché possono essere usati anche per rappresentare attività cicliche che non termineranno mai.
All'interno dei nodi, l'azione può essere descritta in diversi modi:

- informalmente, in linguaggio naturale
- può richiamare un'altra attività (comportamento simile alla chiamata a funzione)
- può invocare un'operazione di un'altra classe
- può anche essere rappresentata in codice, usando il linguaggio per le azioni di UML

Un nodo azione viene eseguito quando è presente almeno un token su ogni flusso in entrata. Una volta terminata l'azione, si genera un token su ogni flusso uscente dal nodo.
I nodi di decisione modificano il flusso dell'attività dopo aver valutato una guardia: una volta arrivato un token si valutano le guardie sulle uscite dal nodo di decisione: se nessuna guardia è vera il token rimane bloccato sul nodo, se c'è almeno un flusso con guardia vera il token viene instradato su quella, se più di un flusso ha una guardia vera la scelta sarà non deterministica.
I nodi di fusione, rappresentati con un rombo come quelli di decisione, riuniscono i flussi precedentemente separati da una decisione: hanno più flussi entranti e un solo flusso uscente.

Fork e **join** vengono usati per rappresentare i flussi paralleli: il fork ha un flusso in entrata che viene splittato in più flussi paralleli, un join ha più flussi paralleli in entrata e un solo flusso di uscita.

I nodi azione, oltre ai token, possono far fluire anche nodi oggetto.

Per dare più informazioni sull'attività rappresentata possiamo partizionarla in **swimlane**: non modificano la semantica del diagramma, servono solo a raggruppare visivamente le azioni secondo qualche criterio. Sono pensate per rendere più semplice la lettura di un diagramma, ma se usate a sporposito possono complicarla.

Ci sono anche nodi azione specializzati nel generare ed accettare eventi. I nodi di ricezione se hanno flussi in entrata spno attivi quando hanno un token su ognuno, se non hanno flussi in entrata sono attivi per tutta la vita dell'attività, e generano un token su ogni flusso uscente alla ricezione del segnale.

5.5 Component, package e deployment diagrams
Sono tutti e tre diagrammi di modellazione statica

5.5.1 Component diagrams
Il termine componente assume significati diversi a seconda del contesto in cui viene usato, e quindi UML ne da una definizione il più generale possibile: "una scatola nera, rimpiazzabile e componibile, il cui comportamento esterno è completamente definito dalle interfacce".
Una classe è una componente, ma di grana troppo fine per descrivere l'architettura software di un sistema.
Un componente è un'entità logica, quindi non tangibile, realizzate da artefatti (entità fisiche).
L'entità logica Java Beans sarà realizzata da un file JAR, un sottosistema sarà realizzato da un insieme di classi, un progetto WebRatio sarà realizzato da un file JAR ecc.
I diagrammi delle componenti rappresentano le componenti che costituiscono il sistema e le dipendenze tra esse.
I componenti sono rappresentati con rettangoli (con qualche differenza di notazione tra le diverse versioni di UML), le interfacce fornite sono rappresentate con un lollipop e le interfacce richieste con un socket. La composizione è data da un lollipop che si incastra in un socket.
Una **porta** raggruppa un insieme semanticamente coeso di interfacce.
È possibile avere anche una vista interna dei componenti, che fa vedere come sono assemblati i componenti al loro interno.

5.5.2 Package diagrams
In UML un package è un costrutto che raggruppa un numero arbitrario di elementi UML: non solo classi come nei package software ma anche use cases e qualunque altro tipo di diagramma UML. Come nei linguaggi di programmazione, ogni package definisce un namespace, che definisce il confine all'interno del quale i nomi devono essere univoci.

5.5.3 Deployment diagram
Rappresenta la relazione tra software e hardware in un sistema: mostra su quale nodo computazionale vengono eseguiti i vari componenti del sistema. Ha una forte relazione col component diagram.
È composto da **nodi** e **connessioni**: i nodi rappresentano qualsiasi cosa che possa eseguire del software, che può essere fisico (qualunque dispositivo che possa eseguire sw) o virtuale (VM, browser...). Una connessione rappresenta un canale di comunicazioni tra nodi, e può essere etichettata col tipo di connessione. Sui nodi sono dislocati degli artefatti, di cui possono essere rappresentate le dipendenze. Un esempio di dipendenza tra artefatti è una pagina web che dipende dalla tabella di un database.

6. Persistenza

La persistenza consiste nel salvataggio dei dati di lavoro di un'applicazione su un supporto di memorizzazione non volatile, in modo da poter essere recuperati ad un'esecuzione successiva. Può essere ottenuta memorizzando sul file system attraverso la serializzazione, in formato binario o human readable (XML), oppure su un database.
La **serializzazione** è il processo per salvare l'intero stato di un'oggetto su un supporto di memorizzazione lineare o per trasmetterlo via rete. Nel caso di serializzazione binaria l'oggetto viene convertito in una sequenza di bit, e in Java può essere fatta su tutti gli oggetti che implementano l'interfaccia Serializable. Per serializzare in formato human readable invece tipicamente si usa XML, interrogabile attraverso espressioni XPath, linguaggio standard W3C per la localizzazione dei nodi di un documento XML. In Java, la serializzazione/deserializzazione XML avviene attraverso la libreria XStream.
La serializzazione va usata quando ci sono così pochi dati da non dover richiedere tutte le funzionalità offerte da un **DBMS (DataBase Management System)**, che invece va usato quando abbiamo bisogno di interrogazioni dichiarative efficienti e non definite a priori, gestione degli accessi concorrenti a granularità molto fine, supporto alle transazioni.
Nel corso di BD abbiamo visto solo RDBMS, normali DBMS relazionali che vedono i dati rappresentati da semplici strutture tabellari. Il 90% dei DBMS sul mercato sono RDBMS, e anche se molti di questi offrono funzionalità object relational queste non vengono utilizzate più di tanto.
Esistono anche **OODBMS (Object Oriented DBMS)** e **ORDBMS (Object Relational DBMS**, basati sull'Object Relational Model). Gli **OODBMS** memorizzano direttamente gli oggetti, ottenendo quella che si chiama **persistenza ortogonale**: gli oggetti in memoria e gli oggetti memorizzati nel DB sono identici. Offrono una modalità di interrogazione principalmente navigazionale, si percorrono dei cammini attraverso i riferimenti tra gli oggetti. Questo paradigma è seguito dai sistemi NOSQL, pensati per chi ha esigenze di persistenza diverse da quelle offerte dal modello relazionale, ma ha bisogno di interrogazioni che fanno molto uso della navigazione tra gli oggetti. E' molto usato anche nei database interni degli applicativi di CAD, CAM e CASE.
Gli **ORDBMS** invece rappresentano un modello ibrido: mantengono l'idea di tabella, ma eliminano il vincolo che ogni cella debba contenere un valore atomico: in un database OR nelle celle delle

tabelle possiamo memorizzare oggetti. Permette anche di modellare le associazioni per riferimento, evitando di dover replicare un oggetto su più tabelle: le chiavi esterne del modello relazionale infatti hanno una copia memorizzata in ogni tabella che le riferisce: se per valori atomici non è un grosso problema, memorizzando oggetti sarebbe un overhead non indifferente. Caratteristiche OR sono state introdotte in SQL nel 1999 e nel 2003. Ai tipi user definied, creati con **CREATE TYPE**, possono essere assegnati non solo campi ma anche metodi e possono essere definite relazioni di ereditarietà (anche se quest'ultima è poco usata). Progettare uno schema logico OR è particolarmente difficile perché aumentano le possibilità, e quindi le scelte che bisognerà prendere, ad esempio se e quando usare chiavi esterne o ref: queste scelte impatteranno sull'efficienza e dovranno quindi essere ben ragionate.

6.1 Object Relational Mapping

Se i sistemi OODBMS non ci vanno bene, perché abbiamo bisogno di fare query per cui è più adatto il modello relazionale, e degli ORDBMS non ci fidiamo abbastanza, la soluzione è l'**Object Relational Mapping**: si fa "a mano" la conversione tra oggetti in memoria e tabelle di un RDBMS. La conversione modello OO-modello relazionale richiede molto tempo, molto codice e quindi molto rischio di errori, e poiché i due modelli si basano su concetti diversi avremo freuenti **impedance mismatch**: mancanza di corrispindenza tra concetti OO e concetti relazionali. Il modello OO infatti descrive il dominio in termini di oggetti e delle loro proprietà, che possono essere attributi (valori) o associazioni ad altri oggetti, mentre il modello relazionale si basa sulla rappresentazione di associazioni tra valori.
Mappare gli oggetti in un modello relazionale è come smontare e rimontare la macchina invece di parcheggiarla. Esistono però delle tecniche standard per semplificarci il lavoro: ad ogni classe facciamo corrispondere una tabella, e per gli attributi di tipo collezione o strutturato facciamo una tabella a parte. Le associazioni invece verranno mappate diversamente al seconda della loro molteplicità: un'associazione uno-a-uno o uno-a-molti verrà mappata con l'inserimento di una chiave esterna nella tabella che rappresenta la classe dal lato uno dell'associazione. In caso di associazione uno a uno, metteremo la chiave nella tabella da cui pensiamo che dovremmo accedervi più spesso. Un'associazione molti a molti sarà mappata con una tabella aggiuntiva con due chiavi esterne, una per tabella.
L'ereditarietà può essere mappata verticalmente, orizzontalmente o attraverso filtered mapping: nel **vertical mapping** per mappare la classe Persona da cui discendono Cliente e Impiegato abbiamo una tabella Persona con gli attributi comuni e due tabelle Cliente e Impiegato con gli attributi specifici ed una chiave esterna verso persona. Nel **mapping orizzontale** invece eliminiamo la radice della gerarchia: non abbiamo più la tabella Persona ma due tabelle Cliente ed Impiegato, ed ognuna avrà sia gli attributi comuni ad entrambe (nome, cognome,..) sia i suoi specifici**.**
Nel **filtered mapping** infine mettiamo tutto in una sola tabella Persona. Questo è molto simile alla traduzione delle generalizzazioni da modello concettuale E-R a modello relazionale.

6.1.1 Integrazione dell'Object Relational Mapping nel codice

Come integrare nel nostro sistema il codice necessario all'ORM? Ci sono tre strategie possibili

1. **forza bruta**: l'interazione col DB è completamente integrata nel codice dell'applicazione, introducendo ad esempio in ogni classe del dominio i metodi Open e Save per leggere/scrivere l'oggetto sul DB.
2. **DAO (Data Access Objects):** scriviamo a mano uno strato di codice per l'ORM, che ci fornirà degli oggetti necessari all'accesso ai dati (Data Access Objects, DAO). È simile alla soluzione precedente, ma separa l'applicazione dallo strato di accesso ai dati, migliorando usabilità e manutenibilità
3. **Persistence framework**, come LINQ per .NET.

La **forza bruta** è un approccio da evitare, se non in casi molto semplici, perché accoppia fortemente le classi del dominio al DB: richiede quindi che chi sviluppa le classi del dominio conosca dettagliatamente il DB usato, e inoltre viola l'information hiding perché all'applicazione non vengono nascosti i dettagli del DB (modifiche alla struttura della base di dati impatteranno anche sulle classi di business).

Nell'approccio **DAO** invece si crea uno strato, chiamato DAO o persistence layer, che si occupa solo dell'interazione col database: le classi di business per accedere al DB dovranno invocare i metodi di una classe del persistence layer, che costruirà lo statement SQL necessario, lo invierà al DB attraverso i driver, riceverà il risultato e lo passerà alla classe di business. Tipicamente, avremo una classe DAO per ogni classe del dominio. In questo modo abbiamo un buon information hiding: modifiche al database influenzeranno solo il persistence layer e non l'intera applicazione, però stiamo ancora facendo tutto a mano.

I **persistence framework** invece fanno da cuscinetto tra gli oggetti in memoria e il DBMS, permettendoci di ignorare il fatto che saranno salvati su un database: è un approccio black box alla persistenza, in cui l'applicazione salva e recupera i dati da "qualcosa" di cui non conosce il funzionamento interno: l'unico momento in cui il programmatore vede il DB è quando configura il framework. Gli statement SQL inviati al DBMS saranno autogenerati dal framework. In questo modo l'incapsulamento è completo: interagiamo col DB solo al momento di configurare il framework, dopo è tutto nascosto. Questo ci elimina tutta la parte di codice che converte le tuple in oggetti (stimata essere il 30/40% di tutte le applicazioni JDBC). Elimina inoltre tutti i problemi derivanti da variazioni di SQL vendor specific: posso cambiare il DB che sta sotto semplicemente comunicando al framework che sto usando un altro DBMS, senza dovermi preoccupare di cambiare tutti i dettagli vendor specific nelle query perché vengono create direttamente dal framework.
L'utilizzo di persistence framework permette inoltre di pensare le applicazioni in maniera completamente object-oriented, modellando e gestendo i dati senza vincoli legati al modello relazionale.
Quando definiamo un oggetto come persistente, ogni modifica verrà propagata automaticamente al DB: il persistence framework gestirà automaticamente il mapping tra gli oggetti e le tabelle del DB.

Perché allora non si usano solo i persistence framework? Riducono notevolmente l'efficienza, le query generate automaticamente sono molto più lunghe e complesse di una query che fa la stessa cosa ma scritta a mano. Inoltre mi pongono dei vincoli su come deve essere strutturato il

database, e questo potrebbe essere un problema per sistema legacy in cui non possiamo modificare la struttura del DB.

6.1.2 Hibernate

Hibernate è un persistence framework open source per Java. Un'applicazione che sfrutta Hibernate per la gestione della persistenza avrà:

- Le classi Java del dominio applicativo
- Una base di dati
- Un file XML che definisce il mapping di ogni classe persistente, che deve soddisfare la grammatica specificata nel DTD hibernate-mapping-3.0.dtd
- Un file di configurazione (sempre in XML) di Hibernate. Contiene le informazioni necessarie alla connessione alla base di dati come DBMS, diver JDBC, credenziali ecc. Deve soddisfare la grammatica specificata nel DTD hibernate-configuration-3.0.dtd
- Le interfacce, contenute nel package org.hibernate, per l'accesso alla base di dati (Session, Transaction e Query).

Esistono tre diversi approcci alla persistenza con Hibernate: seguendo un approccio **top down** il database verrà generato a partire dalle classi di dominio, seguendo un approccio **bottom up** invece il codice delle classi persistenti sarà generato da Hibernate a partire dalla struttura di un database preesistente. Con l'approccio **meet in the middle** invece abbiamo sia un diagramma di classi del dominio sia una base di dati preesistente.

Le classi del dominio che vogliamo rendere persistenti (PoJo, Plain old Java objects) devono avere alcuni requisiti:

- Un campo di tipo long da usare come chiave primaria
- Un costruttore senza argomenti
- Tutti gli attributi devono avere getter e setter
- I campi corrispondenti a collezioni devono avere tipo interfaccia
- I metodi equals() e hashCode() devono essere definiti sulla base della **business key**: l'insieme di attributi che identifica univocamente l'oggetto nel dominio del sistema (quindi occhio che **chiave primaria ≠ business key**)

Esempio di file di mapping Hibernate (in qualche esame è uscito fuori come domanda facoltativa):

```
<!DOCTYPE hibernate-mapping PUBLIC
     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
     "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.hbm">

<class name="Event" table="EVENTS">
     <id name="id" column="EVENT_ID">
          <generator class="increment"/>
     </id>
```

```
    <property name="date" type="timestamp" column="EVENT_DATE"/>
    <property name="title"/>
</class>
</hibernate-mapping>
```

Per quanto riguarda le interfacce, Hibernate ne fornisce di tre tipi: l'interfaccia **Session**, le cui istanze rappresentano sessioni di comunicazione tra l'applicazione e la base di dati, che offre metodi per il salvataggio/caricamento di oggetti dalla/sulla base di dati; l'interfaccia **Transaction**, le cui istanze rappresentano una transazione: gestisce le transazioni in un sistema che accede a più basi di dati all'interno di un'unica unità di lavoro. E' la più disaccoppiata dall'applicazione: non richiede l'utilizzo dell'API JDBC per impostare una transazione. L'interfaccia **Query** infine è quella che permette di definire delle query, sia in SQL che in HQL (linguaggio di query per Hibernate) che potranno usare al loro interno delle variabili speciali.

7. Design patterns

Un design pattern è una soluzione generale, semplice ed elegante ad un problema ricorrente di design o di programmazione. Dev'essere una soluzione non ovvia ed ampiamente utilizzata, in modo che sia arricchita dall'esperienza di diversi utenti. Esistono molti cataloghi di design pattern, solitamente relativi ad un particolare dominio applicativo o tipologia di applicazioni. Il più importante libro sui design pattern, che contiene tutti quelli più impotanti, è quello della Gang of Four. L'utilizzo di design pattern aiuta ad applicare i principi di programmazione object oriented e ad avere dei design migliori.
Essendo soluzioni generiche a problemi frequenti, quindi applicabili in diversi contesti, non sono legate ad un particolare linguaggio o piattaforma e costituiscono un'**unità di riuso**.
Altre unità di riuso sono le API/librerie, i componenti riusabili (COTS) e i framework: applicazioni "incomplete" che assumono il controllo dell'applicazione. Un **framework** è un insieme di classi e interfacce cooperanti che realizzano un design riusabile e customizzabile per un certo dominio applicativo. Ci dettano l'architettura del software: mentre con una libreria siamo noi a chiamarne le funzioni, decidendo il flusso di controllo del sistema, usando un framework sarà lui ad averne il controllo: noi ci limitiamo a creare delle sottoclassi che customizzano il framework per la nostra applicazione. Per questo usare un framework è più complesso che usare una libreria: per usare un framework infatti dobbiamo prima capirne la logica applicativa, perché dovremo basarci su di essa per sviluppare. Con una libreria invece ho solo un insieme di funzioni da chiamare quando voglio.Inoltre, mentre in una libreria le operazioni (le funzioni) sono già pronte e devo solo chiamarle, in un framework le dovrò scrivere.
Un'altra importante unità di riuso è la product line: famiglie di prodotti basate su un insieme comune di componenti (core asset base) a cui poi vengono aggiunte le componenti di ciascun software specifico. Anche i modelli del dominio, del business, dei requisiti, del design sono unità di riuso.

7.1 Design pattern GRASP

I GRASP sono un insieme di 9 pattern per l'assegnazione di responsabilità nel software. Alcuni non li considerano neanche dei pattern ma semplicemente dei principi di buona programmazione, alla base della progettazione OO: chi vuole imparare a sviluppare bene in object oriented dovrebbe imparare questi pattern.

GRASP pattern controller
Problema: stabilire qual è il primo oggetto oltre lo strato di UI a ricevere e coordinare un'operazione di sistema (=un evento di input princiaple del sistema).
Soluzione: assegno la responsabilità ad una classe che rappresenta il sistema complessivo, oppure ad una classe che rappresenta lo scenario di un caso d'uso all'interno del quale si verifica l'evento.

7.2 Gang of Four design patterns

Si suddividono in creazionali, strutturali e comportamentali. La descrizione di un pattern segue un template preciso:

- nome significativo
- descrizione del problema, per capire quando va utilizzato il pattern
- descrizione astratta della soluzione
- risultati e tradeoffs conseguenti all'applicazione del design pattern.

I pattern creazionali risolvono il problema "chi crea un oggetto di tipo A?". Un esempio di pattern creazionale è il GRASP creator, che assegna alla classe B la responsabilità di creare un'istanza della classe A se è vera almeno una di queste condizioni:

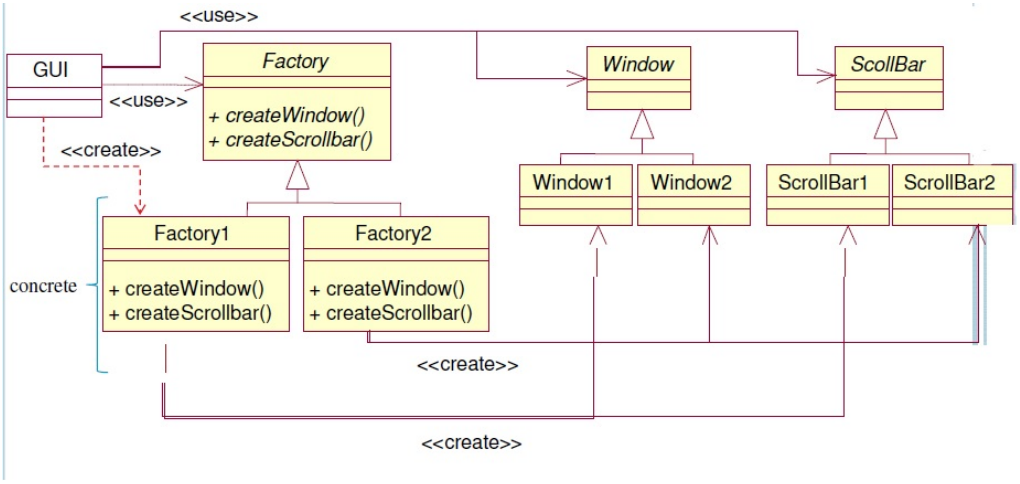
- B contiene o aggrega oggetti di tipo A
- B utilizza strettamente A
- B possiede i dati per inizializzare A

Come detto prima questo pattern è molto intuitivo per chi programma in OO, può venire in mente anche senza averlo mai sentito parlare di design pattern. Ci sono però dei casi in cui applicare il dp Creator non conviene.

7.2.1 Factory

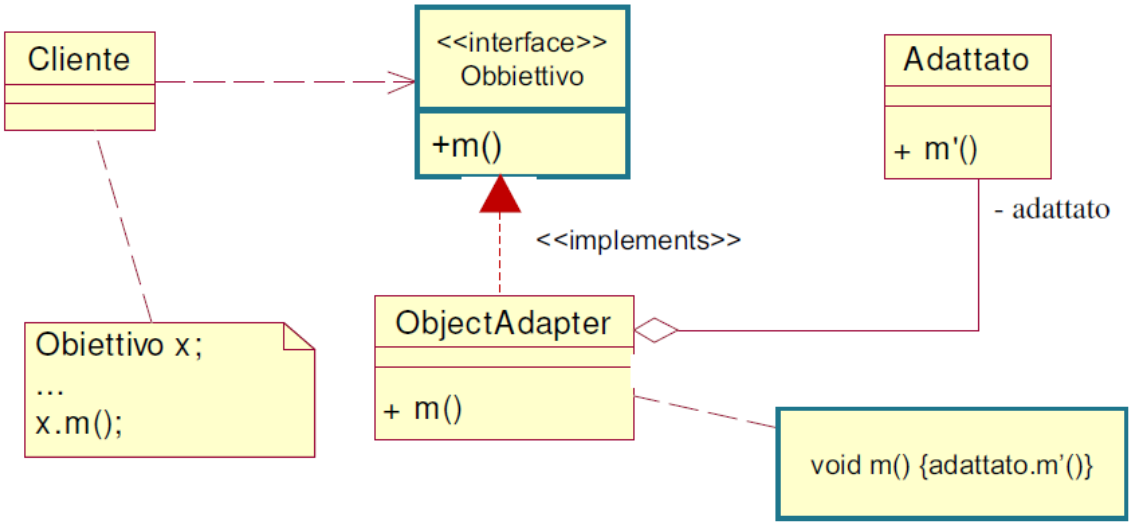
Design pattern creazionale in cui si ha una classe il cui compito è creare oggetti di una certa classe. Va adottato quando si vuole nascondere una logica di creazione complessa, quando si vuole separare la logica di creazione dalla pura logica applicativa oppure quando si vogliono adottare strategie di gestione della memoria (caching o riciclaggio di oggetti).
Un evoluzione del pattern Factory è l'**Abstract Factory**: fornisce un'interfaccia per creare non un singolo prodotto ma una famiglia di prodotti, in modo che non ci sia la necessità di creare istanze di classi concrete all'interno del codice.
Un caso in cui potrebbe servirci l'abstract factory è quello in cui vogliamo progettare una GUI platform independent: non dovrà quindi sapere quali classi concrete istanzia, perché quello sarà un dettaglio dipendente dalla piattaforma. Dobbiamo però evitare che sbagli gli accoppiamenti, evitando ad esempio che accoppi una scroll bar per la piattaforma 1 ad una finestra per la piattaforma 2.
Per fare questo la GUI non chiama i costruttori degli oggetti, ma utilizza i metodi di una factory. Avrò una factory per ogni piattaforma supportata dalla GUI, e discenderanno tutte da una stessa classe astratta AbstractFactory, in modo da poterle sostituire cambiando solo una riga di codice (quella in cui si istanzia la factory) senza effetti collaterali sul resto della GUI.
L'utilizzo del patrern abstract factory isola le classi concrete, in modo che i client non sappiano che tipo di classi concrete stanno utilizzando: rende estremamente facile cambiare la famiglia di prodotti in uso (basta una sola riga di codice).

Lo svantaggio è che per introdurre nuove (famiglie di) classi dovremo andare a modificare l'interfaccia dell'abstract factory e ad introdurre nuove classi astratte e concrete.



7.2.2 Adapter

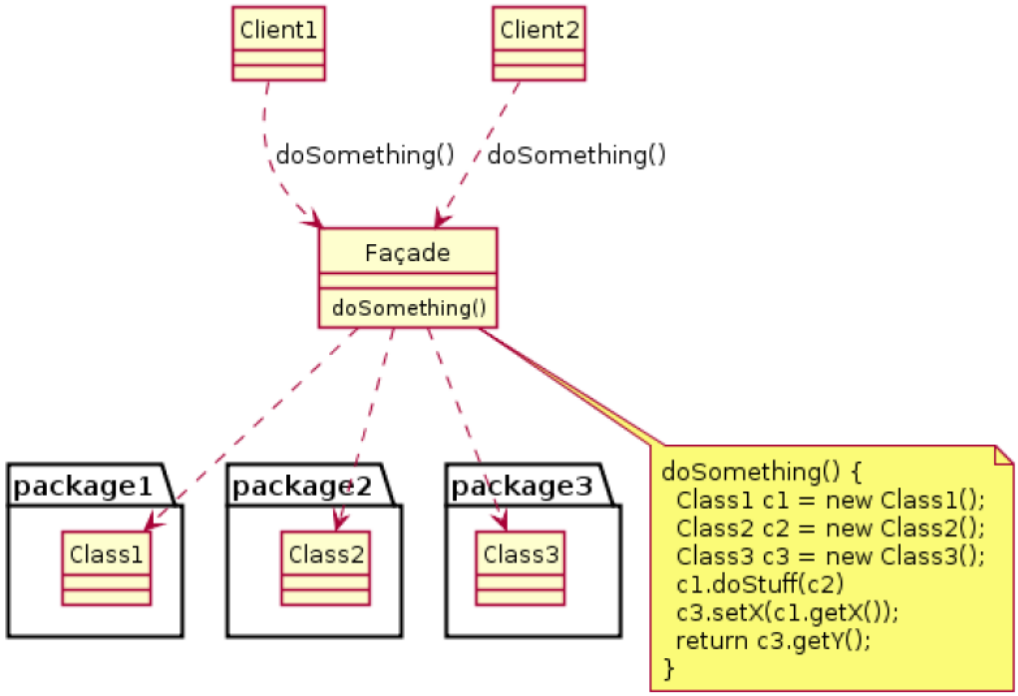
Design pattern strutturale che serve a convertire l'interfaccia offerta da una classe in un'interfaccia attesa dal client, permettendo a classi incompatibili di lavorare assieme. Avremo la classe adapter, con al suo interno un campo della classe da adattare.
Se ad esempio stiamo lavorando su un tool di disegno, che ha tre classi Punto, Linea e Quadrato tutte e tre discendenti da figura. Vogliamo aggiungere un'altra classe Circle, che però offre un'interfaccia diversa, e non la possiamo modificare perché non ne abbiamo il sorgente. La soluzione è creare una classe adapter Cerchio, che offre l'interfaccia attesa, con all'interno un campo di tipo Circle. I metodi della classe Cerchio, che fanno parte dell'interfaccia attesa, non faranno altro che invocare i corrispondenti metodi nella classi Circle, permettendo al sistema di utilizzarla senza rendersi conto che è una classe diversa dalle altre.
Ce n'è di due tipi: object adapter (quello dell'esempio delle slides) e class adapter, in cui l'adapter eredita sia l'interfaccia attesa sia l'interfaccia adattata. Se implementato in un linguaggio senza ereditarietà multipla, l'interfaccia attesa dovrà per forza essere un'interfaccia e non una classe.



7.2.3 Façade

Questo design pattern strutturale serve a semplificare l'accesso a sottosistemi che forniscono interfacce complesse, ed è molto usato nel disegno 2D e 3D, nei compilatori e nei sistemi di controllo di robot. Si utilizza quando una classe client, per portare a termine un'operazione, ha

bisogno di invocare diversi metodi di diverse classi di un sottosistema: il design pattern Façade interpone una classe tra il client e il sottosistema, che offrirà un metodo per compiere l'operazione desiderata. Questo metodo al suo interno invocherà tutti i metodi che la classe client avrebbe dovuto invocare senza l'utilizzo del pattern. Nell'esempio del robot, per far spostare un oggetto ad un braccio robotico andranno invocati diversi metodi di diversi sottosistemi: telecamera, movimento del braccio, controllo della pinza per afferrare l'oggetto ecc. Con l'uso del pattern Façade avremo un solo metodo spostaOggetto() che si smazza tutto il lavoro, riducendo le linee di codice del client.
L'uso di questo pattern promouove un accoppiamento debole fra client e sottosistema, nascondendo al client le componenti complesse del sottosistema ma senza impedirgli di usarle direttamente, se dovesse servire. Façade e Adapter sono entrambi wrapper, ma mentre Façade semplifica le interfacce Adapter le converte e basta.



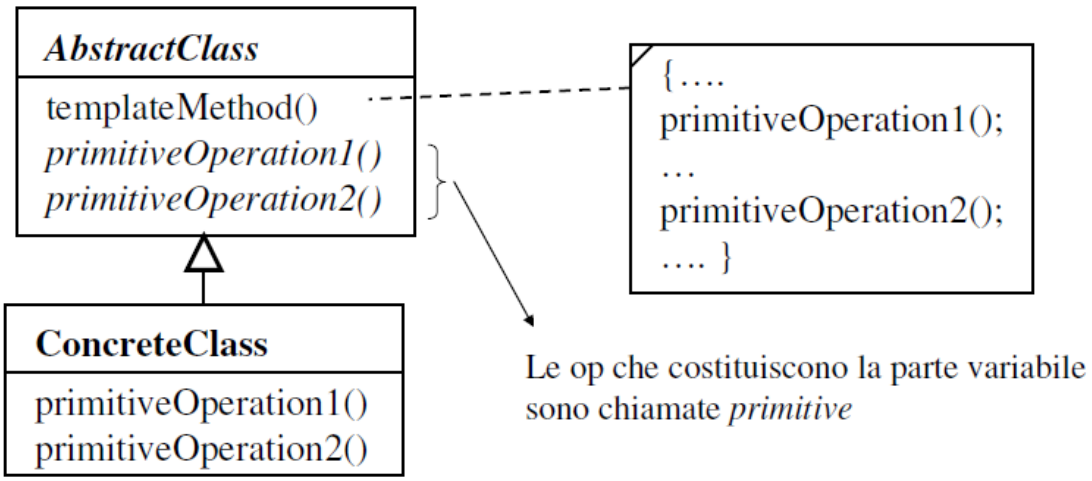
7.2.4 Template Method

Design pattern comportamentale che permette di definire lo scheletro di un algoritmo in un metodo, lasciando al client la responsabilità di definire le parti mancanti. Serve a ridurre al minimo la duplicazione di codice, definendo le parti invariabili di un algoritmo una volta per tutte e lasciando che siano le sottoclassi a definire le parti variabili. È il pattern alla base del concetto di framework: operazioni incomplete da completare per avere l'applicazione funzionante. Si basa sull'**Hollywood principle**: "non chiamate, richiameremo noi": c'è **inversione del controllo** tra sottoclassi e superclassi. Generalmente sono le sottoclassi ad invocare metodi delle superclassi, così invece avviene il contrario: le superclassi invocano metodi che le sottoclassi ridefiniranno nel modo necessario.
Un esempio è una classe riusabile, lato client, per il login degli utenti, che dovrà

1. **chiedere id e password**
2. autenticare l'utente producendo un oggetto che incapsuli le informazioni necessarie
3. **mostrare all'utente un display animato durante l'autenticazione**

4. notificare all'utente quando il login è completo, rendendo l'oggetto prodotto dall'autenticazione disponibile all'app

I punti in grassetto sono quelli invariabili: su qualunque sistema mi stia loggando dovrò chiedere username e password e mostrare un'animazione durante l'autenticazione. I passi necessari all'autenticazione e l'oggetto prodotto in seguito al login invece dipenderanno dallo specifico sistema su cui mi sto loggando.
E' una tecnica fondamentale per ridurre duplicazioni di codice ed evitare di imporre la ridefinizione di tanti metodi alle sottoclassi, ma è importante avere ben chiaro dall'inizio cosa va messo nella superclasse e cosa va lasciato da definire alle sottoclassi.



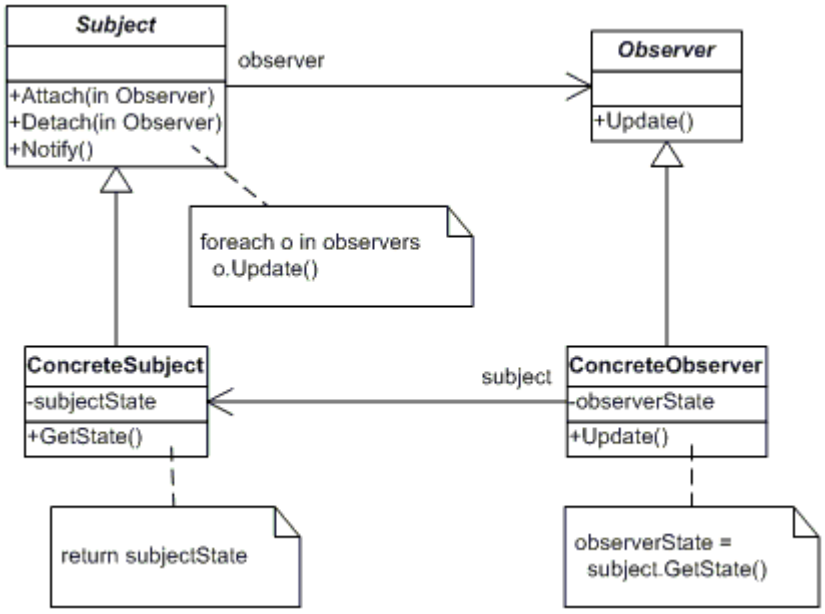
7.2.5 Observer

Il pattern comportamentale Observer serve quando abbiamo degli oggetti che vogliono essere informati sui cambiamenti di stato di un altro oggetto. Definisce una dipendenza uno-a-molti tra gli oggetti in modo che, quando l'oggetto dal lato 1 cambia stato, tutti gli altri sono notificati e aggiornati automaticamente. I partecipanti sono:

- Una classe astratta Subject, che fornisce i metodi Attach, Detach e Notify.
- Una classe concreta ConcreteSubject, che estende Subject.
- Una classe astratta Observer, che fornisce un metodo Update.
- Una (o più) classi concrete ConcreteObserver, sottoclassi di Observer, che conterranno al loro interno un campo con lo stato dell'oggetto osservato.

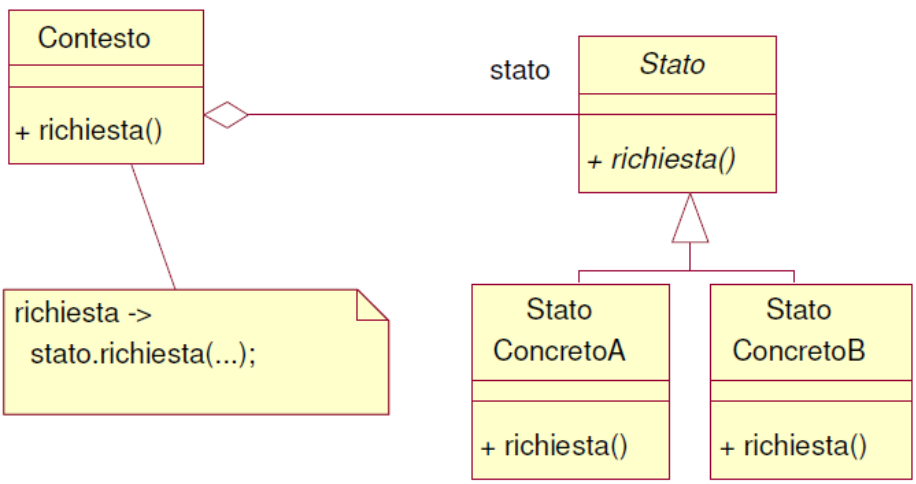
Il metodo Attach viene invocato dai ConcreteObservers per registrarsi come osservatori: il Subject avrà al suo interno una collection di reference verso gli osservatori per notificarli quando lo stato cambia. Il metodo Notify verrà invocato dal ConcreteSubject quando il suo stato cambia: scopo del metodo è andare ad invocare il metodo Update degli Observer: l'invocazione di quest'ultimo fa sapere all'osservatore su cui è stato invocato che il soggetto osservato ha cambiato stato.
Applicando questo pattern otteniamo un collegamento molto lasco ed astratto tra soggetto e osservatori: l'unica cosa che devono conoscere delle rispettive interfacce sono i metodi Attach, Detach, Notifify e Update.

Questo pattern è alla base delle interfacce grafiche, ed è utilizzato anche in tutte quelle applicazioni in cui vengono offerte più viste di un'unica fonte di dati: ad esempio un foglio di calcolo, su cui abbiamo creato diversi grafici: quando cambiamo i dati contenuto nel foglio cambiano anche i grafici.



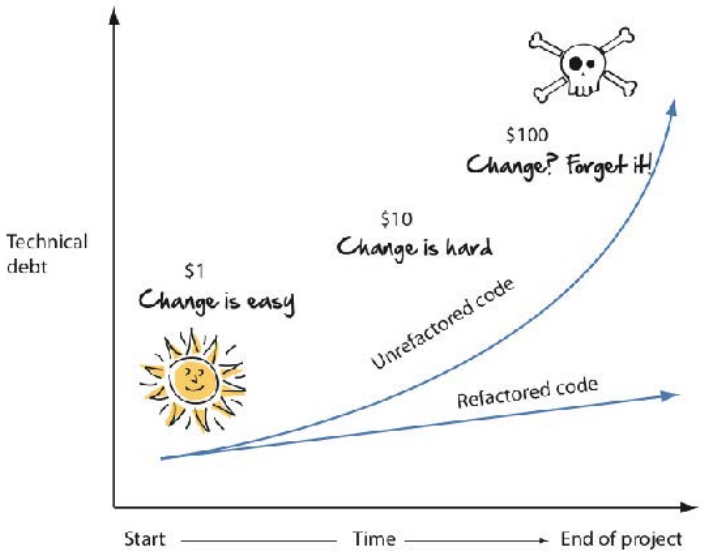
7.2.6 State

Design pattern comportamentale pensato per avere oggetti che cambiano il loro compoartamento al variare dello stato esterno. Per fare questo utilizza polimorfismo e gerarchie di ereditarietà: mettiamo in una classe (astratta) i metodi il cui comportamento deve variare al variare dello stato, e creiamo una sottoclasse per ogni stato. In ogni sottoclasse, i metodi saranno implementati in maniera diversa. In questo modo i client possono vedere l'oggetto col tipo della superclasse ed invocarne i metodi, che avranno un comportamento diverso a seconda della classe concreta con cui è stat istanziato l'oggetto. Senza utilizzare questo pattern, per ottenere lo stesso risultato dovremmo avere lunghi switch/if per sapere in che stato si trova l'oggetto e a seconda di questo eseguire un metodo diverso. L'effetto collaterale del design pattern State è che aumenta il numero di classi del nostro sistema: se prima avevamo una classe che poteva trovarsi in tre stati ora ne avremmo 4, la superclasse con una sottoclasse per ogni stato.



8. Refactoring

Per Fowler, il refactoring è "un cambiamento della struttura sintattica del software che non ha effetti sul comportamento". Viene modificata la struttura del codice, per renderlo più chiaro e leggibile, non la semantica: è come trasformare l'espressione aritmetica (x-1)(x+1) in (x^2-1). Non va quindi confuso con bug fixing, cambi di linguaggio o piattaforma (porting) o cambiamenti del design: sono tutte cose diverse.
Fare refactoring frequentemente è importante perché previene la cosiddetta **design erosion/decay**, ovvero il peggiorare della struttura del codice col passare del tempo: anche un codice che inizialmente aveva un buon design, se nel tempo viene aggiornato senza mai fare refactoring la qualità del design decrescerà col tempo, fino ad arrivare al punto in cui sarà più conveniente rimpiazzare del tutto il codice esistente che andare a metterci le mani.



Si vede però che, anche facendo refactoring, il costo della manutenzione aumenta comunque col tempo: un minimo di decadimento nella qualità del design del sistema è inevitabile quando un sistema viene mantenuto aggiornato.
Il refactoring quindi preserva la leggibilità del codice, aumentandone la manutenibilità. Permette anche di trovare bug che non rimanevano nascosti nella prima stesura del codice.
Quando va fatto il refactoring? Conviene non pianificare lunghi cicli di refactoring dopo lunghi cicli di programmazione, ma rifattorizzare molto spesso, ad esempio ogni volta che vogliamo aggiungere una nuova funzionalità al sistema, ogni volta che troviamo un bug e ogni volta che viene rilevato un **code smell**. I sostenitori dei metodi agili affermano che l'unità di misura per il tempo tra un refactoring e l'altro dovrebbero essere i minuti.
Un **code smell** è un indicatore che nel codice c'è qualcosa che non va: potrebbe essere solo un problema di stile, come un nome non standard, o qualcosa che riduce la leggibilità, ma potrebbe anche essere nascondere qualcosa di più grave, come un problema di design. I tool di analisi come STAN4J sono in grado di rilevare automaticamente alcuni code smell.
La presenza di un code smell non indica necessariamente qualcosa che non va: ad esempio un tipico code smell è un metodo troppo lungo, ma non è detto che sia un problema: potrebbe essere necessario avere un metodo così lungo.
Alcuni code smell che possono venirci segnalati possono essere raggruppati in quelli causati da troppo codice o da troppi poco codice. Tra quelli causati dal troppo codice ci sono metodi troppo lunghi, classi troppo grandi, duplicazione di codice, codice che non viene mai eseguito (dead code),

liste di parametri troppo lunghe. Quelli invece relativi al troppo poco codice sono le classi di soli dati o le catch clause vuote.

I **cloni software** (duplicazioni di codice) sono un tipo di code smell particolarmente problematico perché propagano i bug: un errore nel codice clonato andrà corretto a mano in ogni copia, e anche in assenza di bug complicano la manutenzione perché ogni modifica andrà ripetuta in ogni clone.

Il refactoring va iniziato solo quando il codice passa tutti i test: a questo punto si cercano i code smell e si determina il refactoring più adatto; dopo averlo applicato si ripetono i test per essere sicuri che il codice funzioni ancora (e che quindi il refactoring non ne abbia modificato il comportamento). È importante evitare di fare il passo più lungo della gamba: se applichiamo troppi refactoring tutti insieme il rischio di introdurre errori aumenta. La sequenza di operazioni dovrebbe essere **esecuzione con esito positivo dei test -> individuare code smells -> determinare refactoring adatto -> applicare refactoring -> rieseguire i test.**
I refactoring possibili sono tantissimi, raccolti in libri come quello di Fowler, e possono essere complessi o low-level. I low level refactoring, come rinominare o estrarre un metodo, sono il punto di partenza per i refactoring più complessi: questi ultimi possono essere usati per introdurre dei design pattern.
I refactoring possono essere eseguiti manualmente o attraverso tool di supporto, tipicamente offerti dagli IDE. I tool però offrono solo un sottoinsieme dei refactoring possibili, solitamente un sottoinsieme dei refactoring più semplici. Esistono plugin per estendere i refactoring disponibili in un IDE (JDeodorant per Eclipse, ReSharper per Visual Studio). Sono molto comodi perché si occupano anche di gestire gli effetti collaterali di un refactoring: se ad esempio rinomino una classe o un metodo, dovrò riportare la modifica in tutti i punti in cui il metodo o la classe venivano usati: ma se per rinominare uso un tool, sarà lui ad occuparsene.
Con Eclipse possiamo anche estrarre un'interfaccia da una classe, selezionando a mano quali metodi vanno inclusi; spostare in una superclasse (push up) o in una sottoclasse (pull down) campi e metodi; esplicitare una inner class anonima; o spostare i metodi (**move method**): quest'ultimo si applica quando abbiamo classi con troppo behavior, o troppo accoppiate, che collaborano troppo: come trovare però i metodi da spostare? Dei buoni candidati sono quelli che sembrano riferirsi più ad altre classi che alla classe cui appartengono. Per non modificare l'interfaccia del sistema, è possibile lasciare il metodo vecchio, che conterrà solo una chiamata al metodo spostato.
Anche i troppi punti di decisione in una stessa espressione sono un code smell.

8.1 Low-level refactoring

Replace temp with query
Offerto sia da ReSharper che da Eclipse, serve a limitare l'utilizzo di variabili locali all'interno di metodi. L'idea è di rimpiazzare le variabili locali con un metodo query, che ottiene l'informazione che prima era memorizzata dalla variabile locale. Il tradeoff qui è con l'efficienza: non conviene applicare questo refactoring quando l'operazione da incapsulare nel metodo query è frequente, in questo caso mi conviene tenere la variabile locale invece di ricalcolarla ogni volta.

Replace parameter with method
Cercare di passare meno parametri possibili ai metodi, facendo in modo che calcolino da se tutto quello che possono e che ricevano dall'esterno solo lo stretto indispensabile.

Extract class
Da applicare quando abbiamo una god class: creiamo una nuova classe e ci spostiamo alcuni attributi e operazioni della god class. Ovviamente non va fatto a caso, ma spostando attributi e operazioni correlate. Come per il refactoring move method, è possibile lasciare le deleghe (metodi col vecchio nome che chiamano il metodo spostato nella nuova classe) nella classe originale.

8.2 Refactoring complessi

Solitamente, quando applicati introducono un design pattern nel nostro codice.

Replace inheritance with delegation
Da usare quando una sottoclasse usa solo una parte della sua superclasse, e vuole evitare di ereditare anche tutto il resto. Se ad esempio voglio definire uno stack su vector, se definisco la classe Stack come erede di Vector si porterà dietro anche ad esempio l'operazione insertElementAt, che su uno stack non ha senso. Applicando il refactoring Replace inheritance with delegation invece di definire Stack come erede di Vector mettiamo una variabile d'istanza di tipo Vector all'interno della classe Stack, offrendo all'esterno solo i metodi che hanno senso su uno stack.

Replace conditional with polymorphism
Questo refactoring serve ad introdurre lo State pattern nel codice: sostituisce le decisioni condizionali prese basandosi sul tipo di un oggetto con comportamenti basati sul polimorfismo, in modo che la decisione venga presa, più efficientemente, dall'ambiente di esecuzione. Va creata una sottoclasse per ogni decisione, con un metodo base astratto che verrà ridefinito in ogni sottoclasse, in modo che abbia il comportamento desiderato.

Separate domain from presentation
Va applicato quando abbiamo un'applicazione che mischia interfaccia e business logic, senza "separation of concerns". In questa situazione potremmo avere una god class Window che contiene tutto: ci dovremo estrarre le classi relative alla business logic applicando diversi refactoring low level (extract method, extract class ecc.). È uno dei refactoring più complessi.

9. Testing

Nelle slides c'è la descrizione del processo di rilevamento-correzione dei bug.
Un **fault** è un errore nel codice che porta ad un **failure**, che è la manifestazione dell'errore. La catena con cui si verificano è: errore umano che porta ad un fault che può portare ad un failure. Può portare perché non è detto che un errore nel codice poi si manifesti: se infatti il sistema non riceve mai un input che solleciti le istruzioni errate il fault può rimanere nascosto per anni. Per questo la fase di testing è fondamentale durante lo sviluppo di un progetto.
A rendere difficile la fase di testing è in particolare la scelta degli input giusti che riescano a rilevare i failure. Spesso non è condotta in modo adeguato non solo per la sua difficoltà intrinseca ma anche e soprattutto per i costi e perché in genere è l'ultima fase del processo di sviluppo: siccome i progetti software sono quasi sempre in ritardo si tende a tagliare su questa fase.
Il **testing** è una procedura sistematica che esegue un sistema detto SUT con l'intento di rilevare i failure. Per farlo utilizza una serie di input, che sono un sottoinsieme degli input possibili, e verifica che il risultato ottenuto sia uguale al risultato atteso. Una volta rilevato un failure si esegue il **debugging**, il processo per rilevare un bug/fault e rimuoverlo, che si compone in due fasi: fault localization (trovare l'errore) e rimozione del fault.
Il testing esaustivo, con tutti gli input possibili, è impossibile in contesti reali: anche un semplicissimo programma che somma due interi a 32 bit dovrebbe sommare tutti i numeri da 2^{-32} a 2^{32}, per un totale di 2^{64} numeri: eseguendo un'operazione al millisecondo ci metteremmo più di 500.000 anni. Quindi, un buon testing inizierà dalla scelta degli input migliori, quelli che hanno più possibilità di far saltar fuori gli errori. Abbiamo due approcci di scelta degli input: **white box** e **black box**. Il **white box testing** è basato sulla conoscenza del SUT: sappiamo com'è costruito il sistema e usiamo questa conoscenza per scegliere i test e gli input migliori. È chiamato anche structural testing, e ne è un esempio lo statement coverage testing, che prevede che durante l'esecuzione dei test venga eseguito ogni singolo statement del codice del progetto.
Nel **black box testing** (o functional testing) invece non sappiamo nulla del funzionamento del SUT, la scelta degli input solitamente si fa a partire da specifiche e requisiti del sistema. In questo approccio è molto importante l'esperienza del tester. Non esiste un metodo migliore in assoluto, molti testi di ingegneria del software affermano che i due metodi sono complementari e dovrebbero essere utilizzati entrambi su ogni progetto. Uno studio del 1987 però afferma che con tester esperti il testing funzionale rileva più bug del testing strutturale.

Un **testcase** è, secondo la definizione dell'IEEE Software Engineering Standard Glossary, un'insieme di input per rivelare gli errori in un sw o verificarne l'aderenza ad uno standard. Una **testsuite** è un insieme di testcase.
Cos'è concretamente un testcase dipende dal contesto: per un semplice programma che preso un input fornisce un output sarà solo un'insieme di input. per qualcosa di più complesso come una web app sarà una sequenza di operazioni.
Il testing si svolge su diversi livelli:

- **testing di unità**: la singola unità di codice (funzione, metodo, classe, a seconda di cosa stiamo sviluppando) viene testata isolandola il più possibile dal resto, attraverso l'utilizzo di stub e mock objects. Solitamente, lo unit testing viene svolto con approccio white box.
- **testing di integrazione**: i moduli vengono testati insieme, composti in quello che poi diventerà il sistema definitivo. Viene testata la comunicazione tra i moduli e si verifica quali informazioni vengono scambiate: è in questa fase che escono fuori gli errori causati dalle interfacce tra moduli, come le precondizioni non rispettate, i parametri messi in ordine sbagliato ecc. Solitamente è la fase in cui si verificano più problemi.

- **testing di sistema**: in questa fase si verifica che il sistema risponda ai requisiti e che soddisfi i bisogni dell'utente, che viene coinvolto nelle fasi finali di questo stadio (acceptance testing). Solitamente viene svolto con approccio black box, visto che uno degli scopi principali è verificare l'aderenza ai requisiti. In questa fase non si verifica solamente la correttezza dei risultati, ma anche altri aspetti come performance, usabilità e sicurezza.
- **testing di regressione**: è il testing svolto dopo il rilascio, nelle fasi di manutenzione: poiché ogni modifica al sistema comporta il rischo di effetti collaterali anche in zone che non sembrerebbero essere coinvolte dall'intervento, ad ogni modifica introdotta si riesegue l'intera test suite per verificare che la nuova modifica non abbia side effects.

JUnit: praticamente uguale a NUnit ma su Eclipse e Java invece che Visual Studio e .NET.
Un **failure** è un'asserzione che non viene verificata, un **error** quando si verifica un errore a run-time (eccezione lanciata).

9.1 White box e black box testing

Dato che il testing esaustivo è impraticabile, uno dei principali problemi del testing è quello di trovare dei buoni input per i test case, quelli in grado di portare i failure a manifestarsi. Il white box testing è basato sulla conoscenza della struttura del SUT, il black box tesing invece ne conosce solo il comportamento atteso.

9.1.1 White box testing

Come possiamo valutare la bontà di una test suite? Non dal numero di fault trovati, perché quello dipende anche dal codice testato (su un codice che non contiene errori, anche la migliore test suite non ne dovrà trovare). Una scelta più sensata è misurare "quanto approfonditamente" va a testare il SUT: è questo il concetto di **copertura (coverage)**. Come unità di misura utilizza **le unità di copertura**, cioè elementi che abbiano queste due proprietà:

- è possibile contare il numero di unità nel software
- è possibile identificarle quando vengono eseguite dalla test suite (hit)

La copertura del test sarà il numero di hit, ovvero di unità di copertura eseguite dalla test suite. Come possibile, semplice unità di copertura possiamo scegliere le linee di codice (**code coverage**): misuriamo quante linee di codice vengono eseguite dalla test suite, e la copertura sarà data dalla percentuale di linee eseguite rispetto alle linee totali. La coverage può essere basata non solo sul codice, ma anche su modelli come il grafo di flusso di controllo, le state machines (per codice OO) e anche su requisiti e specifiche (usata nel black box testing).
Il **control flow graph** è un grafo che mostra tutti i possibili cammini che possono essere attraversati durante l'esecuzione di una porzione di codice. Ci sono quattro criteri di copertura basati sul control flow graph:

- **statement coverage**: guarda quanti statement si attraversano. Il meno potente dei quattro.
- **branch coverage**: guarda quanti punti di decisione si attraversano: è il livello di copertura minimo prescritto dallo standard IEEE per unit test.
- **multiple condition coverage**: controlla la copertura di tutti le combinazioni possibili delle condizioni dei nodi decisionali

- **all path coverage**: copertura di tutti i cammini del CFG. Un all path coverage del 100% è impossibile.

Di solito si sceglie un criterio con cui misurare la coverage dei test, si sceglie una percentuale di coverage che la testsuite dovrà raggiungere e si crea una testsuite seguendo il criterio scelto. La scelta del criterio è un tradeoff tra la probabilità di trovare più fault e la complessità di realizzare i test.
Lo **statement coverage** cerca i cammini che attraversano tutti i nodi statement, ed è il più debole criterio di copertura, perché lascia scoperti molti archi.
La **branch coverage** prevede di attraversare tutti gli archi del grafo: tutte le diramazioni dei punti di decisione, tutti i case degli switch e tutti i loop. Se applichiamo la branch coverage avremo automaticamente anche lo statement coverage (tutti gli archi del grafo sono coperti).
La **multiple condition coverage** prevede che

- ogni condizione atomica sia coperta
- per le condizioni composte da più condizioni atomiche, ogni combinazione di condizioni atomiche deve essere coperta.

Include branch e statement coverage: copre tutti i branch e tutti gli statement, ma non tutti i path.
L'**all path coverage**, il più potente dei quattro, prevede che tutti i path del CFG vengano coperti. Non è possibile adottarlo quando nel SUT ci sono dei loop: il numero di cammini crescerebbe troppo, dipendendo dal numero di volte che eseguo il loop. Questo può essere risolto non considerando il numero di volte che si esegue il loop, ma solo se viene eseguito o no. Un altro problema dell'all path coverage è che potrebbero esserci cammini per cui non c'è nessun input che porta ad eseguirli (infeasible paths). Sembra poco rilevante, dopotutto se un cammino non è percorribile non ha neanche senso cercare di testarlo, peccato che il problema di stabilire se un cammino è infeasible o no sia indecidibile.
A differenza di quello che avveniva con le tecniche precedenti, l'all path coverage non implica la multiple condition coverage (e viceversa).
Tutte le tecniche di white box testing hanno lo svantaggio di produrre un gran numero di casi di test: diventa inutilizzabile per il system testing, in genere viene usato solamente per lo unit testing. Inoltre, non è in grado di rilevare fallimenti dovuti a missing features errors, errori dovuti alla mancanza di funzionalità richieste dai requisiti.

9.1.2 Black box testing

Essendo slegato dalla conoscenza del funzionamento del sistema, può essere usato con qualsiasi tipo di software senza dover necessariamente conoscere le tecnologie che stanno alla base, la piattaforma o il linguaggio utilizzato, e può essere utilizzato sia per lo unit testing sia per il system testing. Casi di test per il testing black box possono essere scritti non appena sono disponibili i requisiti. Anche per il testing black box esistono diverse tecniche, come l'**equivalence partitioning**: cerco di suddividere il dominio degli input in classi di input che manifestano lo stesso comportamento: sono quindi classi di equivalenza. Il primo passo di solito è suddividere in input legali e illegali: i primi porteranno ad un risultato, i secondi ad un errore/eccezione I secondi verranno usati per il **negative testing**: quando sto testando come il sistema si comporta nei casi di errore. Queste classi andranno poi a loro volta suddivise.
Il partizionamento può essere **unidimensionale** o **multidimensionale**: il partizionamento unidimensionale prende in consideraizone una variabile di input alla volta ignorando le altre, il

partizionamento multidimensionale considera tutte le variabili assieme: di solito produce molte classi di input difficili da gestire manualmente.

Per la **boundary value analysis** (BVA, analisi dei valori limite) si scelgono i valori vicini al confine tra una classe di equivalenza e un'altra, assumendo che la maggior parte degli errori si verifichi per valori limite (esempio classico: ho messo un < invece di un <=). Questo principio, oltre a derivare dall'esperienza, è stato anche provato da alcuni studi empirici. Per applicare questa tecnica si procede per:

1. partizionamento unidimensionale degli input
2. identificazione dei confini di ogni partizione
3. selezione degli input in modo da comprendere i confini e i punti vicini ai confini

Infine nell'**approccio three-step** i casi di test vengono generati a partire dagli use cases. Bisogna generare un insieme completo di scenari per ogni use case, e per ogni scenario trovare le condizioni che mi permettono di eseguirlo (quasi-test case). Per ogni quasi-test case, identifico i valori di input e avrò il test case corrispondente.

10. Manutenzione ed evoluzione

La manutenzione di un software è la modifica del prodotto dopo il rilascio, per correggere difetti, migliorare le prestazioni o altri attributi, o per adattare il prodotto ad un ambiente modificato. È una fase necessaria perché le modifiche al software sono inevitabili: i bisogni dell'utente cambiano, il contesto in cui viene utilizzato il software si modifica: vengono promulgate nuove leggi, arrivano nuove piattaforme da supportare (esempio recente, i dispositivi mobili), sorge il bisogno mi migliore prestazioni ecc.
Le **leggi di Lehman** sono cinque leggi derivate da studi empirici sull'evoluzione di molti sistemi software di grandi dimensioni. Le prime due sono

1. **cambiamento continuo**: ogni sistema software, se non viene cambiato di continuo, diventa progressivamente meno utile
2. **complessità in aumento**: ogni cambiamento apportato al sistema ne aumenta la complessità.

La manutenzione del software è fondamentale perché durante gli anni le organizzazioni hanno fatto molti investimenti per i loro sistemi software, sono una risorsa critica per l'organizzazione che li usa, e che quindi ne vorrà mantenere il valore: Sommerville stima che nelle software house l'80% del tempo sia speso in operazioni di manutenzione, e solo il 20% nello sviluppo di nuovi progetti.
Il processo di manutenzione solitamente parte da una richiesta di modifica da parte del cliente e se, come accade nella maggior parte dei casi, la persona che si occupa della modifica non è la stessa che ha sviluppato il sistema, per prima cose dovrà comprenderlo, studiandone la documentazione ed il codice. Una volta capita la struttura del sistema bisognerà **localizzare** le porizioni di codice da modificare, quindi implementare i cambiamenti. Quasi sicuramente, questi cambiamenti avranno **effetti collaterali**, che andranno gestiti. Infine, con un **test di regressione** (testiamo anche le parti non modificate per verificare l'assenza di effetti collaterali) verifichiamo che tutto funzioni.

La manutenzione può essere

- **correttiva**, quando ha lo scopo di eliminare dei difetti del software
- **adattiva**, per adattare il software ad un ambiente di utilizzo (sistema operativo, hardware...) diverso da quello per cui era stato progettato inizialmente
- **migliorativa**, per aggiungere o modificare funzionalità del sistema in modo da soddisfare nuovi requisiti
- **preventiva**, per prevenire problemi futuri: anche se non abbiamo ancora ricevuto una richiesta di modifica, potrebbe essere necessario un refactoring per rendere il codice più gestibile per modifiche future, o interventi per prevenire un futuro degrado delle prestazioni. Solitamente le aziende non la fanno, perché sono troppo prese dalle altre.

Secondo Sommerville il 65% degli interventi di manutenzione sono migliorativi, il 18% adattivi e il 17% correttivi. Alcuni fanno una distinzione tra manutenzione ed evoluzione, considerando gli interventi correttivi e preventivi come manutenzione, quelli adattivi e migliorativi come evoluzione.

Gran parte della difficoltà degli interventi di manutenzione è data dalla comprensione del codice scritto da altri, soprattutto quando nel processo di sviluppo non si sono seguite le best practices, non si è prodotta sufficiente documentazione, o se si tratta di codice legacy non pensato per essere modificato in futuro. Queste difficoltà possono essere ridotte di molto adottando e rispettando processi di sviluppo che producano un'adeguata documentazione, tengano conto in ogni fase la necessità di modifiche future, prevedano frequenti refactoring e adottino convenzioni nella scrittura del codice in modo da renderlo più leggibile.
È fondamentale poi, prima di procedere con le modifiche, valutare bene l'impatto che avranno sul sistema, e fare opportuni test di regressione per assicurarsi che non ci siano effetti collaterali indesiderati.

La **documentazione** è fondamentale per la manutenzione, anche se una documentazione troppo corposa invecchia molto rapidamente. Deve essere una **live documentation**: sempre allineata col codice, non troppo lunga, pensata per facilitare i task di manutenzione.

I costi della manutenzione sono alti principalmente a causa della fase di comprensione del codice. Oltre a questo, il costo della manutenzione aumenta man mano che il software invecchia: questo perché la manutenzione (non preventiva) degrada la struttura del software e la qualità del design rendendo più difficili gli interventi successivi.

10.1 Sistemi legacy

I **sistemi legacy** sono quei sistemi per cui l'attività di manutenzione è diventata prevalente su tutte le altre: implementati diversi decenni fa, con linguaggi e tecnologie obsolete. Decenni di manutenzione ne hanno deteriorato enormemente la struttura, e molto probabilmente la documentazione (se esiste) non sarà più allineata al codice. Inoltre, possono contenere regole di business che non sono documentate altrove, e sarebbe molto difficile e costoso riprodurle su un altro sistema.
Come decidere se conviene o no buttare un sistema legacy? Queste domande ci aiutano a capire se la situazione è diventata troppo critica: se rispondiamo positivamente, vuol dire che ci conviene sostituire il sistema invece di mantenerlo:

- Il costo della manutenzione è troppo elevato?
- L'affidabilità del sistema è inaccettabile?
- È difficile adattare il sistema entro una scadenza ragionevole?
- Le prestazioni del sistema non rispettano i vincoli?
- Le funzionalità del sistema sono di utilità limitata?
- È possibile fare lo stesso lavoro meglio, più rapidamente o spendendo meno usando altri sistemi?
- Il costo di manutenzione dell'hardware è tale da giustificare la sostituzione con hardware nuovo più economico?

Se decidiamo di mantenere un sistema legacy, il nostro obbiettivo sarà migliorare la qualità del software e semplificare la manutenzione contenendo i costi.
Gli approcci principali alla manutenzione di sistemi legacy sono:

- la **ridocumentazione**, se quella che abbiamo è insufficiente, non allineata o ancora peggio non l'abbiamo.
- il **restructuring/refactoring**, permigliorare il codice quando è mal strutturato, ad esempio eliminando i goto.
- il **reverse engineering**, per ricostruire il design del sistema a partire dal codice, se non abbiamo a disposizione il design del sistema o se quello che abbiamo non è più allineato. A differenza del processo di forward engineering con cui si progetta un nuovo sistema, il reverse engineering parte dal livello di astrazione più basso (il codice) per arrivare al più alto 8specifiche dei requisiti e diagrammi UML).
- il **reengineering**, che si compone di reverse engineering, modifica di specifiche e design ed infine forward engineering di un nuovo sistema, serve a sviluppare un nuovo sistema basandosi su quello esistente: può essere ad un porting di un'applicazione da un linguaggio/piattaforma ad un'altra.

11. Extreme Programming

Extreme Programming (XP) è un processo di sviluppo per software OO, ideato per piccoli gruppi di sviluppatori (tra le 4 e le 20 persone). È inadatto per sistemi grandi e complessi o safety critical. Non utilizza UML o altri linguaggi di analisi o design, ma passa direttamente dalle user stories alla codifica. È stato ideato nel 1999 da Kent Beck, Ward Cunningham e Ron Jeffries dopo il fallimento del progetto C3 (un sistema per page e contributi che doveva rimpiazzare i sistemi legacy di Chrysler) a cui Beck aveva partecipato. Si chiama "extreme" perché estremizza le best practices: se le iterazioni corte sono buone, allora saranno il più corte possibile (nell'ordine dei giorni invece che delle settimane); se i test e l'integrazione sono importanti allora li faremo più volte al giorno; se la code review va fatta di frequente la faremo al momento stesso della codifica (pair programming).
I requisiti sono documentati con le **user stories**: sintetici casi d'uso scritti direttamente dal cliente e che, invece dei lunghi, formali e dettagliati use cases dei modelli plan driven sono dei semplici esempi, abbastanza corti da essere scritti su un post-it.
Gli sviluppatori, dopo aver compreso le user stories, le suddividono in punti e ad ogni punto attribuiscono un tempo di rilascio: a questo punto sarà il cliente a dire quali punti vuole che siano sviluppati per primi. Si segue un approccio di **small releases**: frequenti rilasci (uno ogni 1-4 settimane) di funzionalità che creano valore concreto, e che aiutano il cliente a capire meglio i suoi bisogni. In questo modo inoltre il cliente avrà fiducia nel progresso del suo progetto, perché lo vede in prima persona.
Per arrivare dalle user stories alle classi del linguaggio si utilizzano le **carte CRC (Class Responsibility Collaboration)**: schede su cui gli sviluppatori scriveranno le possibili classi del sistema, le loro responsabilità e le loro collaborazioni con altre classi.
In XP non si utilizzano CASE tool come Visual Paradigm: al massimo, quando un minimo di class/sequence diagram si rendono necessari per farsi capire, si fanno alla lavagna. Non si conservano documenti di design: vale il principio di **tacit/implicit knowledge**. Per avere sempre ben chiaro il design in assenza di documentazione si cerca di mantenere il codice il più chiaro possibile, attraverso refactoring continui e rispetto costante delle convenzioni, in modo che sia sempre facilmente leggibile per tutti. Anche i commenti nel codice vengono evitati il più possibile: il codice deve essere autodocumentato, comprensibile da solo attraverso struttura chiara e nomi significativi.
N.B.: queste sono le prescrizioni di Kent Beck nel suo libro del 1999: nella pratica, le aziende che adottano XP non le seguono pedissequamente: soprattutto per quanto riguarda la documentazione, conservarne un minimo spesso è necessario.
XP non pianifica per il riuso, ma perché lo sviluppo sia il più semplice possibile (principio KISS, simple code and simple design). Si ragiona per esempi (**think by example**): si inizia con un esempio, si scrive un algoritmo ad hoc e poi lo si generalizza o particolarizza a seconda delle esigenze. In XP, il codice è di tutti (**collective code ownership**), tutti possono modificare il codice scritto da altri, non è necessario chiedere autorizzazioni. Gli stand up meeting servono anche a questo, per sapere sempre chi sta facendo cosa ed evitare di andargli a incasinare un lavoro in corso.
Le **code conventions** vengono stabilite all'inizio e seguite pedissequamente, usando anche tool per semplificarci il lavoro.
Nel **pair programming** si hanno sempre due sviluppatori per postazione: solitamente è il meno esperto a scrivere il codice, e l'altro ad osservare e commentare. Le coppie vengono cambiate spesso. È un punto controverso dell'extreme programming, studi empirici per verificare se davvero il pair programming aumenta la produttività hanno dato risultati constrastanti. Tra i suoi punti di

forza ci sono la continua revisione del codice che riduce la possibilità di bug, il training, il fatto che ogni modulo sia così conosciuto nel dettaglio da almeno due persone e il ridursi della possibilità che il programmatore si blocchi in un punto senza riuscire ad andare avanti.
Altro principio seguito da XP è lo **unit testing**: per ogni classe vengono scritti test cases per verificarne il funzionamento, si eseguono ad ogni modifica apportata alla classe e dovranno essere tutti superati prima di poter andare avanti. Si segue il **test driven development**: i test vengono scritti prima del software che dovranno testare. I test devono essere eseguiti anche dopo ogni **refactoring**, che deve essere guidato dal principio "once and only once": eliminare totalmente la duplicaizone di codice, se ci accorgiamo che due porzioni di codice svolgono uno stesso lavoro vanno unificate in un unico modulo.
Molto importanti anche i test funzionali di accettazione (**acceptance testing**), che verificano che una user story sia stata implementata correttamente: durante lo sviluppo i test di accettazione possono anche non essere superati al 100%: seguendo il principio incrementale delle small releases infatti avremo molti rilasci intermedi, che non implementeranno tutte le funzionalità: la percentuale di test di accettazione superati è indice del progresso dello sviluppo.
Per evitare l'incubo dell'integrazione finale si cerca di avere sempre un sistema funzionante e aggiornato (**continuous integration**): l'integrazione viene svolta ogni volta che una user story è completata.

12. Metriche di qualità

Molti progetti falliscono per mancanza di qualità del software, solitamente perché quest'ultima non viene tenuta costantemente sotto controllo, magari vengono fatti solo alcuni controlli alla fine del processo di sviluppo: questo è un male perché prima un problema di qualità viene scoperto più sarà facile da risolvere, quindi il controllo della qualità dovrà essere il più vicino possibile allo sviluppatore, possibilmente integrandolo direttamente nell'IDE. La **program analysis** è il processo di analisi automatica di un programma: può essere statica (fatta senza eseguire il programma, analizzando solo proprietà verificabili a compile-time) o dinamica (che analizza il comportamento a run-time del programma). Quando stiamo analizzando programmi che producono un codice oggetto intermedio (come il bytecode di Java o l'IL del framework .NET) solitamente l'analisi statica viene svolta sul codice sorgente, e solo in alcuni casi sul codice oggetto,

Per l'analisi di programmi Java in ambiente Eclipse useremo il tool STAN4J. Con STAN4J possiamo vedere come i singoli artefatti software (metodi, classi, package, librerie) si compongono tra loro per costruire artefatti di livello più alto e le loro interdipendenze. STAN4J analizza più il bytecode che il codice Java, permettendoci di analizzare anche codice già compilato (ad esempio librerie proprietarie). Ci permette di analizzare il progetto a diversi livelli di dettaglio: a livello di classe, preferibile in caso di progetti di grosse dimensioni, vedremo solo le relazioni tra le classi senza alcuni informazione su campi e metodi; a livello di membro, invece, avremo anche le informazioni su campi e metodi: questo ci permette di analizzare metriche non disponibili a livello di classe come le metriche sui metodi e i grafi di dipendenza tra membri.

12.1 Elenco di metriche

Una metrica è una funzione che mappa un artefatto software su un numero, che indica una qualche proprietà dell'artefatto. Può essere qualcosa di molto semplice come il numero di linee di un metodo o di classi di un package, ma può anche indicare qualcosa di più complesso(metriche CK, metriche di Martin). STAN4J ci presenta le metriche già suddivise per categorie:

12.1.1 Count Metrics

Metriche che tengono semplicemente il conto di qualcosa, come

- **Number of Libraries**
- **Number of Packages**
- **Number of Units:** il numero di classi, interfacce o enum top level (quindi non annidate) all'interno dell'applicazione
- **Number of Methods**: il numero di metodi di una certa classe, costruttori (espliciti e impliciti) ed inizializzatori inclusi.
- **Number of Fields**: il numero di campi di una certa classe (sono esclusi i campi delle classi annidate)
- **Number of Instructions**: il numero di istruzioni bytecode di un metodo
- **Estimated Lines of Code (ELOC):** numero spprossimativo delle linee di codice sorgente che compongono un artefatto (esclusi gli import, le linee vuote ed i commenti)
- **ELOC / Unit (Estimated Lines of Code per Top Level Class):** il valore medio di ELOC per tutte le classi top level dell'applicazione

12.1.2 Complexity Metrics

Metriche che misurano la complessità dell'applicazione

- **Complessità ciclomatica:** misura il numero di punti di decisione sul grafo del controllo di flusso di un metodo, incrementato di 1. Un punto di decisione è uno statement che rende il flusso di esecuzione non lineare: può essere un if, un ciclo, uno switch case, una catch clause ecc. E' molto importante in fase di testing, perché più sarà alta più test andranno fatti: Stan4J lo considera buono fino a 15, medio fino a 20, preoccupante più di 20. Può sembrare che Stan4J a volte dia un numero errato, ma questo perché l'analisi avviene a livello di bytecode, e ad esempio un if con un or nella condizione (es. if(x==1 || x==2)) durante la compilazione viene spezzato in due if. Una complessità ciclomatica alta, oltre a complicare la fase di testing, aumenta la possibità di errori nel codice (fault).

12.1.3 Robert C. Martin metrics

Metriche per sistemi orientati agli oggetti, presentate da Robert C. Martin nel 2002. Sono metriche a livello di package.

- **Afferent Couplings (Ca):** indica il numeo di unità esterne al package che dipendono da unità interne al package (dipendenze entranti). Indica la responsabilità del package: un package con alto Ca avrà un elevato numero di responsabilità e viceversa.
- **Efferent Couplings (Ce):** indica il numero di unità esterne al package da cui dipendono le unità interne al package (dipendenze uscenti). Indica l'indipendenza del package: un package con alto Ce indicherà un package con bassa indipendenza (cioè che ha molte dipendenze uscenti) e viceversa.
- **Instability (I):** è il rapporto tra le dipendenze uscenti e le dipendenze totali: **I = Ce/(Ca+Ce)**. E' compreso tra 0 e 1 ed è tanto più buono quanto è più vicino allo 0: un package con I=0 è infatti completamente stabile (non ha dipendenze uscenti) mentre un package con I=1 è completamente instabile (ha solo dipendenze uscenti).
- **Abstractness (A):** è il rapporto tra il numero di unità astratte (interfacce o classi astratte) rispetto al numero di unità totali. E' compreso tra 0 e 1, dove 1 è un package completamente astratto e 0 un package completamente concreto.
- **Distance (D):** indica il bilanciamento tra abstractness ed instability: è data dalla formula **D = A + I - 1** , ed è sempre compresa tra -1 e +1: i valori desiderati sono quelli intorno allo zero (sequenza principale).

12.1.4 Chidamber & Kemerer (CK) metrics

Metriche per sistemi OO, orientate alla classe

- **Lines of Code per Class (LOCC)**: somma delle linee di codice di tutti i metodi della classe(linee bianche e commenti escluse)
- **Weighted Methods per Class (WMC):** la somma delle complessità dei metodi di una classe (misurata con una metrica di complessità a scelta). Se i metodi hanno tutti complessità 1, equivale al numero di metodi nella classe. E' un indice della complessità della classe: una classe con molti metodi complessi avrà più probabilità di contenere errori, inoltre più metodi ha una classe più è probabile che sia molto legata all'applicazione in cui si trova, e questo ne limita il riuso. Per questo bisogna cercare di mantenere il WMC basso.
- **Depth of the Inheritance Tree (DIT)**: distanza massima di una classe dalla radice dell'albero dell'ereditarietà. Alberi dell'ereditarietà profondi possono aumentare la complessità del progetto: una classe che si trova in profondità nella gerarchia potrà ereditare più metodi, e questo rende più complesso predire il suo comportamento. Maggiore è la profondità di una classe nella gerarchia, maggiore è il riuso dei metodi ereditati.
- **Number of Children (NOC)**: numero di sottoclassi eredi di una certa classe. Un valore di NOC alto indica un alto livello di riuso della superclasse, ma diminuisce l'astrazione dalla classe radice, aumentando la possibilità che la classe figlia non sia un membro appropriato della classe radice. Inoltre, al crescere dle NOC cresce il numero di test case necessari a testare ogni classe figlia.

- **Response for a Class (RFC)**: numero di metodi che possono essere eseguiti in risposta ad un messaggio ricevuto dalla classe. Un valore alto di RFC aumenta la complessità progettuale della classe e la difficoltà di testing.
- **Coupling Between Object classes (CBO)**: indica il numero di accoppiamenti di una classe, cioè il numero di classi che utilizza durante il suo funzionamento. Va tenuto al minimo perché un eccessivo accoppiamento peggiora modularità e riuso: per riutilizzae la classe dovrò portarmi dietro tutte le sue dipendenze, quindi meno dipendenze ha più è riusable. Valori alti di CBO complicano testing e modifiche.
- **Lack of Cohesion in Methods (LCOM)**: indica la coesione di una classe, misurata andando a vedere quanti metodi accedono alle stesse variabili: se tutti i metodi accedono alle stesse variabili, la classe sarà coesa. Per misurare la LCOM si considerano gli insiemi di variabili accedute da ogni metodo della classe: siano $M_1,...,M_n$ i metodi della classe C, e $I_1,...,I_n$ gli insiemi di variabili accedute da ciascun metodo; siano P e Q gli insiemi rispettivamente delle intersezioni vuote e non vuote degli insiemi I dei metodi della classe (formalmente: $P = \{I_i, I_j \mid I_i \cap J_i = \emptyset\}$ e $Q = \{I_i, I_j \mid I_i \cap J_i \neq \emptyset\}$), il valore di LCOM sarà dato dal numero di intersezioni vuote meno il numero di intersezioni non vuote: $|P| - |Q|$ se $|P| > |Q|$, 0 altrimenti. Maggiore è il numero di metodi che condividono attributi della classe, maggiore è la coesione della classe, quindi maggiore è il valore di LCOM minore è la coesione tra i metodi della classe.