

INTRODUZIONE

L'introduzione alla Software Engineering definisce questa disciplina come l'applicazione dei principi dell'ingegneria (progettazione e validazione) alla produzione del software, rendendolo un prodotto industriale. Senza un approccio ingegneristico, si incorre in scarsa qualità del prodotto e scarsa competitività, che possono manifestarsi in cost overrun (superamento dei costi) e time overrun (superamento dei tempi). L'esigenza di adottare un approccio ingegneristico nacque con il passaggio dallo sviluppo di semplici programmi (codificati) allo sviluppo di veri e propri prodotti software. Sebbene tale approccio disciplinato e rigoroso abbia portato notevoli vantaggi, non ha completamente risolto la crisi del software.

Il prodotto software ha aspetti accidentali e aspetti essenziali:

Aspetti Accidentali: Sono superabili con il progresso tecnologico e includono il Ciclo di vita del Software.

Il Ciclo di vita del Software è l'intervallo temporale in cui un prodotto software esiste, dalla creazione alla dismissione. Si compone di 3 Stadi e 6 Fasi:

- Stadio 1: Sviluppo (composto da 6 fasi sequenziali o iterabili):

Requisiti: Definizione di "cosa" il software deve fornire.

Specifiche (o analisi dei requisiti): Dettaglio dei requisiti.

Pianificazione.

Progetto (preliminare e dettagliato): Definizione di "come" realizzare il software.

Codifica.

Integrazione.

- Stadio 2: Manutenzione: Copre circa il 60% dei costi dell'intero ciclo di vita.

- Stadio 3: Dismissione.

Il Testing non è una fase separata, ma un'attività che si svolge durante l'intero sviluppo in due modi:

- Verifica (alla fine di ogni fase): Controlla se la fase è stata ben svolta.

- Validazione (alla fine dello sviluppo): Controlla se il prodotto finale è buono.

Il testing non può mai garantire un prodotto totalmente privo di difetti.

La Defect Removal Efficiency (DRE) si riferisce alla percentuale di difetti trovati prima del rilascio del prodotto. Ad esempio, se si trovano 900 difetti prima del rilascio e gli utenti ne trovano 100 entro 90 giorni, la DRE è del 90%.

Aspetti Essenziali: Non sono superabili con il progresso tecnologico.

- Complessità: Riferita al numero possibile di configurazioni.

- Conformità: Il software deve adattarsi ad ambienti preesistenti (hardware e non), non il contrario.

- Cambiabilità: Il software può subire modifiche sostanziali durante il suo periodo d'uso.

- Invisibilità: Il software in esecuzione è invisibile e non ha proprietà geometriche.

Aspetti di costo del prodotto software

Il costo è solitamente proporzionale al quadrato della dimensione ($C=aS^2$), il che significa che fare due prodotti di dimensione $S/2$ costa meno che farne uno di dimensione S .

Una volta creato, produrre una replica non costa nulla.

Vendere un prodotto di dimensione doppia rispetto a uno venduto al prezzo p richiede un prezzo 4 volte superiore a parità di ampiezza di mercato, o un mercato 4 volte maggiore per venderlo allo stesso prezzo.

Definizioni

Prodotto Sw (o Sw): Codice + Documentazione.

Artefatto: Prodotto Sw intermedio (documento requisiti, specifica, progetto, non eseguibile).

Codice: Prodotto Sw finale (artefatto eseguibile).

Sistema Sw: Insieme organizzato di prodotti Sw.

Cliente: Soggetto che ordina il prodotto Sw.

Sviluppatore: Soggetto che lo produce.

Utente: Soggetto che lo usa.

Sw Interno: Cliente e sviluppatore coincidono.

Sw a Contratto: Cliente e sviluppatore sono soggetti differenti.

Affidabilità del Software (Sw Reliability)

Informalmente, è la fiducia riposta nel prodotto software. Formalmente, è la probabilità che il prodotto software funzioni correttamente in un determinato intervallo temporale (mission time).

Perché il software funzioni correttamente, non deve presentare:

- Difetti (defect): Anomalia presente in un prodotto Sw.
- Guasti (failure): Comportamento anomalo del prodotto Sw dovuto a un difetto.
- Errori: Azione errata di chi introduce un difetto nel prodotto Sw.

I guasti avvengono a causa di difetti, che sono introdotti per errore. Non è detto che un difetto causi necessariamente un guasto durante l'utilizzo.

Intuitivamente, meno difetti significano maggiore affidabilità, ma la relazione tra affidabilità osservata e difetti latenti non è semplice. Eliminare difetti da parti raramente usate del software ha poco effetto sull'affidabilità percepita.

La regola 10-90 indica che il 90% del tempo di esecuzione totale è speso eseguendo solo il 10% delle istruzioni (il nucleo del programma). Il miglioramento dell'affidabilità dipende dalla localizzazione del difetto (se appartiene o meno al nucleo).

L'affidabilità osservata dipende dal profilo operativo (come è usato il prodotto) e quindi può variare a seconda dell'utente.

Confronto tra Affidabilità Hardware (Hw) e Software (Sw)

- Guasti Software: Dovuti a difetti latenti nei programmi; il software non si consuma. Il sistema continua a guastarsi a meno di correzioni. Dopo la riparazione, l'affidabilità del software può aumentare o diminuire. L'obiettivo è la crescita dell'affidabilità (far decrescere la frequenza di guasto).

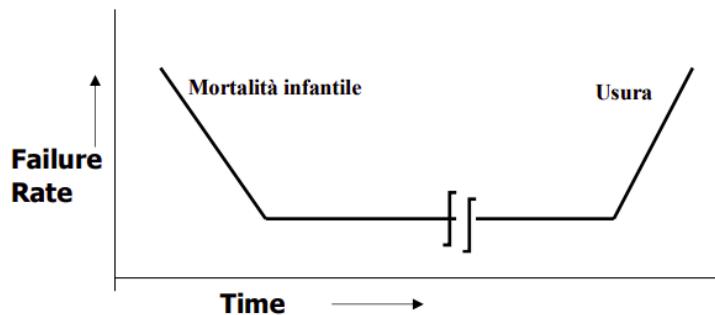
- Guasti Hardware: Quasi sempre dovuti a consumo, deterioramento o rottura dei componenti. Per riparare un difetto hardware, si sostituisce il componente. Dopo la riparazione, l'affidabilità hardware torna come era. L'obiettivo è la stabilità dell'affidabilità (mantenere la frequenza di guasto costante).

La curva di frequenza di guasto del software idealizzata dovrebbe partire alta (mortalità infantile) e scendere a zero nel tempo. La realtà è diversa: la frequenza di guasto risale dopo un po' di tempo.

Per migliorare la curva e stabilizzare il software, si può ricorrere alla software rejuvenation, definendo un periodo per "resettare" il software a uno stato più stabile. Sebbene il software non si usuri di per sé, l'interazione con l'ambiente del sistema operativo può contribuire a malfunzionamenti.

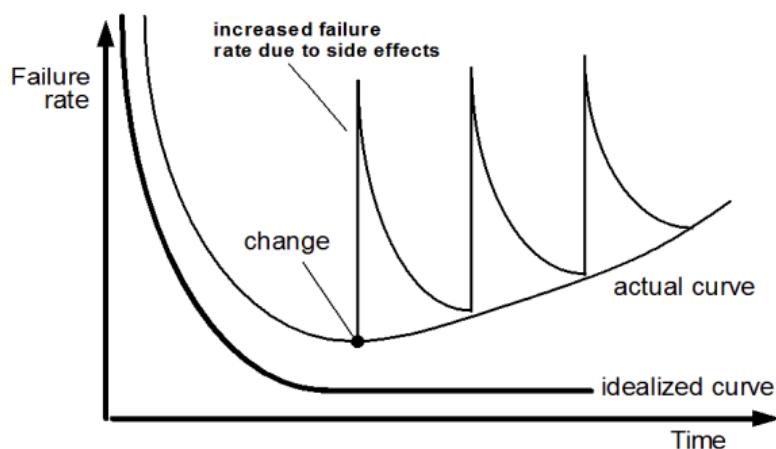
Nella realtà: andamento frequenza di guasto hardware

(effetto dell'eliminazione dei componenti difettosi prima, e dell'usura poi)



Andamento frequenza di guasto software

(effetto dell'eliminazione dei difetti prima, e dell'invecchiamento per manutenzione poi)



Disponibilità del Software (Sw Availability)

È la percentuale di tempo in cui il software è risultato usabile nel corso della sua vita.

Dipende dal numero di guasti e dal tempo necessario per ripararli. È una metrica importante per sistemi dove la caduta del servizio causa perdite economiche e di sicurezza.

Per verificare l'affidabilità del software si usa il Statistical Testing, dove il software viene testato per un periodo, registrando i fallimenti, per stimare l'affidabilità in base al tempo di utilizzo.

Fasi della Produzione del Software

Nel corso degli anni, la produzione del software ha attraversato diverse fasi:

- Fase di Abilità: Prevalgono aspetti di lavoro individuale e creativo.
- Fase Artigianale: Il software è prodotto da piccoli gruppi specializzati, spesso con alta professionalità.

- Fase Industriale: L'attività di sviluppo e manutenzione è pianificata e coordinata, con supporto di strumenti automatici.

Caratteristiche del Software

Il software può essere considerato un insieme di elementi che formano una "configurazione" che include:

- Programmi.
- Documenti.
- Dati multimediali.

Viene realizzato dall'ingegnere del software applicando un processo che porta a risultati di qualità elevata, seguendo un approccio ingegneristico.

Le sue caratteristiche sono: va "ingegnerizzato", non si consuma, è complesso, invisibile, si conforma e si cambia.

PROCESSO SOFTWARE

Il Processo Software è una serie di attività necessarie per realizzare il prodotto software rispettando tempi, costi e caratteristiche di qualità desiderate. Segue un ciclo di vita articolato in tre stadi:

1. Sviluppo: Comprende due tipi di fasi:

- Fasi di definizione: Si occupano del "cosa" il software deve fornire, definendo i requisiti e producendo le specifiche.
- Fasi di produzione: Definiscono "come" realizzare quanto definito, includendo progettazione, codifica, integrazione e rilascio al cliente.

2. Manutenzione: Supporta il software realizzato, prevedendo al suo interno fasi di definizione e/o produzione.

3. Dismissione: Il momento in cui il software viene ritirato.

Durante ogni fase del ciclo di vita, si effettua il testing di quanto prodotto, utilizzando tecniche di Verifica e Validazione (V&V) applicate sia ai prodotti intermedi che a quello finale.

Tipi di Manutenzione:

- Correttiva: Elimina difetti (fault) che causano guasti (failure).
- Adattativa: Adatta il software ai cambiamenti dell'ambiente operativo (es. aggiornamenti hardware o processi).
- Perfettiva (o Evolutiva): Estende il software con funzionalità aggiuntive; è la più comune e costosa perché allarga i requisiti.
- Preventiva (o Software Reengineering): Consiste nel fare modifiche che rendano più semplici future correzioni, adattamenti o miglioramenti. Si applica tipicamente per processi con documentazione obsoleta, partendo dal codice per ricostruire la documentazione di progetto e specifica, risultando molto costosa.

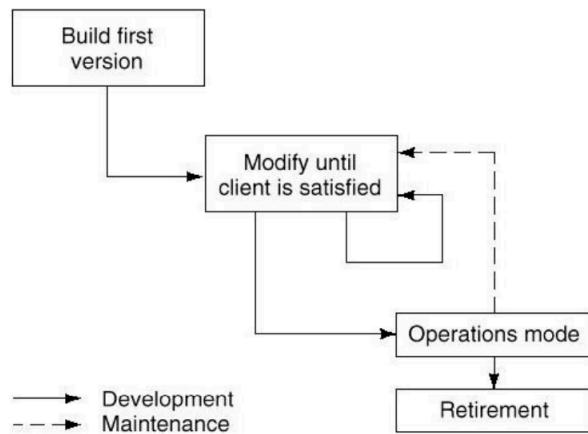
Il ciclo di vita del prodotto software come intervallo di tempo dalla nascita dell'esigenza di costruire un prodotto software all'istante in cui viene dismesso, ma esiste una definizione più dettagliata.

Il ciclo di vita del prodotto software include fasi come definizione dei requisiti, specifica, pianificazione, progettazione (preliminare e dettagliata), codifica, integrazione, testing, uso, manutenzione e dismissione. Queste fasi possono sovrapporsi o essere eseguite in modo iterativo, soprattutto il testing. Il Modello di Ciclo di Vita del software specifica come eseguire

queste attività e il loro ordine. Non esiste un modello migliore in assoluto; la scelta dipende dalla natura dell'applicazione, dalla maturità dell'organizzazione, dai vincoli del cliente

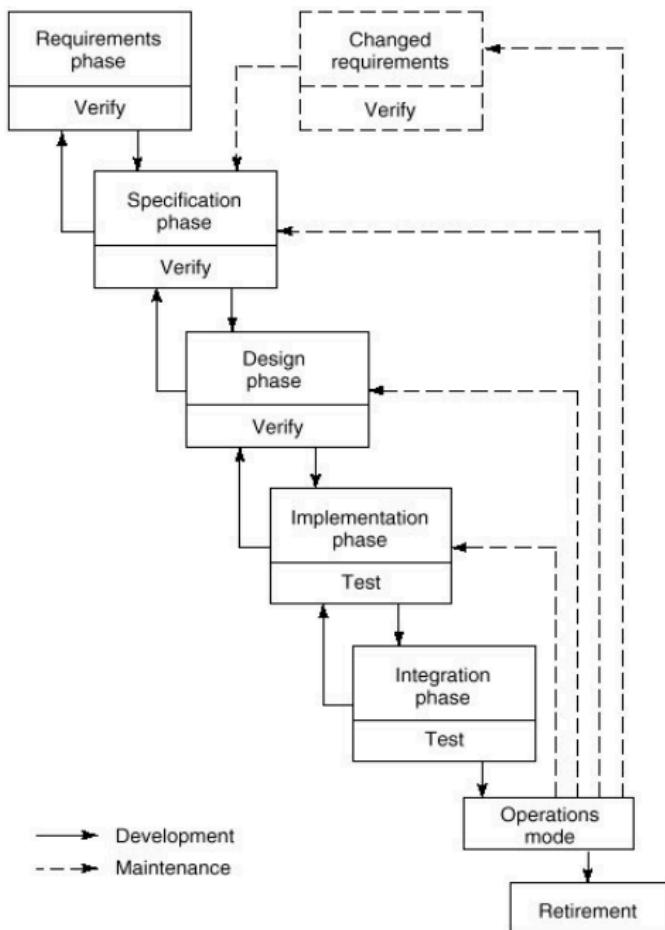
Modelli di Ciclo di Vita del Software

L'assenza di un modello porta alla modalità di sviluppo "Build & Fix" o "Fix-it-Later", dove il software viene sviluppato e rilavorato finché non soddisfa il cliente, potendo portare a una "crisi del software" se si itera continuamente



Modello Waterfall

Un primo modello nato nel 1970, due anni dopo la nascita dell'ISW, è il Modello Waterfall.



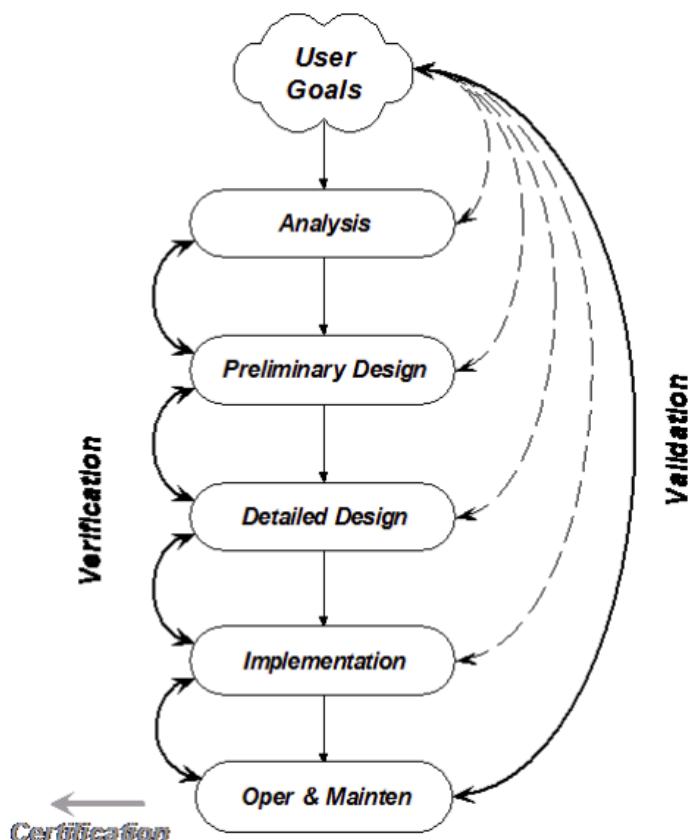
Le attività sono svolte in rigida sequenza: si passa alla fase successiva solo quando quella precedente è stata completata e verificata.

Limiti:

Assume che una volta finita la specifica, non si tornerà più sui requisiti, il che è irrealistico (es. modifiche normative).

Assenza di feedback continuo dal cliente, che è coinvolto solo all'inizio e alla fine, rischiando un prodotto che non soddisfa pienamente le esigenze.

I modelli successivi cercano di sfruttare i vantaggi e andare oltre gli svantaggi del modello waterfall.



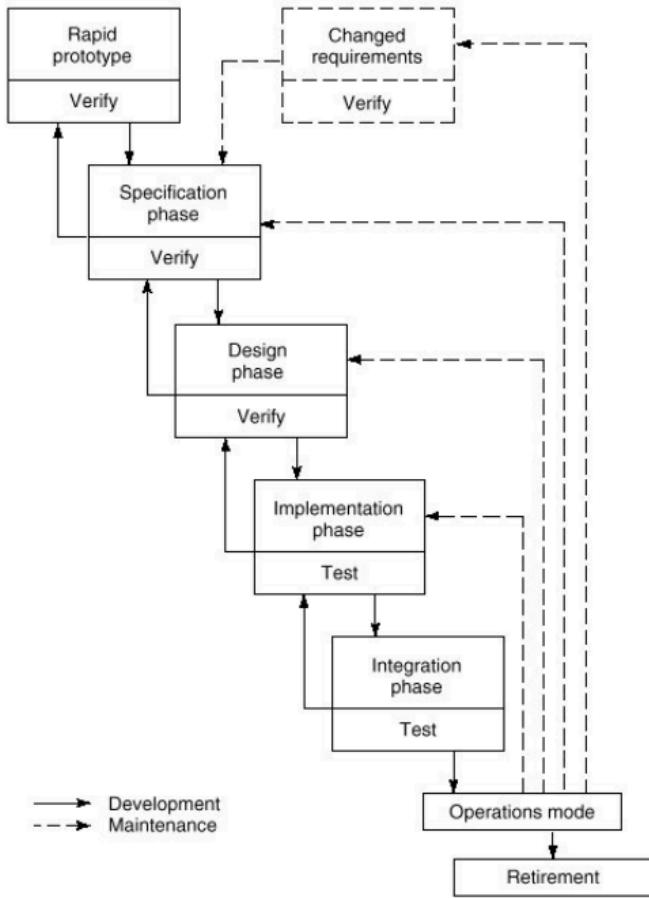
Prima di passare agli altri modelli distinguiamo verifica e validazione:

- Verifica: Controlla la correttezza del progetto rispetto al documento di specifica ricevuto in input. Si occupa di "fare bene il prodotto".
- Validazione: Capisce se ciò che è stato richiesto è stato correttamente realizzato, cioè se il prodotto soddisfa gli obiettivi dell'utente. Si occupa di "fare il prodotto giusto".

In basso a sinistra Certificazione: è la dichiarazione formale relativa alla qualità del prodotto. (es. DOCG: esiste uno standard di riferimento e certifico che il mio prodotto soddisfa lo standard).

Nel caso dei prodotti software non esiste lo standard diretto quanto delle certificazioni della capacità di un'organizzazione dell'utilizzo delle tecniche migliori (maturità dell'organizzazione vista prima).

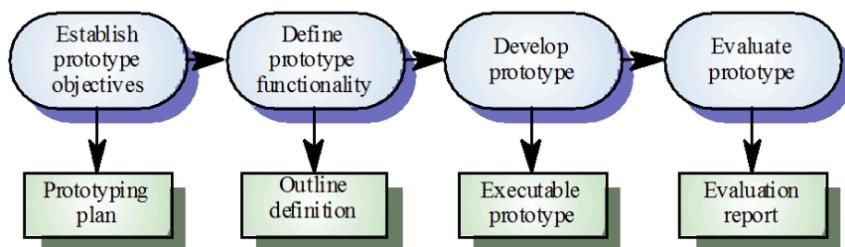
Il modello a Prototipo Rapido è stato il primo modello migliorativo post waterfall.



Si concentra sulla prima fase, chiamata Rapid Prototype, per superare le ambiguità nella comunicazione dei requisiti con il cliente.

Il prototipo è l'implementazione dell'interfaccia del prodotto, fornito rapidamente (max una settimana/10 giorni).

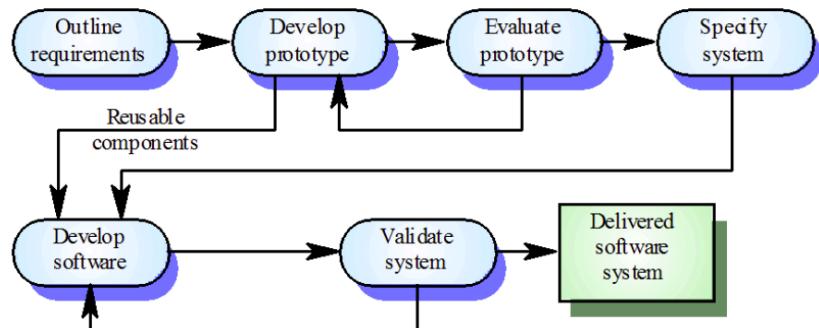
Ha come Obiettivo: Tirare fuori (Requirements Elicitation) e convalidare (Requirements Validation) i requisiti, permettendo all'utente di sperimentare il sistema.



A partire da questo modello si è pensato di usare il prototipo anche come strumento di specifica, ossia come qualcosa da cui partire per poi sviluppare piano piano il progetto. Ma i requisiti non funzionali non possono essere adeguatamente testati con il prototipo, l'implementazione non ha valenza legale e funzioni critiche di sicurezza potrebbero essere impossibili da soddisfare per un semplice prototipo.

Il prototipo dovrebbe essere "Usa e Getta": utilizzato solo per la fase dei requisiti, dopodiché il progetto vero e proprio segue un modello Waterfall, perciò non deve essere considerato come prodotto finale.

Alcuni dei componenti del prototipo potranno essere in alcuni casi utilizzati per la realizzazione dell'interfaccia grafica finale.



Principale svantaggio Rapid Prototype Model: È difficile convincere l'utente della complessità del progetto effettivo dopo aver visto un prototipo rapido, mettendo pressione al team.

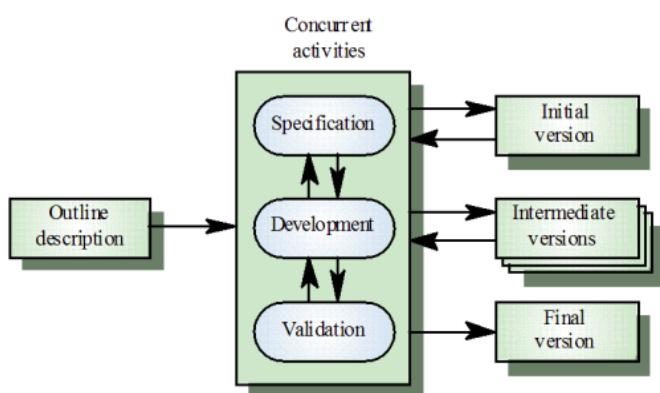
Si usano tecniche di Visual Programming (drag & drop) per la rapidità. Importante far capire al cliente che il progetto vero e proprio non può basarsi solo su questa tecnologia ma è molto più complesso e richiede molto più tempo, infatti non sono adatte per lavori di gruppo o ottimizzazione del codice, rendendo difficile la manutenzione.

Con il tempo, invece di modificare l'approccio Waterfall per migliorarlo, si sono introdotti approcci che rompessero la struttura rigida e monolitica del modello in favore di altro. Tali approcci fanno uso della Process Iteration: piuttosto che sviluppare il prodotto in modo monolitico come sequenza di macrofasi si è pensato di introdurre iterazioni che lavorassero su parti del prodotto più piccole (anche perché nel tempo i requisiti cambiano).

Due approcci: Sviluppo Incrementale e Sviluppo a Spirale.

Sviluppo Incrementale

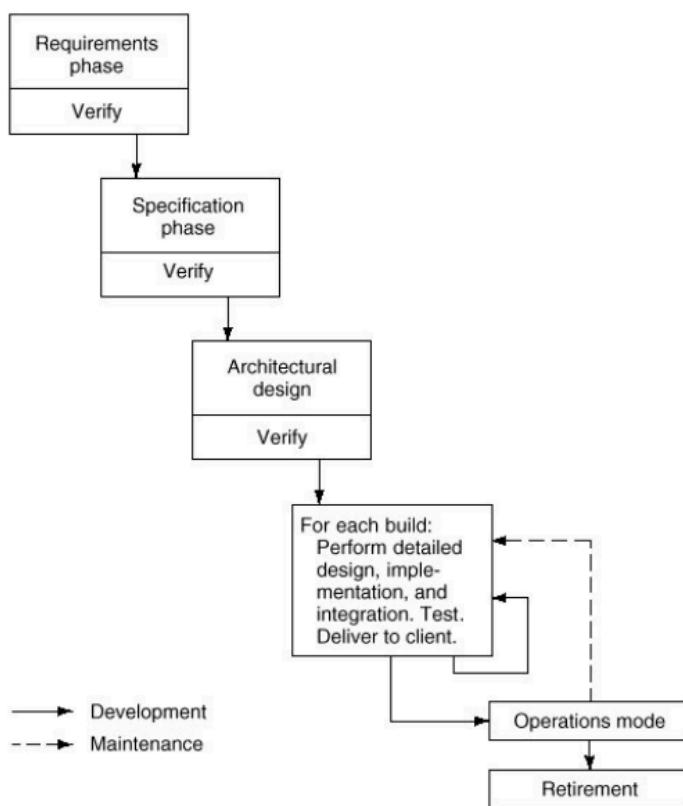
L'idea dello sviluppo incrementale è sviluppare e consegnare il prodotto in incrementi successivi (build). Oltre ai costi ridotti un altro vantaggio sta nel fatto che il cliente vede continuamente i progressi e riceve feedback continuo. Inoltre non si perdono nemmeno i benefici Waterfall dal momento che ciò viene fatto con un approccio rigoroso e strutturato.



Si parte da una outline description (descrizione d'insieme) del prodotto e ogni attività può essere svolta in modo concorrente da team separati risparmiando tempi. La prima build sarà l'Initial Version (molto limitata), si prosegue con le successive fino alla finale.

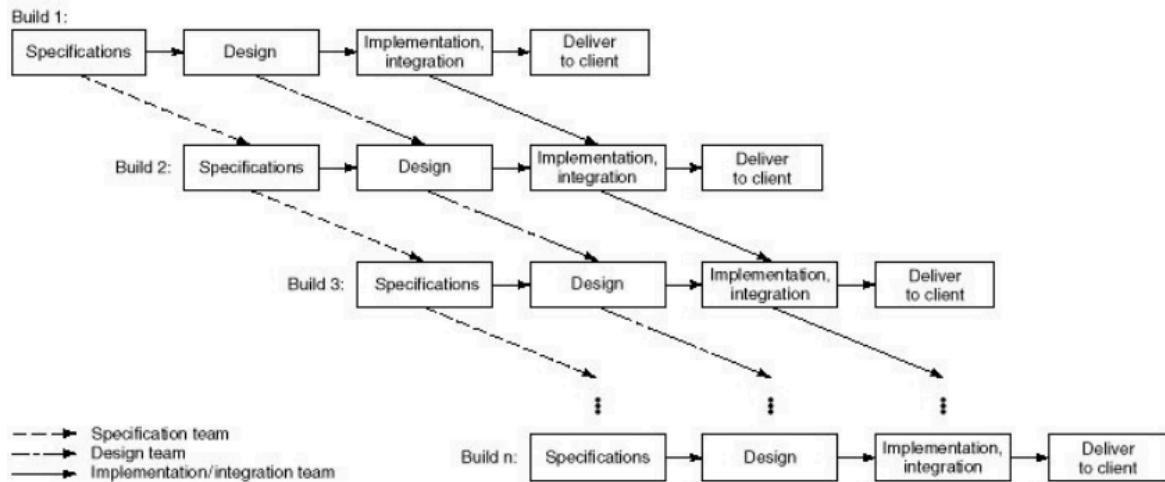
L'approccio Incrementale può essere realizzato in due versioni alternative:

- Overall Architecture (più conservativa): identica al Waterfall fino alla progettazione architetturale, poi procede in modo incrementale a partire dalla progettazione dettagliata. Le componenti dell'architettura sono trattate come build. (Ogni fase di progettazione è divisa in architetturale o preliminare e di dettaglio. Nella prima si progetta l'architettura del software, individuando le componenti principali del prodotto software e le relazioni che esistono tra loro. Dopo averle individuate, questa versione considera ciascuna componente come se fosse una build procedendo quindi con l'approccio incrementale). Dopodiché si va in modalità operativa (manutenzione finché il prodotto non viene dismesso).



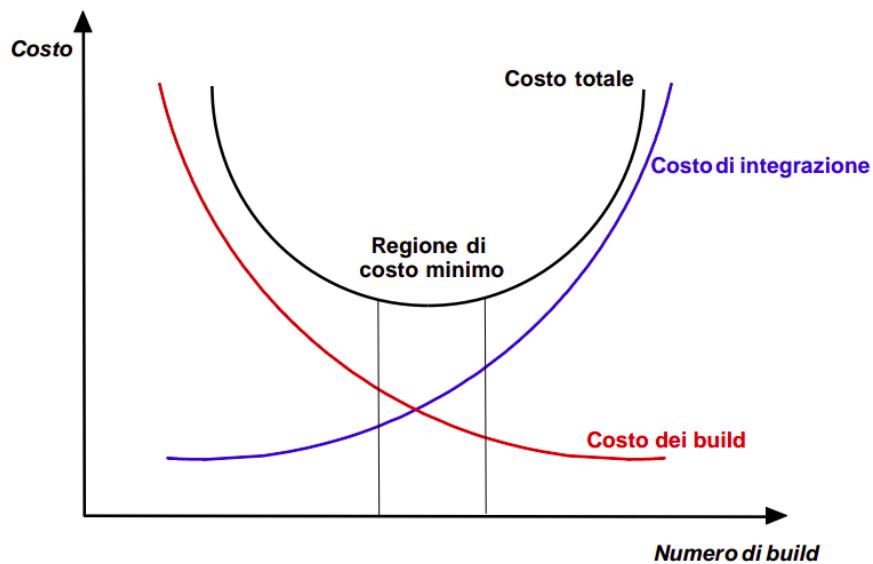
- Senza Overall Architecture (più rischiosa): build sono identificati dai requisiti di alto livello con priorità. Si lavora sul Build 1 come fosse Waterfall, poi sul Build 2, ecc.. Sebbene più veloce all'inizio con team paralleli, presenta rischi di integrazione con build precedenti,

poiché non si hanno informazioni sull'interazione delle componenti.



Con l'architettura software sapevo esattamente quali componenti interagissero tra loro e come integrarli di conseguenza. Senza architettura non ho queste informazioni ed è possibile che alla k-esima build scopri problemi di integrazione con build che avevo già consegnato.

A questo punto per l'approccio incrementale si deve sapere quale sia il numero di build più conveniente in cui suddividere il prodotto. Si sceglie in base all'impatto sui costi: più build, minor costo per modifica, ma maggiore costo di integrazione. Si cerca una regione di costo minimo.



Facciamo un confronto tra Waterfall e Incrementale.

Nel modello Waterfall è più difficile accomodare modifiche ai requisiti, mentre in quello incrementale requisiti suddivisi in classi di priorità e facilmente modificabili (modificare un requisito che ha impatto sul singolo build significa lavorare su una più piccola parte di codice).

Nel primo si può avere feedback dal cliente solo una volta terminato lo sviluppo, nel secondo continuo feedback dal momento che gli mando continuamente roba.

Nel primo le fasi sono condotte in rigida sequenza (l'output di una costituisce l'input per la

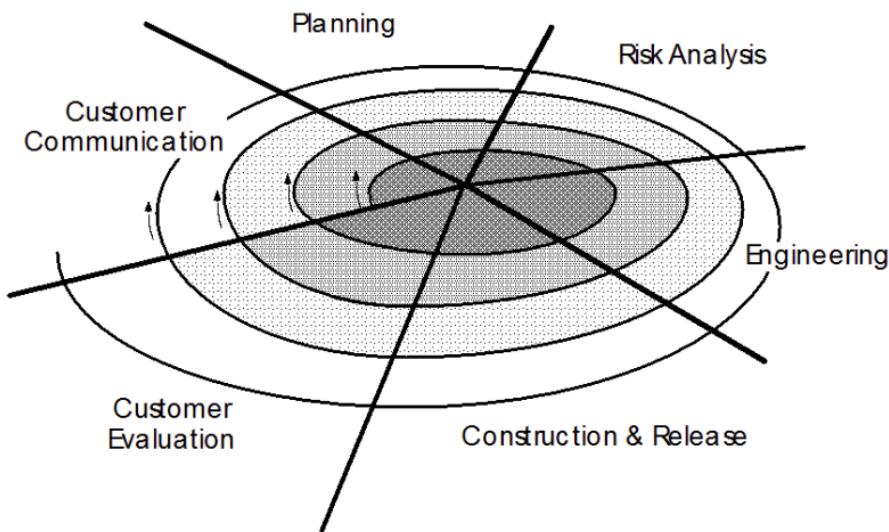
successiva e si passa alla prossima solo dopo verifica), nel secondo le fasi possono essere condotte in parallelo.

Nel primo tutte le fasi sono sviluppate sull'intero prodotto mentre nel modello incrementale, anche considerando la versione con overall architecture a partire dalla fase di progettazione dettagliata il resto delle fasi sono effettuate sul singolo build.

Nel primo caso il team di sviluppo è costituito da un gran numero di persone, nel secondo posso avere diversi team di sviluppo, ciascuno con piccole dimensioni.

Sviluppo a Spirale

Si parte dalla spirale più piccola andando sempre in esterno. La dimensione radiale rappresenta l'incremento dei costi, quella angolare l'avanzamento del tempo (dalla spirale più interna all'inizio fino a quella più esterna alla fine). Ad ogni cerchio della spirale faccio sempre le stesse attività iterativamente.



Si divide il piano in vari settori:

Si parte da dove stanno le frecce iniziando quindi con la Customer Communication cioè l'interazione con il cliente,

Poi si Pianificano le attività da svolgere,

Per arrivare a un settore che è caratteristica peculiare di questo modello: la Risk Analysis.

Se questa analisi porta a una valutazione secondo la quale i rischi sono eccessivi allora potrebbe essere conveniente valutare l'idea di fermare il progetto (e questo vale per ogni iterazione).

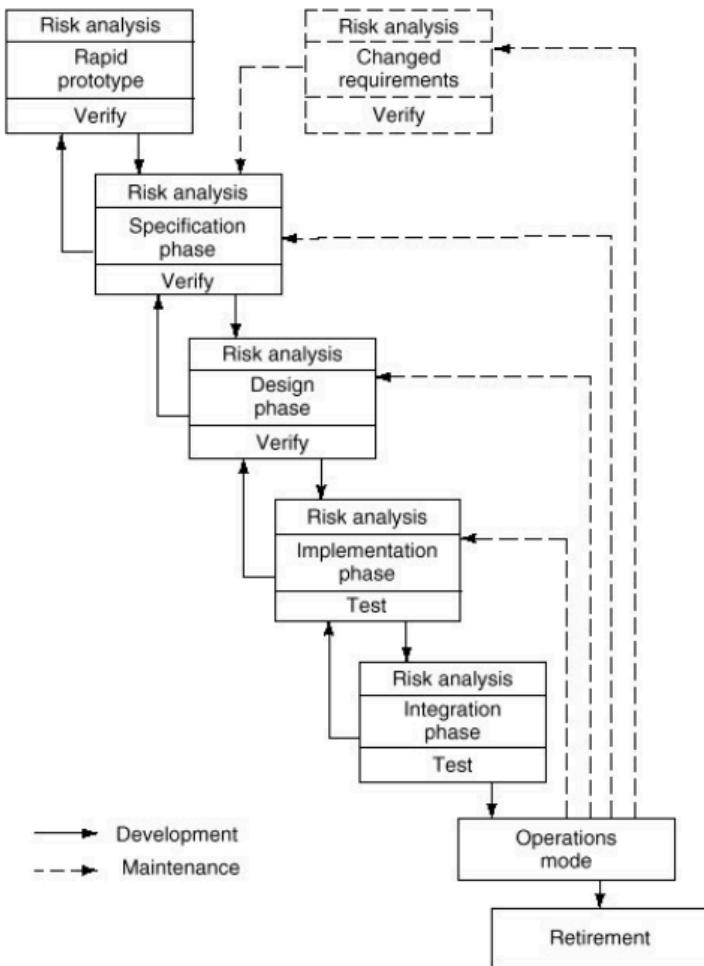
Se conviene procedere si passa all'Ingegnerizzazione (specifiche, progettazione etc..), poi si fa Costruzione e Rilascio (codifica, costruzione e rilascio) e infine

Customer Evaluation (si consegna al cliente quella che può essere una build) per poi ripartire.

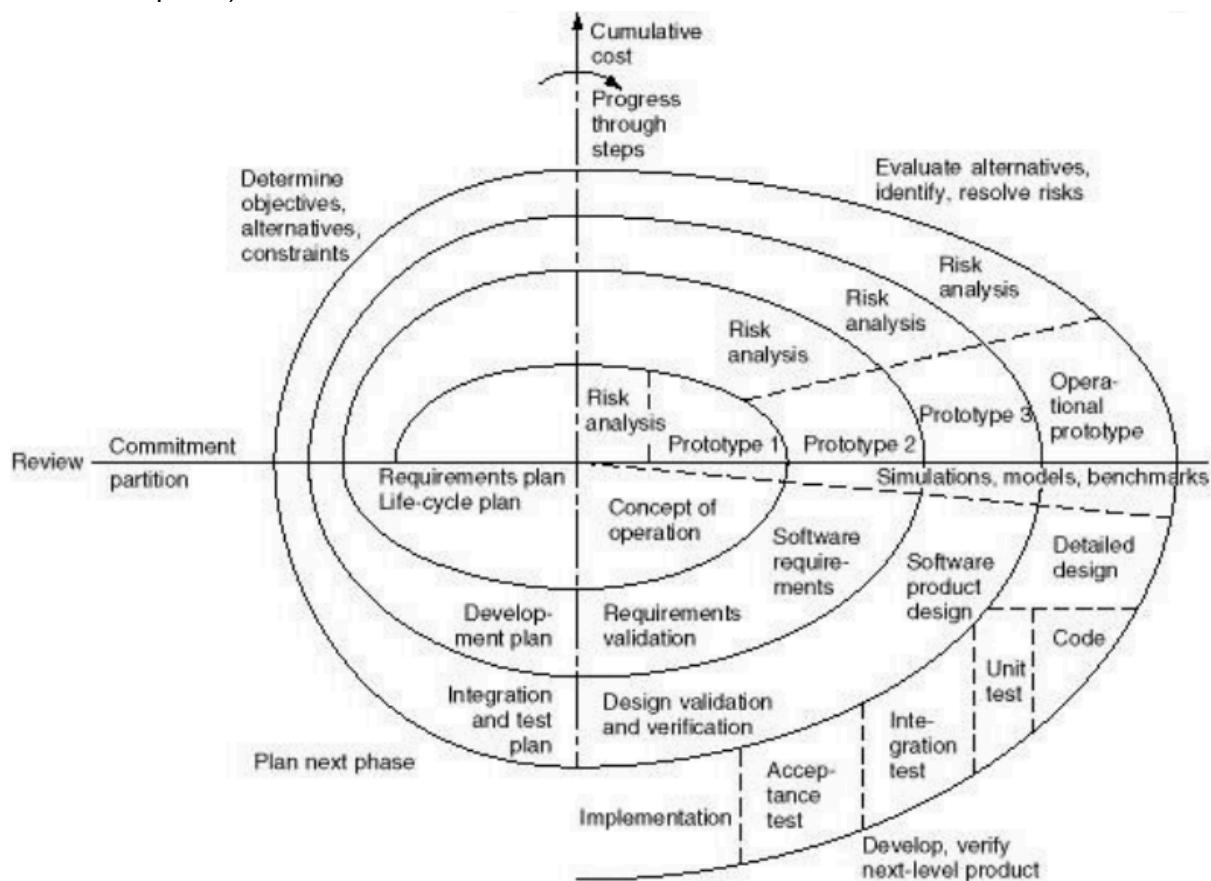
Ogni cerchio corrisponde a una build; anche la manutenzione è considerata un cerchio aggiuntivo.

Esiste anche questa rappresentazione del modello a spirale semplificata che è utile per mostrare la differenza rispetto al Waterfall: ad ogni blocco viene aggiunta inizialmente una

fase di risk analysis che deve essere effettuata prima di passare alla successiva.



Un'altra rappresentazione del modello a spirale è quella di Boehm (la prima versione del modello a spirale)



Anche qui centrale la fase di analisi dei rischi, dove vediamo nella spirale dei Prototipi. Questi sono ovviamente diversi da quanto visto nel modello a prototipo rapido, sono infatti prototipi creati appositamente per facilitare l'analisi dei rischi.

Nella fasi di Develop invece si fa uso di simulazioni, test e benchmark per aiutarsi, infatti è applicato con successo solo per software di tipo interno (es. software scientifici per la NASA), non per software a contratto. Non ha senso dire al cliente a un mese dalla consegna che l'analisi dei rischi suggerisce problemi, dato che ci sono questioni legali e di contratto. Un'altra caratteristica è che si devono avere persone molto competenti per l'attività di analisi di rischi. In realtà questa attività viene ovviamente fatta in un qualsiasi progetto software, ma nel modello a spirale diventa un elemento chiave.

Risk Management

In ogni tipo di progetto, non solo software, viene realizzata l'attività di risk management. Si tratta della serie di strumenti e tecniche per identificare e minimizzare i rischi.

Definizione di rischio: La probabilità che possa capitare una circostanza avversa durante lo sviluppo del software.

Tipi di rischio in base all'effetto:

- Project Risks: Effetti su tempo e risorse.
- Product Risks: Effetti su qualità e performance del software in produzione.

- Business Risks: Effetti sull'organizzazione che sviluppa il software.

Risk	Risk type	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organisational management with different priorities.
Hardware unavailability	Project	Hardware which is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool under-performance	Product	CASE tools which support the project do not perform as anticipated
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Bisogna considerare quindi tutto ciò che potrebbe avere impatto negativo sul progetto e la probabilità che accada.

Il risk management è un sottoprocesso del processo software con diverse attività sequenziali:

1. Risk Identification: Output è un documento con tutti i possibili rischi.

Rischi tipici nel software: tecnologia, persone, organizzazione, tool di supporto, requisiti, stima.

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. Software components which should be reused contain defects which limit their functionality.
People	It is impossible to recruit staff with the skills required. Key staff are ill and unavailable at critical times. Required training for staff is not available.
Organisational	The organisation is restructured so that different management are responsible for the project. Organisational financial problems force reductions in the project budget.
Tools	The code generated by CASE tools is inefficient. CASE tools cannot be integrated.
Requirements	Changes to requirements which require major design rework are proposed. Customers fail to understand the impact of requirements changes.
Estimation	The time required to develop the software is underestimated. The rate of defect repair is underestimated. The size of the software is underestimated.

2. Risk Analysis: Output è una lista di rischi con priorità. Si associa probabilità di occorrenza (very low, low, moderate, high, very high) ed effetto (catastrofico, serio, tollerabile,

insignificante) per ogni rischio.

Risk	Probability	Effects
Organisational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staff are ill at critical times in the project.	Moderate	Serious
Software components which should be reused contain defects which limit their functionality.	Moderate	Serious
Changes to requirements which require major design rework are proposed.	Moderate	Serious
The organisation is restructured so that different management are responsible for the project.	High	Serious
The database used in the system cannot process as many transactions per second as expected.	Moderate	Serious
The time required to develop the software is underestimated.	High	Serious
CASE tools cannot be integrated.	High	Tolerable
Customers fail to understand the impact of requirements changes.	Moderate	Tolerable
Required training for staff is not available.	Moderate	Tolerable
The rate of defect repair is underestimated.	Moderate	Tolerable
The size of the software is underestimated.	High	Tolerable
The code generated by CASE tools is inefficient.	Moderate	Insignificant

I rischi ad alta priorità sono quelli con effetto catastrofico o serio con probabilità almeno alta. Si identificano poi i "top-ten risks" (possono essere anche di più), classificandoli in ordine di importanza.

3. Risk Planning: Ci si concentra sui rischi a priorità maggiore e si pensano strategie per reagire.

- Avoidance Strategies: Ridurre la probabilità di occorrenza.
- Minimisation Strategies: Ridurre l'effetto del rischio sul progetto o prodotto.

Sarebbe ideale applicare entrambe le strategie ad ogni rischio ma ogni rischio è diverso e per alcuni è conveniente usare una strategia piuttosto che un'altra.

Se nessuna delle due è possibile, si adottano Contingency Plans (piani B).

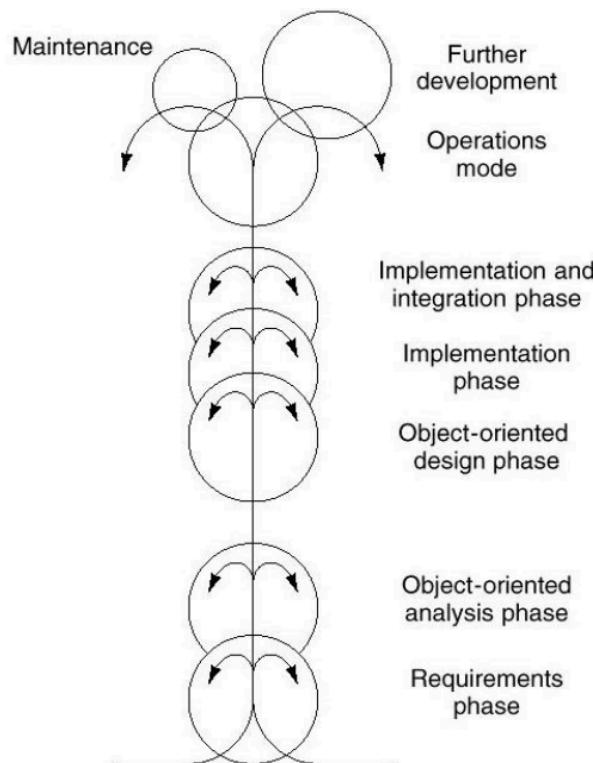
Risk	Strategy
Organisational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Recruitment problems	Alert customer of potential difficulties and the possibility of delays, investigate buying-in components.
Staff illness	Reorganise team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact, maximise information hiding in the design.
Organisational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying in components, investigate use of a program generator.

4. Risk Monitoring: Effettuato periodicamente durante lo sviluppo per rianalizzare rischi cambiati o identificare nuovi fattori di rischio ("campanelli d'allarme").

Risk type	Potential indicators
Technology	Late delivery of hardware or support software, many reported technology problems
People	Poor staff morale, poor relationships amongst team member, job availability
Organisational	organisational gossip, lack of action by senior management
Tools	reluctance by team members to use tools, complaints about CASE tools, demands for higher-powered workstations
Requirements	many requirements change requests, customer complaints
Estimation	failure to meet agreed schedule, failure to clear reported defects

Modello Object Oriented o a Fontana

Prevede un approccio object oriented all'analisi dei requisiti e alla progettazione del software.



Guardando la figura l'etichetta object-oriented sta soltanto sulla fase di analisi dei requisiti e sulla fase di progettazione. Infatti un software progettato secondo questo approccio potrebbe essere implementato anche facendo uso di un linguaggio di programmazione non object oriented. Si differenzia dal Waterfall per la possibilità per ogni fase (cerchio) di essere sviluppata in modo concorrente e iterativo (la concorrenza viene messa in evidenza in quanto le fasi (i cerchi) non sono distinte l'una dall'altra ma si sovrappongono, ossia ad esempio mentre faccio la fase di definizione requisiti posso già iniziare anche l'analisi dei

requisiti (mentre in waterfall sarebbe stato in rigida sequenza)).

L'iterazione viene invece messa in evidenza dalle frecce: quelle all'interno dei cerchi rappresentano le iterazioni intra-fase (all'interno di una singola fase) e quelle che escono dal cerchio che invece sono inter-fase.

La manutenzione è più semplice perché il software prodotto è più facile da modificare (cerchio piccolo in alto a sinistra).

Un altro modello è quello di Ingegneria Simultanea o Concorrente.

L'obiettivo è ridurre costi e tempi tramite un approccio concorrente allo sviluppo di prodotto e del processo ad esso associato. Quindi le fasi di sviluppo coesistono invece di essere eseguite sequenziali. Questo approccio è molto efficace, ma richiede strumenti software che per la gestione efficace del progetto, lo sharing di informazioni e il project management. Si fanno riunioni online per team distribuiti condividendo file e informazioni.

Di tutt'altra natura è il modello basato su Metodi Formali.

Modello utile solo per i software critici. L'analisi dei requisiti è realizzata usando tecniche e linguaggi di specifica formali (matematici), che permettono una specificazione non ambigua e l'uso di tecniche di verifica automatizzate.

Un esempio di realizzazione di questo modello è la cleanroom software engineering (ing. soft. in camera sterile) dove già a partire dall'analisi dei requisiti si deve fare molta attenzione a non inserire errori.

La maggioranza dei modelli visti si applicano con successo per software a contratto (cliente che commissiona). Quando il software è realizzato dall'organizzazione tramite investimenti per il mercato (senza un cliente specifico), si adottano approcci diversi.

Vedremo due modelli di cicli di vita organizzati da due organizzazioni differenti: la prima che sviluppa software di tipo commerciale (pubblica chiedendo ai clienti di pagare la licenza d'uso) e la seconda che invece sviluppa software di tipo freeware.

Modello Microsoft ("Synchronize-And-Stabilize")

All'interno dell'azienda, si adotta un approccio iterativo, incrementale e concorrente che esalta la creatività delle persone coinvolte nello sviluppo di prodotti software. L'approccio adottato da Microsoft, noto come "Synchronize-And-Stabilize" è basato su:

- Sincronizzazione quotidiana: Attività svolte individualmente o in piccoli team (3-8 persone), assemblando componenti in un daily build da testare e correggere. Gli sviluppatori scaricano codice, lavorano, e lo inseriscono nel repository entro una certa ora per la compilazione del daily build.

(Es saltare: Lo sviluppatore si collega ad uno strumento di configuration management (repository centralizzato del codice) e in base all'utente scarica sulla macchina la parte di prodotto su cui dovrà lavorare quella giornata, sarà libero di realizzare quanto richiesto come preferisce, l'unico vincolo è che dovrà inserire in repository quanto fatto entro una certa ora. Il sistema raccoglie tutto, compila, e crea un eseguibile (daily build). Chiaramente non si riesce tipicamente a realizzare subito una build funzionante, l'idea è che chi ha realizzato la parte di codice che non va bene si dovrà fermare oltre tempo stabilito per sistemare)

-Stabilizzazione: periodicamente viene fatta un'attività di stabilizzazione in cui avviene un incremento significativo del prodotto (milestone), tipicamente 3-4 volte nell'arco dell'intera realizzazione del software.

Per quel che riguarda invece il ciclo di sviluppo Microsoft:

1. Planning: Carattere tecnico-gestionale. Si genera la product vision, si parte con il Vision Statement ossia una breve descrizione che articola la visione che c'è dietro il prodotto (necessaria perché non ho un cliente che mi chiede il software ma io azienda lo voglio fare) coinvolgendo Product e Program Managers (coordinati dal Project Manager, responsabile dell'intero progetto). Si realizza poi un Documento di Specifica (dei requisiti) più tecnico. Infine, Schedule and Feature Team Formation, basandosi sul documento di specifica si pianificano le attività e formano i team (Program Manager, 3-8 sviluppatori, 1 tester per sviluppatore).
2. Development: Si definiscono un numero limitato di sottoprogetti, ognuno rappresenta una milestone. Si progetta, codifica e debugga. Il primo sottoprogetto include le funzionalità più critiche.
3. Stabilization (da non confondere con la stabilizzazione periodica): Fase che precede l'immissione sul mercato. Si effettua testing interno (alpha testino) e testing esterno (beta testing) con utenti selezionati (OEMs produttori hardware, ISVs produttori software). Si arriva alla versione finale del prodotto.

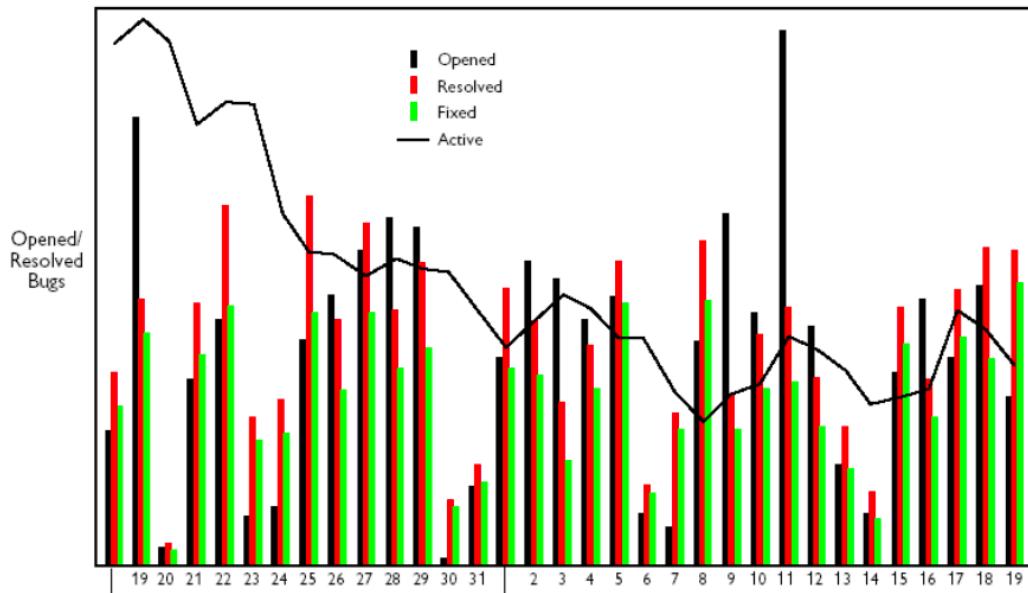
Strategie e Principi

La prima strategia è dedita a definire prodotto e processo e consiste in "considerare la creatività come elemento essenziale" e prevede 5 principi di realizzazione:

- a) dividere il progetto in milestone, tipicamente 3 o 4,
- b) definire una product vision e una specifica funzionale che evolverà durante il progetto (non congelata come in Waterfall),
- c) selezionare funzionalità e relative priorità in base all'utente (dove l'utente è il Product Manager, non esiste cliente poiché non è software a contratto),
- d) definire un'architettura modulare per distribuire i compiti tra i team.
- e) assegnare task elementari e limitare le risorse (Microsoft ha persone molto competenti e ciò è vantaggioso in generale ma svantaggioso perché si tratta di persone incontrollabili, se gli si impongono degli standard se ne vanno. Per controllarle indirettamente si assegnano quindi dei task molto elementari senza dirgli esattamente come risolverli ma limito le risorse ossia gli dico di farlo entro un certo tempo).

La seconda strategia riguarda lo sviluppo e la consegna dei prodotti e consiste nel "lavorare in parallelo con frequenti sincronizzazioni".

- a) si vuole sincronizzare il lavoro dei team di sviluppo con il daily build,
- b) avere sempre un prodotto da consegnare per ogni piattaforma e mercato,
- c) usare lo stesso linguaggio di programmazione all'interno dello stesso sito di sviluppo,
- d) testare continuamente il prodotto
- e) usare metriche per il supporto delle decisioni (in realtà quest'ultimo principio è usato in modo molto limitato perché non essendovi un contratto da rispettare a tutti i costi l'attenzione risiede più che altro nella promessa che l'azienda fa al mercato riguardo il tempo di consegna).



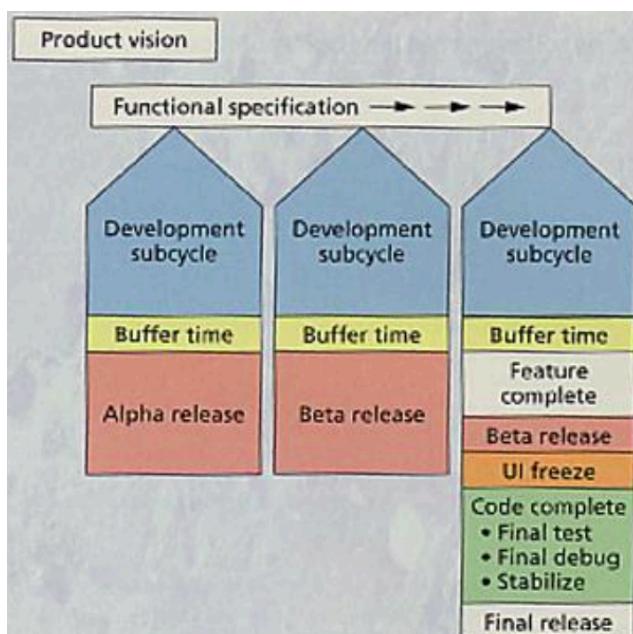
Quindi le metriche sono alla fine limitate, e lo si capisce dal seguente grafico: sulle ascisse le date, le barre rappresentano i dati raccolti ogni giorno relativamente a tre elementi dal punto di vista della qualità del software:

barra nera == opened bugs (problemi riscontrati durante la giornata nel compilare il daily build),

barra rossa == Resolved Bugs post debugging ho capito qual è il problema

barra verde == Fixed Bugs ho effettivamente risolto il problema aggiustando il codice.

La barretta contorta che attraversa il grafico indicherebbe il numero atteso di difetti attivi, ossia quelli che ancora devo scovare. Sebbene si stabilisca un obiettivo di qualità legato ai difetti attivi, il prodotto viene consegnato anche con molti difetti se la data promessa è vicina, poiché rispettare la data è prioritario per l'immagine aziendale.



Nel grafico vediamo come si parta dalla vision per poi avere una specifica che va avanti durante l'intero sviluppo a differenza del waterfall. In questo caso vediamo come il progetto sia diviso in tre cicli di sviluppo, la fine di ognuno delle quali rappresenta una milestone. In

ognuno di questi cicli si procede chiaramente con la fase di development, poi c'è un buffer time (nella fase iniziale si definisce lo scheduling di ciò che si farà e si usano tecniche di stima di tempi, costi.. per capire quando l'azienda sarà in grado di rilasciare il prodotto). Nonostante queste stime siano abbastanza precise, durante lo sviluppo spesso le cose non vanno come pianificato e quindi a ogni milestone si aggiunge un buffer time per cercare di risolvere eventuali ritardi). Si ricorda che il primo sottoprogetto deve contenere le funzionalità critiche affinché se qualcosa non andasse non si è andati troppo avanti. Dopo la prima milestone esce l'alpha release per il testing interno, nel secondo milestone si rilascia la versione Beta e infine dopo la terza milestone si arriva a Feature Complete, non si potranno aggiungere funzionalità, poi ancora rilascio versione beta e poi la UI freeze (congelamento della user interface. La prima cosa su cui si lavora infatti è l'interfaccia utente, che viene verificata in laboratori di usabilità molto avanzati presenti in sede microsoft e che una volta superati i controlli "congela" ossia non potrà più essere toccata fino al rilascio). Si arriva poi al rilascio definitivo.

Synch-and-Stabilize	Sequential Development
Product development and testing done in parallel	Separate phases done in sequence
Vision statement and evolving specification	Complete "frozen" specification and detailed design before building the product
Features prioritized and built in 3 or 4 milestone subprojects	Trying to build all pieces of a product simultaneously
Frequent synchronizations (daily builds) and intermediate stabilizations (milestones)	One late and large integration and system test phase at the project's end
"Fixed" release and ship dates and multiple release cycles	Aiming for feature and product "perfection" in each project cycle
Customer feedback continuous in the development process	Feedback primarily after development as inputs for future projects
Product and process design so large teams work like small teams	Working primarily as a large group of individuals in a separate functional department

Modello Netspace

Anche alla Netscape (browser competitor di microsoft e altri prodotti simili a microsoft) adottava un modello simile a synchronize-and-stabilize, adattato alle applicazioni Internet (browser e server):

In Netscape già la dimensione dello staff cambiava: 3 sviluppatori per un tester (vs 1:1 in Microsoft).

Riguardo il processo software vi era invece scarso effort di pianificazione (eccetto che sui prodotti server che erano la fonte di guadagno dell'azienda.). Documentazione incompleta (attività considerate marginali evitate) e scarso controllo sullo stato di avanzamento del progetto (lasciato all'esperienza e influenza dei PM). Scarso controllo sulle ispezioni del codice (meno tester) e pochi dati storici per il supporto alle decisioni.

Ciò che era il vision statement in microsoft qui diventa una vision generata da Advanced Planning Meeting APM ossia riunioni in cui si discuteva di ciò che si sarebbe potuto realizzare in base alle opportunità di mercato. In queste riunioni erano coinvolti gli esperti di marketing, sviluppatori ed executives (esperti imprenditoriali proprietari dell'azienda).

Da qui la vision, la pianificazione delle attività di sviluppo, si crea la specifica funzionale con gli ingegneri e talvolta col supporto dei product manager (marketing) e si pianificano le varie attività allocando budget, risorse umane e denaro con la discutendone anche con gli executives.

Dopodiché lo sviluppo viene monitorato con meeting periodici (First Executive Review) in cui gli sviluppatori si incontrano con gli executive per informarli dello stato di avanzamento del lavoro.

Poi finita la fase di sviluppo si passa al rilascio dell'alpha e poi due beta che in questo caso venivano rilasciate a tutti permettendo feedback più ampio.

Poi si completava il codice e dopo la stabilizzazione e debugging rilascio RTM (release to manufacturing, copia "gold" sarebbe la versione definitiva).

È stato proprio il fatto di non basarsi su dati reali come supporto alle decisioni e fidarsi dell'esperienza dei PM che ha portato al fallimento dell'azienda. Solo un terzo degli sviluppatori software su prodotti browser, il resto sui server (fonte effettiva di profitto per l'azienda). 660 persone in totale di staff, nemmeno confrontabile con Microsoft, tuttavia competitor serio sul fronte browser.

La mancanza di dati reali a supporto delle decisioni e l'eccessiva fiducia nell'esperienza dei PM, unita all'intervento degli executive che bloccarono l'implementazione di una funzionalità critica chiamata Active Channel: (aggiornamento automatico della pagina) per non ritardare il rilascio, portarono al fallimento dell'azienda.

Metodi Agili

Nati all'inizio degli anni 2000 in reazione alla formalizzazione eccessiva dello sviluppo software (es. Waterfall), per sfruttare la creatività degli sviluppatori e rendere lo sviluppo meno pesante e più agile.

In realtà il modello Agile estende ulteriormente le caratteristiche dell'approccio incrementale puntando su una comunicazione intensa tra sviluppatori e cliente/utente, e su fast feedback. L'Agile Manifesto definisce i valori e i principi:

Tra i valori:

Individui e interazioni contano di più che processi e strumenti,

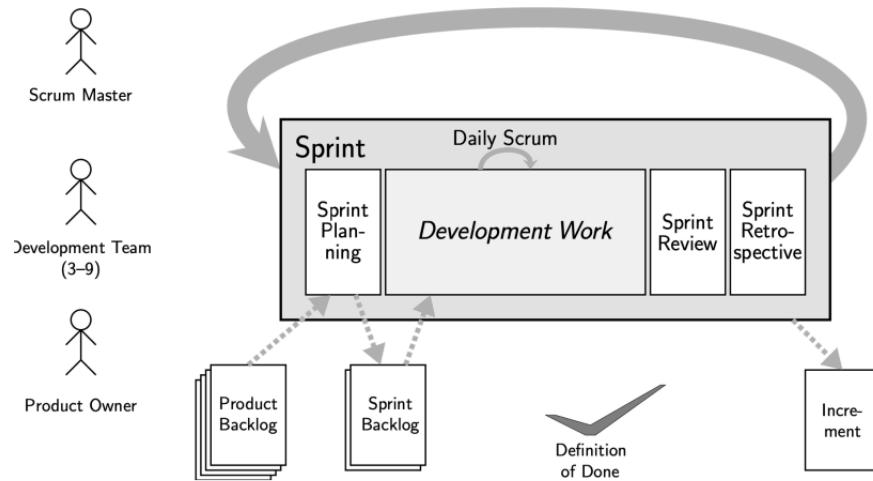
Si punta a produrre software funzionante piuttosto che documentazione,

Conta la comunicazione con il cliente piuttosto che negoziare ai fini di un contratto,

Conta avere team che rispondono ai cambiamenti piuttosto che seguire un rigido piano di lavoro.

Questa sembra un'estremizzazione dell'approccio Build & Fix, ma ciò che cambia è che comunque esiste una sistematica organizzazione. In aggiunta di questi valori contiene dodici principi guida ulteriori.

Uno dei metodi agili più in auge è Scrum.



Basato sul concetto di knowledge-management. Tre ruoli principali:

- Scrum Master: Assicura che la metodologia sia compresa bene e implementata correttamente dal team di sviluppo e dal product owner, non partecipa direttamente alle attività di sviluppo ma garantisce il rispetto delle regole scrum interagendo con i team.
- Product Owner: Capo progetto che gestisce e assegna priorità ai requisiti, contenuti nel Product Backlog (documento con tutte le attività da fare e requisiti).
- Development Team (3-9 persone): Si occupa di sviluppo testing, attività tecniche.

Cinque eventi diversi:

1. Sprint: Corrisponde all'idea di build, un incremento successivo del software, tipicamente richiede 2/4 settimane (quindi build molto piccoli, per garantire feedback continuo).
2. Sprint Planning: Requisiti dal Product Backlog vengono trasferiti nello Sprint Backlog (quanto da fare per quello specifico sprint),
3. Development Work: Il team di sviluppo lavora allo Sprint con meeting quotidiani (daily Scrum meeting) per organizzarsi e identificare problemi.
4. Sprint Review: Alla fine dello Sprint, l'incremento è presentato al cliente.
5. Sprint Retrospective: Si confronta quanto fatto con gli obiettivi iniziali per pianificare lo Sprint successivo, includendo obiettivi non raggiunti (piuttosto che consegnare più tardi lo Sprint posticipo le attività mancanti nel successivo, concetto di timeboxing).

Artefatti prodotti nella parte bassa della figura:

- Product Backlog,
- Sprint Backlog,
- Definition of Done e Incremento: il team stesso definisce cosa significa aver raggiunto l'obiettivo, secondo l'idea di self-organizing.

Scrum è veloce e agile, con una minima forma di processo e organizzazione imposta al team.

User Story (concetto importante nella Definizione dei Requisiti)

Pratica comune nello sviluppo Agile (spesso con Scrum) per far interagire gli utenti in modo più efficace. Infatti l'obiettivo principale dell'ingegnere del software è aiutare gli sviluppatori a

costruire software di qualità (qualità intesa come capacità di soddisfare le esigenze degli utenti). L'utente è colui che userà il prodotto e per questo è l'elemento principale su cui focalizzarsi. Invece di chiedere all'utente come vorrebbe il prodotto, l'idea è di definire questi requisiti ad alto livello iniziali con il concetto di User Story. Definisce un requisito utente come una "storia" breve (una frase) dal punto di vista dell'utente:

Formato Standard: "As a <role>, I want <goal> so that <benefit>."

I requisiti complessi possono essere Epics (suddivisibili in User Stories più piccole). Le User Story popolano il Product Backlog e lo Sprint Backlog in Scrum.

L'obiettivo di tutti questi modelli è stato quello di organizzare le attività di sviluppo, di modo da evitare l'approccio Build & Fix che tanti danni ha fatto nel passato. L'uso dei metodi Agili sono diversi dal Build & Fit, infatti anche questi metodi nonostante siano più "leggeri" rispetto a un waterfall sono comunque organizzati in modo più sistematico.

A differenza dei prodotti fisici (es. DOCG), non esistono "etichette" dirette per certificare la qualità del software. Per ovviare a questo, si certificano non i software ma le organizzazioni che sviluppano il software.

Il modello standard in questo senso è il CMM.

CMM (Capability Maturity Model)

Introdotto nel 1993 dal SEI (Software Engineering Institute), misura il livello di maturità del processo software di un'organizzazione (efficacia nell'uso delle tecniche di ingegneria del software). Attualmente si usa il CMMI (CMM Integrated), che integra altre certificazioni.

Parlando del CMM, il modello è basato su un questionario e schema valutativo a 5 livelli additivo (se certificato a un livello, si è certificato anche per i livelli inferiori):

- Lvl 1 Initial: certifica di base ogni possibile organizzazione. "Success depends on heroes" Il successo dipende dagli "eroi" (persone specializzate), non c'è un vero processo organizzativo.
- Lvl 2 Repeatable: si utilizzano tecniche di base di project management che permettono la ripetibilità (minimo di pianificazione e monitoraggio).
- Lvl 3 Defined: L'organizzazione ha un Processo documentato, standardizzato e integrato per lo sviluppo software.
- Lvl 4 Managed: sì tecniche di misurazione del processo per capirne l'andamento.
- Lvl 5 Optimizing: oltre a controllare e misurare il processo, posso anche migliorarlo.

Per certificarsi, l'organizzazione segue le linee guida del SEI per le KPA (Key Process Area) (18 in totale) relative a ciascun livello. Ogni KPA descrive: obiettivi, impegni e responsabilità, capacità e risorse necessarie, attività da realizzare, metodi di monitoring della realizzazione, metodi di verifica della realizzazione.

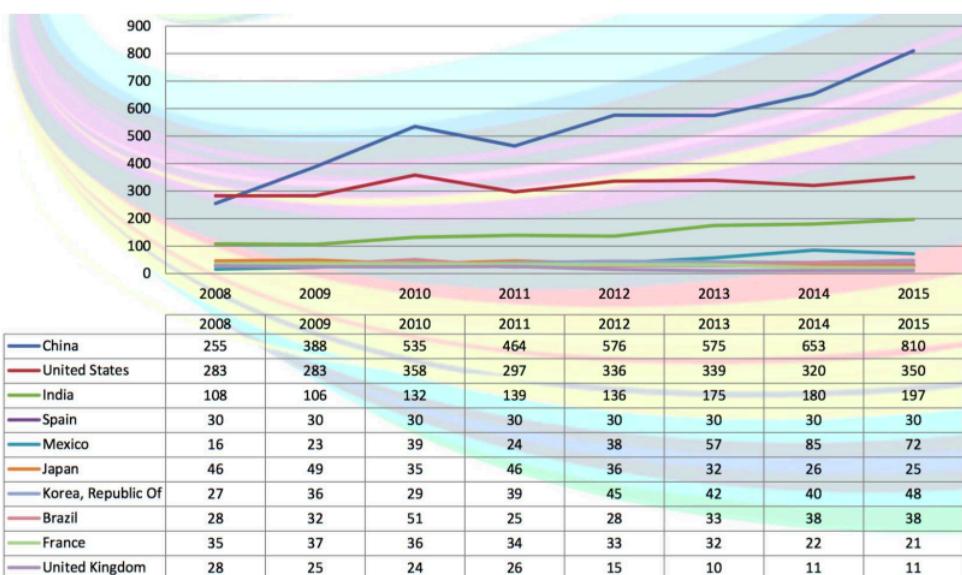
La certificazione CMMI è richiesta come vincolo in vari domini applicativi. Più facile certificarsi per piccole organizzazioni in quanto toccare l'organigramma di grandi organizzazioni per adattarlo alle KPA è molto complesso. NB essendo un modello additivo per certificarmi a livello 3 devo soddisfare anche le KPA a livello 2.

			Result
Level	Characteristic	Key Process Areas	
Optimizing (5)	Continuous process capability improvement	Process change management Technology change management Defect prevention	Productivity & Quality
Managed (4)	Product quality planning; tracking of measured software process	Software quality management Quantitative process management	
Defined (3)	Software process defined and institutionalized to provide product quality control	Peer reviews Intergroup coordination Software product engineering Integrated software management Training program Organization process definition Organization process focus	
Repeatable (2)	Management oversight and tracking project; stable planning and product baselines	Software configuration management Software quality assurance Software subcontract management Software project tracking & oversight Software project planning Requirements management	
Initial (1)	Ad hoc (success depends on heroes)	"People"	Risk

A lvl 2 devo dimostrare di eseguire correttamente 6 delle 18 KPA definite: tra queste software configuration management (capacità di gestire la configurazione dei prodotti come controllo delle versioni), software quality assurance (attività di verifica e convalida), software subcontract management (se prendo un contratto di grandi dimensioni devo poter appaltare parti del progetto a terze parti garantendo al contempo la qualità di quanto fatto da loro), ... A lvl 3 Peer Reviews (tecniche formali di verifica), Training Program (corretta formazione del personale) etc...

A lvl 5 Defect Prevention: si è in grado di utilizzare approcci formali per prevenire l'introduzione di difetti nel software.

Le statistiche suggeriscono come lvl 5 per aziende che necessitano software sicuri per attività critiche (vedi Boeing). Chiaramente si certifica il reparto software dell'azienda e non l'azienda nella sua interezza. La maggior parte delle organizzazioni sono certificate a lvl 3 perché per buona parte dei domini di mercato il vincolo è quello.



REQUISITI SOFTWARE

Un requisito software è una descrizione dei servizi che un sistema software deve fornire, insieme ai vincoli da rispettare durante lo sviluppo e la manutenzione. Secondo lo standard IEEE, un requisito si articola in tre punti:

- Una condizione/capacità necessaria per risolvere un problema o raggiungere un obiettivo dell'utente.
- Una condizione/capacità che il sistema deve possedere per soddisfare un contratto, uno standard, una specifica o altri documenti imposti (non necessariamente derivante dall'utente).
- Una rappresentazione documentale di una condizione/capacità come descritto nei punti precedenti.

La complessità nella definizione dei requisiti deriva dal fatto che possono esistere a diversi livelli di astrazione e possono cambiare man mano che il progetto evolve.

Se una compagnia/cliente avvia un contratto per un software di grandi dimensioni deve definire le sue necessità in modo astratto (ossia ad alto livello di astrazione) tale per cui non esista una soluzione preconfezionata (viene detto semplicemente ad alto livello ciò che si vuole).

Infatti questi requisiti devono essere descritti in modo che diversi contractor possano partecipare al bando di gara per ottenere il contratto e realizzare il software. Ogni contractor farà la propria proposta dicendo come secondo lui dovrà essere realizzato il software per realizzare quelle necessità e la compagnia decide a quale assegnare il contratto. Dopodiché il contractor dovrà scrivere una system definition per il cliente molto più dettagliata in modo che questo capisca cosa il software dovrà fare. Entrambi i documenti sono definibili come documenti di requisiti.

Ogni requisito in realtà deve essere valutato a due dimensioni: livello di astrazione e tipologia. Riguardo il livello di astrazione ne esistono due tipi:

- Requisiti Utente (Requirement Definition): Tipicamente scritti in linguaggio naturale, eventualmente con diagrammi, per descrivere servizi e vincoli operativi. Devono essere comprensibili a tutti, poiché sono scritti per e con il cliente.
- Requisiti di Sistema (Specification): Specificati in un documento ben strutturato che descrive dettagliatamente i servizi che il software deve fornire. Possono usare linguaggi specifici (es. linguaggi di modellazione).

Il documento contenente sia i requisiti utente che quelli di sistema è quello che costituisce il contratto tra cliente e fornitore.

Esiste anche in alcuni casi un altro tipo di requisito, il Software Specification, ancora più dettagliato dei requisiti di sistema e spesso utilizzato per software critici (linguaggi formali per prevenire errori).

Ora un breve richiamo di definizioni di termini:

- Cliente: Persona o organizzazione che paga per la fornitura del software.
- Fornitore: Persona o organizzazione che produce il software per il cliente.
- Utente Finale: Persona che interagisce direttamente con il prodotto software, non necessariamente corrispondente al cliente.

Esempio (saltare):

“Requisito utente:

1. Il sistema software deve fornire un mezzo per rappresentare e visualizzare file esterni generati da altri tool.

Requisito di sistema:

1.1 L'utente deve avere la possibilità di definire il tipo dei file esterni.

1.2 Ad ogni tipo di file esterno deve essere associato il tool che lo ha generato.

1.3 Ogni tipo di file esterno deve essere rappresentato mediante una specifica icona sullo schermo.

1.4 L'utente deve avere la possibilità di definire l'icona che rappresenta il tipo di file esterno.

1.5 Quando l'utente seleziona un'icona che rappresenta un file esterno, deve poter essere eseguito il tool in grado di visualizzare il file.”

In questo esempio è chiaro il differente livello di astrazione.

Lettori dei requisiti:

- Requisiti Utente: Manager del cliente, utenti finali del sistema, eventuali ingegneri del cliente, manager del contractor, system architects.

- Requisiti di Sistema: System end-users, ingegneri del cliente, software developers (progettisti e sviluppatori di codice), system architects. (Nota: i client manager e contract manager non sono più lettori principali a questo livello).

La classificazione dei requisiti avviene lungo due dimensioni: una basata sul livello d'astrazione e l'altro basato sul concetto di categoria.

Esistono due principali categorie di requisiti: Funzionali e non Funzionali.

I Requisiti Funzionali descrivono le funzionalità del sistema software in termini di servizi forniti agli utenti e come il sistema software si comporta di fronte a specifici input e situazioni particolari.

Es: “Il sistema software deve fornire un appropriato visualizzatore per i documenti memorizzati.”

Si noti come questo esempio sia requisito utente in quanto molto ad alto livello, descrive precisamente cosa vuole l'utente in termini di sue necessità e basta una frase per descrive.

I Requisiti Non Funzionali (anche detti extrafunzionali) coprono una gamma più ampia di requisiti, infatti sono tutti quei requisiti che descrivono le proprietà del sistema software in relazione a determinati servizi o funzioni e possono anche essere relativi al processo di sviluppo.

- Possono riferirsi a caratteristiche di qualità del software (es. efficienza, affidabilità, sicurezza, ...).

- Possono riferirsi a caratteristiche del processo di sviluppo (es. standard di processo, uso di ambienti CASE, linguaggi di programmazione, metodi di sviluppo imposti dal cliente).

- Possono riferirsi a caratteristiche esterne (es. interoperabilità, vincoli legislativi, privacy).

Es 1 (saltare): “Il tempo di risposta del sistema all'inserimento della password utente deve essere inferiore a 10 sec”.

Es 2: “I documenti di progetto (derivable) devono essere conformi allo standard ABC”.

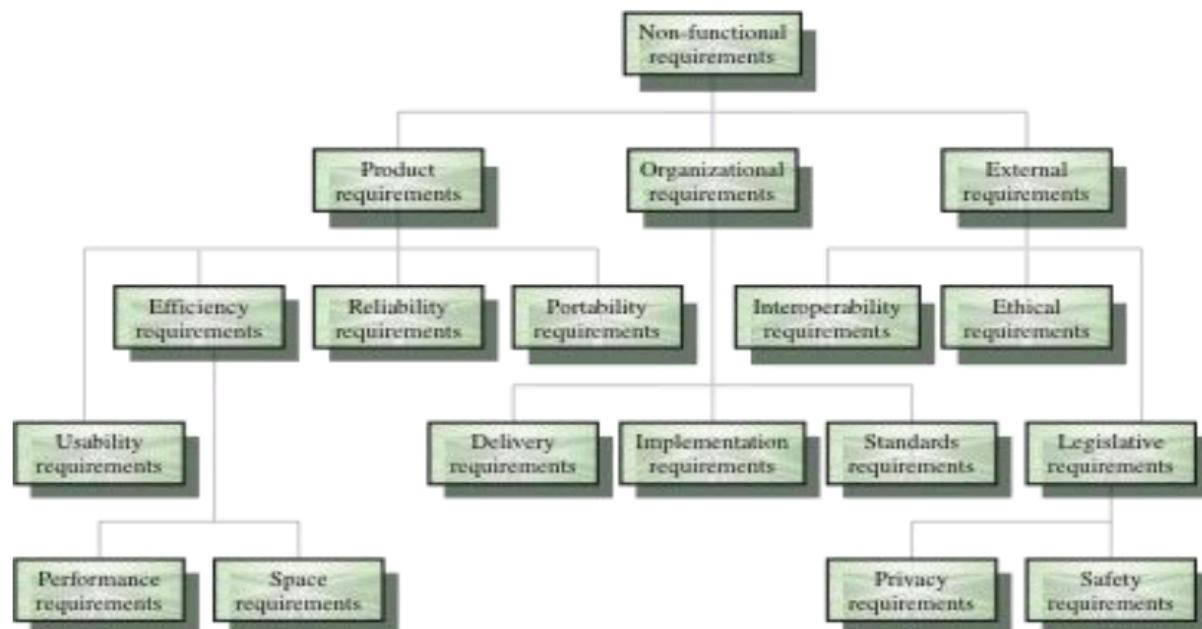
Es 3: “Il sistema software non deve rilasciare ai suoi operatori nessuna informazione personale relativa ai clienti, tranne nominativo e identificatore”.

Requisiti di Dominio: "terza" categoria di requisiti, che si sovrappone con le altre due. Infatti un requisito di dominio rappresenta un requisito funzionale o non funzionale che deriva dal dominio applicativo.

Es: "I documenti di rendiconto contabile, secondo la normativa ABC, devono essere stampati alla ricezione e cancellati immediatamente".

I requisiti non funzionali sono più difficili da trattare rispetto a quelli funzionali a causa della loro ampia gamma rispetto a quelli funzionali, ma soprattutto perché per identificare le caratteristiche che il mio software dovrà assumere (quindi i relativi requisiti non funzionali) devo far riferimento a un modello di qualità, che le descrive in modo preciso.

Qui di seguito una gerarchia dei requisiti non funzionali



Non esiste comunque una strategia migliore per l'identificazione dei requisiti non funzionali: si può prima decidere di scrivere i funzionali ma spesso si opera in modo congiunto. La cosa importante è però essere molto attenti a definire questi requisiti fin dall'inizio dal momento che questi tipicamente sono scritti in linguaggio naturale e quindi possono comportare problemi.

Problemi Comuni nella Definizione dei Requisiti:

- Ambiguità: il requisito non deve poter essere interpretato in modi diversi.

ES: specificare un tempo senza far riferimento al fuso orario

ES: significato di "appropriato visualizzatore": dal punto di vista dell'utente magari un visualizzatore appropriato per il documento in questione (per word visualizzatore word, per excel excel etc..) mentre dal punto di vista dello sviluppatore magari implementare un generico visualizzatore di testo che mostri il contenuto del documento.

- Incompletezza: I requisiti non includono tutte le caratteristiche richieste; il prototipo rapido può aiutare a superare questa difficoltà. (spesso neanche il cliente sa definire bene quello che vuole)

- Inconsistenza: Conflitti o contraddizioni tra diversi requisiti

ES: Req 1 "ogni form di input deve contenere non più di 5 campi editabili dall'utente" e Req 2

“nella forma di input relativa all’inserimento di dati anagrafici deve introdurre nome, cognome, anno, nascita, cell, fax...”

- Verificabilità: I requisiti, soprattutto quelli non funzionali, devono essere espressi in modo quantitativo per poter dimostrare il loro rispetto.

Alcuni requisiti sono facili da misurare come efficienza (speed) e dimensioni, ma altri meno come l’usabilità (“facile da usare”, per misurarla abbiamo il training time (inteso come il “tempo di addestramento” necessario per un utente affinché apprenda ad usare il software). ES: per un software per aerei un aggiornamento prevede un training time anche molto lungo affinché ci si assicuri che i piloti siano a piena conoscenza di quanto realizzato) oppure ancora number of help frames (i frame nel software che spiegano in dettaglio quando ci punto sopra cosa fa quella funzionalità).

Ulteriore requisito difficile da definire in termini di misure è la Portabilità: facilità nel portare un prodotto software in esecuzione su una piattaforma differente rispetto a quella nativa per cui è stata sviluppata (da iOS a Windows). Una misura potrebbe essere la percentuale di istruzioni dipendenti dalla piattaforma nativa (come requisito dico di far sì che questa percentuale non superi un tot affinché in futuro possa sistemare il codice per spostarmi in un’altra piattaforma).

Come scriviamo i requisiti? Sono tipicamente espressi in linguaggio naturale seguendo però alcune linee guida per evitare problemi, tra cui evitare l’uso di termini tecnici, usare un formato standard per ogni requisito, evidenziare le parti fondamentali per ogni requisito, utilizzare il linguaggio naturale in modo consistente (es. uso di “deve” per requisiti necessari e “dovrebbe” per quelli desiderabili).

3.5.1 Adding nodes to a design

3.5.1.1 The editor shall provide a facility for users to add nodes of a specified type to their design.

3.5.1.2 The sequence of actions to add a node should be as follows:

1. The user should select the type of node to be added.
2. The user should move the cursor to the approximate node position in the diagram and indicate that the node symbol should be added at that point.
3. The user should then drag the node symbol to its final position.

Rationale: The user is the best person to decide where to position a node on the diagram. This approach gives the user direct control over node type selection and positioning.

Specification: ECLIPSE/WS/Tools/DE/FS/Section 3.5.1

Qui un esempio di standard per definire requisito utente, numerazione rigorosa e roba importante in grassetto, poi man mano che scendo specifica di come si vorrebbe ad alto livello che si comporti il software in accordo con il requisito.

Poi Rationale che sarebbe il perché esiste quel requisito e Specification che è un puntatore al corrispondente requisito sistema (alla specifica di questo requisito utente).

Quando invece si arriva a scrivere i requisiti di sistema la possibile scelta di linguaggio è molto più ampia rispetto al naturale dei requisiti utente ,richiedono molta precisione in quanto saranno questi i requisiti usati come base per il progetto software.

Diverse possibili notazioni:

Si considera informale l'utilizzo di linguaggio naturale strutturato (comunque più specifico di quello utente) e formale quello di specifiche matematiche (non ambigue per definizione e verificabili in modo automatico, utilizzate per software critici in quanto molto costosi), e tra questi due estremi le notazioni semi-formali: PDL come pseudocodice, poi utilizzo di notazioni grafiche.

- basato su *form in linguaggio naturale*

ECLIPSE/Workstation/Tools/DE/FS/3.5.1

Function	Add node
Description	Adds a node to an existing design. The user selects the type of node, and its position. When added to the design, the node becomes the current selection. The user chooses the node position by moving the cursor to the area where the node is added.
Inputs	Node type, Node position, Design identifier.
Source	Node type and Node position are input by the user, Design identifier from the database.
Outputs	Design identifier.
Destination	The design database. The design is committed to the database on completion of the operation.
Requires	Design graph rooted at input design identifier.
Pre-condition	The design is open and displayed on the user's screen.
Post-condition	The design is unchanged apart from the addition of a node of the specified type at the given position.
Side-effects	None
Definition:	ECLIPSE/Workstation/Tools/DE/RD/3.5.1

Anzitutto vediamo il passaggio dal requisito utente di prima al corrispettivo requisito di sistema in linguaggio naturale strutturato. Si tratta di definire nei dettagli cosa devo realizzare, intuitivamente la “dichiarazione” della funzione (so esclusivamente la firma: parametri di input e output).

Infatti si legge nell'esempio solo la descrizione, gli input, output (design identifier sta a rappresentare il fatto che si aggiorna il progetto con il nuovo nodo),
Source come sorgente dei valori dei parametri in input, destination (database del progetto perché il progetto deve essere aggiornato in relazione all'aggiunta del nodo),
Requisiti (l'utente deve avere un progetto aperto, identificato da un certo design identifier),
Pre-condizione (condizione che deve essere vera affinché possa eseguire la funzione, in questo caso quando il progetto è aperto e mostrato sullo schermo),
Post-condizione (condizione che deve essere vera dopo aver eseguito la funzione, cambia solo l'aggiunta del nodo),
Effetti collaterali e Definition che punta al requisito utente corrispettivo.
Starà allo sviluppatore realizzare la “definizione” della funzione. Nonostante questa specifica sia comunque vicina a quanto necessario per descrivere la funzione, può comunque comportare problemi di ambiguità e per questo spesso si preferisce far riferimento a linguaggi semi-formali come PDL.

- basato su **PDL** (*Java-like*)

```
class ATM {
    // declarations here
    public static void main (String args[]) throws InvalidCard {
        try {
            thisCard.read () ; // may throw InvalidCard exception
            pin = KeyPad.readPin () ; attempts = 1 ;
            while ( !thisCard.pin.equals (pin) & attempts < 4 )
                { pin = KeyPad.readPin () ; attempts = attempts + 1 ;
                }
                if ( !thisCard.pin.equals (pin) )
                    throw new InvalidCard ("Bad PIN");
            thisBalance = thisCard.getBalance () ;
            do { Screen.prompt (" Please select a service ") ;
                service = Screen.touchKey () ;
                switch (service) {
                    case Services.withdrawalWithReceipt:
                        receiptRequired = true ;

```

Si descrive il comportamento per l'implementazione del comportamento di un ATM. Viene descritto in java-like cosa succede quando metto la carta (quindi anzitutto potrebbe non leggerla come valida, poi provo il pin e ogni pin corrisponde a un tentativo per un massimo di 3, poi se scrivo bene il pin arrivo alla schermata per gestire il mio conto etc...).

Usando pseudocodice evito ambiguità che potrebbero esservi nell'uso di linguaggio naturale strutturato. Tuttavia PDL ha uno svantaggio: ci troviamo ancora nella fase di specifica del software e quindi della descrizione di cosa deve fare il software e non come deve essere implementato (ciò sarà a carico dalla fase di progettazione in poi). Il rischio che quindi si corre con PDL è che si pesti i piedi ai lavori dei progettisti, si deve usare lo pseudocodice non per dettagli algoritmici.

Quindi più che un esempio visto sopra di solito è conveniente usare PDL esclusivamente per la definizione di interfaccia, che si limitano a dire cosa fare invece di come (equivalenza di dichiarazione di funzione piuttosto che definizione).

- specifica di interfaccia basata su **PDL**

```
interface PrintServer {

    // defines an abstract printer server
    // requires: interface Printer, interface PrintDoc
    // provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter

    void initialize ( Printer p ) ;
    void print ( Printer p, PrintDoc d ) ;
    void displayPrintQueue ( Printer p ) ;
    void cancelPrintJob ( Printer p, PrintDoc d ) ;
    void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
```

Quanto viene scritto sia per i requisiti utente (definizione) che per requisiti di sistema (specifiche) viene inserito nel documento guida di ogni progetto di sviluppo software: il Documento di Analisi dei Requisiti (o Documento di Specifica).

Describe COSA il sistema deve fornire (dominio del problema), non COME deve essere sviluppato (dominio della soluzione, compito delle fasi successive). Interviene continuamente durante lo sviluppo e la manutenzione; la sua assenza può portare a tecniche di reverse-engineering per ricavarlo a partire dal software.

Ruoli che contribuiscono: Cliente, Manager (per investimenti), System Engineers (per sviluppo), System Test Engineers (per generare Plan Test e test case), System Maintenance Engineers (la manutenzione più comune non è quella correttiva ma quella perfettiva, che indirizza le modifiche ai requisiti).

Le modifiche ai requisiti richiedono la modifica del documento, per cui è importante mantenere link di tracciabilità per valutarne l'impatto. Per la stesura si usa un template basato sullo standard IEEE 830-1998:

- Preface: Lettori attesi, cronologia delle versioni, riepilogo delle modifiche.
- Introduction: Scopo, breve descrizione del sistema, interazione con altri sistemi, ambito nel contesto aziendale.
- Glossary: Definizione di termini tecnici.
- User requirements definition: Requisiti utente funzionali e non funzionali.
- System architecture: Panoramica di alto livello dei componenti del sistema.
- System requirements: Specifiche dei requisiti di sistema funzionali e non funzionali.
- System models: Descrizione delle relazioni tra i componenti del sistema e tra il sistema e il suo ambiente.
- System evolution: Assunzioni sulla base del sistema e cambiamenti previsti (evoluzione hardware, bisogni utente).
- Appendices: Informazioni specifiche sull'applicazione (es. descrizioni hardware e database).
- Index: Indice dei contenuti, indice alfabetico, lista dei diagrammi.

REQ ENGINEERING

Varia in base al dominio applicativo (es. software critico allora specifica matematica), alle persone coinvolte e all'organizzazione (usiamo un approccio Object Oriented):

1. Studio di Fattibilità:

Fase preliminare che stabilisce se procedere con lo sviluppo. Si basa su una descrizione sommaria del sistema e delle necessità utente. Deve essere svolta rapidamente in quanto se l'esito è no devo passare a un'altra. Informazioni raccolte tramite colloqui con Client Manager, Ingegneri del Software esperti nel dominio, Esperti di tecnologie, Utenti finali. Lo studio di fattibilità produce come risultato un report che stabilisce l'opportunità o meno di procedere con lo sviluppo.

2. Identificazione e Analisi dei Requisiti e 3. Specifica dei Requisiti:

Nel caso in cui lo studio di fattibilità abbia dato esito positivo si parte con l'Identificazione e Analisi di requisiti. Si ha un incontro tra team di sviluppo, cliente e utenti finali per identificare i requisiti utente, dalla cui analisi si generano quelli di sistema. L'incontro coinvolge gli stakeholder (coloro che hanno interesse diretto o indiretto nei requisiti del sistema).

I task principali di questa attività sono:

- Comprensione del Dominio: L'analista deve studiare in tempi brevi il dominio applicativo. Importante soprattutto dal punto di vista dell'analista software (es. se il sistema software

riguarderà un ufficio postale, allora l'analista ne deve comprendere il funzionamento).

- Raccolta dei Requisiti: Interazione con gli stakeholder.
- Classificazione dei requisiti: l'insieme dei requisiti viene poi suddiviso in sottoinsiemi (es. requisiti riguardanti gestione dei dati, riguardanti l'interfaccia utente etc..)
- Risoluzione dei conflitti: Identificazione di conflitti/contraddizioni.
- Assegnazione delle priorità: si dà priorità ai requisiti anche mediante interazione con stakeholder (importante soprattutto quando si utilizza un approccio incrementale per cui prima devo gestire quelli di maggior importanza)
- Verifica dei requisiti: per verificarne completezza e consistenza.

Tecniche di Identificazione dei Requisiti:

- Prototipazione: Costruire un prototipo per comprendere meglio le esigenze utente e ridurre rischi.
 - Casi d'uso: Basati su scenari, chiedendo al cliente di descrivere come userebbe il sistema
 - Etnografia: L'analista osserva il lavoro di un'organizzazione quando non si ottengono informazioni dirette dall'utente.
- Tecniche di analisi (e specifica) dei requisiti (più che di tecniche qui si parla di metodi):
- Semi-formali: Basate su modelli del sistema (es. metodi di analisi strutturata o orientata agli oggetti).
 - Formali: Specifiche matematiche (es. macchine a stati finiti, Petri Net, ...).

4. Convalida dei Requisiti:

È necessario trovare difetti per evitare costosi rework in fasi avanzate. I controlli includono quindi validità, consistenza, completezza, realizzabilità, verificabilità e ci sono delle tecniche che supportano il lavoro dell'analista nella convalida dei requisiti, tra cui:

- Revisioni Informali: Esame dei requisiti da parte di colleghi (es. come in microsoft per cui per ogni sviluppatore c'è un tester, in questo caso però ci si riferisce ai requisiti)
- Revisioni Formali: Prevedono tecniche di Walkthrough (discussione passo-passo del documento con tutti i contributori per identificare problemi) e Ispezioni (molto più formali, con un team di ispezione e ruoli precisi, processo definito, e riunioni di confronto. Sono costose ma efficaci).
- Prototipazione
- Generazione di Test Case: per cui tramite la generazione di test case capisco se ho scritto bene o male un requisito (se si ha problema nel generare test case è un problema legato ai requisiti e non chi lo genera tipicamente)
- Analisi di consistenza automatizzata (per requisiti formali).

5. Gestione dei Requisiti:

I requisiti cambiano continuamente, si ritoccano anche durante e dopo lo sviluppo (contrario del modello Waterfall).

Quindi la Gestione dei Requisiti rappresenta il processo di identificazione e controllo delle modifiche subite da essi durante il ciclo di vita del software.

I requisiti di un sistema software possono essere classificati in:

- Requisiti Stabili: hanno probabilità minima di esser modificati nel tempo
- Requisiti Volatili: elevata probabilità di modifica. Questi possono essere mutabili (modifiche legate a cambiamenti dell'ambiente operativo), emergenti (modifiche causate da una migliore comprensione del sistema software dopo aver interagito con gli stakeholder), consequenziali (dovute all'introduzione di sistemi informatici nel flusso di lavoro es. se alle

poste i bollettini a un certo punto vengono pagati digitalmente) e di compatibilità (legate a cambiamenti nei sistemi e nei processi aziendali).

Tali eventuali modifiche dei requisiti devono essere pianificate mediante:

- identificazione univoca dei requisiti in questione
- gestione delle modifiche tramite analisi di costi, dell'impatto e della realizzazione
- politiche di tracciabilità per capire esattamente quali sono le relazioni tra requisiti e progetto del sistema software
- uso di tool CASE per supporto alle modifiche (devo usare strumenti che mi supportino in questo lavoro) (lo strumento per eccellenza per la requirements engineering in generale è IBM Doors)

L'uso di specifiche formali richiede effort maggiore, deve però essere affiancato da una fase di verifica. La fase di specifica con la specifica formale è molto costosa, tuttavia comporta tipicamente la riduzione dei costi per le fasi successive di progettazione.

Esempi di specifiche formali: (spesso sono affiancate da una sintassi visuale più semplici).

Petri Net

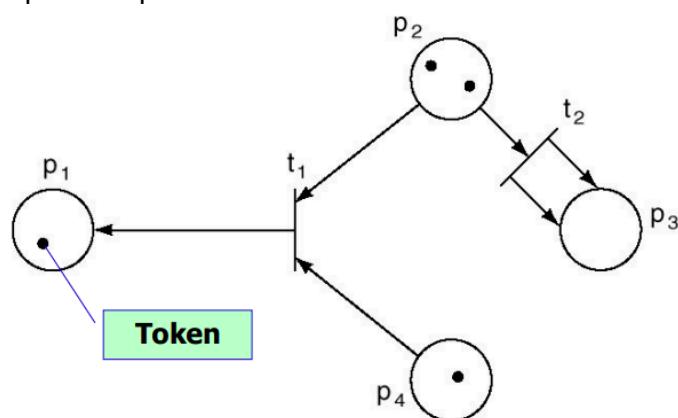
Introdotte ancor prima della nascita della ISW per specificare sistemi con elevato grado di concorrenza e problemi di temporizzazione e sincronizzazione come le telecomunicazioni.

La versione originale fornisce una sintassi visuale nella quale esistono tre costrutti fondamentali:

Place (cerchio): rappresenta una componente software o una condizione,

Transitions (barre): Rappresentano un'azione o un evento,

Archi Orientati: (frecce) collegano sempre o posti a transizioni o transizioni a posti, indicando input e output.



Questo sistema formale serve a capire come un certo sistema evolve nel tempo in fase di esecuzione.

Per rappresentare l'evoluzione del sistema in questo senso si utilizza un altro elemento, il Token (pallini neri inseriti all'interno dei places). L'operazione per cui inserisco token nei places è detta Marking, che può essere iniziale (distribuzione iniziale di token), intermedi e marking finale.

Per definire l'evoluzione del sistema è necessario definire delle regole. Anzitutto diremo che una Transizione è Abilitata se esiste almeno un token all'interno di ogni posto collegato in ingresso. (Nel caso sopra sia t1 che t2 sono abilitate)

Ora l'idea è che a partire dallo stato iniziale il sistema potrebbe evolvere svolgendo la

transizione t1 o la t2, l'idea è di svolgerle tutte per trovare eventuali inconsistenze.

Si definisce ora come Firing di una transizione come il prelievo da parte del place in ingresso e consegna al place in uscita di tanti token quanti sono gli archi uscenti.

(esempio: se è t1 a scattare allora si tolgoano un token da p2 e uno da p4 e se ne mette uno a p1 (uno solo perché un solo arco in uscita)).

Quindi stato iniziale $S_0 = (1, 2, 0, 1)$ (dentro p1 1 token, in p2 2 etc..).

Quindi stato $S_1 = (2, 1, 0, 0)$. A questo punto t1 non è più abilitata perché in p4 non ci sono più token ma resta abilitata t2 perché 1 token in p2. A questo punto scatta t2 e quindi p2 avrà 0 token e p3 2 token poiché 2 archi in uscita.

Quindi $S_2 = (2, 0, 2, 0)$ e a questo punto dato che non ci sono più transizioni abilitate S_2 è stato finale.

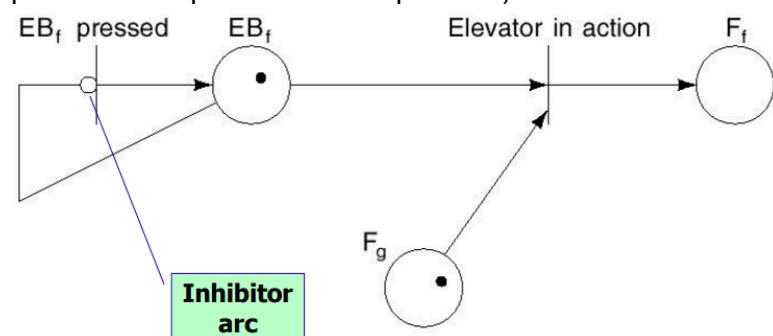
Facendo invece partire prima t2 avrò $S_1 = (1, 1, 2, 1)$, ancora una volta mi accorgo di avere un bivio poiché sia t1 che t2 sono abilitate, scelgo t1 $S_2 = (2, 0, 2, 0)$ ed è finale. Se facevo scattare t2 invece $S_2 = (1, 0, 4, 1)$ anch'esso finale.)

Un altro elemento della Petri Net è l'arco inibitore (pallino bianco con barra): la transizione è abilitata sse non c'è nessun token in ingresso.

L'obiettivo del modello in definitiva è verificare che in tutte le possibili esecuzioni il sistema funzioni correttamente.

(Es: saltare) Dopo aver capito le regole generale della rete di Petri vediamole applicate al caso specifico di un software.

Ipotizziamo ad esempio di voler analizzare il comportamento di un software che regola la centralina di controllo di un ascensore (considerabile sistema critico in quanto un danno potrebbe comportare danni a persone).



Viene esplicitato via rete di Petri un requisito in particolare: se l'ascensore si trova al piano terra e un utente lo chiama dal primo piano deve andare al primo piano. Si necessitano in questo caso 3 posti:

F_g == ascensore al ground floor,

F_f == ascensore al first floor e

E_{Bf} == elevator button first floor (si chiama l'ascensore al primo piano).

Il token al posto F_g specifica che l'ascensore si trova al piano terra, mentre il token in E_{Bf} specifica che il pulsante è stato premuto. Per rappresentare il fatto che l'ascensore si muove uso la transizione elevator in action: tale transizione è abilitata come detto nelle regole se esiste almeno un token sugli archi in ingresso. Nel nostro caso vediamo che è abilitata, quindi si arriva al nuovo stato per cui non vi è alcun token in E_{Bf} o F_g ma un token in F_f . Ci serve una transizione che specifichi la pressione del pulsante al primo piano E_{Bf} , a questo serve E_{Bf} pressed dove è presente un arco inibitore.

Quando quindi E_{Bf} non ha alcun token per rappresentare il fatto che qualcuno preme il

pulsante la self transition inserisce il token in Ebf, e se ha un token abilitato e qualcuno ripreme il pulsante non succede nulla.

(FINE Esempio)

Usando una specifica formale anche un requisito semplice richiede parecchio lavoro, è molto costoso utilizzarle (notazione molto più ampia del linguaggio naturale).

Soltamente limitata alla parte di controllo del software (non ha senso usarla per l'interfaccia). Ulteriore costo sta nel fatto che oltre che costruire il modello questo deve anche essere validato (devo verificare che i comportamenti corrispondano con il mio software quindi convalida molto importante). Il principale vantaggio è che esistono tool automatici che verificano il tutto.

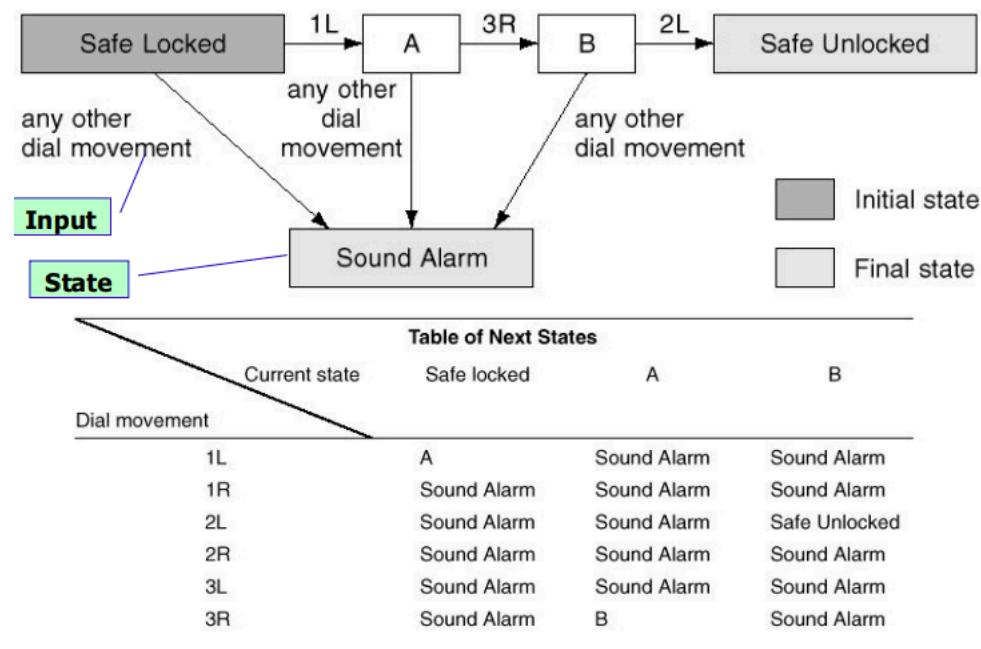
A fronte di alcune limitazioni della Petri Net, nel corso del tempo sono stati sviluppati diversi "dialetti" di questa specifica formale. Una limitazione lampante ad esempio è che ogni transizione rappresenta un'azione che è istantanea, per cui non vi è un tempo associato alla relativa esecuzione.

Tale limitazione è superata da un dialetto chiamato GSPN (generalised stochastic petri net): Associa un tempo ad ogni transizione per convalidare tempi prestazionali del sistema (basarsi anche sul tempo di esecuzione di specifiche funzioni del software).

Un altro ancora è CPN (colored PN): Introduce colori per i token per descrivere comportamenti diversi per classi di utenti (in base a chi lo utilizza).

Un'altra caratteristica della PN è che il concetto di stato è rappresentato indirettamente (per conoscerlo devo vedere la distribuzione di token nei place).

La notazione che ora vediamo invece prevede un altro approccio: Specifica Formale con Macchine a Stati Finiti FSM.



L'obiettivo è, come per PN, rappresentare la dinamica (evoluzione) di un sistema attraverso un insieme di stati. La differenza principale è che la primitiva di base permette di rappresentare direttamente lo stato del sistema.

La primitiva di base è il rettangolo che rappresenta lo stato, l'arco orientato rappresenta un evento che porta il passaggio a uno stato successivo. Stati iniziali con grigio più scuro, finali più chiaro.

(ES: saltare) L'esempio qui presente rappresenta il comportamento di una cassaforte. Si parte dalla cassaforte chiusa, che per essere aperta prevede il movimento di una "manopola". Stati intermedi per rappresentare il passaggio dopo ogni movimento. (es. 1L == un movimento a sx). Qualsiasi movimento sbagliato porta allo stato Sound Alarm. (FINE esempio)

Un problema, come in PN, è che rappresentando un sistema molto complesso posso avere un numero di stati enorme. Ciò che si fa normalmente è associare a ogni FSM una Table of Next States con colonne stato e righe azione.

Sia FSM che PN non sono notazioni create per la specifica di software, una notazione formale creata appositamente è il linguaggio Z. La primitiva di base è il concetto di schema. Sia con PN che con FSM i due concetti fondamentali erano stato e azione per descrivere l'evoluzione del sistema, in questo caso lo schema basta a rappresentare entrambe. Ogni schema Z ha il seguente formato: Nome, dichiarazioni di variabili e predici che agiscono sulle variabili.

(ES: saltare) Vediamo un esempio sia per rappresentare lo stato che l'azione. Si specifica lo schema del bottone di un ascensore.

esempio di specifica di stato

Button_State	
floor_buttons, elevator_buttons	: P Button
buttons	: P Button
pushed	: P Button
floor_buttons	\cap elevator_buttons = \emptyset
floor_buttons	\cup elevator_buttons = buttons

Abstract Initial State

Button_init := [Button_State' | pushed' = \emptyset]

Si distinguono i bottoni dentro l'ascensore e sui piani per chiamarlo. Si definiscono le variabili con il nome e per definire il tipo si usa la notazione ": P Button" dove in questo caso Button è l'insieme di tutti i pulsanti e P denota l'insieme potenza (insieme di tutti i possibili sottinsiemi).

Di questo tipo si definiscono i bottoni dentro e fuori ascensore, i bottoni e i bottoni premuti. Si definisce lo stato iniziale come "Button_init := [Button_State' | pushed' = \emptyset]" (nello stato iniziale l'insieme dei bottoni dell'ascensore ha stato tale per cui non è premuto, l'insieme pushed è vuoto).

Nella parte dei predici specifico come deve comportarsi lo stato del bottone: l'intersezione delle variabili floor buttons e elevator buttons è vuoto (sono pulsanti distinti) mentre la loro unione sono tutti i bottoni.

Vediamo invece l'utilizzo dello schema per un'azione: premere il bottone.

esempio di specifica di operazione

```
Push_Button
ΔButton_State
button?: Button
(button? ∈ buttons) ∧
(((button? ⊏ pushed) ∧ (pushed' = pushed ∪ {button?})) ∨
((button? ∈ pushed) ∧ (pushed' = pushed)))
```

Delta rappresenta gli stati su cui agisce l'azione (in questo caso lo schema definito prima). Nella parte declarations, trattandosi di un'operazione (funzione), definisco i parametri della funzione (Button). Il fatto che sia parametro di input è evidenziato da ? (output sarebbe stato !). Nella zona predicati definisco invece come agisce quest'operazione sulle variabili dello stato che prende in considerazione (button_state). La prima riga mi dice che il parametro di input deve appartenere all'insieme dei buttons, messo in AND con ciò che specifica che se il bottone è spento si accende, se è già acceso invece non si fa nulla. (se il bottone non appartiene a pushed allora ("implicito" nell'and) il nuovo valore dell'insieme di pushed (nuovo valore perché ') è uguale al vecchio valore + il bottone che ho appena premuto, oppure se il bottone appartiene a pushed allora il nuovo insieme è uguale al precedente).

(FINE esempio)

Specifiche semi-formali (livello intermedio tra linguaggio naturale e specifiche formali).

L'approccio principale è costruire un modello del sistema (rappresentazione astratta) per facilitare la comprensione del suo funzionamento. Per descrivere completamente il sistema è necessario costruire diversi modelli, che lo rappresentino da vari punti di vista (informazioni, funzioni e comportamento dinamico (evoluzione)).

Esistono due principali metodi di specifica semiformali;

- Metodi di analisi strutturata (o procedurale).
- Metodi di analisi orientata agli oggetti.

Per costruire questi modelli del sistema si necessita di un linguaggio di modellazione. Ne esistono di diversi tipi, ognuno dei quali definisce un modello specifico.

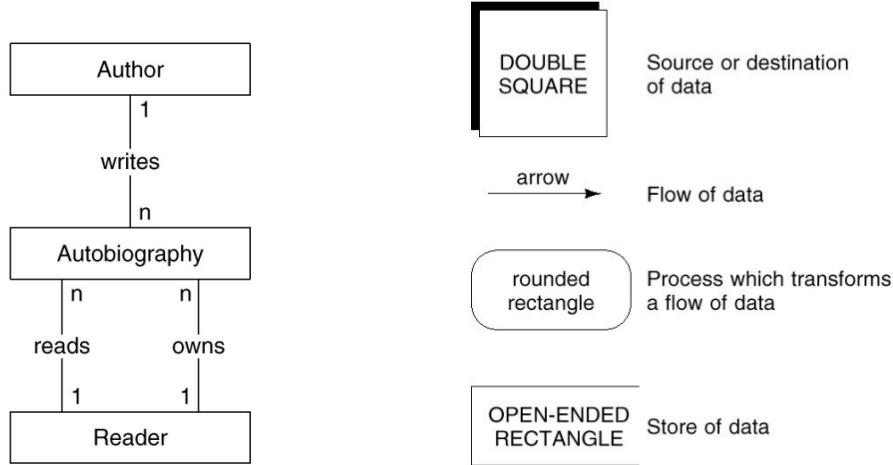
Per descrivere la specifica semi-formale di un sistema software si usano 3 tipi di modelli:

- Modello dei dati: rappresenta gli aspetti statici e strutturali relativi ai dati (data requirements): ERD (not UML), class diagram (UML)
- Modello comportamentale: rappresenta gli aspetti funzionali del sistema (functional requirements): data flow diagram (not UML), use case diagram (UML), activity diagram (UML), interaction diagram (UML)
- Modello dinamico: rappresenta gli aspetti di "controllo" e di come le funzioni del modello comportamentale modificano i dati introdotti nel modello dei dati: state diagram (UML).

UML (Unified Modeling Language) linguaggio standard di modellazione utilizzato per l'approccio Object Oriented. I not UML sono linguaggi di modellazione che venivano usati prima dell'arrivo dei metodi OO, quando ancora si usavano solo metodi di analisi strutturata. Si tratta di ERD e Data Flow Diagram.

ERD (Entity Relationship Diagram) diagramma entità relazione per costruire modello dei dati. Il Data Flow Diagram invece serve a definire modelli comportamentali. (come a basi di dati entità-relazione). In particolare DFD utilizza quattro costrutti per rappresentare le varie funzioni del software e come elaborano i dati, come partano da una sorgente e arrivino ad una destinazione. Il diagramma non deve essere letto in ordine temporale, ma solo in termini di come i dati viaggiano all'interno del software. Normalmente si costruisce una gerarchia di DFD dove per ogni raffinamento si ha un dettaglio maggiore.

(ERD e DFD)



Uno dei metodi di analisi strutturata più usati prima dell'avvento degli object oriented era lo SSA (Structured System Analysis). Composto da 9 step e basato sul concetto di step-wise refinement, l'obiettivo è fornire passo passo una guida al fine di completare l'attività di specifica:

Step 1: Si partire producendo il Data Flow Diagram.

In particolare si dovrebbe utilizzare il documento dei requisiti utente o il prototipo (all'epoca si utilizzava principalmente waterfall e prototipo rapido) per identificare i flussi di dati, le sorgenti e destinazioni di dati e i processi (funzioni) che trasformano i dati. Inoltre si vuole che si proceda per raffinamenti successivi aggiungendo dei dati ai data flow esistenti (gerarchia livelli astrazione).

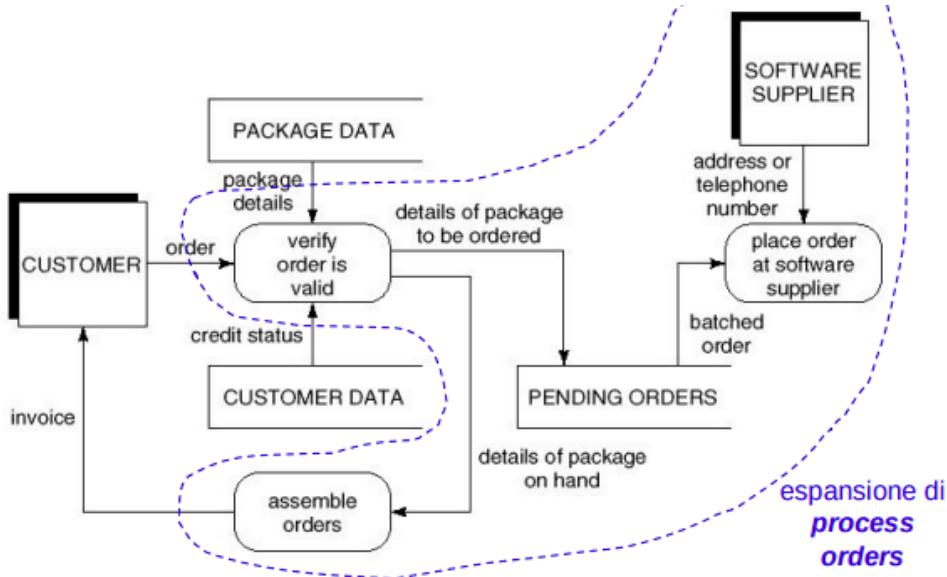
Step 2: Decidere quali sezioni devono essere automatizzate e come.

Si parte dall'analisi costi-benefici per decidere quali sezioni DFD automatizzare, e si decide come devono esserlo: Batch o Online.

Online Processing: i dati sono analizzati real time, online (es. quando inserisco i dati nel form e il software si accorge se manca qualcosa).

Batch Processing: operazioni che avvengono per lotti, (es. posso decidere di accumularne ed effettuare le operazioni insieme).

(ES: saltare)



Nell'esempio di DFD con secondo raffinamento di prima il processo “verify order is valid” è sicuramente online, mentre “place order at software supplier” batch (infatti archivio i dettagli ordine nel pending orders per poi magari successivamente chiamare i fornitori “software supplier” per inviare gli ordini).

(FINE es)

I successivi tre passi sono relativi rispettivi a un raffinamento di data flow, processi e data stores.

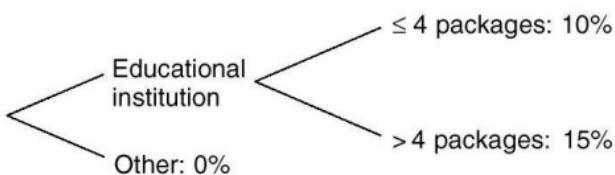
Step 3: Prevede quindi di determinare i dettagli del data flow.

(ES: saltare) nel DFD di prima ho un arco “order” da cliente a processo, devo raffinarlo per capirne la vera e propria struttura ottenendo quindi dettagli come order_identification, customer_details, package_details. Si procede poi ad approfondire ulteriormente questi dati per vedere se sono dati elementari o ancora di tipo strutturato. order_identification ad esempio è un intero a 12 interi, costumer_details è invece informazione strutturata composta da customer_name, customer_address etc... Devo fare l'operazione fino a definire tutto come dato elementare per ogni data flow.

(FINE es)

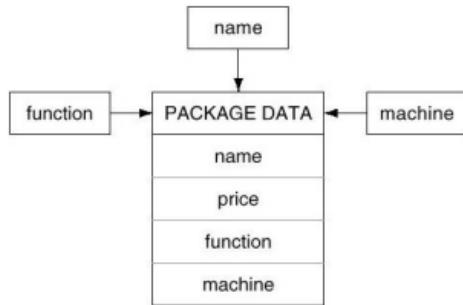
Step 4: Prevede di definire la logica processi.

Utilizzando ad esempio un albero decisionale per definire il comportamento di una funzione (es give_educational_discount):



Step 5: Si occupa dei dettagli dei data store, si definisce il contenuto esatto di ogni archivio e come rappresentarlo (secondo il formato specifico di un certo linguaggio di programmazione).

Ad esempio tramite l'utilizzo di DIAD (data-immediate-access diagram).



Per memorizzare i dettagli di un package li organizzo per nome, prezzo, funzione e machine (sistema operativo). Il DIAD specifica quali sono i campi utilizzabili per effettuare ricerche dirette sui dati (in questo caso name, function e machine, per cui posso chiedere ad esempio quali sono tutti i software di un certo nome, oppure che girano su windows, o che sono elaboratori di testo, ma non in base al prezzo).

Step 6: Prevede di definire le risorse fisiche.

Oggi si sfrutta un DBMS per l'archiviazione di dati vs il vecchio file system. In tal caso ne andava specificato il nome, l'organizzazione in termini di accesso (sequenziale, indicizzato...), il mezzo di memorizzazione, i record il tutto arrivando a livello di dettaglio "field level" (es. il nome lo specifico come una stringa di 30 caratteri...)

Step 7: si determina la specifica dell'Input/Output.

Si devono specificare le forme di input (le maschere di inserimento dei dati) a livello di componenti e layout, ugualmente per gli screen di output e l'output da stampare. Queste informazioni erano necessarie perché all'epoca non erano disponibili interfacce utente di tipo grafico, ma solo testuale e con spazi molto limitati.

Step 8: Determinare il dimensionamento.

Si stima il volume dell'input (es. numero di ordini attesi dal cliente giornalmente), frequenza di ogni report da stampare e la sua scadenza, dimensione e numero di record da passare dalla CPU alla memoria di massa (oggi impensabile in quanto lo spazio di memorizzazione è molto più grande) e dimensione di ogni file (es. file anagrafica clienti).

Step 9: Determinare i requisiti hardware.

Si definiscono quindi i requisiti della memoria di massa, per il backup (all'epoca dischi magnetici o nastro), le caratteristiche dei terminali utente e dispositivi di output, adeguatezza dell'hardware esistente per il software ed eventuali costi per hardware aggiuntivo.

Chiaramente molte di queste scelte sono state abbandonate con l'evoluzione tecnologica, ma grazie a questi step l'analista software procedeva per raffinamenti successivi a completare la specifica.

Il documento ottenuto doveva esser approvato dal cliente per poi passare alla fase di progettazione.

OOA (Object Oriented Analysis)

Questa è ancora una fase di definizione che si concentra su COSA deve fare il prodotto senza occuparsi del COME (compito della fasi di OOD).

OOA e OOD deve fornire una rappresentazione corretta, completa e consistente:

- degli aspetti statici e strutturali relativi ai dati (modello dei dati).
- degli aspetti funzionali del sistema (modello comportamentale).
- degli aspetti di "controllo" e di come le funzioni del modello modificano i dati (modello dinamico).

Un metodo OOA non definisce solo le tecniche, ma anche l'insieme di procedure, strumenti e tecniche per avere un approccio sistematico alla gestione e allo sviluppo.

L'input del metodo OOA è l'insieme dei requisiti utente, l'output è l'insieme dei modelli del sistema che definiscono la specifica del software.

OOA fa principalmente uso di notazioni visuali (diagrammi), ma essendo semiformale queste notazioni sono affiancate da metodi più tradizionali basati su linguaggio naturale.

Lo sviluppo dei modelli di OOA non è sequenziale ma iterativo (si hanno raffinamenti successivi), e la costruzione dei modelli avviene in parallelo.

Esistono diversi metodi OOA: Catalysis, Objectory, Shaler/Mellor, OMT (Object Modeling Technique), Booch, Fusion.

(NB OOD == Object Oriented Design per la fase di progettazione, per ora ci concentriamo su OOA).

I metodi da tenere in considerazione per l'introduzione di UML:

Objectory: basato sui diagrammi dei casi d'uso (scenari)),

OMT (Object Modeling Technique): basato su tecniche di modellazione iterative, focalizzato principalmente sulla parte di analisi OOA,

Booch: (simile a OMT ma focalizzato sulla parte di design OOD).

Per unificare le notazioni usate (classi, oggetti, ...) dai vari metodi di OOA e OOD, è stato proposto e adottato nel '97 come standard OMG (Object Management Group) il linguaggio UML (Unified Modeling Language). UML non è un metodo né un processo di OOA, ma è solo una notazione/linguaggio che non vincola la scelta del metodo. UML si compone di 9 formalismi di base (diagrammi) e un insieme di estensioni. Questi sono:

1. Use Case Diagram: Evidenzia come gli utenti (attori) utilizzano il sistema, molto flessibile per requisiti, specifica e progettazione.
2. Class Diagram: Diagramma strutturale che descrive le classi con proprietà e associazioni che le legano.
3. State Diagram: Rappresenta il comportamento dinamico dei singoli oggetti di una classe in termini di stati possibili e transizioni di stato per effetto di eventi.
4. Activity Diagram: Particolari state diagram che rappresentano modelli di flusso di attività (work-flow), usati per il modello comportamentale.
5. Sequence Diagram: Evidenzia le interazioni che oggetti di classi diversi si scambiano nell'ambito di un determinato caso d'uso, ordinate in sequenza temporale (non evidenziano le relazioni tra oggetti).
6. Collaboration Diagram: Descrive le interazioni (messaggi) tra oggetti, evidenziando le relazioni esistenti tra le singole istanze (semantica identica al Sequence Diagram).

7. Object Diagram: Legato al Class Diagram, rappresenta oggetti e relazioni tra essi in un caso d'uso specifico.
 8. Component Diagram: Evidenzia la strutturazione e le dipendenze tra componenti software.
 9. Deployment Diagram: Evidenzia le configurazioni dei nodi elaborativi di un sistema real-time e i componenti, processi e oggetti assegnati a tali nodi.
- Use Case Diagram: Evidenzia come gli utenti (attori) utilizzano il sistema, molto flessibile per requisiti, specifica e progettazione.

Come detto i metodi OOA sono iterativi, e il punto di partenza cambia in base al modello. L'8 e il 9 sono diagrammi di implementazione che si soffermano più sulla parte di progettazione.

Il Modello dei Dati rappresenta da un punto di vista statico e strutturale l'organizzazione logica dei dati da elaborare, utilizzando il Class Diagram per definire classi, attributi, operazioni e associazioni. È fondamentale perché un sistema software orientato agli oggetti è un insieme di oggetti che "collaborano". La costruzione del modello dei dati è iterativa e incrementale.

Durante la prima iterazione, ci si concentra sulle entity classes, cioè le classi rilevanti per il dominio applicativo, lasciando per dopo le control classes (logica di esecuzione) e boundary classes (interfaccia utente). Per le entity classes, si iniziano a definire attributi e associazioni, mentre le operazioni sono suggerite dal modello comportamentale.

Approcci per l'Identificazione delle Classi:

- Noun Phrase: Si prendono tutte le frasi nominali dai requisiti utente e ogni sostantivo diventa una classe candidata. Nel secondo step, l'analista decide se è Rilevante (entity class), Irrilevante (non appartiene al dominio applicativo) o Fuzzy (non ci sono sufficienti informazioni)
- Common Class Patterns: Basato sulla teoria di classificazione, identifica classi a partire da gruppi predefiniti (es. Concept, Events, Organization, People, Places) usando la sola conoscenza del dominio applicativo e quindi non i requisiti utente. Non è sistematico e può causare ambiguità.
- Use Case Driven: I requisiti utente sono definiti tramite Use Case Diagram (scenari di funzionamento del software). Simile all'approccio noun phrase, assume che l'insieme degli use case sia completo e corretto. Gli attori diventano automaticamente classi entity.
- CRC (Class Responsibility Collaborators): Basato su riunioni con apposite card, dove ogni carta ha nome, responsabilità (cosa fa) e classi con cui si relaziona -> in generale questo approccio è più utile quando le classi sono già state identificate.
- Approccio Mixed: Mixed: Fusione dei precedenti, ad esempio: 1) identificazione iniziale con Common Class Patterns; 2) aggiunta di classi con Noun Phrase e Use Case Driven; 3) verifica con CRC.

Linee guida per l'identificazione di Entity Classes:

- 1) Deve avere uno specifico obiettivo, statement of purpose.
- 2) Deve essere istanziata prevedendo la realizzazione di un insieme di istanze (oggetti). Le "singleton classes" (per cui si prevede una sola istanza) non sono di norma classificabili come entity classes.
- 3) Prevede un insieme di attributi (non un solo attributo).

- 4) Importante distinguere tra elementi modellabili come classi o attributi.
- 5) Prevede per ogni classe un insieme di operazioni (anche se inizialmente trascurate).

Linee guida per la specifica delle classi

Ad ogni classe deve essere associato un Nome significativo, che abbia una lunghezza massima, che sia al singolare e che adotti una convenzione standard (di modo che già solo dalla sintassi del nome capisca se è classe o attributo, es. come abbiamo visto parole multiple congiunte e CamelCase).

Riguardo gli attributi invece devono ovviamente anch'essi adottare una convenzione standard (nel nostro caso minuscolo, snake_case). Riguardo le operazioni queste conviene aggiungerle solo a partire dalla presenza di un modello comportamentale.

(ES: saltare fino a pag 52)

Vediamo ora degli esempi, casi di studio, per approfondire questa roba.

Vedremo per software di gestione iscrizioni università, videonoleggio, contact management e telemarketing.

L'università offre corsi di laurea triennale. Riguardo la struttura si fa riferimento quando era diviso in facoltà per didattica e dipartimento per ricerca. Division == facoltà contiene i dipartimenti, ogni corso di studio è associato a una singola facoltà e un corso di studio potrebbe includere materie di altre facoltà. Il software serve alla gestione dei programmi di studio per ogni studente con corsi propedeutici, obbligatori e in termini di vincoli come conflitti di orari e dimensione massima delle classi.

È poi richiesto che il sistema supporti pre-iscrizioni e iscrizioni, inviando mail con istruzioni per iscriversi e i voti presi nell'ultimo semestre se ci si iscrive dal secondo anno in poi. Al momento dell'iscrizione il sistema deve accettare il piano di studi e convalidarla (alcuni aspetti saranno convalidati automaticamente, altri no).

Nel Video Store ogni cassetta e disco e ogni cliente è identificato da un codice a barre. I clienti possono prenotare video da prendere in una data specifica e il software deve aiutare l'impiegato a rispondere alle domande del cliente (es. se ha un film).

Software di tipo gestionale (rapporto con clienti frontend) Contact Management è una componente importante per sistemi di gestione aziendale chiamati ERP (enterprise resource planning) per la gestione e automatizzazione di attività di backend di un'organizzazione. Componenti tipici: package che gestisce contabilità, produzione... Il Contact Management CMS è package che gestisce le risorse umane. Il CMS serve all'azienda, che si occupa di ricerche di mercato, per gestire i rapporti con i clienti ossia con tutte le organizzazioni che comprano i prodotti (ricerche di mercato) dell'azienda. Tramite il software l'azienda può interagire con clienti passati, presenti e potenziali. Il sistema deve essere disponibile a tutti gli impiegati ma con diversi livelli di accesso. Il sistema gestisce lo scheduling di attività legate al contatto clienti.

Si ha infine il caso di telemarketing. In questo caso si guarda un'azienda che vuole vendere biglietti della lotteria per beneficenza. Questi sistemi di telemarketing si basano sulla capacità del database di gestire la coda delle chiamate. Per incentivare si chiamano i vecchi supporter, si danno dei bonus se si comprano tanti biglietti o se si diventa nuovi supporter. La compagnia non utilizza le pagine gialle per chiamare (random) ma solo potenziali contributori.

Vediamo ora la prima applicazione per l'University Enrolment (A.1 == prima iterazione). Consideriamo il seguente requisito e identifichiamo usando noun phrase le classi candidate: "Each university degree has a number of compulsory courses and a number of elective courses." I sostantivi di questa frase si ha laurea, numero e corsi. Corsi e lauree sono rilevanti, infatti il sistema ne dovrà mantenere informazione. Quindi sicuramente faranno parte del class diagram. Number d'altro canto è irrilevante, concetto generico. Gli altri due concetti sono corso obbligatorio e corso opzionale, che sicuramente sono informazioni da mantenere nel nostro sistema. Tuttavia per memorizzare queste informazioni è davvero necessario introdurre classi a parte o mi basta usare un attributo tipo? Allora per adesso lascio in Fuzzy.

"Each course is at a given level and has a credit-point value, A course can be part of any number of degrees, Each degree specifies minimum total credit points value required for degree completion, Students may combine course offerings into programs of study suited to their individual needs and leading to the degree in which enrolled"

Nella seconda parte si vuole sapere un certo corso in quali lauree è incluso, questa è una tipica associazione, il numero minimo di crediti per laurearsi, etc.. (nel class diagram attributi, classi e associazioni tra classi). Si prende quindi ogni requisito, si utilizza lo stesso approccio dove si identificano le classi rilevanti e non e dove si è indecisi si mette nelle classi Fuzzy. Vedremo come le associazioni sono implementate come attributi. Ultimo requisito: studente rilevante, course offering (corso erogato in diverse istante durante l'anno a diversi canali) fuzzy perché potrebbe essere interessante avere informazioni sull'offerta di un certo corso, programs of study rilevante.

Relevant classes	Fuzzy classes
Course	CompulsoryCourse
Degree	ElectiveCourse
Student	StudyProgram
CourseOffering	

Quanto viene detto dall'analista è leggermente diverso: StudyProgram in fuzzy perché l'obiettivo è comunque minimizzare il numero di classi e probabilmente posso gestire l'informazione in un altro modo.

Passiamo ora a B, prima iterazione del caso di studio Video Store. Primo Requisito: "The video store keeps in stock an extensive library of current and popular movie titles. A particular movie may be held on video tapes or disks."

Video store, stock, library, movies, tapes e disks sostantivi. Videostore irrilevante perché le linee guida dicevano che per identificare una classe è necessario identificare un certo insieme di istanze, e poiché il software è dedicato a un singolo negozietto e non a una catena non ha senso identificare la classe. MovieTitle, VideoTape e VideoDisk sono rilevanti in quanto bisogna memorizzare il codice a barre per ognuno di questi. Stock e Library sono generici.

Altri requisiti: "Video tapes are in either "Beta" or "VHS" format, Video disks are in DVD format, Each movie has a particular rental period (expressed in days), with a rental charge to

that period, The video store must be able to immediately answer any inquiries about a movie's stock availability and how many tapes and/or disks are available for rental, The current condition of each tape and disk must be known and recorded".

Relevant classes	Fuzzy classes
MovieTitle	RentalConditions
VideoMedium	
VideoTape	
VideoDisk (or DVDDisk)	
BetaTape	
VHSTape	

Betatape, VHStape non come attributi ma come classi dal momento che evidentemente necessito di stabilirne diverse proprietà. Videomedium scelta dell'analista di creare una classe dove inserire tutte le proprietà comuni sia a dischi che a cassette come se fosse una gerarchia (figli sono beta e vhs). RentalConditions come fuzzy perché il prestito potrebbe esser gestito sia come classe che come attributi, ancora da decidere.

Passiamo alla prima iterazione Contact Management:

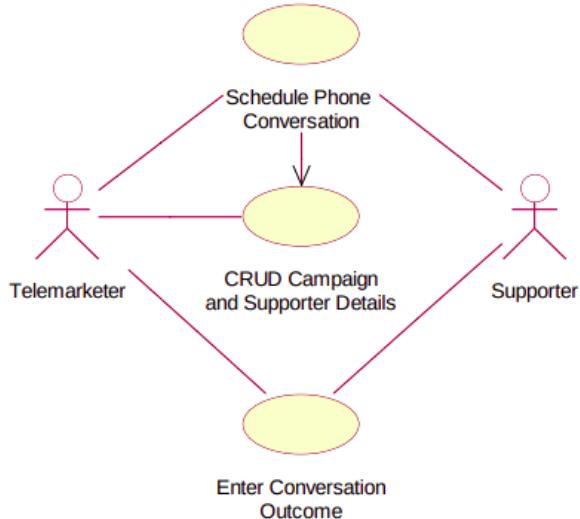
"To "keep in touch" with current and prospective customer base, To store the names, phone numbers, postal and courier addresses, etc. of organizations and contact persons in these organizations, To schedule tasks and events for the employees with regard to relevant contact persons, Employees can schedule tasks and events for other employees or for themselves, A task is a group of events that take place to achieve a result (e.g. to solve customer's problem), Typical types of events are: phone call, visit, sending a fax, arranging for training, etc".

Relevant classes	Fuzzy classes
Organization	CurrentOrg
Contact	ProspectiveOrg
Employee	PostalAddress
Task	CourierAddress
Event	

L'idea è di specificare il concetto di cliente (generico) usando le classi più specifiche Organization e Contact. Similmente a quanto visto con l'uni per corsi obbligatori e non, qui si parla di clienti potenziali o attuali che potrei implementare come classi o con attributi perciò li metto in fuzzy. L'ultimo requisito sembrava evidente dover essere implementato tramite l'uso di attributi, ma postaladdress e courieraddress sono state messe in fuzzy in quanto variano in base all'uso del software: attributi se me la cavo a specificare l'indirizzo come semplice stringa, altrimenti potrebbe essermi utile implementarla con tanti campi diversi come classe.

Vediamo adesso la prima iterazione per il Telemarketing dove invece sfruttiamo l'approccio Use Case Driven: business use case diagram sotto riportato.

Si parla di Business use case diagram dal momento che il livello di astrazione è molto alto. Come già detto in precedenza l'use case diagram mi permette di specificare i possibili scenari di utilizzo del software.

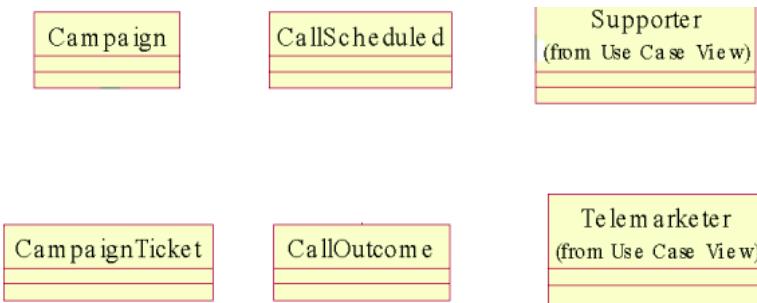


Le ellissi rappresentano la primitiva di caso d'uso, poi si ha l'omino che rappresenta la primitiva attore, ossia colui (o qualcosa, non è detto che sia un essere umano) con cui il software interagisce. In questo caso si hanno due attori: telemarketer (che chiama) e supporter (colui che eventualmente compra biglietti della lotteria). In particolare in questo diagramma il telemarketer rappresenta l'utente di questo diagramma in quanto come vedremo è colui che attiva i 3 casi d'uso, mentre il supporter è un attore che può essere coinvolto durante la loro esecuzione.

Come avevamo specificato inizialmente i telemarketer effettuano le chiamate e alla fine di esse devono registrare i risultati della chiamata -> i due casi d'uso principali sono quello in alto e in basso. Il primo si attiva quando deve fare la chiamata al supporter, il secondo a fine chiamata. Si ha poi un caso d'uso nel mezzo: CRUD Dettagli Campagna e Supporter. CRUD == Create, Read, Update, Delete (le 4 operazioni base che si possono fare con i dati), questo caso d'uso mi serve quando devo compiere una di queste operazioni per i dati relativi sia la campagna che il supporter (es. se il supporter acquista 3 biglietti devo segnare indirizzo, numero etc... e devo aggiornare il fatto che 3 biglietti sono stati venduti). I casi d'uso e gli attori sono connessi da delle linee che rappresentano il coinvolgimento dell'attore o l'attivazione da parte dell'attore di quel caso d'uso. Quando scenderemo a livello d'astrazione specifica vedremo come dovremo introdurre le frecce e non sarà sufficiente l'uso generico di linee. Appare poi il collegamento tra i primi due casi d'uso (la freccia) e indica la relazione generica tra due elementi UML. In questo caso rappresenta il fatto che mentre faccio una chiamata può nascere l'esigenza di aggiornare i dati (CRUD). Come avevamo anticipato non è effettivamente scritto nel diagramma cosa deve fare il software una volta attivato il caso d'uso. Per questa ragione affianco al diagramma per ogni caso d'uso specificato va inclusa una nota testuale in linguaggio naturale specifichi cosa fa il software quando si verifica un caso d'uso.

Ora, dal punto di vista della ricerca di entity classes, tutti gli attori diventano automaticamente delle classi entity. Oltre a questo devo far riferimento alla notazione testuale: Si procede come nel noun phrase evidenziando i sostantivi e ragionandoci sopra. "The telemarketer requests the system that the phone call to a supporter be scheduled and dialed up, Upon successful connection, the telemarketer offers lottery tickets to the supporter. During a conversation, the telemarketer may need to access and modify both

campaign and supporter details (CRUD, create – read – update – delete), Finally, the telemarketer enters the conversation outcome, i.e. the successful or unsuccessful results of the telemarketing action”.



Per la prima volta vediamo la soluzione presentata con il formalismo del class diagram (nome, attributi, operazioni, gli ultimi due per ora vuoti).

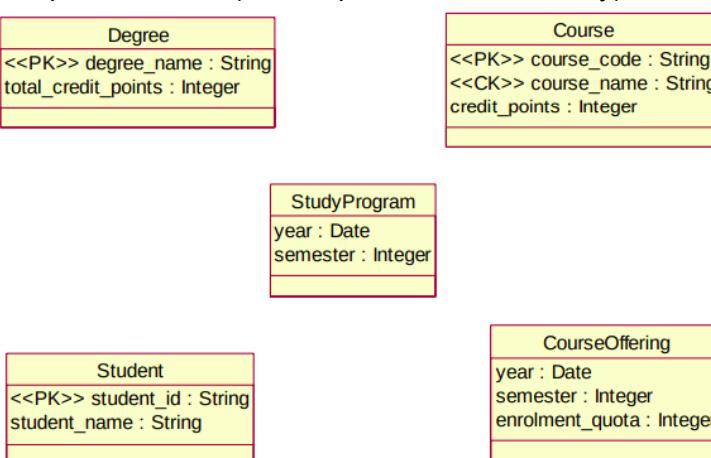
Si passa alla seconda iterazione dell'università.

“A student's choice of courses may be restricted by timetable clashes and by limitations on the number of students who can be enrolled in the current course offering.”

Per questo requisito mi è sufficiente introdurre gli attributi per CourseOffering già definito in precedenza dove enrolment_quota == numero max di studenti.

“A student's proposed program of study is entered in the on-line enrolment system, The system checks the program's consistency and reports any problems, The problems need to be resolved with the help of an academic adviser, The final program of study is subject to academic approval by the delegate of the Head of Division and it is then forwarded to the Registrar”.

Vi è, visti i nuovi requisiti, la necessità di introdurre StudyProgram come classe e non come semplice attributo (dubbio perché stava in fuzzy).



Questi attributi non fanno riferimento solo ai requisiti appena letti, ma anche a quelli della prima iterazione (es. per ogni corso era stato specificato di dover conoscere i crediti -> credit_points in Course). CumpolsoryCourse e ElectiveCourse stavano in Fuzzy ma non riappaiono -> conveniva toglierli.

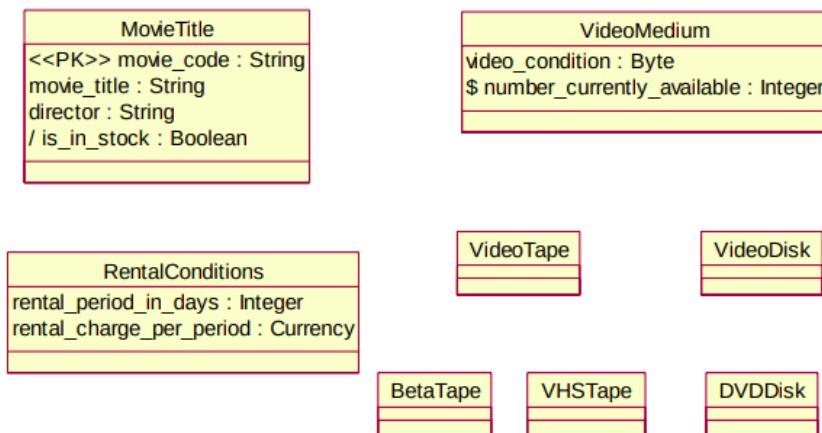
Ci si accorge inoltre che come prefisso di alcuni attributi vi sono <<PK>> Primary Key e <<CK>> Candidate Key dove i simboli <<>> sono chiamati Stereotipi in UML. Una caratteristica importante in UML abbiamo detto essere la possibilità di estensione standard creando un Profilo (es. estendo UML nel settore finanza con il profilo Finanza dove creo primitive UML specifiche per quel dominio). Un profilo rappresenta null'altro che un insieme

di stereotipi.

In questo caso specifico si sta dicendo di voler introdurre un Profilo DBMS (orientato quindi alla traduzione delle classi in tabelle) in cui si annotano gli attributi della classe con caratteristiche tipiche della progettazione database.

Seconda iterazione per VideoStore:

"The rental charge differs depending on video medium: tape or disk (but it is the same for the two categories of tapes: Beta and VHS), The system should accommodate future video storage formats in addition to VHS tapes, Beta tapes and DVD disks, The employees frequently use a movie code, instead of movie title, to identify the movie, The same movie title may have more than one release by different directors".



Chiaramente movie code e title sono attributi della classe movie, e anche director. Capiamo già da come sono posizionate le classi che esiste una sorta di gerarchia. Inoltre il requisito per cui si potrebbero dover introdurre nuove tecnologie per cassette e dvd è soddisfatto introducendo videodisk e videotape come "generalizzazioni". Is_in_stock deriva da un requisito della prima iterazione: il software deve essere utilizzato per rispondere alla query del cliente se un certo film è disponibile o meno. / is_in_stock ha il simbolo / che è standard UML e rappresenta derived attribute: il valore dell'attributo non è inserito dall'utente ma derivato runtime (è il sistema in grado di capire se c'è il film a disposizione o meno).

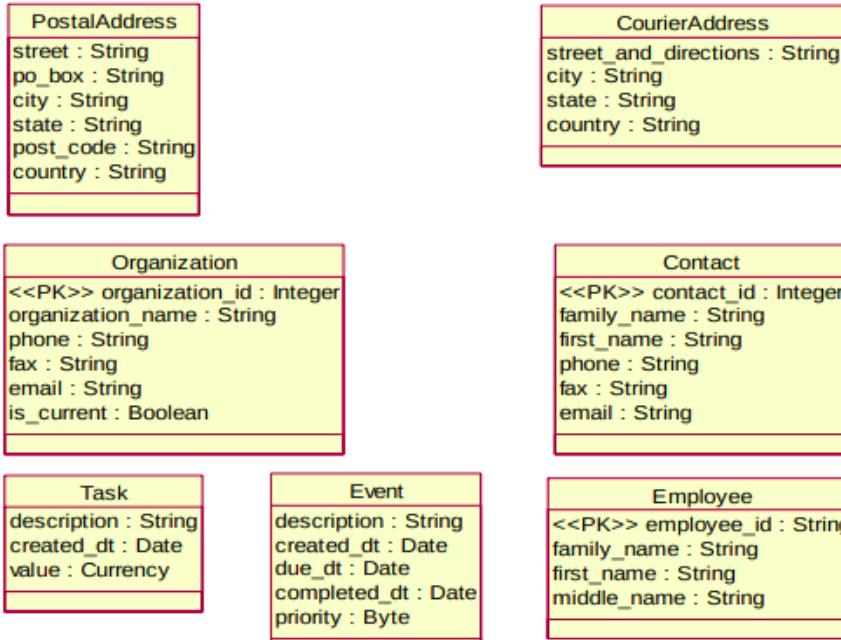
L'altra informazione di cui avevamo bisogno era il numero di VideoMedium disponibili per il noleggio, ciò si applica sia alle cassette che ai dvd quindi la metto nella classe più generica (ereditarietà). \$ number_currently_available NON indica il numero di cassette/dischi che sono disponibili per un certo film. Infatti \$ è un altro simbolo standard UML e rappresenta un attributo statico: singolo valore condiviso tra tutti gli oggetti che creo a partire da quella classe. L'attributo in realtà indica il numero totale di videomedium: incrementato quando ne creo uno e decrementato quando non è più disponibile.

Per soddisfare la richiesta del numero disponibile per uno specifico film devo utilizzare le associazioni tra classe MovieTitle e VideoMedium con is_in_stock e number_currently_available se non ci sono associazioni tra questi due allora is_in_stock sarà a 0.

Tornando all'esempio di Contact Management:

"A customer is considered current if there exists a contract with that customer for delivery of our products or services. Contract management is, however, outside the scope of our system, Reports on contacts based on postal and courier addresses (e.g. find all customers by post code), Date and time of the task creation are recorded, The "money value" of a task

can be stored, Events for the employee are displayed on the employee's screen in the calendar-like pages (one day per page), The priority of each event (low, medium or high) is visually distinguished on the screen, Not all events have a "due time" - some are "untimed", Event creation time cannot be changed, but the due time can, Event completion date and time are recorded, The system stores identifications of employees who created tasks and events, who are scheduled to do the event ("due employee"), and who completed the event" L'ultimo requisito sarà importante per la terza iterazione in quanto si vogliono identificare gli impiegati che hanno creato task e eventi, a cui sono stati assegnati e che li hanno completati.



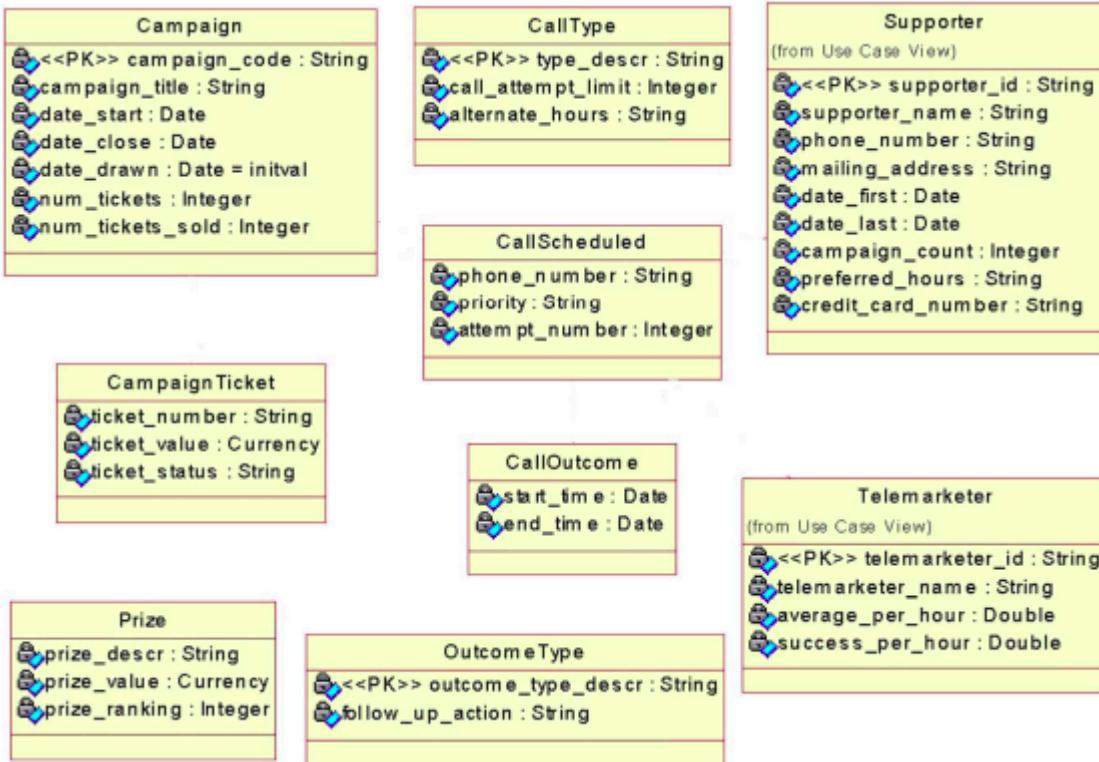
Contract non è classe importante per il mio software, per stabilire se è attuale aggiungo l'attributo `is_current` a `Organization`. Forse conviene utilizzare informazione strutturata per l'indirizzo creando una classe a parte visto che si vogliono cercare i clienti per post code (primo requisito).

`PostalAddress` e `CourierAddress` introdotti come classi a parte, task ed evento hanno attributi `created_dt`, `due_dt` e `completed_dt` di tipo `Date` e `value` per `task` di tipo `Currency` (tipi di dato base in UML). Per definire la priorità degli eventi è presente l'attributo `priority` di tipo `Byte` (si voleva low medium o high, un boolean non bastava quindi uso byte).

La seconda iterazione per il Telemarketing:

"Each campaign: Has a title that is generally used for referring to it, Has also a unique code for internal reference, Runs over a fixed period of time. Soon after the campaign is closed, the prizes are drawn and the holders of winning tickets are advised. Tickets are uniquely numbered within each campaign. The total number of tickets in a campaign, number of tickets sold so far, and the current status of each ticket are known (e.g. available, ordered, paid for, prize winner). To determine the performance of the society's telemarketers, the duration of calls and the successful call outcomes (i.e. resulting in ordered tickets) are recorded. Extensive information about supporters is maintained: Contact details (address, phone number, etc.), Historical details such as the first and most recent dates when a supporter had participated in a campaign, Any known supporter's preferences and constraints (e.g. times not to call, usual credit card number). Telemarketing calls are made

according to their priorities. Calls which are unanswered or where an answering machine was found, are rescheduled: Times of repeat calls are alternated, Number of repeat calls is limited. Limits may be different for different call types (e.g. a normal "solicitation" call may have different limit than a call to remind a supporter of an outstanding payment). Call outcomes are categorized - success (i.e. tickets ordered), no success, call back later, no answer, engaged, answering machine, fax machine, wrong number, disconnected".



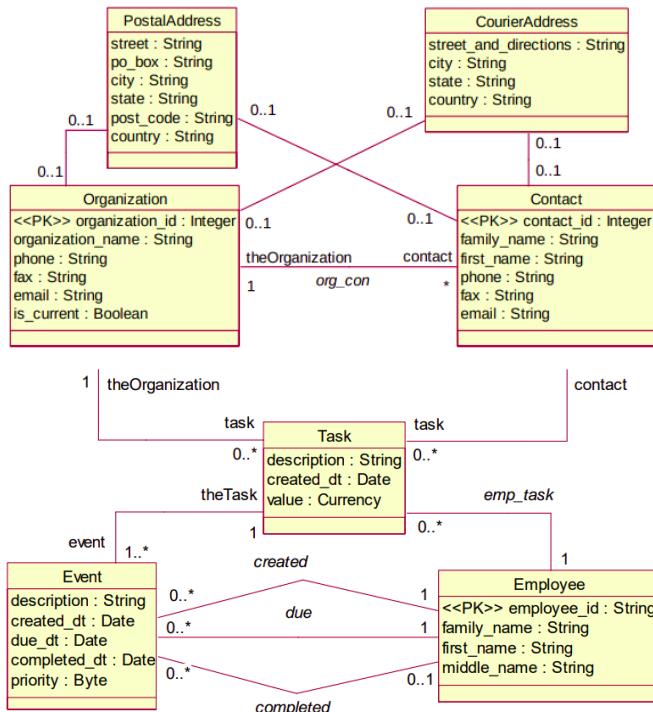
Guardando i nuovi requisiti è conveniente inserire una classe "Prize" per indicare cosa vince chi vince la lotteria. Per lavorare sul terzo requisito devo lavorare sull'associazione tra telemarketer, chiamata da fare e risultato chiamata (lo faccio per ogni chiamata di un certo telemarketer e capisco così le sue prestazioni). average_per_hour e success_per_hour sono gli attributi di telemarketer che ne indicano le prestazioni di cui parlavamo prima, ed il valore di essi dipende dall'associazione che esiste tra Telemarketer, CallScheduled e CallOutcome. Una cosa che probabilmente manca in CallScheduled è la durata della chiamata, fattore che serve per determinare le performance del telemarketer. Un'osservazione è da fare sulle classi CallOutcome e OutcomeType. Si era ipotizzato di inserire OutcomeType come attributo di CallOutcome, invece è stata creata una classe a parte con un occhio di riguardo proprio all'implementazione. Sapendo infatti che all'ora vengono fatte ad esempio 500 chiamate tra tutti i telemarketer, la seconda soluzione permette di utilizzare molta meno memoria. Mettendo gli attributi in una sola classe avrei dovuto memorizzare per ogni CallScheduled 5 attributi, con due classi distinte invece di oggetti OutcomeType non ne creo 500 all'ora, ma solo un numero pari al numero possibile di outcome. Ora ognuna delle 500 CallOutCome sarà associata al rispettivo OutcomeType -> di outcometype ne ho tipo 15, ho risparmiato un sacco.

Con la prima iterazione abbiamo identificato le prime entity classes, con la seconda definito eventualmente ulteriori classi e gli attributi, con la terza definiamo le associazioni per finire la definizione del modello di dati prima di passare al comportamentale. Quando si incontra un

attributo che ha come tipo di dato una classe, quell'attributo avrà come valore un riferimento a un oggetto di quella classe. Quindi in generale un attributo rappresenta un'associazione se sono identificati con tipo di dato non elementare ma classe. Quando si ha un'associazione ternaria questa può essere rimpiazzata con un ciclo di associazioni binarie, rimuovendo una delle associazioni in quanto potrò ad esempio raggiungere la classe A da B in modo transitivo passando da C (risparmio spazio ma perdo in termini di efficienza).

Così come detto per classi e attributi, anche le associazioni devono avere nomi significativi e devono usare ovviamente la stessa convenzione usata per gli attributi. Inoltre è fondamentale esprimere la molteplicità delle associazioni, che come negli schemi ER deve essere espressa ad entrambe le estremità. Infine è importante assegnare i nomi di ruolo (rolename) alle estremità delle associazioni, specificano il ruolo giocato dagli oggetti di ciascuna classe che partecipano a quella associazione.

Si aggiungono le associazioni Partiamo dalla terza iterazione di Contact Management.
Il requisito: "The system allows producing various reports on our contacts based on postal and courier addresses".



Vediamo una delle limitazioni di UML se vogliamo porre il vincolo affinché ogni contatto abbia un solo indirizzo tra CourierAddress e PostalAddress (solo uno dei due). Per risolvere la cosa posso farlo informalmente aggiungendo una nota testuale, altrimenti in modo formale si può utilizzare un linguaggio detto OCL che permette di associare dei vincoli ai diagrammi UML.

Qui vediamo l'esempio di un'associazione completamente specificata: abbiamo infatti nella associazione sotto nome (org_con), specificità e nome di ruolo. L'associazione mette in relazione Contact e Organization e ha le seguenti molteplicità: fissato un Contatto questo lavora in una e una sola organizzazione (uno a sx) mentre fissata l'organizzazione questa è costituita da almeno 1 ma anche più contatti che vi lavorano

Ora vediamo i role_name, che sono i nomi che esprimono il ruolo con cui gli oggetti di una classe partecipano all'associazione e normalmente sono quelli utilizzati per implementare

l'associazione. Per implementarla infatti dovrò inserire nella classe Contatto dovrò mantenere l'informazione relativa all'organizzazione in cui lavora (quindi aggiungerò a Contact l'attributo theOrganization che è di tipo Organization). D'altra parte in Organization potrò mettere un attributo contact che indica l'insieme di oggetti Contact che lavorano nell'organizzazione.

Uno dei primi requisiti diceva che un task è un gruppo di eventi -> un evento fa parte di un task e un task ha 1..* eventi. Per tener traccia dell'impiegato che ha creato il task: relazione emp_task dato un task affidato a un solo impiegato, dato un impiegato può aver creato 0..* task.

Vediamo ora le associazioni che legano la classe impiegato con la classe evento. L'ultimo requisito richiesto in C2 chiedeva che il sistema memorizzasse gli impiegati che creano task e eventi, a cui è assegnato un evento e che completano l'evento. Ciò significa che per ogni evento deve esserci un impiegato che crea quell'evento, uno a cui è assegnato e uno che lo completa (possono essere differenti). Vedendo il modello un Impiegato può aver creato, aver avuto assegnati, aver completato 0..* eventi. Fissano evento avrà 1 impiegato che lo ha creato, 1 impiegato a cui è stato assegnato e 0..1 impiegato che l'ha completato (quindi dato evento sicuro qualcuno l'ha creato e a qualcuno è stato assegnato, ma non è detto che sia stato completato).

(FINE ESEMPIO)

Associazioni: Le associazioni collegano le classi, hanno nomi significativi e devono esprimere la molteplicità ad entrambe le estremità (es. 0..1, * per 1:N). UML presenta limitazioni nel porre vincoli complessi sulle associazioni (es. un solo indirizzo tra due tipi), risolvibili con note testuali o linguaggi formali come OCL.

Tipi Particolari di Associazioni (Contenimento): UML fornisce due primitive per le relazioni di contenimento (whole-part) tra una classe composta (superset, il contenitore) e una o più classi componenti (subset). Questo tipo di relazione può assumere 4 significati in base alla "forza" del legame che esiste tra classe contenitore e classe contenuta:

- Aggregation (\diamond): Rappresenta un contenimento più debole (semantica per riferimento). Corrisponde a relazioni di tipo Has (es. Facoltà has Dipartimento, no fixed dependency né fixed property) e Member (es. Meeting e Coordinatore, non vi sono proprietà speciali di contenimento eccetto la membership), dove non c'è "existence-dependency". Implementata come associazione standard.
- Composition (\blacklozenge): Rappresenta un contenimento più forte (semantica per valore). Corrisponde a relazioni di tipo ExclusiveOwns (transitività (se A contiene B e B contiene C allora A contiene C), (se A contiene B B non può contenere A), Existence-dependency (se cancello l'oggetto contenitore elimino anche i contenuti) e Fixed Property (se un'istanza della classe contenuta è messa in relazione con un'istanza della classe contenitore non può esser messa in relazione con nessun'altra istanza contenitore); es. Book e Chapter) e Owns (transitività, asimmetria, existence-dependency, ma non fixed property; es. Car e Tire). In questo caso se l'oggetto contenitore viene cancellato, anche il contenuto viene eliminato.

Quindi quando un requisito implica il contenimento di una classe in un'altra uso relazioni di associazione di contenimento, da decidere se Aggr. o Composition.

Ereditarietà (Generalizzazione): UML usa il concetto di generalizzazione (freccia dalla sottoclasse alla superclasse) per rappresentare la condivisione di attributi e operazioni tra classi.

- Sostituibilità: Un oggetto della sottoclasse può essere usato al posto di un oggetto della superclasse e non viceversa. (Immaginiamo di avere una classe A specializzata in una sottoclasse B e di creare i rispettivi oggetti a e b. Dire a = b è corretto secondo la sostituibilità in quanto la classe B sa fare le stesse cose che fa A (le eredita) e anche altro in più, mentre b = a non va bene in quanto la classe A non sa fare tutto ciò che fa B).
- Polimorfismo: una stessa operazione può avere implementazioni diverse nelle sottoclassi (una sottoclasse può ridefinire il comportamento di alcuni metodi ereditati in base alle proprie necessità).

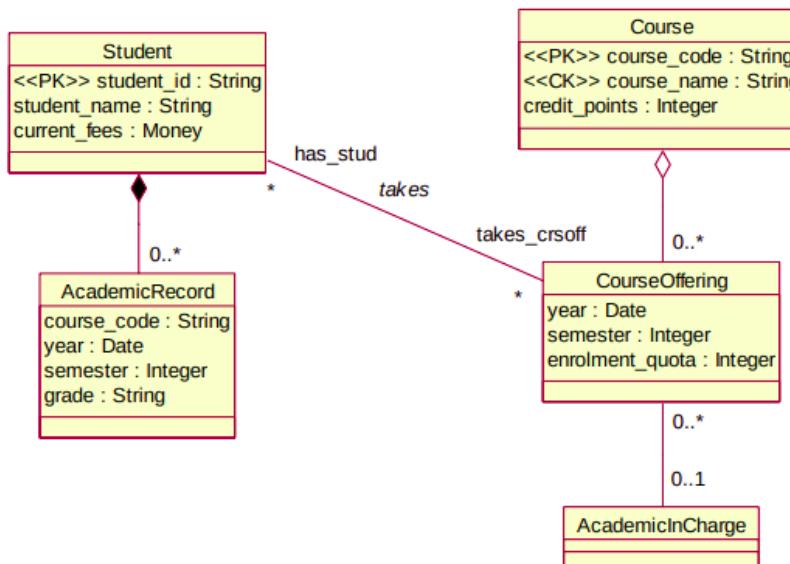
(ES: saltare fino a pag 55) Date A e B classi dove B Composition di A allora quando creo i due oggetti da A e B l'oggetto creato da A avrà al proprio interno l'oggetto creato da B, e se cancello l'oggetto A cancello anche l'oggetto B (existence dependency) (semantica per valore in quanto l'oggetto contenitore contiene l'oggetto contenuto tra gli attributi).

Stesso scenario ma A e B legate da Aggregation: semantica per riferimento significa che l'oggetto A avrà tra gli attributi un riferimento all'oggetto B, che vive nel suo spazio di memoria (quindi se cancello l'oggetto A cancello il puntatore ma non B).

Dal punto di vista implementativo quindi un'associazione di tipo aggregation si implementa come associazione standard tra due classi.

Terza Iterazione per Università:

"The student's academic record to be available on demand. The record to include information about the student's grades in each course that the student enrolled in (and has not withdrawn without penalty). Each course has one academic in charge of a course, but additional academics may also teach in it: There may be a different academic in charge of a course each semester, There may be different academics for each course each semester"



AcademicRecord come classe per registrare la carriera universitaria dello studente dove ogni volta che lo studente supera l'esame ne creo un nuovo oggetto. Fissato lo studente, questo può fare da 0 a N esami (0..*). Takes è l'associazione tra Student e i corsi che sta seguendo in quel momento (* sta per molteplicità 1:N).

AcademicInCharge sta per docente, fissato il docente questo può insegnare 0..* corsi "offerti", fissato il corso questo (secondo l'ultimo requisito sopra) in realtà dovrebbe poter essere insegnato anche da più docenti e sicuramente da almeno 1, quindi non 0..1 ma 1..*.

Analizziamo ora l'uso di Composition e Aggregation.

La composition a sinistra indica un legame molto forte tra gli oggetti studente e gli oggetti AcademicRecord, se si cancella dall'archivio uno studente si cancellano anche i dati relativi agli esami da lui sostenuti (dichiarazione forte, ma non è errore). Riguardo l'aggregation a destra, ci sta dicendo che se cancello ad es. il corso ISW allora non saranno cancellate tutte le istanze del corso negli anni passati (corretto, se si cancellassero si perderebbero formazioni importanti legate anche agli studenti che hanno seguito e sostenuto l'esame).

Per rappresentare la generalizzazione in UML si usa una linea avente freccia diretta verso la superclasse. Capiamo di doverla usare quando nei requisiti leggiamo tipo "può essere" o "è del tipo di"... Generalizzazioni piuttosto chiare, relazione tra MovieTitle e VideoMedium available permette di capire quanti e quali supporti video ho legati a un film specifico, si associa in particolare un oggetto MovieTitle a 0..* oggetti VideoMedium utilizzabili per noleggiarlo.

Se 0 il film non è disponibile per il noleggio -> attributo di MovieTitle is_in_stock diventa falso.

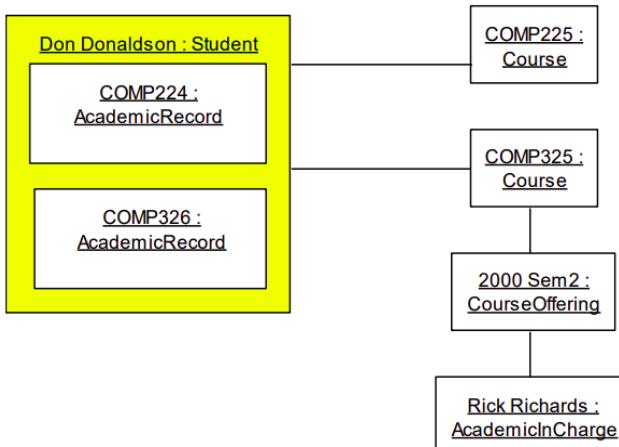
RentalConditions, per ogni VideoMedium che viene noleggiato si applica una specifica RentalCondition, mentre fissato un oggetto RentalCondition questa può essere associata a uno o più VideoMedium (più perché più clienti possono scegliere la stessa). Si noti come il nome della classe (Esempio B3) VideoMedium, VideoTape, VideoDisk siano in corsivo. Ciò non è casuale, si indicano delle Classi Astratte (ossia non si possono creare oggetti a partire da quelle classi, posso creare solo in questo caso BetaTape, VHSTape e DVDDisk unici oggetti effettivamente concreti).

Perché allora non utilizzarle direttamente? Se l'avessi fatto avrei dovuto però specificare l'associazione available con MovieTitle e apply con RentalConditions per ognuna delle tre classi, sarebbe quindi stato corretto sintatticamente ma non avrebbe espresso il significato dietro la generalizzazione per cui quelle associazioni valgono per tutti. Con questa terza iterazione abbiamo concluso la parte relativa alle associazioni e in generale il primo step sul Modello dei Dati.

Passiamo al Modello Comportamentale, vedremo che lavorare su di esso ci permetterà di tornare al Class Diagram e aggiungere anche le operazioni. Prima di far ciò vediamo un altro diagramma UML che può essere utile nel Modello dei Dati come complemento del Class Diagram: l'Object Diagram.

Esso è per la rappresentazione grafica di istanze di classi ed è usato:

- per mostrare esplicitamente relazioni complesse tra classi (es. sopra mostrare la relazione available direttamente tra oggetto MovieTitle e DVDDisk),
- illustrare le modifiche ai singoli oggetti durante l'evoluzione del sistema (ogni oggetto creato da una classe ha uno stato definito dai valori dei suoi attributi, che possono cambiare nel tempo),
- illustrare la collaborazione tra oggetti durante l'evoluzione del sistema.



Lo studente Don Donaldson ha superato due esami: COMP225 e COMP326, il fatto che esistesse la relazione di Composition viene esplicitato dal fatto che gli AcademicRecord sono contenuti in Student (se cancello lo studente cancello anche gli esami che ha fatto). Dall'altra parte l'associazione tra l'oggetto corso e courseoffering è rappresentata normalmente (semantica di riferimento come visto).

In realtà questo esempio sarebbe sbagliato in riferimento all'esempio A.3 in quanto in tale diagramma non vi era relazione diretta tra studente e corso, ma solo con CourseOffering. Si ricorda che stiamo usando una specifica semiformale quindi vogliamo costruire un modello del software che lo rappresenti da tutti i punti di vista: statico (dati), funzionale (comportamentale) e di controllo (dinamico).

(FINE ESEMPIO)

Il modello comportamentale rappresenta gli aspetti funzionali del sistema, evidenziando come gli oggetti collaborano ed interagiscono per offrire i servizi del software.

Poiché stiamo costruendo il modello dal punto di vista object oriented, alla fine il software sarà popolato da un certo numero di oggetti che collaborano scambiandosi messaggi (messaggio == metodo che chiede a un altro oggetto di eseguire un metodo).

Utilizza i Diagrammi di:

- Diagramma di Casi d'Uso (per descrivere i possibili scenari di funzionamento)
- Activity Diagram (per descrivere il flusso di elaborazione)
- Sequence Diagram (per descrivere l'interazione tra oggetti)
- Collaboration Diagram (per descrivere l'interazione tra oggetti)

Ultimi due anche detti diagrammi d'interazione. Ancora il modello è costruito in modo incrementale e iterativo, si sfruttano di volta in volta le informazioni del modello dei dati che a sua volta fa uso del modello comportamentale per aggiungere altre classi (inizialmente ci siamo soffermati sulle entity classes, arriveranno le control (per esecuzione) e boundary (per interfaccia) classes).

Iniziamo dall'Use Case Diagram.

Durante OOA si concentra su COSA il sistema deve fare. Un caso d'uso rappresenta:

- una funzionalità completa (dato uno scenario che ha un flusso principale e possibili flussi alternativi non deve identificare molteplici casi d'uso, ma uno solo che rappresenti quindi un singolo scenario di funzionamento)
- una funzionalità visibile dall'esterno (fa riferimento a un possibile uso di quello scenario da parte dell'utente)

- comportamento ortogonale (ogni caso d'uso deve essere indipendente, ciò segue direttamente dal fatto che ogni caso d'uso deve rappresentare funzionalità completa)
- una funzionalità originata da un attore del sistema (si deve avere almeno un attore che attiva il caso d'uso, se nessun attore lo attiva allora non è caso d'uso)
- una funzionalità che produce un risultato significativo per un attore.

I casi d'uso vengono identificati dall'insieme di requisiti utente e soffermandosi sugli utenti principali dei casi d'uso: gli attori (e le loro necessità).

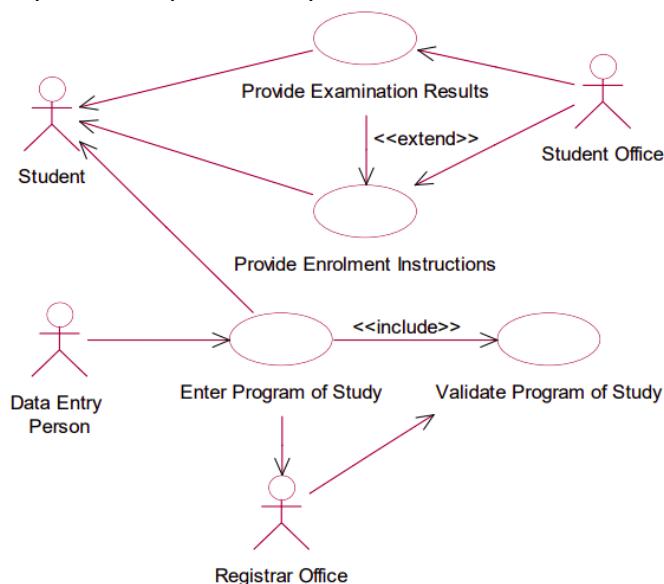
La sintassi fornita da UML prevede graficamente come l'attore (omino) e caso d'uso (ellisse). Tali elementi sono collegati tra loro tramite relazioni, quattro tipi:

- Associazione: Lega attore e caso d'uso (con archi orientati che indicano attivazione o coinvolgimento) (se da attore a caso d'uso l'attore l'ha attivato, altrimenti attore coinvolto nello svolgimento del caso d'uso)
- Include ed Extend: si identificano tra due casi d'uso, devo specificare se si tratta di Stereotipo Extend o Include.

Se A <<include>> B, allora se un attore attiva A affinché A venga completato dovrà necessariamente attivare (da un attore differente o lo stesso) e completare B.

Se A <<extend>> B, allora quando un attore attiva B potrei (ma non necessariamente!) attivare (da un attore differente o lo stesso) anche A.

- Generalizzazione: Relazione tra attori, dove un sotto-attore può attivare tutti i casi d'uso del super-attore più i suoi specifici.

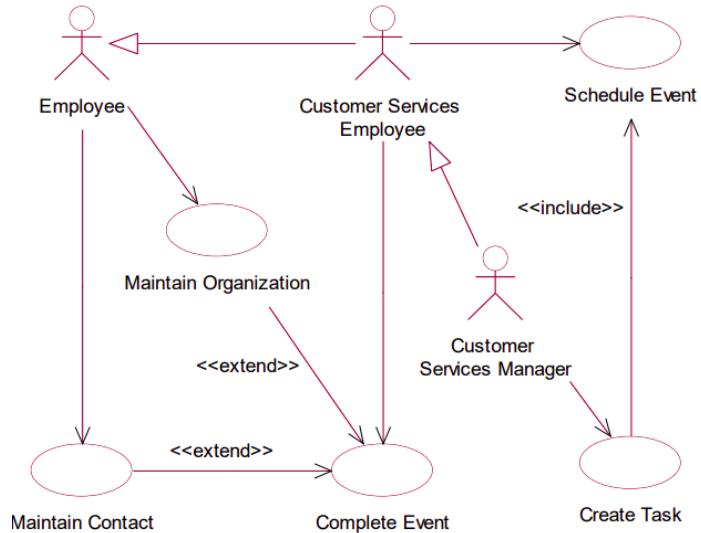


(ES: saltare fino a pag 59)

Abbiamo detto che il software per l'università deve prima dell'iscrizione dello studente inviargli le istruzioni per iscriversi insieme ai risultati dell'ultimo semestre. Durante l'iscrizione lo studente consegna il piano di studi e la tizia in segreteria lo inserisce nel software che lo convalida. Questi scenari di funzionamento sono descritti dai quattro casi d'uso presenti in figura. Gli attori sono gli impiegati della segreteria studenti (student office) che manda le istruzioni insieme ai voti, impiegati segreteria didattica per i corsi di studio specifici (registrar office) che riceve e i piani di studio e li convalida, data entry person che fa parte della segreteria e inserisce i dati dello studente nel sistema ed infine lo studente in sé.

I primi tre sono attori che attivano casi d'uso, lo studente è attore coinvolto dall'esecuzione di

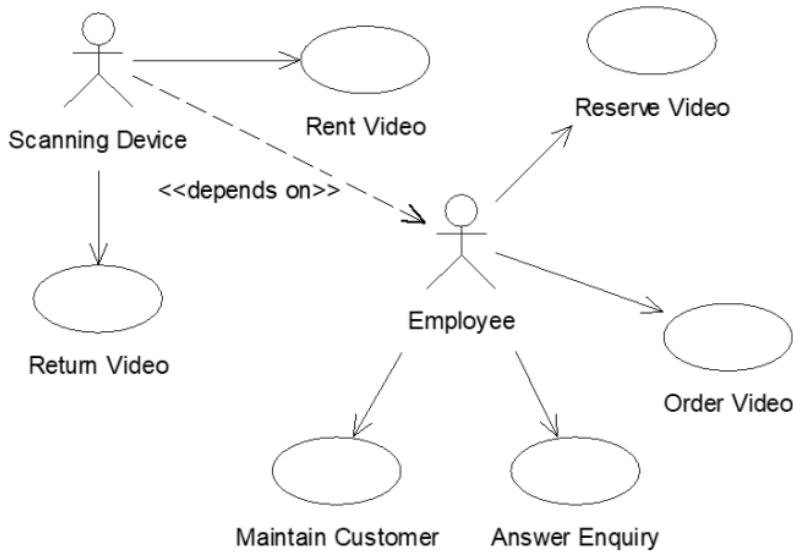
alcuni casi d'uso (es. provide examination results verranno mandati per email allo studente). Vediamo ora extend e include. Enter program study include validate program study significa che per completare l'enter devo completare il validate, starebbe a significare che non posso inserire un piano di studi e lasciarlo là senza sapere se è valido o meno, corretto. Provide Examination Result extend provide enrolment instructions, ossia se mando il secondo potrei anche inviare i dati legati ai risultati dell'ultimo semestre (ma non necessariamente). Corretto in quanto se uno studente si iscrive al primo anno sicuramente riceve le istruzioni sul come iscriversi ma non ha risultati dello scorso semestre quindi il primo caso d'uso non si attiva, al contrario potrebbe attivarsi se si parla di uno studente che si iscrive dal secondo anno in poi.



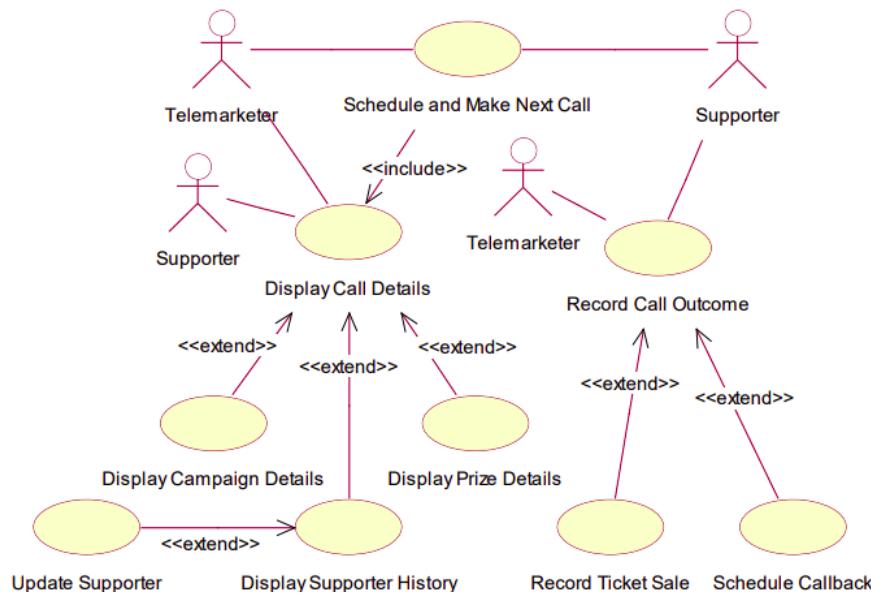
Ricordiamo che il software di contact management aveva l'obiettivo di gestire i contatti dell'azienda che si occupa di ricerca di mercato. Gestire un contatto nel software significava creare task relativi ad attività di contact management, ogni task presenta diversi eventi. Quindi casi d'uso Create Task, Schedule Event, Complete Event, Mantain Organization e Maintain Contact (le ultime due tutte le operazioni di base (CRUD) che possono essere fatte sia sulle operazioni che sui contatti). Create Task include Schedule Event, quindi quando attivo il primo deve essere attivato e portato a termine anche il secondo (corretto perché quando creo un task questo è formato da eventi che devono essere schedulati). Maintain Contact e Maintain Organization sono in extend con Complete Event, anche questo ha senso perché quando attivo Complete Event (mentre sto completando un evento) potrei aver necessità di gestire le informazioni di una contact person o di un'organizzazione a cui sto facendo riferimento. Riguardo gli attori, questo software poteva essere usato da tutti gli impiegati con accessibilità diversa in base al grado. Si osserva quindi come l'impiegato generico possa attivare i casi d'uso Mantain Contact e Organization per recuperare le informazioni basi di organizzazioni/contact person. Poi l'attore Customer Services Employee (già più specifico) che può schedulare gli eventi e completarli. Vi è infine un ultimo attore, il Customer Services Manager, l'unico che può creare task.

Generalizzazioni: l'impiegato può essere specializzato in un impiegato delle risorse clienti, che a sua volta può essere manager del reparto. Gli attori più specifici in questa gerarchia possono fare quanto fanno gli attori generici più alcune cose specifiche che fa in quanto specializzato (da qui l'attivazione di certi casi d'uso), quindi un superattore come CSM può non solo creare task, ma anche schedulare eventi e accedere a informazioni di organizzazioni e contact person (ereditarietà). Similmente a quanto visto nel modello dei dati

anche qui avrei potuto evitare la generalizzazione collegando tutto e facendo un macello, ma avrei mancato di semantica per capire le relazioni tra attori coinvolti nei casi d'uso.



Tornando ora al caso di studio VideoStore, l'attore principale è sicuramente Employee che usa il software per gestire tutto, ma vediamo un secondo attore Scanning Device che è collegato all'impiegato con una relazione <<depends on>> (anche questa associazione generale, tratteggiata in realtà e non continua come si vede). Tale relazione indica che la pistola per scannerizzare è "dipendente" dall'impiegato, necessita l'impiegato perché si attivino e completino i casi d'uso associati ad essa, nb non è generalizzazione! Quindi è come se fosse l'impiegato ad essere associato a Rent e Return Video, ma serve lo strumento/attore di scanning per farlo.



Prima di vedere l'approccio formale vediamo per completezza il diagramma casi d'uso del Telemarketing. Schedule and Make Next Call e Record Call Outcome due casi d'uso fondamentali che fa il Telemarketer. Display Call Details attivato a fronte di una telefonata perché il telemarketer sappia chi sta chiamando e perché. Diversi casi d'uso in extend: per quel che riguarda i risultati telefonata Record Ticket Sale (se il cliente acquista biglietti) e Schedule Callback (se si vuole ripianificare la chiamata al supporter). Per quanto riguarda

invece i dettagli chiamata potrei necessitare dei dettagli campagna, dettagli premi e dettagli storia supporter e eventualmente aggiornare dati supporter. Tutti questi extend sono chiaramente convincenti. Esiste infine un include tra schedule and make next call e display call details anch'esso convincente, infatti quando faccio una nuova chiamata sono necessari i dettagli di chi chiamo e il perché. Un problema nel diagramma tuttavia è che a questo livello di specifica bisogna sempre specificare archi orientati tra attori e casi d'uso, e ciò qui non avviene. Si osserva poi (non è un errore) che vi sono attori duplicati. Altro importante errore è che abbiamo detto definendo il concetto di caso d'uso che ogni caso d'uso deve avere un attore che lo attiva e in questo caso diversi sono lasciati a se stessi, è chiaro come questi dovranno essere attivati dal telemarketer. Risolvo il problema o utilizzando associazioni esplicite o aggiungendo una nota al diagramma.

(FINE ESEMPIO)

I diagramma di casi d'uso in sé non forniscono informazioni complete dal punto di vista comportamentale. Ciò che dobbiamo fare sarà quindi estendere la descrizione specificando i dettagli dietro ciascun caso d'uso.

Due approcci per far ciò:

- Informale per cui in un template inserisco i dettagli in linguaggio naturale per specificare il caso d'uso,
- Formale che fa uso del Diagramma delle Attività (UML) che fa capire esattamente cosa fa il software quando si attiva il caso d'uso.

Vediamo ora come funziona ciò focalizzandoci sul singolo caso d'uso Rent Video.

Con l'approccio Informale:

Brief Description: dove per ogni caso d'uso viene data una descrizione che dica ciò che succede.

Attori.

Precondizioni: condizioni che devono essere vere per abilitare la funzione sul software.

Main Flow: flusso di attività principale legato al caso d'uso.

Alternative Flows: si specificano eventuali flussi alternativi.

Postconditions: condizioni vere dopo che il caso d'uso è stato completato con successo.

Ora passiamo all'approccio formale, per ogni caso d'uso dovrei definire il Diagramma delle Attività.

Il diagramma delle attività ha grande potere espressivo, rappresentando flussi di esecuzione sequenziali e concorrenti, utilizzabile a diversi livelli di astrazione (OOA e OOD).

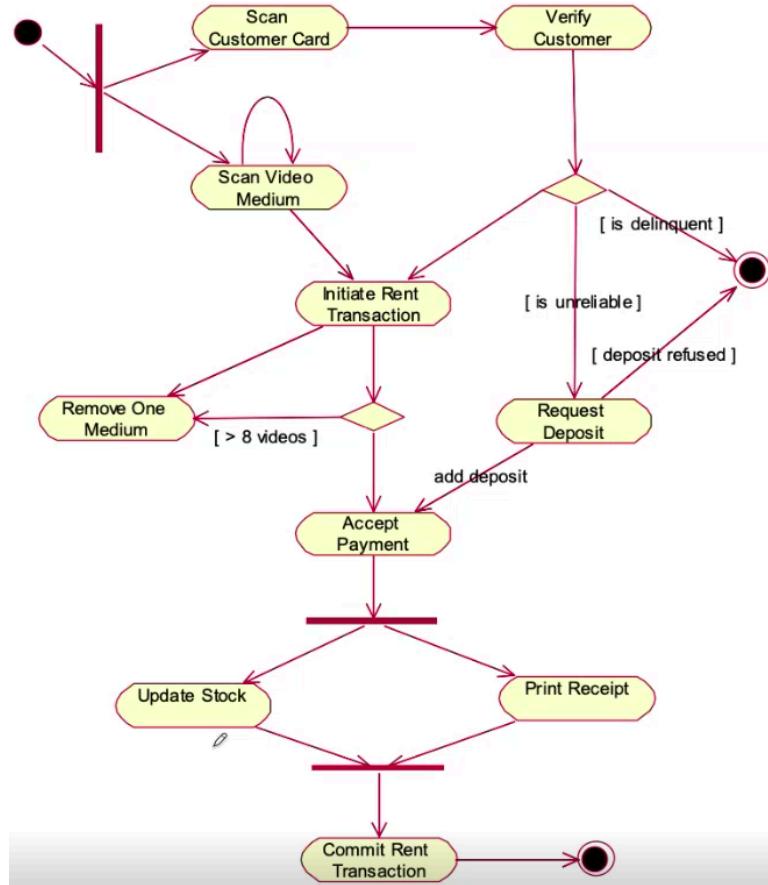
Esso rappresenta in UML una variante degli state diagram, dove i nodi rappresentano le attività mentre le transizioni rappresentano il completamento di un'attività e il passaggio alla successiva.

In realtà questo diagramma si utilizza principalmente in fase di OOD per rappresentare un algoritmo, ma in fase di OOA si usa per rappresentare il flusso di attività nell'esecuzione di un singolo caso d'uso.

Le primitive fornite da UML: si parte da un nodo iniziale che rappresenta l'evento d'inizio, che corrisponde all'attivazione del caso d'uso. Poi le varie attività da svolgere (altri nodi), ognuno di questi nodi sono collegati da transizioni che possono essere associate a specifiche condizioni, si passa a una transizione in uscita se l'attività è stata completata. Per rappresentare flussi concorrenti si sfruttano delle barre di sincronizzazione dette fork-join (simili a quelle viste nelle reti di Petri) mentre per i flussi alternativi si sfruttano nodi

decisionali (branch/merge diamonds, rombi). Importante l'utilizzo di questi diagrammi in quanto mentre con il linguaggio naturale è possibile ambiguità qui l'interpretazione è una.

Descriviamo l'esempio sotto.

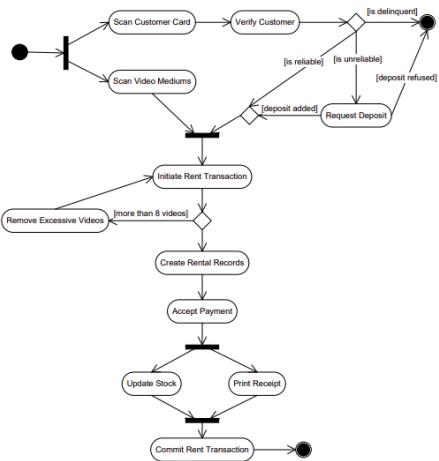


(ES: saltare fino a pag 61)

Nodo iniziale cerchietto pieno (corrisponde all'attore che attiva il caso d'uso), finale cerchietto pieno cerchiato. Troviamo subito una barra che è di fork (in quanto il numero di archi in uscita è maggiore degli entranti) che quindi ci dice che vi sono due flussi in parallelo, il primo scannerizza i video medium noleggiati dal cliente e da qui due possibili uscite, la prima self transition (se il cliente arriva con 5 video devo scannerizzarli tutti) e la seconda mi porta all'attività successiva. La seconda attività sopra è scannerizzare la carta del cliente per poi eseguire l'attività verifica cliente (se è delinquente o meno). Dopo verify customer c'è un nodo decisionale: cliente affidabile allora inizio la transazione di prestito, se delinquente arrivo al nodo finale e finisce la storia, se inaffidabile richiedo il deposito, se rifiutato allora arrivo allo stato finale, se accettato allora lo faccio pagare. NB ogni arco uscente da un rombo (flusso alternativo) deve specificare una condizione, al limite posso lasciarne uno solo vuoto in caso non si verifichino le altre due condizioni (es. in questo caso cliente affidabile o ≤ 8 videos). Tornando al cliente affidabile dopo aver inizializzato la transazione devo verificare che il numero in noleggio sia ≤ 8 , se così non è allora ne rimuovo finché e torno a Initialize Rent Transaction finché la condizione è verificata e scendo a Accept Payment. A questo punto di nuovo fork: il software aggiorna lo stock e allo stesso tempo stampa la ricevuta. Dopodiché join: ciò significa che anche se l'update stock fa prima della stampa ricevuta prima di passare alla fase successiva si deve aspettare. C'è infine la conferma della transazione che porta allo stato finale.

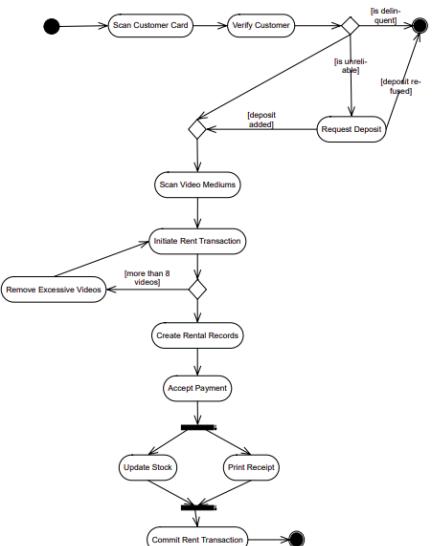
Il diagramma d'esempio presenta vari problemi sia sintattici che semantici: chiaramente la freccia del ciclo deve andare da Remove One Medium a Initialize Rent Transaction e non viceversa. Un altro problema è che se il cliente è considerato indaffidabile non effettuo il controllo sul numero di video, altro problema sta in Scan Video Medium: non esistono condizioni di guardia che mi dicono quando devo ciclare e quando devo passare all'attività successiva. Normalmente per rappresentare il ciclo sul singolo nodo mi serve un nodo di branch (rombo che torna all'attività se devo riscannerizzare altrimenti prosegue), in UML posso rappresentare ciò con la forma vista sopra ma devo necessariamente in tal caso anche esplicitare la condizione di guardia che mi permetta di capire come comportarmi. Ulteriore problema sta nel fatto che Initialize Rent Transaction può essere attivata due volte: sia quando il flusso arriva da Scan Video Medium che quando arriva dal branch post Verify Customer per cui il cliente è affidabile. Serviva sfruttare una join affinché l'attività avvenisse solo quando entrambi erano conclusi.

Da questi ragionamenti si deriva la versione corretta:



Si noti la differenza tra merge e join: uso merge con il rombo quando o vale una condizione o vale l'altra, la join invece con la barra se devo aspettare entrambi i flussi.

Questa versione è sicuramente corretta ma potrebbe essere ancora migliorata, es. il fatto di definire le attività concorrenti iniziali di scan non è conveniente visto che l'impiegato deve scannerizzarle una alla volta.



(FINE ESEMPIO)

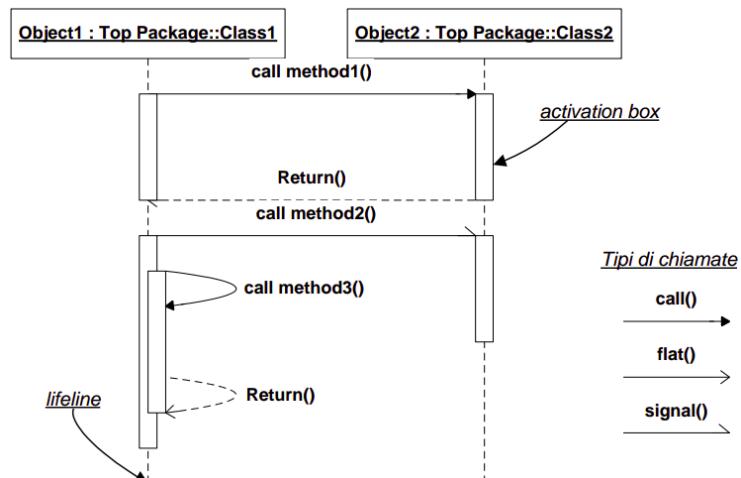
Stiamo utilizzando un linguaggio Object Oriented, tutte le attività presenti nei nodi del diagramma di attività saranno realizzate concretamente al tempo di esecuzione dall'interazione (scambio di messaggi ossia richiesta di far partire metodi) tra oggetti creati dalle classi. Quindi a questo punto ho necessità dell'utilizzo del Modello dei Dati per andare avanti.

Si definiscono a questo punto i Diagrammi di Interazione.

- Sequence Diagram: usato principalmente in fase di specifica (OOA) mostra in modo esplicito le interazioni tra oggetti, che avvengono tramite uno scambio di messaggi (di tipo "richiesta esecuzione attività") e l'ordine temporale. In particolare le attività possono essere mappate come messaggi.

I messaggi possono essere di due tipi:

- Signal (asincrona): rappresenta il fatto che l'oggetto mittente continua l'esecuzione dopo aver inviato il messaggio asincrono.
 - Call (sincrona): rappresenta una richiesta di tipo send-reply in quanto l'oggetto mittente blocca l'esecuzione dopo aver inviato il messaggio in attesa di risposta da parte dell'oggetto destinatario.
 - Flat: se ancora in fase di specifica per certe interazioni non si è scelto in modo definitivo
- Dal punto di vista visuale la notazione è la seguente:

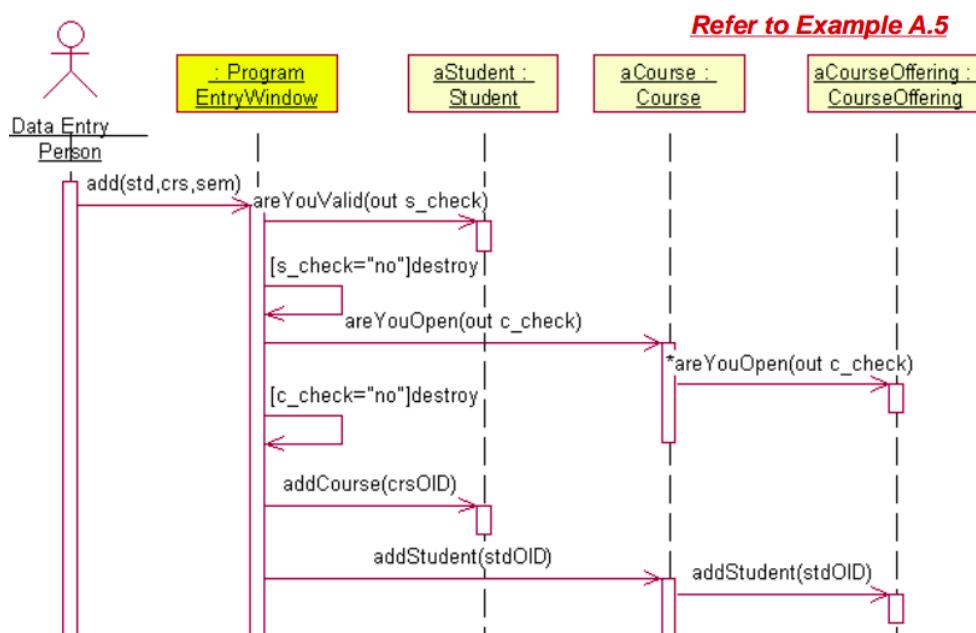


I rettangoli (orizzontali) rappresentano gli oggetti che interagiscono, per ognuno devo dire nome e classe che lo genera. Per ogni oggetto si identifica la lifeline (corso di vita dell'oggetto) dove l'oggetto scambia informazioni con altri, e questo scambio si rappresenta con archi orientati. Quando un oggetto è attivo (in esecuzione/manda messaggi etc..) allora ciò è specificato tramite gli activation box (rettangoli verticali). Importante il fatto che un activation box più piccolo NON implica il fatto che questo rappresenti un arco temporale minore poiché in UML non è definito il concetto di tempo (infatti esistono profili separati per estendere UML con caratteristiche temporali). Con il sequence diagram mi limito a conoscere le sequenze temporali.

- Collaboration Diagram: usato principalmente in fase di progettazione (OOD) e descrive lo scambio di messaggi tra oggetti mediante relazioni tra gli oggetti stessi.

Si tratta di rappresentazioni equivalenti e possono essere generati automaticamente l'uno dall'altro. Esiste la differenza perché il primo rende esplicito l'ordine temporale, mentre il secondo si focalizza proprio sui messaggi che si scambiano gli oggetti. Questi diagrammi sono importanti perché è grazie a loro che siamo in grado di identificare le operazioni delle classi nel class diagram (ed eventuali classi aggiuntive).

(ES: saltare) Nell'esempio precedente es. call method1() significa che l'oggetto1 chiede a oggetto2 di eseguire metodo1, ciò viene fatto tramite call() freccia piena, quindi attende la risposta e non fa niente nell'attesa. Il secondo oggetto invia la risposta Return() al primo tramite signal (freccia con aletta a metà) per cui continua l'esecuzione dopo aver risposto, il primo oggetto allora riprende l'esecuzione. Poi il primo manda una signal method2() per cui non si ferma e come successiva esecuzione call method3() call su se stesso.



Gli oggetti coinvolti dal sequence diagram sono creati dalle classi che conosciamo Student, Course e CourseOffering. Viene anche creato un altro oggetto Program EntryWindow da una classe senza nome che non abbiamo considerato in quanto si tratta di una classe boundary, ossia classi che non gestiscono informazioni ma che si occupano di implementare l'interfaccia tra software e utente. Infine si ha l'attore che scatena la sequenza di interazioni tra oggetti. Si descrive ciò che avviene quando dopo che lo studente ha consegnato il piano di studi il data entry person deve inserirlo nel sistema. L'attore chiede al software tramite l'interfaccia di aggiungere un certo studente std a un certo corso crs in un certo semestre sem. Per rispondere alla richiesta l'oggetto di interfaccia interagirà con gli altri oggetti entity che conosciamo. Per prima cosa si verifica che lo studente sia correttamente iscritto (valid), poi se il corso accetta ancora studenti. Se le cose vanno bene aggiungerò lo studente. Anzitutto l'oggetto interfaccia (boundary) chiede all'oggetto studente se è valido, quindi chiamata al metodo `areYouValid` con parametro di output `s_check` (lo passo vuoto, mi aspetto risposta `s_check` booleana). If `s_check` no allora boundary fa una call a se stesso "`destroy`" di modo che l'attore capisca di non poter inserire i dati (ovviamente apparirà un messaggio a schermo), else boundary chiede all'oggetto corso se è ancora aperto (`out c_check`) (condizione di guardia!). Ora l'oggetto corso non mantiene le informazioni di tutte le istanze del corso quindi a sua volta chiede all'istanza del corso per quel semestre

CourseOffering se è o meno disponibile a prendere altri studenti inoltrando la richiesta iniziale. If c_check no then destroy, else boundary fa aggiungere il corso tra quelli seguiti dallo studente tramite interazione con Studente e aggiunge lo studente al corso tramite interazione con Course che a sua volta inoltrerà la richiesta a CourseOffering. NB qui si usa UML1, molto limitato. In realtà le condizioni di guardia qui presenti (es if s_check no .. else..) si dovrebbero usare nei diagrammi di attività e non nei sequence diagram, andrebbe quindi creato un sequence diagram per ogni possibile interazione. Ciò è superato in UML2 grazie all'introduzione di primitive apposite. Ciò che comunque a noi interessa guardando questo sequence diagram è che possiamo estrarre informazioni molto utili per raffinare il class diagram in termini di operazioni. Infatti ad es. vediamo come boundary chieda all'oggetto studente di eseguire il metodo areYouValid(), quindi Student deve mettere a disposizione quel metodo. Secondo questa logica capisco meccanicamente a partire dal sequence diagram quali metodi devo assegnare alle rispettive classi.

(FINE ESEMPIO)

Torniamo quindi al modello dei dati per aggiungere nuove operazioni ed anche eventuali nuove classi (come le boundary), dall'insieme di operazioni ci sarà quindi possibile definire l'Interfaccia Pubblica di Classe.

Un concetto importante nella programmazione a oggetti è l'information hiding per cui si vuole far vedere all'esterno solo ciò che è necessario per gli oggetti conoscere, nascondendo quanto non è necessario.

In generale si vuole che, piuttosto che rendere immediatamente accessibile agli oggetti gli attributi di una classe (e quindi la possibilità per lui di modificarli direttamente), si definisce un'interfaccia di classe attraverso degli accessor method (metodi di accesso, i getter e setter) che permettono di recuperare e aggiornare i valori degli attributi di una certa classe (es. definisco solo getter per attributi che non voglio vengano modificati).

Quindi l'Interfaccia Pubblica di Classe definisce l'insieme di operazioni che la classe mette a disposizione delle altre classi.

Durante la fase di OOA (dove ci si concentra sul COSA), si determina solo la signature dell'operazione (dichiarazione di funzione): nome, lista di parametri e il tipo di ritorno.

Sarà in fase di OOD che si definirà l'algoritmo che implementa l'operazione. Un'operazione può avere:

- Instance Scope se usata su singolo oggetto,
- Class (static) scope se opera su attributi statici (condivisi da tutti gli oggetti creati a partire da quella classe), preceduta da \$.

Per identificare le operazioni si parte dal sequence diagram e banalmente ogni messaggio inviato ad un oggetto identifica un metodo della classe a cui appartiene tale oggetto.

Inoltre ad ogni classe vanno aggiunte tutte quelle operazioni che appartengono al criterio CRUD, ossia ogni oggetto deve supportare operazioni di create (crea nuova istanza, almeno un costruttore), read (stato di un oggetto, getter), update (lo stato di un oggetto, setter), delete (l'oggetto stesso, serve a sistemare anche quanto fatto dall'oggetto es. prima di eliminarlo chiudo i file etc...).

(ES: saltare)

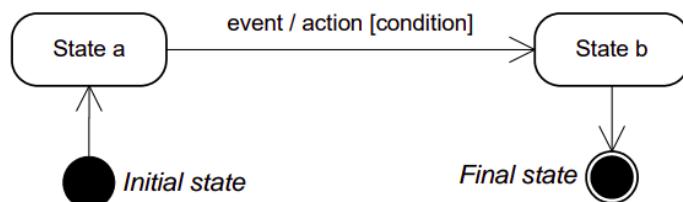
Vediamo l'applicazione da A.6 sequence diagram per l'università all'aggiornamento del diagramma di dati. L'unica cosa da aggiungere qua sotto era addStudent come metodo

anche di Course Oltre che costruire i modelli comportamentale e di dati tuttavia, è necessario costruire anche il modello dinamico affinché la specifica con approccio semiformale mi permetta di costruire un modello completo.

(FINE ESEMPIO)

Modello Dinamico (State Diagrams)

Rappresenta il comportamento dinamico (evoluzione) degli oggetti di una classe in termini di stati possibili, eventi e condizioni che originano transizioni di stato. Molto simile agli Activity Diagram, ma Activity Diagram è un caso particolare di State Diagram.

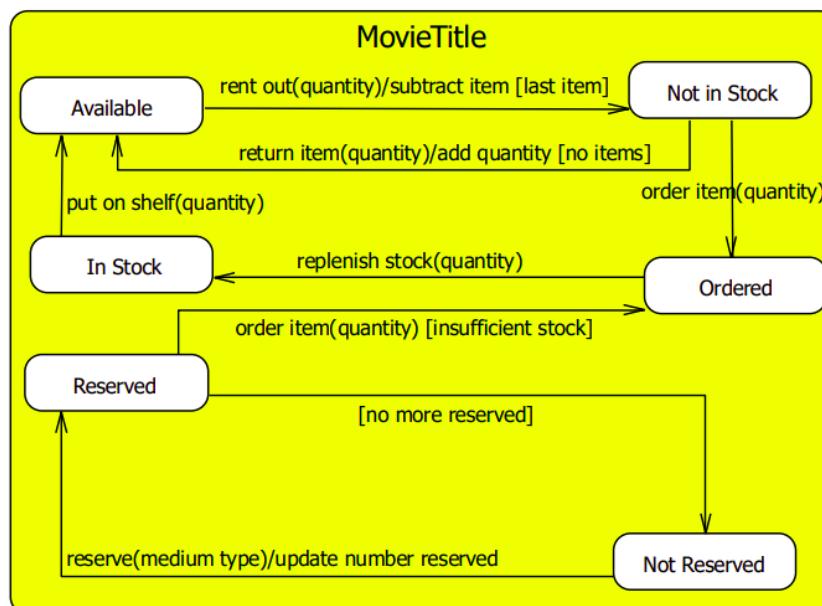


Normalmente il Modello dinamico viene costruito solo per le cose per cui è interessante descriverne il comportamento dinamico, come le classi di controllo (finora abbiamo visto entity classes che gestiscono l'accesso ai dati e boundary classes per l'interfaccia utente, le control classes gestiscono la logica dell'applicazione).

Principalmente quindi questo modello è usato per software per cui è importante conoscere l'evoluzione, come applicazioni real-time e scientifiche (mentre è meno frequente nello sviluppo di applicazioni gestionali).

(ES: saltare)

Vediamo soltanto un esempio visto che nei nostri casi di studio abbiamo visto software di stampo gestionale per capire come viene applicato il diagramma degli stati.



Si vuole quindi rappresentare l'evoluzione dei possibili stati in cui si può trovare l'oggetto creato dalla classe MovieTitle. Qui manca in realtà nodo iniziale e finale, è facile immaginare l'iniziale collegato a Available (creo un oggetto film perché è arrivato e disponibile in magazzino). L'unico arco in uscita da Available porta a Not In Stock con transizione avente notazione completa in quanto ha evento, azione e condizione. Quindi affinché si passi da

Available a Not In Stock è necessario che sia noleggiato l'ultimo elemento disponibile, quando ciò accade esso viene sottratto.

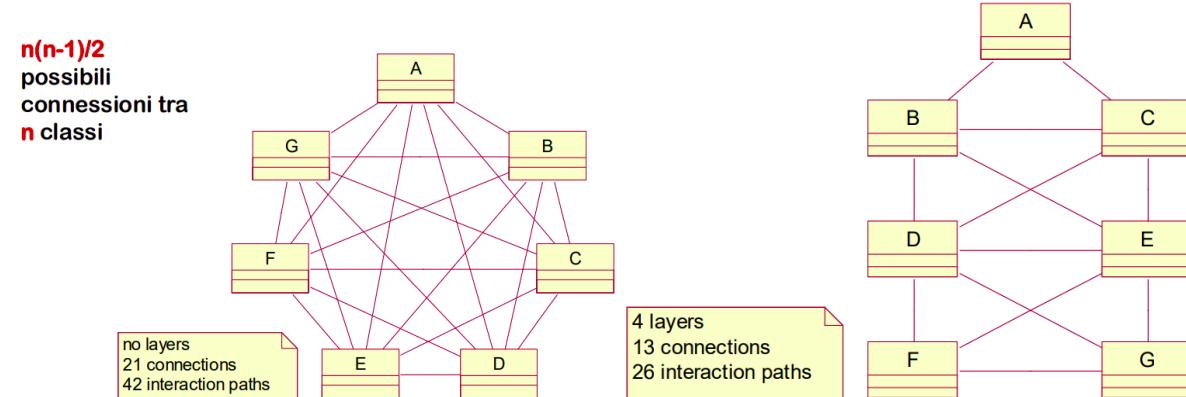
Si resta in Not In Stock finché o viene restituito il film noleggiato (in tal caso si torna in available) oppure se ne vengono ordinati di nuovi (si passa da ad Ordered).

Da Ordered al magazzino stato In Stock e poi si torna ad Available dopo averli messi sullo scaffale. Si poteva passare anche allo stato Ordered nel caso in cui un cliente lo abbia prenotato.

(FINE ESEMPIO)

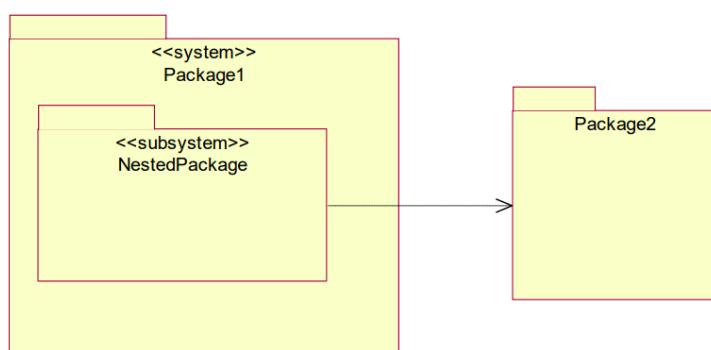
La Gestione della Complessità nei modelli di OOA.

Per sistemi software di grandi dimensioni, la complessità dei modelli (interconnessioni tra classi) deve essere gestita. Le associazioni tra classi nel modello dei dati generano complesse reti di interconnessione, in cui i cammini di comunicazione crescono in modo esponenziale con l'aggiunta di nuove classi. L'approccio utilizzato per far fronte alla complessità è la stratificazione: meccanismo che permette di isolare elementi tenendo conto di come essi devono interagire. In particolare gli elementi che fanno parte di un certo strato possono interagire solo con elementi dello stesso strato o di strati adiacenti. Tramite l'introduzione di questa gerarchia di classi, si passa da complessità esponenziale a polinomiale.



Per poter raggruppare gli elementi in UML si sfrutta la nozione di Package (in un package raggruppo quindi non solo classi ma anche altri elementi UML come use case).

I package possono anche essere annidati (gerarchie di package, il più esterno ha accesso a quelli interni), una classe può appartenere ad un solo package ma comunicare con classi appartenenti ad altri package. Si effettua una dichiarazione di visibilità (public, private, protected) per definire la visibilità delle classi presenti nel package.



Si usa il simbolo cartella per identificare i package, e si possono definire relazioni di dipendenza tra package con la freccia tratteggiata (non continua come in immagine) (relazione generica), quando si usa è opportuno come al solito definire lo stereotipo per capire che tipo di relazione sia.

In UML non esiste il concetto di Package Diagram, questi possono essere creati all'interno di class diagram e use case diagram.

In generale si possono specificare due tipi di relazioni tra package:

- Generalizzazione (implicano anche dipendenza)
- Dipendenza (di uso, accesso, visibilità etc..).

Il criterio che ci permette di definire gli strati, quanti sono e come raggruppare le classi in base alla loro responsabilità è l'approccio BCE (Boundary Control Entity).

- Boundary Package: classi i cui oggetti gestiscono l'interfaccia tra attore e sistema, presentando lo stato all'utente in forma visuale
- Control Package: inseriamo le classi che rappresentano azioni ed attività di uno use case, intercettando l'input dell'utente e controllando l'esecuzione dello scenario.
- Entity Package: classi entity che abbiamo identificato fin dall'inizio, che gestiscono l'accesso alle entità fondamentali del sistema, (corrispondono alle strutture dati) (BCE simile al paradigma in programmazione MVC model view controller dove model rappresentazione dati, view interfaccia e controller gestione logica applicativa).

Un oggetto boundary non può comunicare direttamente con Entity e viceversa!

PIANIFICAZIONE

La fase di pianificazione rientra in un processo più ampio: la Gestione di Progetti Software che comprende pianificazione, monitoraggio e controllo di persone, eventi e processi durante lo sviluppo del prodotto.

Il documento guida utilizzato è l'SPMP (Software Project Management Plan).

La gestione efficace di un progetto software si fonda sulle “quattro P”:

- Persone: Elemento più importante di un software di successo. (il SEI ha addirittura elaborato delle KPA legate alla gestione del personale).
- Prodotto: identifica le caratteristiche del software da sviluppare (obiettivi, dati, funzioni...).
- Processo: definisce il quadro generale entro il quale si stabilisce il piano di sviluppo.
- Progetto: definisce l'insieme di attività da svolgere (compiti, persone, tempi e costi).

(ES: saltare)

Problema: viene chiesto di sviluppare il prodotto software in 3 mesi e dall'analisi dei requisiti si capisce che sia necessario 1 anno/uomo (qm di effort, quantità di lavoro necessaria per svolgere una certa attività).

La soluzione immediata è farci lavorare 4 persone, ognuna 3 mesi/persona per un totale di 12 mesi/persona > 1 anno/persona (==11 mesi/persona). La realtà tuttavia ci dice che i 4 sviluppatori potrebbero impiegare un anno ottenendo un prodotto di qualità inferiore a quello risultante dal lavoro di un singolo sviluppatore. Ciò è dovuto al fatto che si lavora su un qualcosa di complesso, e aumentare il numero di persone significa anche aumentare il numero di interazioni, distribuire le decisioni ed avere eventuali incomprensioni.

Inoltre assegnare un compito che ha impegno pianificato 1 anno/uomo a 4 persone per farlo in tre mesi non tiene conto del fatto che alcuni compiti non possono essere condivisi.

Inoltre se si decidesse di aggiungere un ulteriore sviluppatore quando si osserva che il

progetto è in ritardo allora secondo la Legge di Brooks si rischierebbe di ritardare ulteriormente il progetto a causa del periodo di formazione (per mettersi in pari) e dell'aumento di interazioni.

(FINE ESEMPIO)

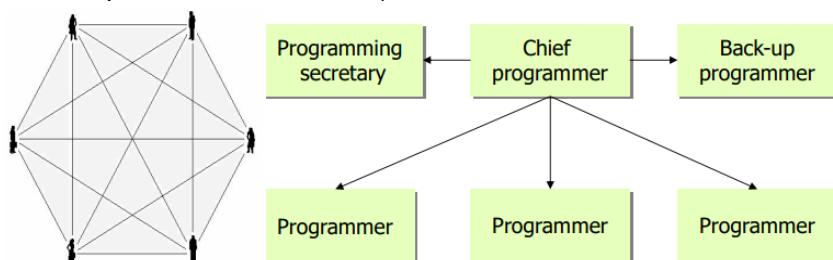
Descriveremo due degli approcci che permettono di organizzare team (persone).

- Democratico (orizzontale) ('70): basato sul concetto di egoless programming, per cui il codice prodotto è il codice del team piuttosto che del singolo sviluppatore che ha contribuito alla singola parte di codice (evitando così attacchi personali quando si trova un difetto del codice).

Vantaggi: atteggiamento positivo nella ricerca di difetti e molto produttivo in caso di problemi difficili da risolvere (es. ambiti di ricerca)

Svantaggi: l'approccio non può essere imposto ma deve nascere spontaneamente, inoltre gli sviluppatori più anziani non desiderano essere valutati al pari dei più giovani.

- Approccio con Capo Programmatore (gerarchico, verticale): basato sul concetto di specializzazione (ogni partecipante svolge i compiti per i quali è stato formato) e gerarchia (ogni sviluppatore comunica esclusivamente con il capo-programmatore che dirige le attività ed è responsabile dei risultati).



Vantaggi: (funziona in domini specifici già organizzati in termini gerarchici (militare, difesa))
Diminuisce il numero di canali di comunicazione e quindi diminuiscono anche i problemi in base alla legge di Brooks.

Svantaggi: richiede personale molto esperto per ricoprire i vari incarichi; il capoprogrammatore è sia manager che tecnico con grande esperienza, il programmatore di backup che può dover anche sostituire il capoprogrammatore è responsabile di attività di testing, poi il segretario di programmazione che è responsabile della documentazione e dell'archivio di produzione.

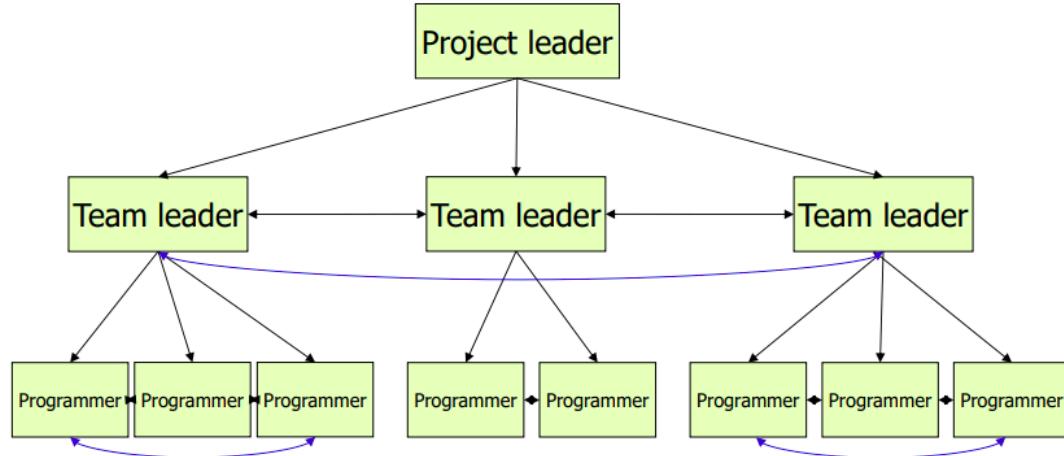
Questi due approcci appena visti sono agli estremi, ovviamente ne esistono molti altri "in between". Si è assistito in particolare ad una evoluzione degli approcci, dove il capoprogrammatore essendo sia manager che tecnico risulta essere il valutatore di se stesso quindi va sostituito da due individui:

- Team Leader (responsabile di aspetti tecnici)
- Team Manager (responsabile di aspetti gestionali)

Tuttavia nasce un problema: in questo approccio leader e manager non hanno canali di comunicazione. Soluzioni a ciò è identificare aree di responsabilità condivise per permettere ai due di comunicare e introdurre un altro livello di gestione con un project leader responsabile di progetto e che comunica con i team leader.

Ulteriore evoluzione è l'approccio che prende in considerazione anche l'approccio

orizzontale, con decision making decentralizzato per cui si introducono canali di comunicazione tra programmatori, tra team leader sfruttando quindi i vantaggi dell'approccio orizzontale.



Le componenti fondamentali di una corretta pianificazione sono:

- Scoping (comprendere il problema e lavoro da svolgere),
- Stime (prevedere tempi, costi ed effort),
- Rischi (definire le modalità di analisi e gestione dei rischi),
- Schedule (allocare attività e allocare le risorse disponibili per ogni attività),
- Strategia di controllo (avere gli strumenti per capire se si sta procedendo correttamente).

Le attività di stima di tempi, costi ed effort nei progetti software sono effettuate con gli obiettivi di: ridurre al minimo il grado di incertezza e limitare i rischi comportati da una stima. Risulta quindi necessario usare tecniche per incrementare l'affidabilità e l'accuratezza di una stima. Le tecniche di stima possono basarsi principalmente su:

- 1- stime su progetti simili già completati
- 2- "tecniche di scomposizione" (approccio bottom-up)
- 3- modelli algoritmici empirici (più accurato)

Le tecniche di scomposizione utilizzano un approccio divide et impera. Si basano su stime dimensionali (come LOC Lines of Code o FP Function Points) e sulla suddivisione dei task e delle funzioni con relativa stima di allocazione dell'effort.

I modelli algoritmici empirici invece si basano su dati storici e su relazioni del tipo $d=f(v_i)$ dove d è il valore da stimare (es. costo, effort, durata) mentre v_i sono le variabili indipendenti (es. LOC o FP stimati)

Il problema di entrambe le tecniche è che richiedono come variabile indipendente (parametro da fornire in input alla tecnica) una stima dimensionale del software (tipicamente tramite LOC) e quindi si torna al problema del sottostimare la dimensione del software che fornirà un valore scorretto in input e di conseguenza un risultato altrettanto sottostimato. La vera dimensione del software la saprà solo dopo la fase di codifica e integrazione, ed è difficile stimare prima LOC. Sono quindi state introdotte unità di misura alternative (come i punti funzione FP) che però purtroppo non sono compatibili con diversi modelli algoritmici empirici.

Functions	estimated LOC	LOC/pm	\$/LOC	Cost	Effort (MM)
UICF	2340	315	14	32,000	7.4
2DGA	5380	220	20	107,000	24.4
3DGA	6800	220	20	136,000	30.9
DBM	3350	240	18	60,000	13.9
CGDF	4950	200	22	109,000	24.7
PCF	2140	140	28	60,000	15.2
DAM	8400	300	18	151,000	28.0
Totals	33,360			655,000	144.5

Si stima guardando componenti più piccole: UICF gestione interfaccia utente, 2DGA e 3 analisi geometria a 2 e 3 dimensioni, DBM (database management), CGDF strumento di computer graphics, PCF gestione periferiche, DAM per l'analisi strutturale del progetto.

Devo anzitutto fornire il numero stimato di LOC per ogni componente, difficile tipicamente fare queste stime (tipicamente si fa ciò quando si sta aggiornando un progetto es. aggiungendo funzionalità di modo che non si sottostimi troppo). A questo punto chi fa uso della tecnica deve fornire come input anche LOC/pm (parametro di effort, produttività, quantità di lavoro per lo sviluppo quindi righe di codice per mese/uomo, quante righe di codice produce una persona in un mese lavorativo) e \$/LOC (parametro di costo, costo per riga di codice).

Forniti questi tre input tramite la tecnica di scomposizione posso derivare le colonne di Cost e Effort (MM) (semplicemente moltiplico \$/LOC per estimated LOC per il primo e per il secondo divido estimated LOC per LOC/pm).

NB: effort e costi calcolati riguardano solo la fase di progettazione (dal post specifica al rilascio).

Questa tecnica funziona solo se siamo in grado di avere dati relativi a produttività LOC/pm e costo \$/LOC ma soprattutto se siamo in grado di stimare accuratamente il numero di righe di codice per componente, e per farlo come anticipato è fondamentale lavorare con analogia dei progetti simili passati. Tuttavia un altro grosso problema sta nel fatto che questi dati non servono più a nulla se arriva un cliente che ci chiede di realizzare lo stesso identico progetto già fatto in passato ma su un linguaggio diverso. Questa dipendenza dalle righe di codice è un problema che è stato affrontato con l'introduzione di un'unità di misura specifica svincolata dal linguaggio di programmazione: FP (Function Point).

Misura il numero di funzionalità che il prodotto software contiene basandosi sul documento di specifica. Per questa ragione questa tecnica prevede la stima prima dell'implementazione (ossia calcolo il numero di FP a partire dal documento di specifica che ho già scritto, senza fare stime). Un FP incorpora una certa quantità standard di funzionalità che poi sarà usata come riferimento per calcolare la quantità complessiva di funzionalità.

La IFPUG (international function point user group) è l'organizzazione internazionale che fornisce il manuale con le modalità con cui calcolare il numero di FP a partire dal documento

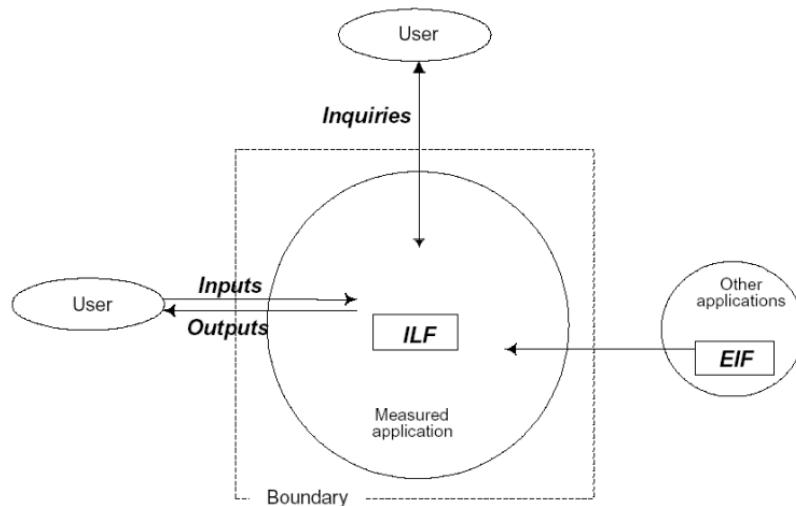
di specifica, il manuale è continuamente aggiornato e molto dettagliato.

Il conteggio FP è effettuato seguendo due step:

1) si calcola anzitutto l'Unadjusted Function point Count (UFC): si tiene conto esclusivamente della funzionalità che il prodotto deve offrire, senza guardare alla complessità tecnica.

2) si moltiplica l'UFC con il Technical Complexity Factor TFC.

$$FP = UFC \times TFC$$



UFC è calcolato rilevando 5 valori (detti elementi di conteggio) dal documento di specifica: Il cerchio grande rappresenta il software sul quale viene fatta la misura di punti funzione, tale software ha come confine Boundary (oltre il quale vi sono gli utenti e gli altri sistemi con cui il software interagisce).

Dei 5 elementi di conteggio citati prima due fanno riferimento ai dati (informazioni che il sistema software deve gestire o scambiare con altri) mentre gli altri tre fanno riferimento alle interazioni con il mondo esterno in termini di utenti del software stesso.

Nella Categoria Dati si devono contare:

- ILF (Internal Logical Files): gruppo di dati o informazioni di controllo generato, utilizzato o mantenuto dal sistema software (es. dati mantenuti dal software in database).

- EIF (External Interface Files): gruppo di dati o informazioni di controllo passate direttamente o condivise tra applicazioni.

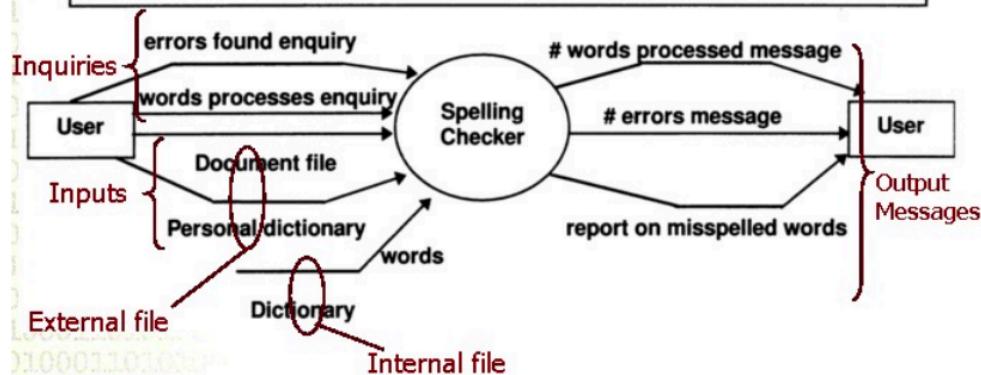
Il termine file non va inteso come tradizionale elemento nel file system ma come gruppo di elementi correlati tra loro.

Categoria interazioni (transazioni) si devono contare:

- EI (External Inputs): elementi che entrano nel software e modificano lo stato degli ILF.
- EO (External Outputs): dati/informazioni di controllo prodotti dal software e forniti all'utente.
- EQ (External Inquiries): tutte le combinazioni input/output dove un input causa e genera un output immediato senza cambiare lo stato degli ILF (senza modificare le informazioni gestite dal software).

Importante puntualizzare che non possono esserci elementi ripetuti in categorie diverse!

Spell-Checker Spec: The checker accepts as input a document file and an optional personal dictionary file. The checker lists all words not contained in either of these files. The user can query the number of words processed and the number of spelling errors found at any stage during processing.



(ES: saltare)

Si fa riferimento di un software il cui documento di specifica sia racchiuso in quel rettangolo sopra. Si tratta di un software che controlla gli errori ortografici nei testi. Il software accetta come input un documento e un dizionario personale opzionale e sceglie come parole sbagliate tutte quelle che non fanno parte del suo dizionario interno e del dizionario personale opzionale fornito.

Si immagina che vengano inviati documenti molto grandi, quindi durante la fase di processing l'utente può chiedere al sistema quante parole sono state processate fino a quel momento e quanti errori sono stati trovati. A questo punto per contare il numero di elementi di conteggio riformulo il grafico di prima, il software confinato nello Spelling Checker e poi gli utenti esterni.

Contiamo anzitutto il numero di file Interni (dati che lo spell checker contiene internamente e utilizza in fase di esecuzione) -> dizionario interno (1).

I file esterni sono invece il documento su cui fare spell checking e un eventuale dizionario (2).

Le inquiries sono a tempo di esecuzione il numero di parole processate e il numero di errori trovati (2).

Gli output prodotti sono il report finale con tutte le parole non parte dei dizionari e gli eventuali output delle 2 inquiries (3).

Gli input forniti dall'utente sono di nuovo il document file e il personal dict (2).

Importante puntualizzare che non possono esserci elementi ripetuti in categorie diverse! In questo caso infatti gli input forniti dall'utente non rappresentano il document e il personal dict in sé, ma il nome che li rappresenta (es. percorso file) mentre parlando di file esterni si intende il file nella sua interezza necessario per far funzionare il software.

(FINE ESEMPIO)

Una volta ottenuti questi numeri sfrutto una tabella fornita dal manuale di conteggio dove inserire i numeri contati nella prima colonna.

Function Type	Functional Complexity	Complexity Totals	Function Type Totals
ILFs	Low 1 Average High	X 7 = X 10 = 10 X 15 = _____	_____ 10
EIFs	Low 2 Average High	X 5 = X 7 = 14 X 10 = _____	_____ 14
EIs	Low 2 Average High	X 3 = X 4 = 8 X 6 = _____	_____ 8
EOs	Low 3 Average High	X 4 = X 5 = 15 X 7 = _____	_____ 15
EOs	Low 2 Average High	X 3 = X 4 = 8 X 6 = _____	_____ 8
Total Unadjusted Function Point Count			55

Per ognuno di essi decido la corrispettiva complessità funzionale, e in base ad essa decido per quanto moltiplicarli. Il manuale di conteggio potrà fornirmi dettagli riguardo ILF in questo caso, ad esempio potrebbe dirmi se una tabella ha numero di colonne inferiore a 10 allora Low, altrimenti etc...

Abbiamo quindi finalmente calcolato UFC (unadjusted perché finora abbiamo considerato questi elementi solo dal punto di vista della loro complessità funzionale)

Difficilmente ci si sbaglia di troppo perché il manuale di conteggio è molto dettagliato.

Per ottenere gli FP definitivi devo però considerare anche la difficoltà tecnica oltre a quella funzionale. La difficoltà tecnica è calcolata a partire da 14 fattori, detti Fattori di Degree of Influence, che possono avere influenza più o meno forte nel caso specifico del software dove sto effettuando il calcolo degli FP.

Ad ognuno dei fattori è associato un valore intero compreso tra 0 e 5, dove 0 influenza irrilevante e 5 influenza essenziale.

Alcuni di questi fattori sono:

se l'applicazione ha necessità di effettuare backup e recovery in modo affidabile (nel caso spell checker è 0, nel caso di un qualsiasi sistema d'archiviazione forse anche 5),

data communications,

distributed data processing (è software locale o distribuito, nel caso spell checker 0 in quanto lo eseguo nel mio pc),

prestazioni,

online data entry (è significativo l'ingresso dei dati o meno),

facilità d'uso,

....

riusabilità,

facilità d'installazione, se è necessario installarlo su più siti, facilità di modifica (manutenibilità).

Una volta associati i valori interi per ognuno dei 14 fattori, posso calcolare TCF come segue:

$$TCF = 0.65 + 0.01 \sum_{j=1}^{14} F_j$$

+35% adjusted perché se metto a tutti gli F_j 5 allora ottengo $0.65 + 0.7 = 1.35$ mentre se metto 0 ottengo 0.65. Si parla di aggiustare perché UCF verrà "aggiustato" di questa percentuale in avanti o indietro per via della moltiplicazione, non sarà mai stravolto ma sistemato in funzione della difficoltà tecnica.

Tuttavia resta un problema, le tecniche di stima di tempi costi ed effort maggiormente utilizzate non si basano su FP ma su LOC. La stessa funzione realizzata con un certo linguaggio di programmazione avrà uno specifico numero di LOC diverso dagli altri. Per tradurre quindi FP in LOC devo scegliere un determinato linguaggio. Sono state quindi sviluppate delle tabelline dette di backfiring.

Language	Nominal level	Source statements per function point		
		Low	Mean	High
First generation	1.00	220	320	500
Basic assembly	1.00	200	320	450
Macro assembly	1.50	130	213	300
C	2.50	60	128	170
Basic (interpreted)	2.50	70	128	165
Second generation	3.00	55	107	165
Fortran	3.00	75	107	160
Algol	3.00	68	107	165
Cobol	3.00	65	107	170
CMS2	3.00	70	107	135
Jovial	3.00	70	107	165
Pascal	3.50	50	91	125
Third generation	4.00	45	80	125
PL/I	4.00	65	80	95
Modula 2	4.00	70	80	90
Ada 83	4.50	60	71	80
Lisp	5.00	25	64	80
Forth	5.00	27	64	85
Quick Basic	5.50	38	58	90
C++	6.00	30	53	125
Ada 9X	6.50	28	49	110
Database	8.00	25	40	75
Visual Basic (Windows)	10.00	20	32	37
APL (default value)	10.00	10	32	45
Smalltalk	15.00	15	21	40
Generators	20.00	10	16	20
Screen painters	20.00	8	16	30
SQL	27.00	7	12	15
Spreadsheets	50.00	3	6	9

Per ogni linguaggio è fornito il source statement per punti funzione (a quante LOC in uno specifico linguaggio di programmazione corrisponde il singolo punto funzione). Chiaramente analisi statistica quindi: lower bound, numero medio e upper bound. Calcolare LOC corrispondenti tramite la tabella e fornirle in input alla mia tecnica di stima di effort e tempi che sarà accurato visto che ho preso in considerazione FP e corrispondente LOC via backfiring.

Con queste tabelle grazie al calcolo del numero di LOC possiamo associare valori quantitativi per rappresentare la capacità di espressione, da qui il Nominal Level della tabella che aumenta all'aumentare del livello del linguaggio.

Calcolati i FP dal doc di specifica e i corrispettivi LOC possiamo utilizzare oltre che la tecnica di tabelle di scomposizione anche i modelli algoritmici.

Vediamo di questi ultimi in particolare:

COCOMO (COmputational COst MOdel), modello introdotto da Barry Boehm nel 1981. Questo modello serve a determinare l'effort a partire da LOC e a partire dall'effort calcolare anche la durata e i costi di sviluppo.

COCOMO comprende 3 modelli:

- Basic per stime iniziali,
- Intermediate (usato dopo aver suddiviso il sistema in sottinsiemi)
- Advanced (usato dopo aver identificato i moduli facenti parte ciascun sottosistema e le relazioni tra essi..).

La stima dell'effort viene effettuata a partire dalla stima delle dimensioni del prodotto in KLOC e dalla stima del modello di sviluppo del prodotto, che misura il livello intrinseco di difficoltà nello sviluppo, tra:

- organic: (prodotti di piccole dimensioni)
- semidetached: (prodotti di dimensioni intermedie)
- embedded: (prodotti complessi)

Analizzeremo la versione originale, ma nel 95 è stato introdotto COCOMO II.

Prendiamo per vedere il funzionamento il caso di modello Intermediate e Organic.

Passo 1:

In generale il calcolo dell'effort è effettuato con formule del tipo

$$\text{Effort Nominale} = a(\text{KLOC})^b,$$

cioè che cambia sostanzialmente il valore di a e b in funzione del modello e modo.

Nel caso Intermediate Organic allora $a = 3.2$ e $b = 1.05$.

Si parla di Effort Nominale in quanto questo valore non tiene conto di caratteristiche sul costo (simile a quanto visto per i punti funzione con i fattori di degree influence).

Passo 2:

L'effort nominale deve essere "aggiustato" moltiplicandolo per un fattore moltiplicativo C il cui valore (similmente a quanto visto per il calcolo di FP) è basato su 15 cost drivers. In particolare C si ottiene come produttoria dei cost driver C_i . Ogni C_i determina la complessità del fattore i che influenza il progetto e ognuno di essi può assumere un valore che si scosta in più o in meno dal valore unitario (nominale, se scelgo tutti i C_i con valore unitario allora effort = effort nominale).

Cost Drivers	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Database size		0.94	1.00	1.08	1.16	
Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
Execution time constraint			1.00	1.11	1.30	1.66
Main storage constraint			1.00	1.06	1.21	1.56
Virtual machine volatility*	0.87		1.00	1.15	1.30	
Computer turnaround time	0.87		1.00	1.07	1.15	
Personnel Attributes						
Analyst capabilities	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Programmer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience*	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project Attributes						
Use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

*For a given software product, the underlying virtual machine is the complex of hardware and software (operating system, database management system) it calls on to accomplish its task.

Nel caso di FP erano fattori che avevano influenza sulla complessità tecnica del prodotto e che “aggiustavano” di +35% UnadjustedFPCost. In questo caso invece i cost driver hanno un rating: valore nominale 1.00 (infatti poi li metto in produttoria e se sono tutti 1 allora effort = effort nominale), il rating scende o sale a seconda dei casi, se scende ho risparmi in termini di costo se invece sale avrò costo aggiuntivo.

I 15 fattori sono divisi in 4 gruppi (legati a caratteristiche di Prodotto, Piattaforma d'esecuzione, Personale e Progetto).

Required Software Reliability (l'applicazione ha requisiti di affidabilità stringenti o no? Se non ne ha posso scegliere Very Low -> risparmio sul costo perché il prodotto non ha quella esigenza, se invece alti posso aumentare molto l'effort!), dimensione database, complessità di prodotto (per cui è definita anche Extra High con influenza Ci 1.65), vincoli di tempo d'esecuzione (es. per software real time), memorizzazione, volatilità virtual machine, tempo di risposta turnaround.

Poi per il calcolo dell'effort contano anche le caratteristiche del personale -> competenze analisti software (in questo caso se le competenze sono alte allora si risparmia sui costi! 0.71 Very High), conoscenza del dominio applicativo, competenza programmatore, conoscenza del sistema operativo dove si definisce il software, conoscenza linguaggio programmazione e lo stesso vale per gli attributi di progetto: se uso di pratiche moderne di programmazione allora l'effort diminuisce e lo stesso per l'uso di strumenti software.

L'ultimo cost driver, “Required development schedule” è l'unico che fornisce un valore maggiore di 1 sia a sx che a dx -> conviene stare al valore nominale.

Questo attributo indica il fatto che il mio progetto abbia o meno tempo stringente di consegna, sia che abbia tempi molto stringenti che non l'effort aumenta. Il motivo intuitivamente sta nel fatto che se devo fare qualcosa entro poco tempo mi ci metterò tanto quindi grande effort, se invece ho tanto tempo mi ci metterò comunque parecchio per cercare la soluzione migliore.

Applicando il prodotto per il fattore C non si modificherà mai l'effort di un ordine di grandezza, ma similmente a quanto visto per l'aggiustamento di UFC si "aggiusterà" l'effort nominale di un tot. Per capire quale rating conviene scegliere ci sono parametri standard da seguire,

Cost Driver	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
RELY	Effect: slight inconvenience	Low, easily recoverable losses	Moderate, recoverable losses	High financial loss	Risk to human life	
DATA		$\frac{DB \text{ bytes}}{\text{Prog. DSI}} < 10$	$10 \leq \frac{D}{P} < 100$	$100 \leq \frac{D}{P} < 1000$	$\frac{D}{P} \geq 1000$	
CPLX	See next slide					
Computer attributes						
TIME			$\leq 50\% \text{ use of available execution time}$	70%	85%	95%
STOR			$\leq 50\% \text{ use of available storage}$	70%	85%	95%
VIRT		Major change every 12 months Minor: 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 weeks Minor: 2 days	
TURN		Interactive	Average turnaround $< 4 \text{ hours}$	4-12 hours	>12 hours	
Personnel attributes						
ACAP	15th percentile*	35th percentile	55th percentile	75th percentile	90th percentile	
AEXP	$\leq 4 \text{ months experience}$	1 year	3 years	6 years	12 years	
PCAP	15th percentile*	35th percentile	55th percentile	75th percentile	90th percentile	
VEXP	$\leq 1 \text{ month experience}$	4 months	1 year	3 years		
LEXP	$\leq 1 \text{ month experience}$	4 months	1 year	3 years		
Project attributes						
MODP	No use	Beginning use	Some use	General use	Routine use	
TOOL	Basic microprocessor tools	Basic mini tools	Basic midi/maxi tools	Strong main programming, test tools	Add requirements, design, management, documentation tools	
SCED	75% of nominal	85%	100%	130%	160%	

* Team rating criteria: analysis (programming) ability, efficiency, ability to communicate and cooperate

es: Vediamo com. se dalla reliability dipende la vita delle persone allora very high.

Riguardo la dimensione si calcola rapporto dimensione database e numero d'istruzioni, se minore di 10 low etc...

La cosa utile di COCOMO è che oltre che stimare l'effort mi permette di determinare anche la durata del progetto. Sapendo che un software ha un effort di certi MM e sarà sviluppato da un certo team di persone, COCOMO permette di stimare il tempo che impiegheranno a svilupparlo usando ancora un'altra formula del tipo:

Tempo = $c(E)^d$ dove c e d cambiano a seconda del modello e del modo.

Anche qui le unità di misura hanno significato preciso: l'effort dato in input deve essere MM e il valore di durata restituito è espresso in mesi (tempo alla consegna, dal momento in cui inizio a usare COCOMO quindi tipicamente dopo analisi requisiti fino alla data di rilascio del prodotto al cliente o al mercato, non è quindi il tempo di vita del software ma di sviluppo a partire dal momento in cui uso COCOMO). Per d = Organic è 0.38, Semi-detached 0.35 e Embedded 0.32.

Non possiamo però concludere subito guardando questa formula che impiego meno tempo a sviluppare un embedded rispetto a un organic, in quanto l'effort che gli pongo in input è totalmente diverso (per embedded molto superiore).

Ciò che ci manca ancora da stimare è il costo. Questo valore si ottiene facilmente partizionando l'effort: infatti le persone che sviluppano il progetto ovviamente svolgono ruoli diversi.

Si divide anzitutto lo sviluppo (post analisi requisiti) in tre parti: progettazione preliminare, progettazione di dettaglio (codifica e testing), infine integrazione.

Dell'effort restituito da COCOMO ad es. 16% lo dedico al primo, 62% al secondo e 22% all'ultimo.

Dopodiché ragiono sul personale, partendo dalla progettazione preliminare e così via per le altre fasi.

Per ottenere il costo finale farò i calcoli sapendo il costo MM per ogni membro del personale (non è lo stipendio ma il costo totale per l'organizzazione, chiaramente comprende lo stipendio):

(es. se COCOMO mi darà 100 MM tot, allora 16MM per progettazione preliminare di cui 8 MM per il PM e gli altri 8 per l'analista. Sapendo che il PM ad es. mi costa 8k MM allora 64k solo per metà della progettazione preliminare e così via fino ad arrivare al costo complessivo.)

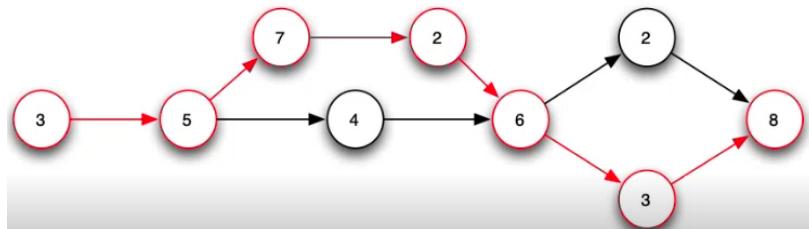
Pianificazione Temporale.

Dopo aver scelto il modello di processo, identificato i task da eseguire, stimato costi ed effort, è necessario effettuare la pianificazione temporale ed il controllo dei progetti, questa consiste nel definire una rete di quei task (che NON sono tutti indipendenti!) in base ai seguenti principi:

- Ripartizione (scomposizione del processo e prodotto in parti (task e funzioni) di dimensioni ragionevoli)
- Interdipendenza (identificazione delle dipendenze reciproche di task)
- Allocazione delle risorse (identificare numero di persone, effort, data inizio/fine da assegnare a ogni task)
- Responsabilità definite (individuare le responsabilità del personale associate a ciascun task)
- Risultati previsti: (definizione dei risultati prodotti al termine di ogni task)
- Punti di Controllo (milestone, identificare momenti in cui è possibile capire se il progetto procede come pianificato).

Gli strumenti che si utilizzano per operare sulla pianificazione temporale sono due:

- Diagrammi PERT (Program Evaluation and Review Technique) grazie al quale definiamo la rete di task come un grafo dove nodo = task e arco = legame di precedenza. Dal diagramma posso determinare il cammino critico (sequenza di task che determina la durata minima del progetto, il cammino più lungo) e stimare il tempo di completamento per ciascun task (es. posso stimare che il task di durata 7 sopra perché venga completato in totale ci metterò $3+8+7=18$) e i limiti temporali di inizio e fine di ciascun task.



(In questo caso il cammino critico è segnato dal cammino in rosso, accumulare un ritardo su uno di questi task significa portarsi dietro un ritardo per l'intero progetto.)

Ciò che PERT non fornisce direttamente sono informazioni precise a livello temporale (non so quando un progetto inizia e finisce a livello calendario), in questo senso si sfrutta la Carta di GANTT che è una pianificazione calendariale delle attività da svolgere.

Poiché in GANTT invece non appaiono le relazioni di precedenza tra task, viene integrata con PERT per fornire informazioni complete.

Tutti questi strumenti (COCOMO, tabelle di scomposizione, GANTT, PERT così come tutta la parte di risk management) fanno parte di un processo più ampio che abbiamo detto essere il Software Project Management, che è guidato da un documento fondamentale:

TITLE PAGE — document number, project and task names, report title, and report date.

LEAD SHEET — document identification numbers, project and task names, report title, customer name, preparers, contract and task identifiers, and report date.

TABLE OF CONTENTS — list of subsection titles and page numbers.

1. INTRODUCTION

1.1 **Purpose** — brief statement of the project's purpose.

1.2 **Background** — brief description that shows where the software products produced by the project fit in the overall system.

1.3 **Organization and Responsibilities**

1.3.1 **Project Personnel** — explanation and diagram of how the development team will organize activities and personnel to carry out the project: types and numbers of personnel assigned, reporting relationships, and team members' authorities and responsibilities (see Section 3 for guidelines on team composition).

1.3.2 **Interfacing Groups** — list of interfacing groups, points of contact, and group responsibilities.

2. STATEMENT OF PROBLEM — brief elaboration of the key requirements, the steps to be done, the steps (numbered) necessary to do it, and the relation (if any) to other projects.

3. TECHNICAL APPROACH

3.1 **Reuse Strategy** — description of the current plan for reusing software from existing systems.

3.2 **Assumptions and Constraints** — that govern the manner in which the work will be performed.

3.3 **Anticipated and Unresolved Problems** — that may affect the work and the expected effect on each phase.

3.4 **Development Environment** — target development machine and programming languages.

3.5 **Activities, Tools, and Products** — for each phase, a matrix showing: a) the major activities to be performed, b) the development methodologies and tools to be applied, and c) the products of the phase (see Section 4). Includes discussion of any unique approaches or activities.

3.6 **Build Strategy** — what portions of the system will be implemented in which builds and the rationale. *Updated at the end of detailed design and after each build.*

4. MANAGEMENT APPROACH

4.1 **Assumptions and Constraints** — that affect the management approach, including project priorities.

4.2 **Resource Requirements** — tabular lists of estimated levels of resources required, including estimates of system size (new and reused LOC and modules), staff effort (managerial, programmer, and support) by phase, training requirements, and computer resources (see Section 3). Includes estimation methods or rationale used. *Updated estimates are added at the end of each phase.*

- 4.3 Milestones and Schedules** — list of work to be done, who will do it, and when it will be completed. Includes development life cycle (phase start and finish dates); build/release dates; delivery dates of required external interfaces; schedule for integration of externally developed software and hardware; list of data, information, documents, software, hardware, and support to be supplied by external sources and delivery dates; list of data, information, documents, software, and support to be delivered to the customer and delivery dates; and schedule for reviews (internal and external). *Updated schedules are added at the end of each phase.*
- 4.4 Metrics** — a table showing, by phase, which metrics will be collected to capture project data for historical analysis and which will be used by management to monitor progress and product quality (see Section 6 and Reference 3). If standard metrics will be collected, references to the relevant standards and procedures will suffice. Describes any measures or data collection methods unique to the project.
- 4.5 Risk Management** — statements of each technical and managerial risk or concern and how it is to be mitigated. *Updated at the end of each phase to incorporate any new concerns.*

5. PRODUCT ASSURANCE

- 5.1 Assumptions and Constraints** — that affect the type and degree of quality control and configuration management to be employed.
- 5.2 Quality Assurance (QA)** — table of methods and standards used to ensure the quality of the development process and products (by phase). Where these do not deviate from published methods and standards, the table references the appropriate documentation. Means of ensuring or promoting quality that are innovative or unique to the project are described explicitly. Identifies the person(s) responsible for QA on the project, and defines his/her functions and products by phase.
- 5.3 Configuration Management (CM)** — table showing products controlled, tools and procedures used to ensure the integrity of the system configuration: when the system is under control, how changes are requested, who makes the changes, etc. Unique procedures are discussed in detail. If standard CM practices are to be applied, references to the appropriate documents are sufficient. Identifies the person responsible for CM and describes this role. *Updated before the beginning of each new phase with detailed CM procedures for the phase, including naming conventions, CM directory designations, reuse libraries, etc.*

6. REFERENCES

- 7. PLAN UPDATE HISTORY** — *development plan lead sheets from each update indicating which sections were updated.*

SPMP (software project management plan).

PROGETTAZIONE

La Progettazione è la fase in cui si decide il COME realizzare il sistema software, prendendo in input il documento di specifica (dall'analisi dei requisiti) e producendo il Documento di Progetto. Si divide in:

- Progetto Architetturale (o preliminare): il sistema software viene partizionato in più componenti.
- Progetto Dettagliato: ciascuna componente viene progettata in modo dettagliato, scegliendo algoritmi e strutture dati.

Principi di progettazione:

- Stepwise Refinement: è una strategia di progettazione top-down consistente nel procedere per raffinamenti successivi, si parte dalla specifica di una funzione (o di dati) in cui ancora non è descritto il funzionamento interno/struttura interna dei dati, per poi di volta in volta aggiungere un livello di dettaglio maggiore (da qui raffinamento) (Legge di Miller: concentrarsi al più su 9 elementi). Il concetto di Raffinamento è complementare a quello di Astrazione.

- Astrazione: concentrarsi solo sugli aspetti essenziali ignorando i dettagli secondari (introdotto da Dijkstra). Nell'ambito del processo software, ogni passo rappresenta un raffinamento (scendo più in dettaglio) del livello di astrazione della soluzione.

Due principali tipi di astrazione:

- Procedurale (es. le funzioni: funzione printf stampa, non serve che io sappia come funziona il codice)
- dei Dati (es. data encapsulation, ossia utilizzo di una struttura dati che astrae l'insieme di azioni eseguite su di essa).

- Decomposizione Modulare e Modularità:

Suddividere il prodotto in segmenti più piccoli (moduli software) per migliorare manutenibilità, comprensibilità e riusabilità. Definizione standard della modularità IEEE 610-12: la modularità è il grado di cui un certo software è costituito da un numero di componenti discrete (moduli) tali che la modifica di un componente abbia impatto minimo sugli altri. Quindi il criterio che deve guidare l'attività di decomposizione modulare è identificare moduli quanto più indipendenti l'uno dall'altro.

Un modulo è un elemento software che:

- contiene istruzioni, logica e strutture dati (sia definizione variabili che il loro utilizzo)
- può essere compilato separatamente e memorizzato in una libreria software
- può essere incluso in un programma
- può essere usato invocando segmenti di modulo identificati da nome e lista di parametri
- può usare altri moduli

Esempi del concetto di modulo possono essere le funzioni o le classi.

Il risultato ottenuto dopo la decomposizione modulare è detto architettura dei moduli. Quindi in generale la decomposizione modulare si basa sul principio del "divide et impera" per cui dividere in moduli mi permette di ridurre la complessità e quindi l'effort, in particolare il costo di riaggregazione è tale per cui la complessità e quindi l'effort di risolvere due moduli insieme è maggiore rispetto a risolverli separatamente per poi riaggregarli.

$$C(p_1) > C(p_2) \Rightarrow E(p_1) > E(p_2)$$

$$C(p_1+p_2) > C(p_1) + C(p_2)$$

↓

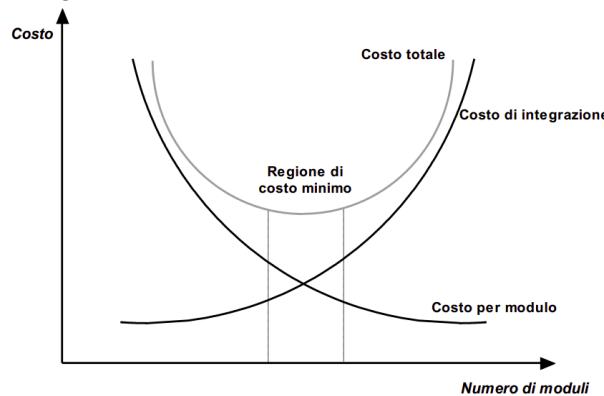
$$E(p_1+p_2) > E(p_1) + E(p_2)$$

Una buona divisione di un prodotto software in moduli è quella che permetti di ottenere:

- Massima coesione (cohesion) interna ai moduli

- Minimo grado di accoppiamento (coupling) esterna ai moduli

La coesione rappresenta le interazioni interne al modulo e deve essere massimizzata, il coupling fa invece riferimento all'interazione tra moduli e deve essere minimizzata. Il problema di queste metriche è che se cerchiamo di massimizzare la coesione operiamo negativamente sul coupling e viceversa, si vuole quindi trovare il numero di moduli che offra il miglior tradeoff, e in questo senso facciamo riferimento esclusivamente al costo.



Coesione: Per eseguire una funzione sono necessarie varie azioni, che possono essere concentrate in un singolo modulo o sparse in tanti.

In altre parole, la coesione misura il grado di interazione interna al modulo tra le azioni di una funzione.

La coesione si misura utilizzando una scala di valori, per un totale di 7:

1. Coincidentale: nessuna relazione tra azioni nel modulo.
 2. Logical: elementi correlati, ma solo uno di essi viene utilizzato dal modulo chiamante
 3. Temporal: relazione temporale tra gli elementi
 4. Procedurale: gli elementi sono correlati in base a una sequenza predefinita di passi
 5. Communicational: leggermente migliore della procedurale, uguale ad essa solo le azioni sono svolte sulla stessa struttura dati
 6. Informational: ogni elemento ha una porzione di codice indipendente e un proprio punto di ingresso e di uscita, inoltre ogni elemento agisce sulla stessa struttura dati.
 7. Funzionale: tutti gli elementi sono correlati dal fatto di svolgere una singola funzione.
- In base al tipo di approccio, possiamo avere come obiettivo o Informational (per paradigma OO) o Functional (per programmazione strutturata).

(ES: saltare)

Es. di coesione coincidentale: stampa la prossima riga, inverti i char della seconda stringa parametro, aggiungi 7 al quinto parametro etc...

sono tutte azioni scorrelate, un modulo del genere non è riusabile o mantenibile.

Spesso vi sono dei vincoli nella dimensione del numero di istruzioni minime per modulo, quindi in tal caso anche per moduli semplici è necessario aggiungere istruzioni di questo tipo per raggiungere la dimensione minima.

Es di coesione logica:

module performing all input and output		
1. Code for all input and output		
2. Code for input only		
3. Code for output only		
4. Code for disk and tape I/O		
5. Code for disk I/O		
6. Code for tape I/O		
7. Code for disk input		
8. Code for disk output		
9. Code for tape input		
10. Code for tape output		
:	:	:
37. Code for keyboard input		

il modulo è diviso in tanti pezzi di codice e so che però se mi servisse ne chiamerò soltanto uno. Problemi di manutenibilità perché le porzioni di codice potrebbero essere interallacciate con altro fuori, per questo ancora solo livello 2 di coesione.

Es coesione temporale: open old_master_file, open new_master_file, open transactionfile etc... azioni correlate a livello temporale.

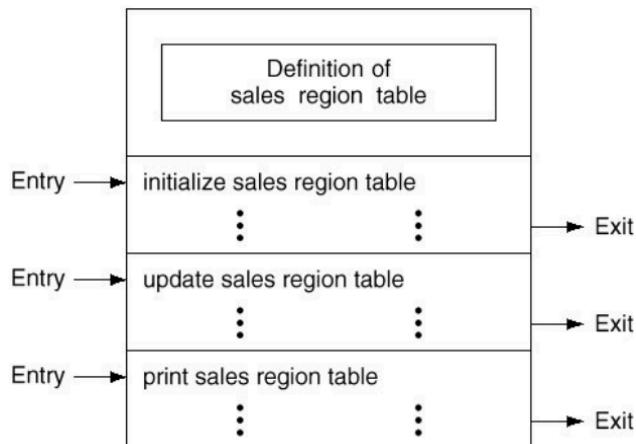
Ancora solo livello 3 di coesione perché i file aperti in questo modulo potrebbero essere aggiornati da altri moduli -> sempre problemi manutenibilità.

Es Coesione procedurale: read part_number from database, use part_number to update... azioni avvengono in sequenza.

Es Communicational aggiunge il fatto che si lavora su una singola variabile/struttura dati.

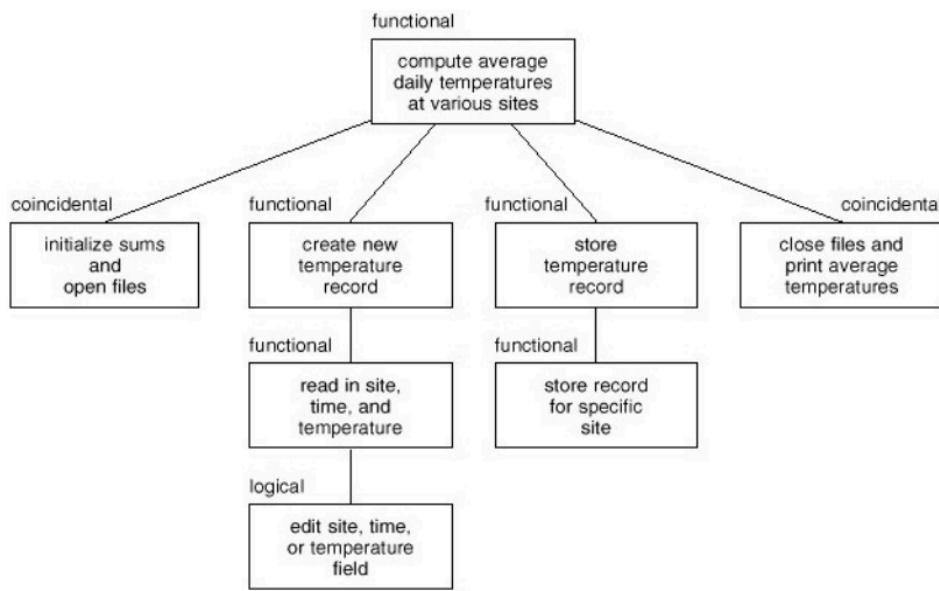
In alcune scale di valori vista la loro somiglianza temporal, procedural e communicational sono combinate in un singolo livello.

Es coesione informational: es. classe dove si ha una struttura dati e una serie di funzioni che lavorano su di essa, ognuna utilizzabile in modo separato.



(FINE ESEMPIO)

La misurazione della coesione ovviamente avviene a livello architetturale, ossia per ciascun modulo individualmente.



A partire dalla descrizione di ogni modulo dobbiamo capire la relativa coesione: vediamo moduli con coesione functional che svolgono una funzione ben precisa (es. memorizzare record temperature, crea record temperature etc..), si hanno poi moduli a coesione coincidentale che svolgono azioni scorrelate (es. inizializza le somme e apri i file, chiudi file e stampa le temperature medie. Si poteva fare di meglio, ad es. creando un modulo dedicato alla gestione dei file, uno a inizializzare le somme e l'altro a stampare le temperature medie), poi moduli a coesione logica che fa varie cose ma ogni volta che lo utilizzo se ne fa una sola in base a ciò che interessa al modulo chiamante.

Dobbiamo ora occuparci di misurare il coupling.

Il coupling misura il grado di accoppiamento tra moduli, anch'esso si misura usando una scala, stavolta a 5 livelli dal peggiore al migliore:

1. Content: un modulo fa riferimento diretto al contenuto di un altro modulo, modificandolo o semplicemente accedendovi (livello peggiore, forte dipendenza tra moduli)
2. Common: due moduli accedono in modalità read e write alla stessa struttura dati
3. Control: un modulo controlla esplicitamente l'esecuzione di un altro modulo
4. Stamp: due moduli interagiscono scambiandosi strutture dati della quale si usano solo alcuni parametri (quindi diciamo che tra i parametri ce ne sono alcuni che non servono all'altro modulo)
5. Data: due moduli interagiscono scambiandosi messaggi, in particolare passando come argomento una struttura dati della quale si usano tutti i parametri

I fattori che influiscono sul grado di accoppiamento sono la quantità di dati condivisi tra moduli, il numero di riferimenti che un modulo ha rispetto ad altri moduli, la complessità dell'interfaccia tra moduli, il livello di controllo che un modulo esercita su un altro.

(ES: saltare)

Content Coupling: example

Degli esempi di content coupling è, dati due moduli p e q,
 se p modifica un'istruzione di q,
 se p fa riferimento a dati locali di q in termini di qualche "displacement" numerico (in

linguaggi a basso livello come assembly, dove si utilizzano i displacement tra moduli per passarsi informazioni),

se p utilizza un'etichetta locale del modulo q (caso del go to, se p fa goto su un'etichetta interna di q, sta saltando direttamente nel suo flusso di controllo interno).

Questo tipo di coupling è molto difficile da trovare in codici attuali, si tratta del livello peggiore di coupling perché ogni cambiamento a q richiede una modifica al modulo p.

Common Coupling: example

Riguardo il common coupling un esempio sono due moduli cca e ccb che accedono alla stessa variabile globale in modalità lettura e scrittura. Si tratta di un grado comunque non buono in quanto capire come si comporta la parte di codice dedicata alla variabile è più difficile se vi sono più moduli che vi accedono e possono modificarla (problemi di sicurezza e integrità dei dati della variabile)

Control Coupling: example

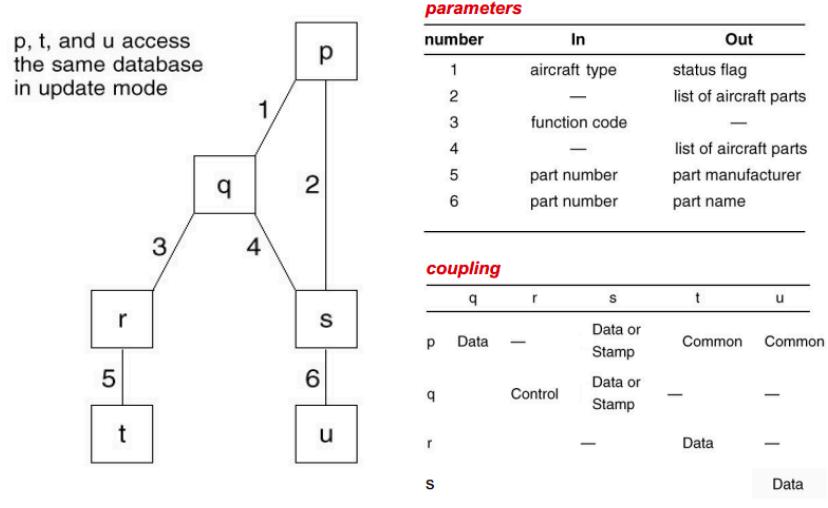
Un esempio per il control coupling è se un modulo p chiama il modulo q chiedendogli di fare qualcosa, dopodiché q invia un flag di ritorno a p che permette a p di svolgere una certa azione (in base a quanto detto da q p si comporta di conseguenza). In questo senso q esercita controllo su p in quanto in base alla sua risposta p si comporta di conseguenza.

Stamp e Data Coupling

~~Stamp~~ Stamp e Data coupling sono infine il livello migliore in quanto i moduli si scambiano informazioni tramite un'interfaccia ben definita scambiandosi messaggi contenenti solo informazioni necessarie se Data altrimenti Stamp.

(FINE ESEMPIO)

Example Structure Chart (oggi chiamato architettura software)



L'interazione tra moduli è evidenziata da degli archi numerati.

I moduli si scambiano dei messaggi, e questi messaggi veicolano dei parametri che possono essere di input o di output.

(ES: saltare)

Es. p e q interagiscono scambiandosi il messaggio 1 che ha come parametro di input tipo di aircraft e output status flag.

Nella tabella sotto invece vediamo per ogni interazione che esiste tra moduli il relativo livello di coupling.

Es. tra p e q viene definito il livello data, si assume cioè che sia aircraft type che status flag siano parametri utilizzati dai rispettivi moduli.

Un altro esempio è l'interazione 4, dove i due moduli si scambiano parametri di output lista di parti di aircraft. Il relativo coupling è definito Data or Stamp poiché evidentemente non si hanno informazioni sull'effettivo utilizzo di questa lista di parti (se ne uso solo alcune allora stamp else data), vale lo stesso per l'interazione 2.

Per l'interazione 3 invece come parametro di input codice di funzione, coupling di controllo in quanto chi manda il messaggio chiede al modulo di eseguire la funzione.

Dopodiché anche 5 e 6 Data Coupling, mentre per l'accoppiamento tra p e t e p e u livello di coupling Common. Nella figura non vi è l'arco che identifica l'interazione in quanto p e t e p e u non si scambiano effettivamente dei messaggi ma accedono allo stesso database in update mode (NB se fosse stato scritto in read mode allora non sarebbe stato accoppiamento di tipo common).

(FINE ESEMPIO)

Information Hiding

Consiste nel progettare e definire i moduli in modo che gli altri moduli vedano solo quanto serve, nascondendo quindi i dettagli implementativi (procedura e dati) che ad essi non sono necessari. I vantaggi si riscontrano quando è necessario apportare modifiche (fasi di testing e manutenzione)

Nel caso dell'uso di Information Hiding invece si usano qualificatori di accesso (private, getter e setter).

Riusabilità:

Fa riferimento all'utilizzo di componenti sviluppati per un prodotto all'interno di un prodotto differente.

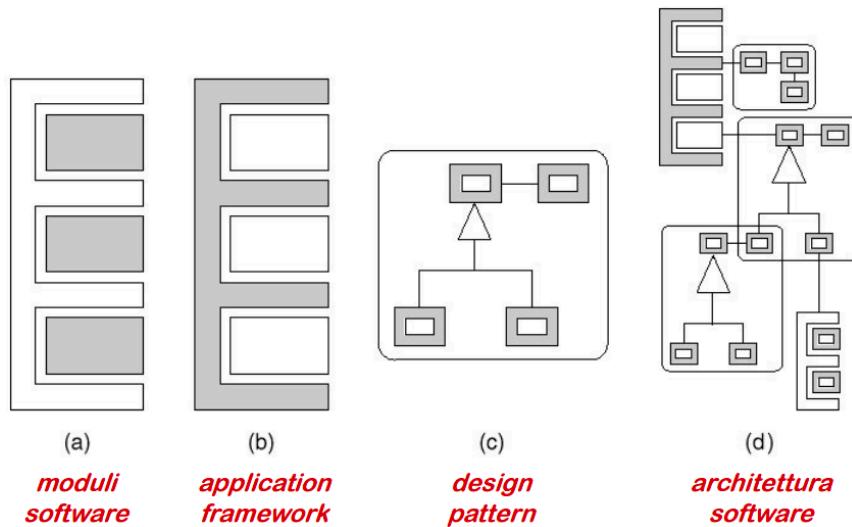
In generale per componente riusabile si fa riferimento a moduli, parte di codice, progetti, parti di documenti, insiemi di test data, stime di costi o tempi etc..

Tra i principali vantaggi la netta diminuzione di costi e tempi di produzione del software e incremento dell'affidabilità dovuto all'uso di componenti già convalidati.

Nella fase di Progetto, la riusabilità si applica a:

- singoli moduli software
- application framework (la logica che tiene insieme i moduli e che viene utilizzata per usare i moduli)
- design pattern (a livello progettuale spesso capita di affrontare problemi ricorrenti, quindi si sono definite soluzioni standard)

- architettura software (comprendenti (a), (b), (c)).



OOD (Object Oriented Design)

Il metodo di progettazione OOD (Object Oriented Design) è costituito da due sottofasi principali:

- Progettazione Preliminare (o Architectural Design/System Design): Si prendono decisioni sull'organizzazione d'insieme del software, definendo la sua architettura.

- Progettazione Dettagliata (Object Design): Si definiscono i dettagli di ciascun modulo in termini algoritmici e di strutture dati.

OOD è un processo iterativo e incrementale e riutilizza i risultati ottenuti dalla fase di OOA.

Architetture di sistema

L'architettura di sistema definisce la struttura delle componenti software e le relazioni tra di esse. Si osserva un'evoluzione delle architetture standard, dalle più vecchie alle più moderne.

Le prime 2 architetture invece sono esempi di architetture software centralizzate (si usa tipicamente un processo del sistema operativo che, mandato in esecuzione, esegue l'intero software).

Le architetture dalla 3 alla 6, sono definite architetture di sistema software distribuito (ossia per cui la sua esecuzione non è centralizzata ma partizionata e distribuita su vari nodi di esecuzione che sono interconnessi attraverso un'infrastruttura di rete che può essere locale, geografica etc...).

a. Architetture Centralizzate

1- Mainframe-based architectures: Utilizzano un computer molto potente (mainframe) che esegue sistemi operativi multiutente. I terminali collegati (solo dispositivi di I/O) non possono eseguire software autonomamente. Tutte le applicazioni sono eseguite sul singolo mainframe, rendendo l'esecuzione centralizzata. Sono ancora usate in domini critici come quello bancario per la loro affidabilità e sicurezza.

2- File Sharing: PC sono collegati in rete (es. Ethernet), l'esecuzione è localizzata su un singolo nodo, quindi è considerata centralizzata.

b. Architetture Distribuite

L'esecuzione del software è partizionata su più nodi interconnessi da un'infrastruttura di rete. Per l'utente, la distribuzione è trasparente, non ci si accorge ad es. che si sta comunicando con un server nell'eseguire un'applicazione.

Nel passaggio da architetture centralizzate a distribuite ha svolto un ruolo chiave l'utilizzo di tecnologia Middleware e rappresenta dello strato software che gestisce la connettività. (es. RPC remote procedure call, invece di chiamare procedure locali ne chiamo di remote. Devo quindi chiedere a qualcun altro di svolgere la procedura e prendere i risultati, i protocolli che permettano che ciò avvenga appunto sono gestiti dal Middleware, altri esempi MOM e ORB).

I principali vantaggi che hanno portato all'utilizzo di sistemi distribuiti:

Condivisione di dati e risorse tra i nodi, Openness (possibilità di gestire anche risorse eterogenee, si vuole ad esempio che un oggetto scritto in java in una macchina Windows possa invocare un metodo di un oggetto implementato in C++ su una macchina Linux), Concurrency (tutti gli oggetti in esecuzione operano in modo concorrente), Scalability (se non si ha più memoria è sufficiente aggiungere un nuovo nodo da cui condividere), Load

Balancing (si vuole distribuire il carico affinché non vi siano processi troppo appesantiti di lavoro), Fault Tolerance (backup dei nodi nel caso qualcosa vada storto), Trasparenza (diversi tipi di trasparenza, es. l'utente non distingue l'uso di risorse remote o locali oppure l'utente non sa che il nodo non funziona se è operativo quello di backup etc...).

Tuttavia non vi sono solo vantaggi ma anche fattori critici:

- Qualità del servizio (problemi di affidabilità es. se non ho la rete non posso usare l'applicazione, di performance, tempo di accesso al disco fisico etc.., si possono fare delle predizioni ma non saranno mai precise)
- Interoperabilità (è necessario far gestire risorse eterogenee ed è difficile)
- Sicurezza (dati condivisi tra vari nodi)

3- Architetture Client/Server: si distingue il ruolo di processi client, di processi server e quelli che svolgono entrambi i ruoli. Il processo client interagisce con l'utente ed è quindi responsabile di prendere in carico la richiesta e fornire una risposta. Il client è solo un intermediario, non produce lui la risposta ma sottomette la richiesta ad un processo server per ottenerla.

Il processo server è invece il processo che attende le richieste da eventuali processi client. Egli nasconde la complessità dell'intero sistema all'utente e al processo client (da qui la trasparenza agli occhi dell'utente). Quando riceve la richiesta il processo server può rispondere direttamente o usare server secondario per rispondere.

Application Layer

rappresenta l'approccio BCE per la rete dei calcolatori

- Presentation Layer (Boundary - B): Aspetti di presentazione, interazione con l'utente.
- Application Processing Layer (Control - C): Logica di esecuzione dell'applicazione.
- Data Management Layer (Entity - E): Gestione dei dati.

Two-Tier C/S Architectures: Si esauriscono in due livelli (client e server) dove possiamo allocare i layer applicativi. Due principali modelli:

- Thin-Client Model: Solo il Presentation Layer è sul client, Application Processing e Data Management sono sul server, "appesantendolo"
- Fat-Client Model: Presentation e Application Processing sono sul client, Data Management sul server. Il client gestisce gran parte della logica. Esistono modelli intermedi.
Si tratta chiaramente di modelli agli estremi, esistono infatti modelli intermedi dove la parte di Application Processing viene partizionato nel Client e nel Server.

Three-Tier Architectures: Evoluzione delle Two-Tier, con un livello specifico per ogni Application Layer. Presentation sul client, Application Processing su un server intermedio, Data Management su un server di backend. Il server intermedio agisce da client e server, e la sua complessità è nascosta al client. Offre migliori performance rispetto al thin-client e più semplice gestione rispetto al fat-client.

N-Tier Architectures: Estensione delle Three-Tier con l'aggiunta di ulteriori strati di backend (es. authentication servers). Le architetture C/S sono ampiamente usate e sono state base per architetture a oggetti distribuiti (Distributed Object Architecture).

4- Architettura ad Oggetti Distribuiti

Nel paradigma object oriented, il singolo oggetto può svolgere sia il ruolo di client (se richiede un metodo) che di server (se lo esegue).

Qui entra quindi molto in gioco il middleware, si vuole che invocare un metodo non comporti differenze rispetto all'approccio centralizzato.

Per raggiungere questo obiettivo è stato realizzato un middleware che è chiamato ORB (Object Request Broker), che funge da agente (broker) per la comunicazione (si parla in questo senso di software bus).

Anche per ORB netta separazione tra interfaccia e implementazione, infatti i servizi offerti da ORB sono specificati (viene definita quindi solo l'interfaccia) in un abstract bus, per poi procedere all'implementazione concreta con la bus implementation.

L'esempio più famoso di abstract bus si chiama CORBA, questo rappresenta uno standard pubblicato da OMG per la specifica dell'interfaccia dei servizi offerti da un ORB (quindi no implementazione, solo standard di specifica).

Tuttavia ciò comportò problemi in quanto l'interoperabilità non era più garantita: se dispositivi eterogenei interagivano e l'implementazione di ORB era una in orbix e l'altra in visibroker (i due principali vendori) non funzionava.

A questo punto l'OMG ha dovuto rilasciare una versione successiva di CORBA in cui venne introdotto un protocollo chiamato IIOP che permette di garantire interoperabilità anche tra ORB eterogenei.

5- Component Based

L'idea è sviluppare software assemblando componenti preconfezionate che implementano funzionalità specifiche.

In questo senso una componente software è un'astrazione che può essere implementata in modi diversi (oo, approccio strutturale etc...), si fa uso del principio di netta separazione tra interfaccia e implementazione. Si parla in questo caso quindi di un riuso black box, in quanto la componente viene riusata semplicemente perché realizza quell'interfaccia.

Aspetti importanti delle componenti software sono quindi:

- capacità di incapsulare strutture software in queste componenti astratte (variabilità, cambiano comportamento in base a come sono utilizzate)
- possibilità di "assemblare" queste componenti collegandoli attraverso l'interfaccia e lo scambio di messaggi (adattabilità)

Vediamo di seguito le principali differenze tra oggetto e componente:

- L'oggetto incapsula servizi; la componente è un'astrazione per costruire sistemi OO.
- L'oggetto ha granularità specifica; la componente ha granularità molto variabile (singolo oggetto a intera applicazione).
- L'oggetto ha identità, stato, comportamento; la componente è un'entità software statica a cui chiedere e da cui ottenere qualcosa.

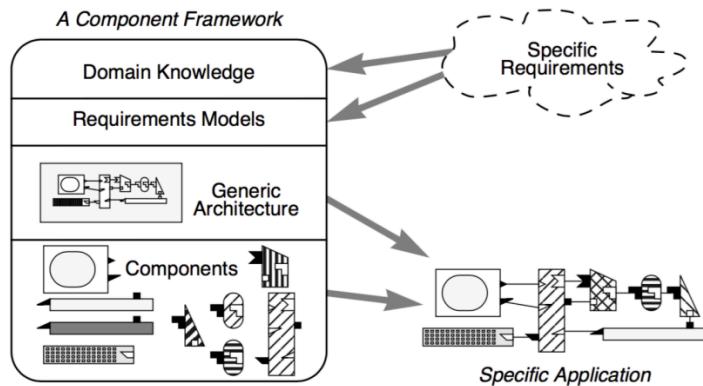
Component framework

Il component framework è l'elemento di cui si fa uso per realizzare applicazioni component based. Fornisce una libreria di componenti, architetture software generiche e cattura requisiti generici per un dominio applicativo. Quindi si fa tipicamente quanto segue: ho dei requisiti specifici per il mio dominio, vedo se vi sono somiglianze con i requisiti generici di un certo

framework e se ve ne sono abbastanza allora implemento le componenti mancanti e il framework potrà permettermi di costruire l'applicazione per soddisfare i miei requisiti specifici.

I programmatori possono quindi aggiungere componenti mancanti, aggiornando il framework per usi futuri.

Tuttavia, non ha avuto grande successo commerciale/industriale nella realizzazione di framework sufficienti.



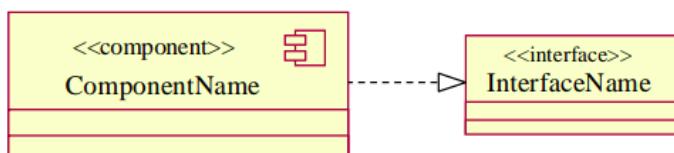
Componenti UML

In UML 1 il concetto di componente era legato a un'entità fisica: un'implementazione di cui posso fare l'allocazione in una certa piattaforma di esecuzione (es. eseguibile, documento word etc..).

Tuttavia ciò comportava un problema: non vi era rappresentazione di come si arrivasse a queste componenti a partire ad esempio dalle classi

Con UML 2 si supera il problema introducendo una modifica sostanziale al concetto di componente: esso rappresenta un elemento che esiste già a livello di progetto.

Il componente viene quindi rappresentato come una sorta di classe con lo stereotipo <<component>> che in modalità blackbox realizza una certa interfaccia, questa realizzazione blackbox è visualizzata con la freccia a triangolo vuoto (simbolo ereditarietà) ma tratteggiata.



In UML 2 inoltre, per superare il gap semantico tra gli elementi a livello di progettazione e livello di esecuzione, il concetto di classe è stato rimpiazzato dal concetto di classe strutturata (per far capire come le classi che progetto sono tradotte in oggetto eseguibile).

6- Architetture Service Oriented

Una SOA è un'architettura software distribuita costituita da più servizi autonomi, che possono essere eseguiti su nodi e piattaforme diverse e implementati in linguaggi diversi. Vengono forniti protocolli standard basati su Internet per consentire ai servizi di comunicare tra loro e scambiare informazioni:

Ogni servizio ha una descrizione, che consente alle applicazioni di scoprire e comunicare con il servizio stesso. La descrizione del servizio definisce il nome del servizio, posizione, e dati richiesti per uno scambio.

Il service provider deve fornire un servizio usabile da più clienti. A differenza delle architetture client-server, le SOA si basano su il concetto di loosely coupled services che possano essere scoperti e collegati anche dai clienti (indicato come consumatore di servizi o servizio richiedenti) con l'assistenza dei broker di servizio.

L'obiettivo è progettare servizi come componenti riutilizzabili e autonomi, minimizzando le dipendenze.

Di seguito i principi di questa architettura

Loose Coupling: servizi il più possibile indipendenti dagli altri (simile per quanto visto parlando di coesione e coupling)

Service Contract: contratto tra consumatore e provider per cui il provider promette al consumatore di rispettare la descrizione mentre il consumatore di usare il servizio secondo le modalità previste

Autonomia: ancora indipendenza da altri servizi

Astrazione: il consumatore deve sapere solo quanto necessario

Riusabilità, Composability e Statelessness: già principi utilizzati nell'architettura basata su componenti

Discoverability: principio esemplare per questa architettura, rappresenta la capacità di un servizio di essere trovato e identificato facilmente da altri servizi o da chi li utilizza.

Software Architectural Broker Patterns

Abbiamo detto che nell'architettura service oriented ritroviamo i due ruoli introdotti con il client/server: il consumatore di servizio (client) e il service provider (server).

Tra le due entità si interpone il service broker che regola e permette la comunicazione tra i due.

Si parla quindi di Software Architectural Broker Patterns.

Per poter utilizzare il broker il service provider, una volta realizzato un servizio, deve registrarsi presso il broker al fine di rendere il servizio disponibile (per soddisfare il principio di Discoverability!) ed il service consumer utilizzerà il broker (registro di servizi) per capire se esiste un servizio che fa al caso suo.

Dopo questa fase iniziale esistono diverse modalità di interazione tra i due: una modalità diretta per cui il consumer comunica direttamente con il provider una volta acquisite le informazioni necessarie dal broker oppure una interazione mediata per cui ogni richiesta da parte del consumer passa attraverso il broker.

In tutto ciò gioca un ruolo fondamentale il concetto di trasparenza, due tipologie:

- Platform Transparency: l'utilizzo del servizio deve essere possibile per qualsiasi piattaforma (sistema operativo etc..) e non si necessita di conoscere i dettagli implementativi che permettono l'esecuzione del servizio.

- Location Transparency: se il provider decide di spostare il servizio su una interfaccia di rete differente, i consumatori non necessitano di essere informati ma soltanto il broker

Si parla in questo senso di brokered communication in quanto al fine di garantire tale trasparenza deve essere il broker a gestire il tutto (es. se cambia interfaccia di rete sarà informato il broker che provvederà affinché il consumer possa continuare a fare richieste normalmente). Per utilizzare la brokered communication, è necessario introdurre il pattern di

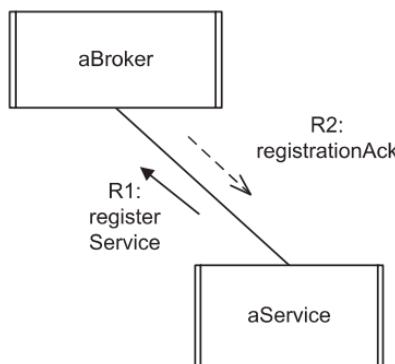
base che è il Service Registration Pattern (il provider deve registrare le informazioni sul servizio presso il broker).

Service Registration Pattern:

Rappresenta semplicemente una richiesta che il service provider fa al broker per registrare il proprio servizio tramite un messaggio dove il provider comunica nome del servizio, una descrizione e l'interfaccia di rete presso cui è disponibile.

I diagrammi di interazione UML per descrivere le interazioni tra oggetti sono: sequence diagram e collaboration diagram (il secondo usato in fase di progettazione). Manca una componente fondamentale nel collaboration diagram che invece è presente nel sequence diagram: l'ordine temporale, quindi alla specifica di ogni messaggio scambiato è associato un sequence number (R1 e R2 in figura) per ricostruire anche l'ordine temporale e passare al sequence. Inoltre in UML 2 il collaboration diagram è stato rinominato communication diagram.

Service Registration Pattern



Il servizio viene registrato presso il broker e una volta ricevuto l'ACK dal broker allora il servizio è stato aggiunto all' "archivio" del broker.

Una volta registrato il servizio i potenziali consumatori possono interagire con il broker e capire dalla descrizione se il servizio è di loro interesse o meno.

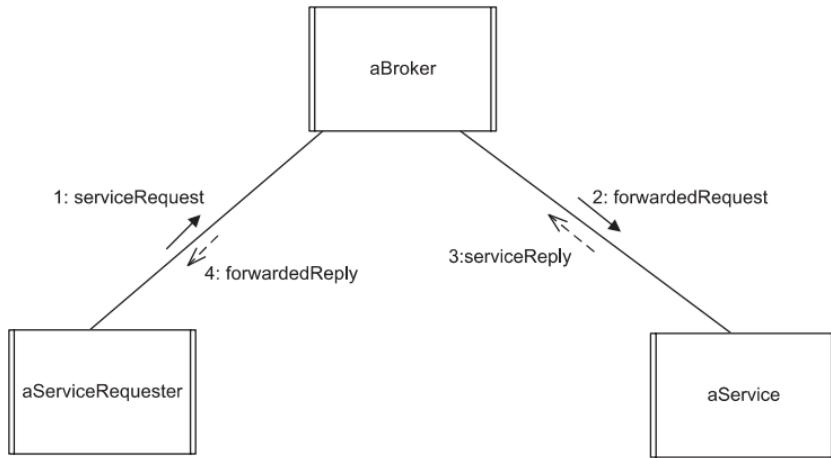
Broker Forwarding Pattern (pagine bianche del telefono):

Il broker si interpone tra provider e consumer anche dopo la fase iniziale per l'utilizzo del servizio.

Si assume che il consumer conosca il nome del service provider ma non sappia l'interfaccia di rete per usare il servizio, quindi lo chiede al provider.

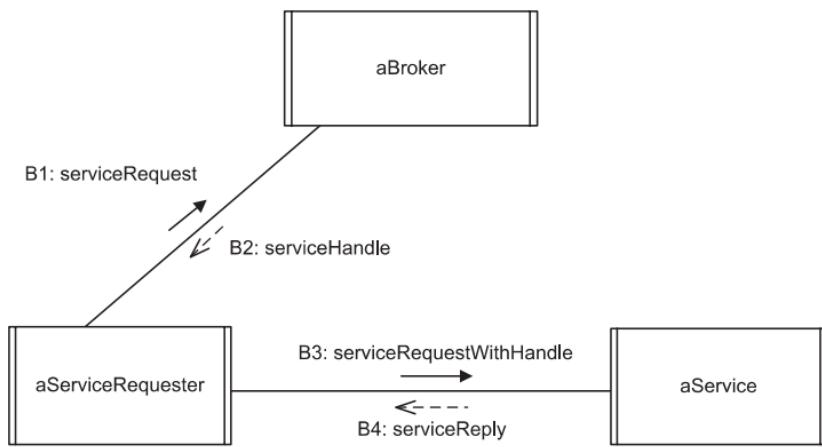
In particolare il client manda un messaggio identificando il servizio richiesto, il broker riceve la richiesta e recupera (in base alle informazioni presenti nel suo archivio dove sono registrati i servizi) l'interfaccia di rete presso cui il servizio è reso disponibile, inoltre quindi la

richiesta al servizio, prende la risposta e la inoltra infine al consumer.



Broker Handle Pattern (pagine bianche del telefono):

Il broker è inizialmente coinvolto per ottenere le informazioni della location del service provider, ma invece di inoltrare il tutto manda le informazioni al consumer affinché la successiva comunicazione sia diretta con il provider

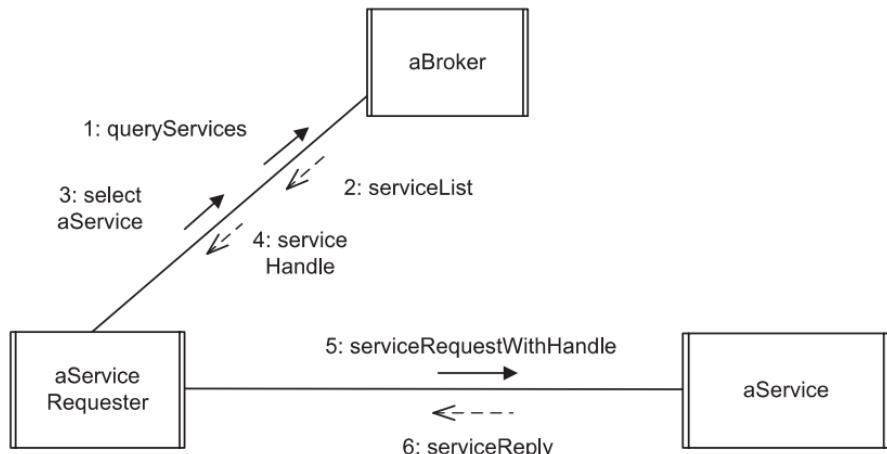


Il vantaggio nell'utilizzo del primo è che è garantita la location transparency: se cambia interfaccia di rete lo viene a sapere il broker che aggiornando il registro di servizi nasconderà automaticamente il tutto al consumer, non funziona con il secondo metodo. Il vantaggio del secondo però è che si scambiano meno messaggi, maggiore efficienza. Nelle pagine bianche conosco il servizio ma non la posizione di esso, nelle pagine gialle conosco il servizio che mi serve ma non lo specifico servizio.

Service Discovery Pattern (pagine gialle):

Il consumer richiede al broker una tipologia di servizio (queryServices), il broker restituisce una lista di servizi che soddisfano la richiesta (sempre tramite l'archivio interno dove i servizi sono registrati), il service requester sceglie il servizio da utilizzare. Da questo punto in poi si

può interagire nelle due modalità viste prima.



Ma in base a cosa potrei scegliere uno specifico servizio da una lista generica?
In base a vari criteri al di là della funzionalità, come QoS (quality of service).

Anche se le SOA sono indipendenti dalle piattaforme, usano i Web Services.
Un Web Service è un servizio che fa uso di protocolli standard internet e per lo scambio di dati (richieste, risposte etc..) usano protocolli XML (Extensible Markup Language).

Si introducono quindi degli standard:

SOAP (Simple Object Access Protocol): protocollo sviluppato dal W3C per permettere lo scambio di informazioni, questi fanno uso di XML come formato di serializzazione e di protocolli standard internet come HTTP e la relativa porta 80.

SOAP come detto è basato sul formato XML e consiste in tre parti:

- una "busta" che definisce un framework per descrivere cosa c'è nel messaggio e come processarlo,
- un insieme di regole per codificare i tipi di dato scambiati tra consumer e provider,
- un modo per rappresentare queste chiamate di procedura remota.

Quando si registra un servizio bisogna anche scrivere la descrizione del servizio.

Questo standard si chiama WSDL (Web Service Description Language), linguaggio ancora basato su XML, permette di fornire SOLO le informazioni necessarie del servizio (cosa fa, dove si trova e come invocarlo) tramite un documento WSDL al broker e quindi al potenziale consumatore (principio di discovery).

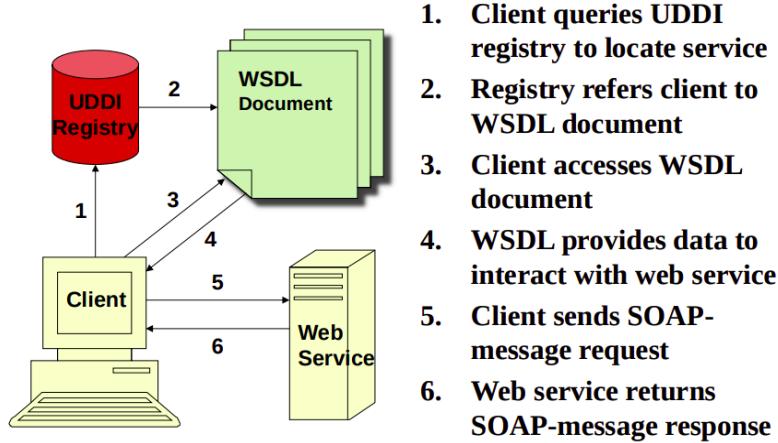
Il documento WSDL oltre alla descrizione contiene ovviamente anche il broker handle, il riferimento che il requester che il consumer può utilizzare per interagire direttamente con il provider.

Un registration service è fornito per far sì che i servizi siano disponibili ai clienti, i servizi vengono registrati con questo registration service. Per i Web services, è fornito un service registry che permette ai servizi di essere pubblicati e localizzati nel www.

UDDI (Universal Description Discovery and Integration): Per realizzare il concetto di service registry viene introdotto un particolare framework UDDI. Tale framework realizza quindi tutte le funzioni base che deve svolgere il broker.

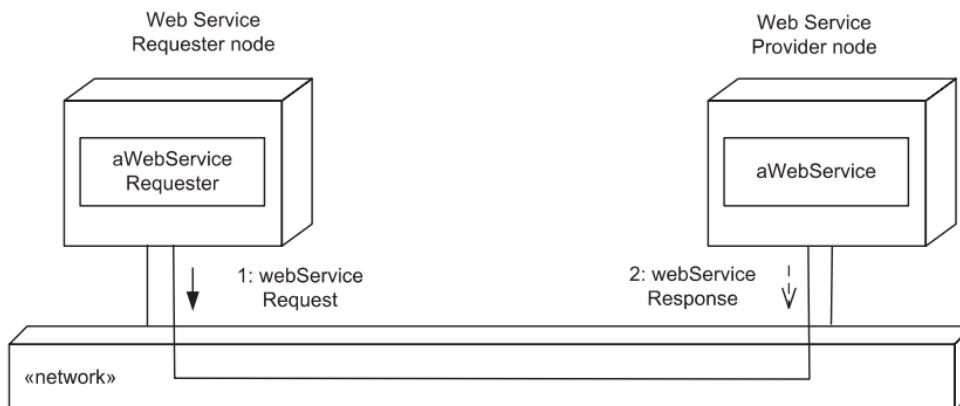
Quindi in generale quando parliamo di tecnologia implementativa a supporto di architetture service oriented si fa riferimento a come realizzare il broker, come descrivere il servizio e come interagire con il servizio, nel caso web services UDDI, WSDL e SOAP.

Tutto il processo riassunto in questa slide:



(ES: saltare)

L'uso del Web Services ha come idea di base di rendere disponibili i servizi tramite interfaccia di rete raggiungibile tramite una piattaforma web (non solo usando HTTP).

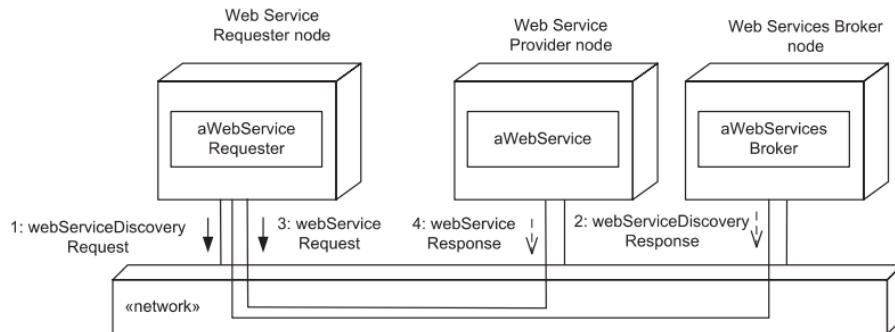


Qui un esempio di utilizzo della tecnologia Web Service. La richiesta viene veicolata al service provider e allo specifico servizio messo a disposizione tramite l'uso del protocollo SOAP.

Il diagramma in figura fa riferimento a un diagramma di implementazione, in particolare un deployment diagram dove si combina la descrizione della piattaforma di esecuzione e l'allocazione di componenti software su questi elementi della piattaforma.

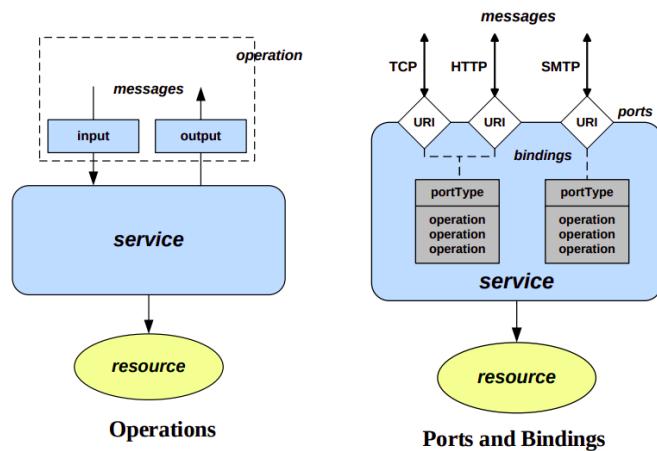
In particolare tre nodi (i parallelogrammi) che rappresentano nodi di esecuzione il primo quello che esegue il consumer (contenente l'oggetto in esecuzione, la componente awebbservice requester), il secondo del provider (contenente l'oggetto ... aWebService) e quello sotto quello che rappresenta l'interfaccia di rete che collega i due (basata ovviamente sul web e quindi protocolli internet, in particolare HTTP).

Qui un esempio di Web Service in cui interviene anche il broker su un altro nodo remoto, il consumatore chiede al broker informazioni sul servizio per ottenere eventualmente il documento WSDL e interagire quindi col provider.



(FINE ESEMPIO)

WSDL mette a disposizione solo gli elementi di base per utilizzare il servizio (quindi niente roba al di là dell'interfaccia tipo QoS), ossia l'insieme di operazioni che possono essere invocate (del tutto simile all'interfaccia pubblica di una classe che mi dice i metodi che posso invocare), la network location dove presso cui raggiungere il servizio ed inviare la relativa richiesta SOAP.



Un documento WSDL come si vede è diviso in due parti separate (livello di astrazione): A sinistra WSDL mette a disposizione le operazioni in termini di messaggi input e output (esiste in realtà anche il messaggio di fault oltre a input e output se errore). Questa parte descritta in XML.

A destra invece si mostra come le operazioni sono concretamente messe a disposizione sull'interfaccia di rete: es. la richiesta è inviata tramite HTTP e raggiunge uno specifico indirizzo di rete URI e lo strumento utilizzato per mettere a disposizione le informazioni a livello di interfaccia di rete cioè il portType.

Un portType è null'altro che un insieme di operazioni messe a disposizione su uno o più URI (endpoint).

Si utilizza maggiormente HTTP (sincrono) per condividere questi messaggi ma si potrebbero utilizzare anche altri protocolli come TCP o SMTP (asincrona).

Nel tempo sono stati introdotti anche ulteriori contributi che non sono alternativi alla tecnologia Web Service ma che si affiancano ad essa, uno di questi è REST (Representational State Transfer).

Questo più che essere una tecnologia implementativa (come WebService) è uno stile architetturale (come gestire l'architettura di un sistema software distribuito usando sempre protocolli standard internet, in particolare HTTP).

REST mette a disposizione un'interfaccia di rete (API) per accedere ad un insieme di risorse messe a disposizione sulla piattaforma web (queste risorse sono di diversi tipi e possono anche essere servizi web).

La cosa fondamentale è che in questo caso non si usa HTTP solo per garantire lo scambio di messaggi tra i partecipanti all'architettura distribuita, ma anche per veicolare queste interazioni.

Caratteristiche: client/server quindi un server mette a disposizione una risorsa con cui si può interagire facendo uso solo di HTTP, ambiente Stateless in quanto non si salva lo stato nell'interazione c/s, vengono però messe a disposizione delle cache per migliorare l'efficienza in una rete basata su REST, interfaccia uniforme in quanto le interazioni sono basate solo e soltanto sui 4 verbi di base HTTP ossia GET PUT POST (aggiornare) DELETE (CreateReadUpdateDelete),

Ogni risorsa deve essere identificata con URL specifico (o URI), inoltre le rappresentazioni delle risorse sono interconnesse usando gli URL.

Tutto si basa sulle risorse web e ogni entità distinguibile è risorsa (sito web, pagina html, documento xml, web service, etc...) e sono identificate da un URL.

La risorsa è tipicamente rappresentata facendo uso di un documento XML.

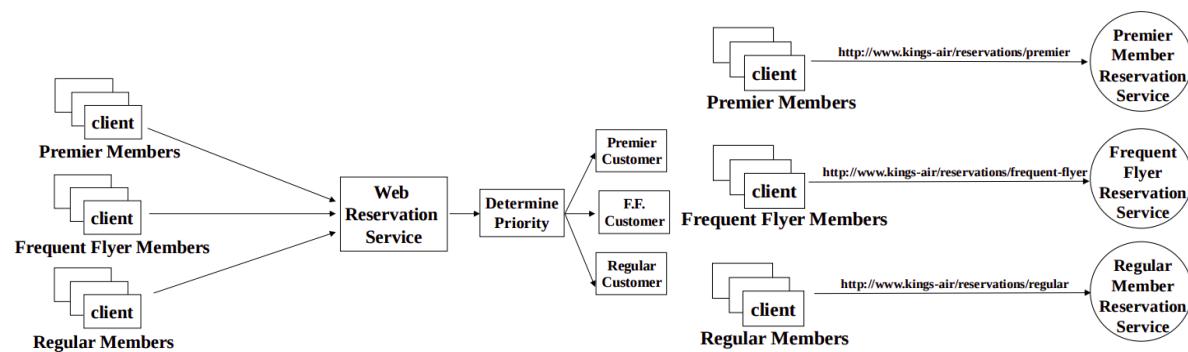
Per interagire con le risorse si utilizzano le RESTful API che usano le 4 operazioni CRUD HTTP in base al "contesto": se si deve prendere una Collection di risorse (es. /users nell'URL) o un singolo Item (es. /users/{id}).

POST può essere usato solo nelle collezioni per aggiungervi un item mentre

GET sia sulle collezioni che sui singoli item (nel primo caso per ottenere la lista di elementi e nel secondo il singolo elemento)

PUT e DELETE utilizzabili solo negli item.

Nell'approccio convenzionale si fa uso di un singolo URL per mettere a disposizione il servizio che offre più tipologie di accesso. L'approccio basato su REST invece fa uso di più URL. (conforme anche con Axiom 0 e non ha collo di bottiglia)



Software Architectural Transaction Patterns

Una Transazione rappresenta una richiesta effettuata da un client che contenga due o più operazioni, che però svolgono una singola funzione logica e devono essere completate interamente o per nulla.

Le proprietà di una transazione sono racchiuse nell'acronimo ACID Atomicity, Consistency, Isolation e Durability.

- Atomicità == la transazione pur essendo insieme di operazioni è vista come unità indivisibile di lavoro, o tutto (committed) o niente (rolled back)
- Consistenza == quando si esegue una transazione (sia di successo che non) il sistema deve essere in uno stato consistente
- Isolation == ogni transazione deve essere eseguita in modo isolato e quindi non essere compromessa da altre transazioni
- Durability == gli effetti prodotti dalla transazione sono permanenti e devono quindi sopravvivere anche ad eventuali system failures. (persistenza)

L'esempio tipico di una transazione è la transazione bancaria. (es. si vogliono trasferire i soldi al conto di un amico: le operazioni sono togliere i soldi dal mio e inviarli al conto dell'amico).

Per garantire le proprietà di questa transazione si utilizza il protocollo (anche in basi di dati) del Two Phase Commit Protocol. Questo protocollo coinvolge un coordinatore.

Vediamo come funzionano le due fasi.

Prima fase: Il coordinatore invia un messaggio di "preparazione al commit" ai servizi. Questi bloccano le risorse (Isolation), eseguono le operazioni e, se tutto va bene, inviano "readyToCommit" al coordinatore. Se manca il messaggio, la transazione è abortita.

Fase 2 (Commit): Se tutti i servizi sono "ready", il coordinatore invia il "commit". I servizi confermano l'operazione, liberano i lock e completano la transazione.

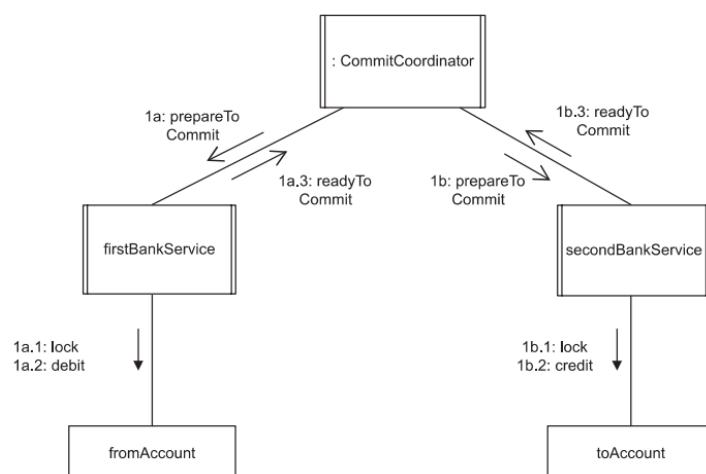
(ES: saltare)

Vi sono due servizi nella transazione di trasferimento bancario ognuno su un'interfaccia di rete differente: firstBankService per prelevare i soldi dal saving account e secondBankService per permettere il deposito del denaro sul secondo conto.

Per coordinare correttamente le attività svolte dai due servizi dobbiamo coinvolgere anche un altro oggetto che svolga il ruolo di coordinatore, il CommitCoordinator.

Vediamo come funzionano le due fasi.

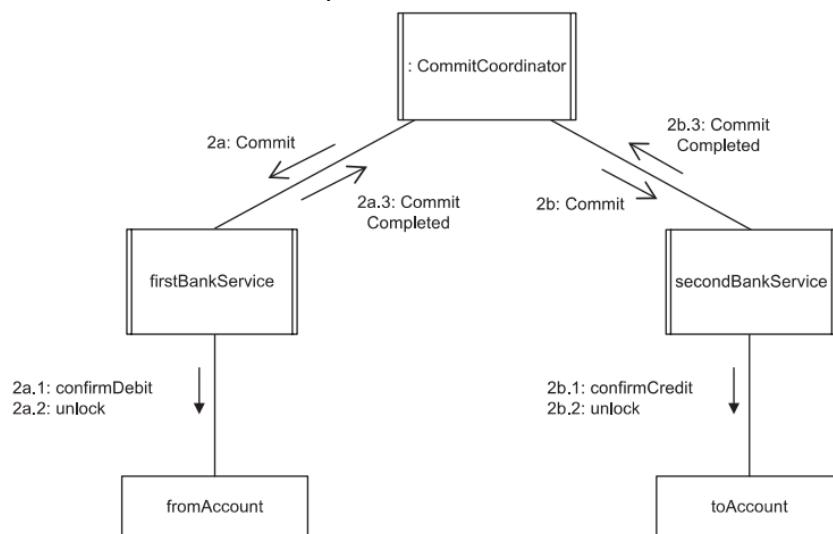
Prima fase:



La transazione è gestita in modo centralizzato per cui sarà il commitcoordinator a gestire l'ordine della transazione.

Il commitcoordinator invia un primo messaggio 1a e 1b ai due servizi coinvolti in cui gli comunica di prepararsi al commit. A fronte di questa richiesta i due servizi effettuano 1a.1 e 1b.1 un lock per garantire l'Isolation e quindi la Consistency (non si vuole che altre transazioni operino mentre i due servizi effettuano l'operazione sui due conti). A questo punto il firstBankService farà l'operazione di debito e l'altro di accredito. Se le operazioni vanno a buon fine i servizi inviano al coordinator un readyToCommit, in assenza di questo messaggio la transazione è abortita.

Se invece a buon fine si passa alla seconda fase:



Viene inviato commit dal coordinatore, confermata l'operazione di addebito e di accredito (2a.1, 2b.1), si libera il lock e viene completata la transazione.

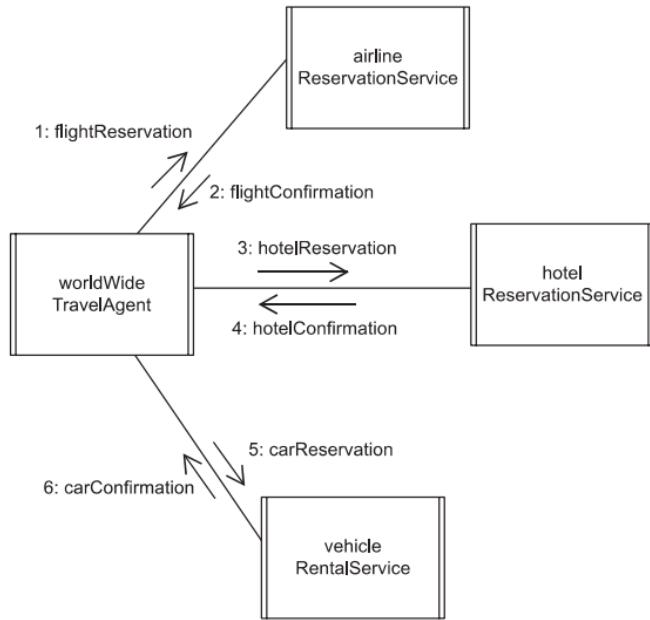
(FINE ESEMPIO)

Compound Transaction Pattern

La compound transaction è una transazione composita, ossia costituita da un insieme di sottotrasazioni, in caso qualcosa vada storto si cerca di salvare il salvabile, ossia se la seconda transazione non va a buon fine ma la prima si farà un rollback "pariziale", ossia rollback solo sulla seconda transazione.

(ES: saltare)

Per illustrare questo tipo di transazione facciamo un esempio concreto: un agente di viaggio deve pianificare il viaggio per un cliente: è necessario prenotare il biglietto aereo, poi prenotare l'albergo e la macchina a noleggio. Piuttosto che vedere questa compound transaction come operazione indivisibile per cui o tutto o niente posso vederla come costituita di tre sottotrasazioni in modo che se riesco a fare la prima ma non la seconda mi salvo la prima.



Qui il caso in cui tutte e tre le sottotransazioni hanno esito positivo. Nel caso in cui una singola sottotransazione non abbia esito positivo, si salvano le prenotazioni che hanno avuto successo.

(FINE ESEMPIO)

Long Living Transaction

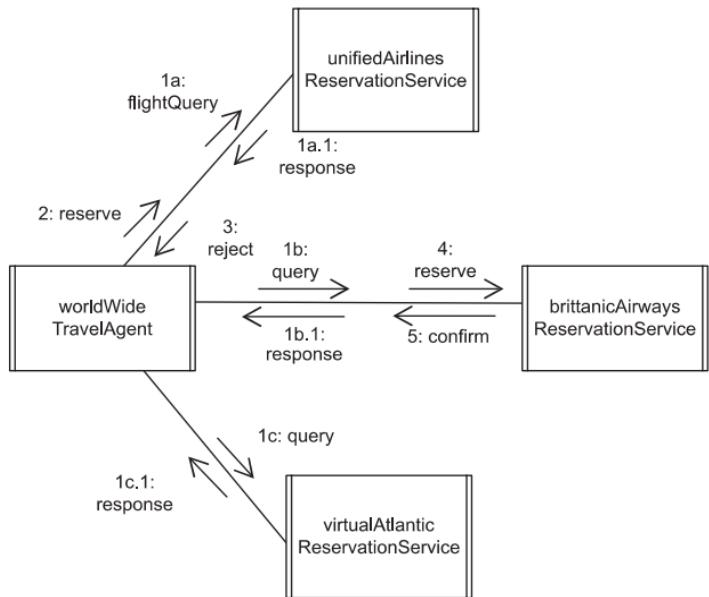
In queste transazioni si ha il **human in the loop** per cui si deve tener conto di possibili ritardi dovute alla decisione di persone coinvolte nelle operazioni.

Il pattern in questo caso prevede di organizzare la transazione distinguendo in essa due o più sottotransazioni separate proprio dall'elemento umano.

(ES: saltare)

Un esempio concreto può essere il considerare una prenotazione aerea in cui c'è il coinvolgimento di un essere umano, dove l'utente fa una ricerca per capire i posti disponibili in un certo volo, scegliere un posto disponibile e confermare la prenotazione. Per effettuare la scelta però l'utente umano impiega del tempo, e in questo periodo può succedere che il posto scelto dall'utente venga prenotato da qualcun altro. Quindi prima di riconfermare la

transazione quindi è necessario un recheck



Nell'esempio in figura il servizio che funge da intermediario tra utente e compagnia aerea (TravelAgent) chiede a 3 compagnie aeree (1a, 1b, 1c) di mostrare i posti disponibili in un certo volo.

Ottenuta la risposta l'utente decide di prenotare il posto in **unifiedAirlines**, (2. **reserve**), il servizio fa un recheck che ha esito negativo e quindi invia 3. **reject** all'utente che dovrà rivolgersi quindi ad un'altra compagnia aerea (in questo caso **britannic**) prenotando e in questo caso il recheck ha esito positivo e quindi la transazione ha successo.

(FINE ESEMPIO)

Negotiation Pattern

Anche chiamato Agent Based Negotiation in quanto subentra la figura di un servizio che lavora per conto dell'utente (l'utente si affida al servizio chiedendogli di far qualcosa al posto suo). Si ha quindi un client agent che lavora per conto dell'utente, egli interagirà con il service agent per soddisfare le esigenze del cliente.

Il service agent mostra una lista di offerte al client agent che si avvicinano a soddisfare la sua richiesta. A questo punto il client agent lavorando per conto dell'utente può decidere se rifiutare/accettare una richiesta etc... (da qui negoziazione)

In particolare il client agent svolge tre tipi di operazioni:

- Proposta di servizio (es. cerca volo roma new york a meno di 1000 euro)
- Richiesta servizio se a fronte dell'offerta (risposta alla proposta) del service agent è soddisfatto
- Rifiutare il servizio altrimenti

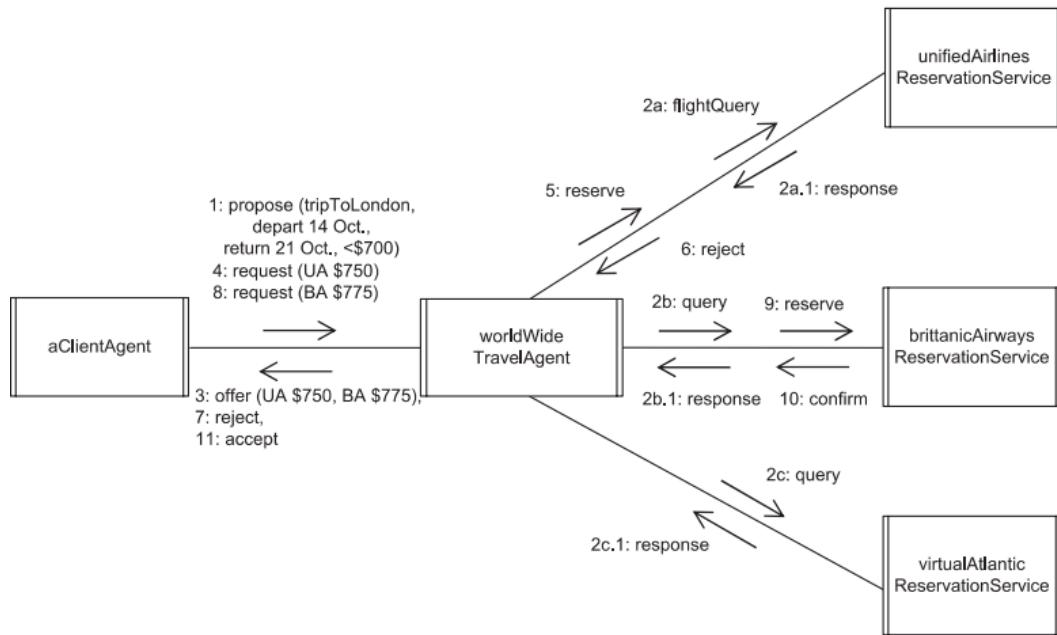
Invece dal punto di vista del service agent:

- Offrire un servizio di fronte alla proposta di servizio del client agent
- Rifiutare una richiesta/proposta del client agent in base alle disponibilità
- Accettare la richiesta/proposta del client agent.

La principale differenza tra richiesta e proposta è che la seconda è negoziabile (es. cerco il volo da roma a ny a meno di 1000 euro ma sono disposto a vedere anche offerte di prezzo lievemente superiori), mentre la richiesta è imperativa.

(ES: saltare)

riguardo la transazione per prenotazione posti vista prima si immagina un servizio che permetta, sapendo che devo andare a new york tot data, di trovare le offerte disponibili (senza che io utente vada a vedere il sito di ogni compagnia aerea).



In questo esempio 1. Il clientagent invia una proposta (negoziabile!) per viaggio a londra il 14 ottobre e ritorno il 21 che costi meno di 700 euro.

A fronte di questa richiesta il serviceagent si confronta con diversi servizi offerti da compagnie aeree inviando in modo concorrente le richieste (2a, 2b, 2c).

Il messaggio che torna al clientagent è una offer con le soluzioni migliori trovate dal serviceagent (unifiedairlines a 750 euro o britannic a 775).

Il client invia quindi un messaggio di richiesta (non negoziabile!) (4) per prenotare UA, ma nel frattempo biglietto già acquistato da qualcun altro quindi reject dalla compagnia al serviceagent e reject dal serviceagent al clientagent (7).

Allora il clientagent richiede quello a 775 di BA, in questo caso la reserve da parte del serviceagent ha successo -> il serviceagent risponde con accettazione finale alla richiesta non negoziabile del clientagent.

(FINE ESEMPIO)

Service Interface Design

I servizi sono quindi elementi che interagiscono tramite scambio di messaggi, ed è quindi fondamentale come visto saper gestire la logica di controllo delle applicazioni.

Per progettare correttamente l'interfaccia dei servizi, si usa gli stessi criteri che si utilizzano per le classi.

Possiamo sfruttare UML 2 e in particolare il concetto di classi strutturate per progettare un servizio come se fosse una classe avente un'interfaccia, eventualmente da ulteriori servizi al suo interno.

Quindi da una parte progettazione del servizio a livello strutturale, dall'altra a livello comportamentale (dinamica attraverso cui questi servizi interagiscono), dove una parte fondamentale è come coordinare questi servizi.

Service Coordination

Due tipi di coordinamento:

Orchestrazione: si ha un elemento centralizzato che coordina l'esecuzione dei servizi (simile a quanto visto nel caso dei two phase commit protocol).

Coreografia: gli aspetti di coordinamento sono decentralizzati (distribuiti).

Nelle SOA reali, tipicamente si usano entrambi gli approcci. I pattern delle transazioni possono essere usati per la coordinazione tra servizi.

Progettazione dettagliata OOD

Nel nostro caso avendo usato un approccio OO fin dall'analisi dei requisiti parleremo di una sottofase di progettazione dettagliata che farà uso di un approccio OO -> Detailed OOD.

Raffina la fase OOA, concentrandosi sulla collaborazione degli oggetti per fornire servizi attraverso casi d'uso e operazioni.

I casi d'uso sono stati introdotti con l'OOA ma sono realizzati tramite collaborazioni in OOD.

La collaboration è costituita da due parti fondamentali:

- una parte comportamentale: rappresenta la dinamica che mostra come gli elementi collaborano tra loro. Si definirà facendo uso dei communication diagrams.

- una parte strutturale: rappresenta gli aspetti statici della collaborazione ed è definita facendo uso del class diagram, usando i composite structure diagram.

Uno degli aspetti più importanti di cui tener conto in fase di progettazione a livello comportamentale è la gestione del controllo. Questa parte ricade nel Control Management.

In una stratificazione corretta delle classi devono esserci oggetti boundary che si devono occupare solo di catturare le richieste dell'utente e inoltrarle agli oggetti di controllo, che conoscono la logica applicativa e interagiranno con gli oggetti entity (oggetti che mantengono le informazioni) al fine di soddisfare la richiesta. Si introduce quindi un oggetto di controllo che si occupi della logica di esecuzione.

(ES: saltare fino a pag 106)

Iniziamo con un esempio tornando al sistema di iscrizione all'università.

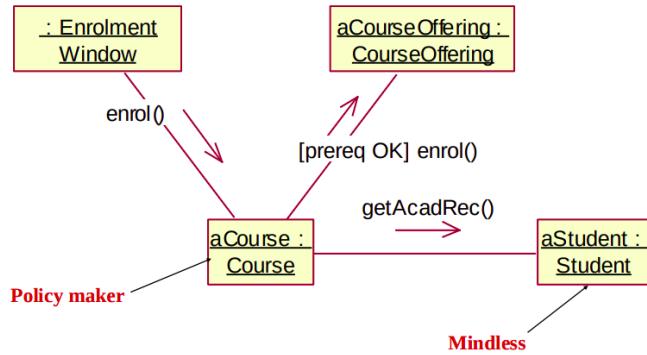
Vediamo come gestire il controllo nel caso in cui si voglia aggiungere uno studente a una specifica courseoffering.

Prima di poter iscrivere lo studente al corso si devono identificare gli esami propedeutici e verificare che lo studente li abbia superati.

Gli oggetti entity di nostro interesse per questa interazione sono classe Studente, Courseoffering e Course (course memorizza le informazioni legate a un corso, courseoffering riguarda un corso istanziato in uno specifico semestre ed anno, e gli esami propedeutici stanno in Course).

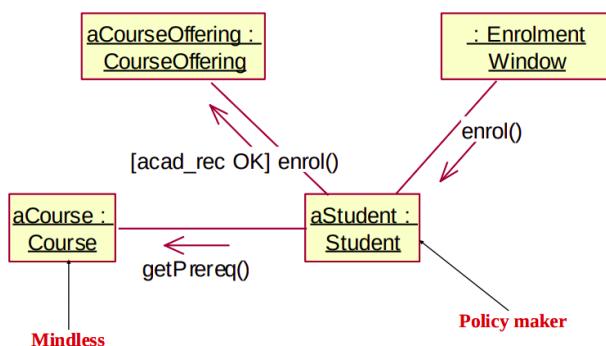
Lato interfaccia utente invece si ha l'oggetto boundary che riceve la richiesta dell'utente (segreteria studenti) di iscrivere lo studente al corso.

Diverse soluzioni per il control management:

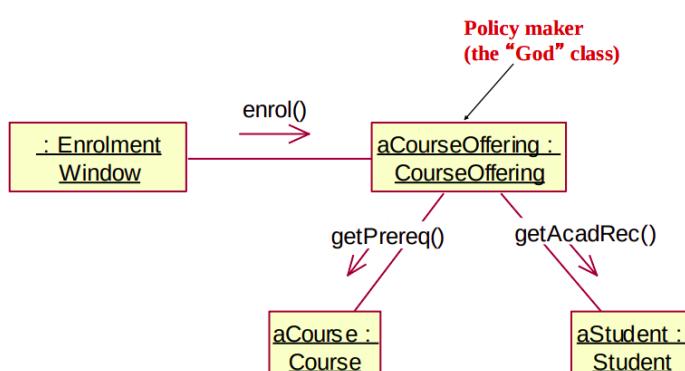


Communication diagram dove l'oggetto enrolment window (boundary) inoltra la richiesta di iscrizione all'oggetto corso, che chiede all'oggetto studente qual è il suo curriculum. Se si verifica che lo studente ha superato i corsi propedeutici allora conferma l'iscrizione inoltrando il messaggio all'oggetto courseoffering, che iscriverà lo studente alla specifica istanza di corso.

Qui chi gestisce il controllo è l'oggetto di corso (Policy Maker) mentre student è mindless perché risponde semplicemente alla richiesta di corso.



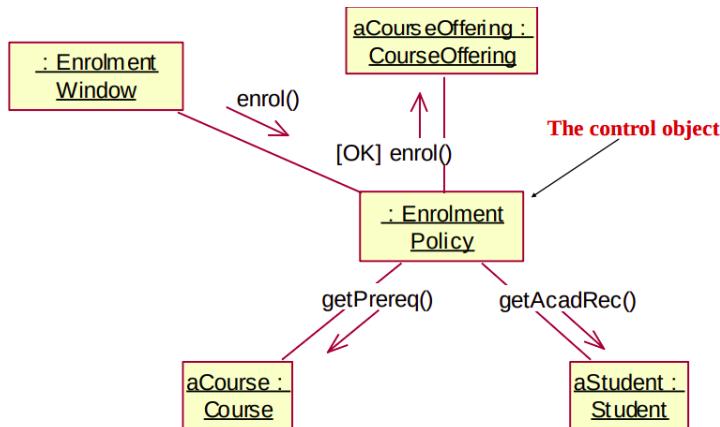
La seconda soluzione prevede invece che la richiesta sia inanzitutto inoltrata allo studente, che chiede al corso quali sono gli esami propedeutici che deve aver passato e se tutto ok (student fa controllo interno per vedere se li ha superati) inoltra a courseoffering. (si invertono i ruoli di controllo)



Si centra il tutto sull'oggetto CourseOffering che diventa classe God in quanto decide tutto. È infatti courseoffering a chiedere a course gli esami propedeuitici e a student i risultati per capire se li ha passati o meno.

In una stratificazione corretta delle classi devono esserci oggetti boundary che si devono occupare solo di catturare le richieste dell'utente e inoltrarle agli oggetti di controllo, che conoscono la logica applicativa e interagiranno con gli oggetti entity (oggetti che mantengono le informazioni) al fine di soddisfare la richiesta.

Ma in questi casi sto dando responsabilità di controllo a un oggetto entity -> non rispettano la stratificazione BCE.



Si introduce quindi un oggetto di controllo che si occupi della logica di esecuzione. Corso fornisce gli esami propedeutici, student la carriera universitaria e courseoffering interpellato solo eventualmente per iscrivere lo studente, si garantisce che le classi entity si limitino ad avere le informazioni senza controllare.

(FINE ESEMPIO)

Isolando la logica di controllo in una classe apposita molti vantaggi tra cui a livello di manutenibilità (eventuali modifiche sul come gestire le iscrizioni sono limitate a questa classe senza dover cercare una classe che si occupasse anche di quello).

In questo caso (così come visto nelle applicazioni service oriented) gioca un ruolo fondamentale il coupling (grado di accoppiamento). Il più desiderabile è l'Intra-Layer Coupling: si vogliono evitare delle dipendenze tra elementi appartenenti a diversi layer applicativi (con layer applicativi si intende BCE).

Si vuole favorire quindi l'Intra-Layer coupling ossia l'interazione all'interno di uno stesso strato piuttosto che l'interazione tra strati differenti.

Si vuole quindi minimizzare l'Inter-Layer Coupling, e uno strumento utile per farlo è la Legge di Demeter.

La Legge di Demeter (anche nota come "don't talk to strangers" in quanto si basa sull'idea di non "comunicare" con oggetti non noti).

Essa afferma che un metodo può inviare messaggi (cioè invocare metodi) solo ai seguenti oggetti:

- 1) L'oggetto stesso (un metodo deve poter invocare i metodi su se stesso, es. usando this in Java e C++)
- 2) Oggetti passati come argomenti nel metodo.
- 3) Oggetto elencato tra gli attributi dell'oggetto stesso (strong law -> attributi ereditati non possono essere usati)
- 4) Un oggetto creato dal metodo
- 5) Un oggetto che fa riferimento a una variabile globale

Come detto la Legge di Demeter favorisce quindi manutenibilità ma anche comprensione del codice.

UML Structured Class

Mentre in UML 1 la classe è intesa come semplice aggregato di dati e operazioni, in UML 2 si mantiene lo stesso simbolo (forma rettangolare) ma diventa “classe strutturata” in quanto contiene elementi detti roles o parti che formano la sua struttura e ne descrivono il comportamento.

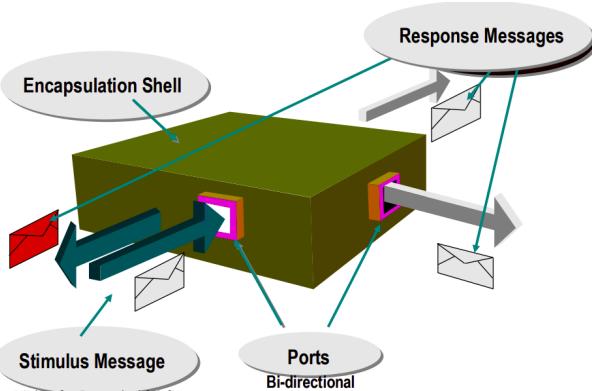
Questo meccanismo è gerarchico in quanto un ruolo/parte di una classe strutturata può essere a sua volta classe strutturata, ciò permette di gestire la complessità in modo stratificato lavorando a diversi livelli di astrazione.

Ogni ruolo rappresenta un elemento partecipante nella realizzazione della struttura interna della classe, e questi ruoli sono interconnessi attraverso il concetto di connettore.

Esiste una differenza tra ruoli e parti: i ruoli seguono una semantica “per riferimento” mentre le parti “per valore”.

In questo senso si utilizza la classe strutturata per definire i building blocks di un’applicazione, nascondendo i dettagli implementativi.

In particolare vi è un incapsulamento molto stretto del comportamento, per cui ogni interazione tra elementi di classi diverse deve avvenire attraverso un meccanismo basato su messaggi.



Si vede quindi concettualmente la classe strutturata come una scatola nera che mette a disposizione servizi (operazioni elencate nell’interfaccia pubblica della classe) nascondendo l’implementazione delle operazioni agli utilizzatori della classe.

Per utilizzare i metodi messi a disposizione dalla classe si utilizzano delle Porte, attraverso le quali inviare messaggi e riceverne eventualmente indietro.

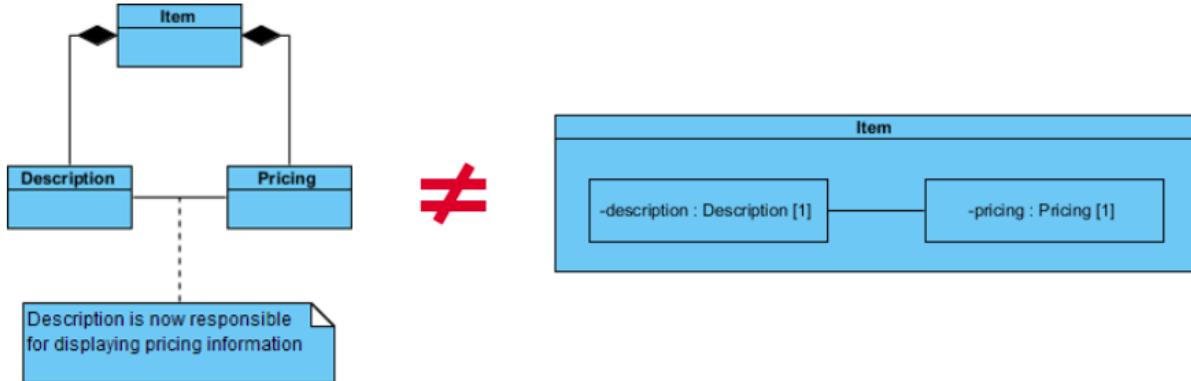
Inoltre all’interno della classe strutturata possono esservi altre classi strutturate, altre classi semplici etc...

Internamente poi deve essere definito come viene realizzata la classe strutturata sia in termini strutturali che comportamentali. Per la parte comportamentale si utilizzano spesso macchine a stati finiti (come la classe evolve durante il suo funzionamento transitando da uno stato iniziale a uno finale).

In questo senso quindi ogni classe può essere vista come un elemento di progettazione autonomo, molto utile in fase di progettazione perché quindi posso suddividere il team per lavorare su classi diverse partendo dall’interfaccia definita in fase di analisi dei requisiti, utile

anche a livello di testing (si testa a livello di unità Unit-Testing fino alla verifica del funzionamento dell'applicazione quando queste unità sono integrate, integration testing).

Class Diagram vs Composite Structure Diagram



Con il class diagram e composition: si rappresenta come item sia costituito da due oggetti interni, uno che descrive l'elemento e l'altro che fornisce informazioni sul prezzo -> se cancello l'oggetto item cancello anche gli oggetti description e pricing in esso contenuti.
Riguardo la classe strutturata invece metto nella classe item due parti: una che è istanza di description e una che è istanza di pricing.

Quindi la classe strutturata rappresenta un meccanismo migliore per rappresentare informazioni tipiche di sistemi software di grandi dimensioni, dove gli elementi vengono rappresentati in modo gerarchico.

In UML 1 il concetto di “componente” descrive un componente fisico, e mancava totalmente il passaggio da classi a livello di progettazione a eseguibile.

Con UML 2 invece, dato che le classi sono descritte in dettaglio tramite classi strutturate e dato che il concetto di “componente” ha subito un cambiamento sostanziale è possibile descrivere il passaggio da progettazione a software effettivo da eseguire.

Questo passaggio viene fatto anzitutto descrivendo la piattaforma di esecuzione sottostante.

Configurazione della piattaforma

La configurazione della piattaforma definisce come le funzionalità hardware/software sono distribuite sui vari nodi fisici dove viene eseguito il sistema.

Questo viene ottenuto attraverso due passi:

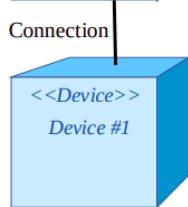
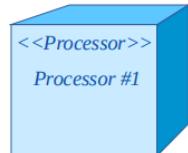
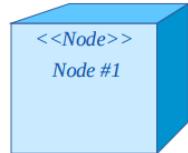
- si definisce la piattaforma hardware sottostante attraverso il Deployment Diagram in UML.
- si descrive come sui nodi fisici di esecuzione vengono allocati i componenti software ricavati durante la fase di progettazione.

Questi componenti software non si chiamano più “components” come in UML 1, ma artefatti.

Deployment Diagram

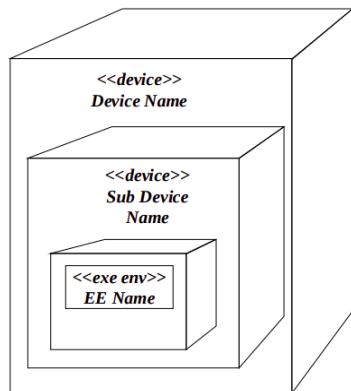
Per quel che riguarda il deployment diagram ogni nodo di esecuzione è rappresentato da un parallelogramma, esistono tre tipi di nodi: può rappresentare o una risorsa computazionale in grado di eseguire il software, o dei nodi processor che eseguono il sistema software, oppure dei “support device” che non hanno capacità di elaborazione e sono controllati dal processore (es. switch di rete).

Le connessioni sono rappresentate mediante archi che collegano i nodi.

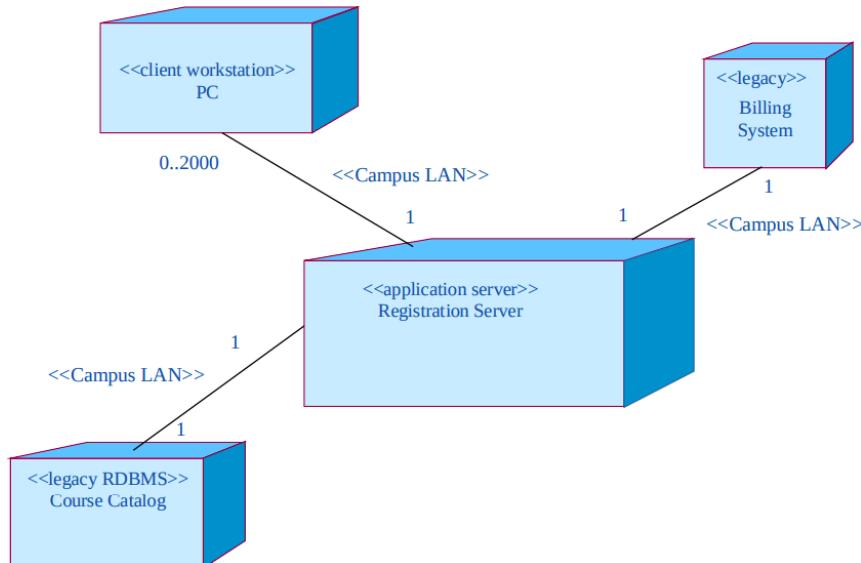


Come detto quindi un nodo rappresenta una risorsa computazionale che ha tipicamente capacità di elaborazione e memoria:

Ne esistono due tipologie principali: Nodi Device se rappresentano proprio la risorsa fisica con capacità di processing o Execution environment se rappresenta particolari piattaforme di esecuzione allocate nel nodo.



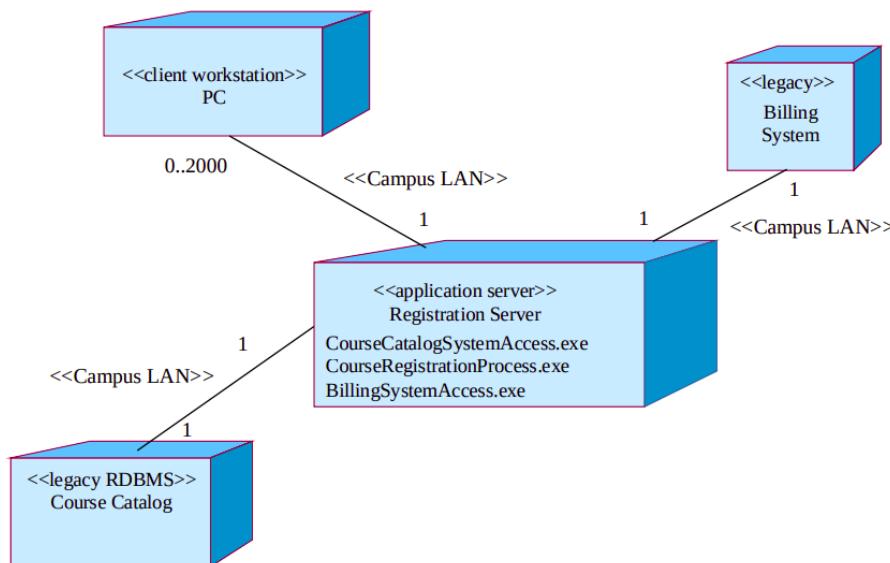
Riguardo i Connector invece rappresenta una connessione tra due nodi del diagram. Oltre alle connessioni si possono utilizzare anche le molteplicità per descrivere quanti nodi partecipano a una connessione nel deployment diagram.



Process-to-Node Allocation

si occupa di come assegnare i vari processi ai dispositivi hardware in esecuzione. Per farlo si tiene conto di vari aspetti:

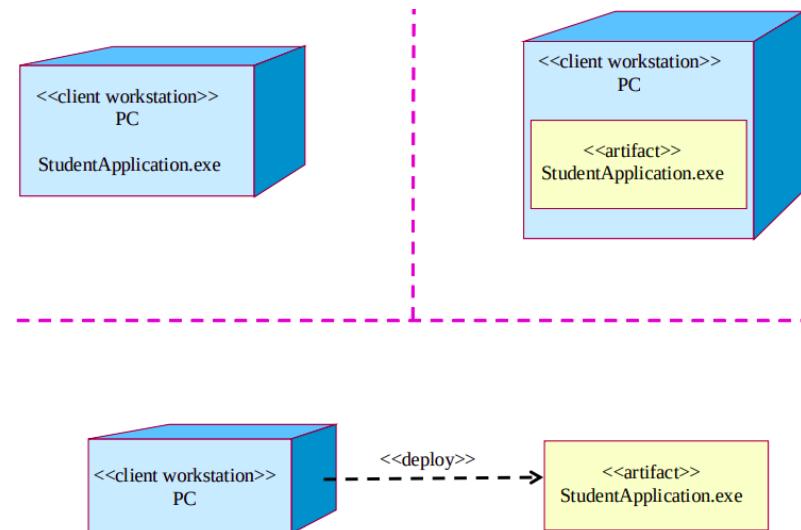
- Pattern di Distribuzione (il carico deve essere distribuito adeguatamente di modo da evitare colli di bottiglia)
- Si vuole trovare un'allocazione che minimizzi i tempi di risposta e aumenti throughput
- Minimizzazione del traffico attraverso la rete (si vorrebbe che processi comunicanti spesso tra loro siano sullo stesso nodo o su nodi vicini per minimizzare traffico)
- In base alla capacità dei nodi (CPU, RAM, spazio)
- In base alla larghezza di banda del mezzo di comunicazione
- In base all'availability dell'hardware e delle connessioni (se connessioni instabili ne devo tener conto per l'allocazione dei processi)
- In base ai Rerouting Requirements (se un nodo fallisce o una connessione si interrompe, il sistema deve poter riallocare i processi)



Si inseriscono quindi i processi nei vari nodi di esecuzione (es. il processo che permette la registrazione e l'accesso al catalogo corsi e al sistema di billing garantendo interoperabilità)

L'attività di Deployment rappresenta ciò che mi permette di assegnare/mappare gli artefatti sui nodi fisici durante l'esecuzione.

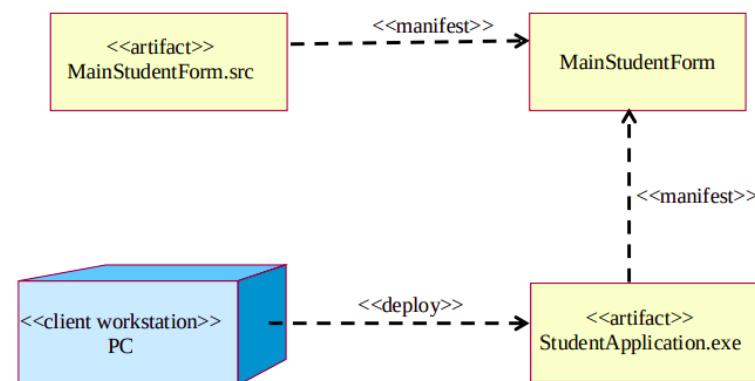
L'artefatto software rappresenta quindi l'entità su cui può essere fatto il deployment verso il nodo fisico, e questi artefatti modellano le entità fisiche che in UML 1 erano rappresentate attraverso il costrutto componente (quindi componente rimpiazzata in UML 2 da artifact). Tra gli artefatti si hanno quindi file, eseguibili, tabelle database, pagine web etc...



Qui sopra tre esempi di come gli artefatti possono essere disposti (deployment) sui nodi, tre modi.

La Manifestazione è una relazione tra un elemento di un modello e il relativo artifact che implementa il modello. Sono tipicamente implementati come set di artefatti e anche in questo caso sono file sorgente, eseguibili, documentazione, ...

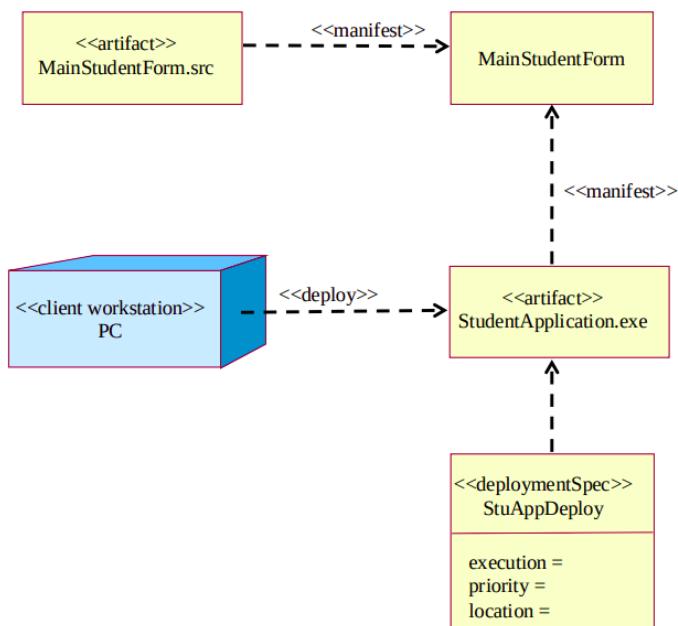
Questi legami di deployment possono anche essere arricchiti di informazioni per specificarne meglio il significato attraverso il Deployment Specification.



(ES: saltare fino a pag)

L'artefatto MainStudentForm/src è codice sorgente manifestazione della classe (elemento di modello) MainStudentForm, ciò viene ancora una volta descritto tramite associazione generica con relativo stereotipo.

Una volta compilato il codice sorgente diventa eseguibile (.exe), che a sua volta è manifestazione di MainStudentForm. Infine il deployment avviene per quel che riguarda proprio questo eseguibile.



L’Oggetto <<DeploymentSpec>> permette di specificare informazioni del tipo come eseguire l’artefatto, dove eseguirlo e la priorità.

L’utilizzo della deployment specification e di questi parametri è utile per arricchire il modello di informazioni utili e per sistemi automatici che permettono automaticamente di effettuare il deployment su nodi di elaborazione.

SOA Casestudy (ES: saltare fino a pag 120)

Parliamo di un software da utilizzare per fare online shopping, partendo da un insieme di requisiti astratti, analisi dei requisiti e poi progettazione effettiva. Il cliente fornisce dettagli personali, memorizzati nel customer account. Se il cliente effettua un ordine e la carta di credito è valida, un ordine di consegna è inviato al fornitore che controlla che il prodotto sia presente nell’inventario, in caso positivo conferma l’ordine e dice al cliente la data consegna prevista. Quando l’ordine è in consegna il cliente è notificato e la carta di credito addebitata.

Si identificano due attori principali che interagiscono con il software: il cliente e il fornitore. Ricordando che ogni caso d’uso deve essere a funzionalità completa (include anche eventuali flussi alternativi etc..), indipendente dagli altri e identificato come tale solo se attivato da almeno un attore, per identificarli lato cliente mi chiedo come il cliente interagisce con il software.

Tre casi d’uso: Sfoglia il catalogo di prodotti, può sottomettere un ordine e se ha sottomesso un ordine può visualizzarne lo stato.

Lato fornitore invece: processare l’ordine richiesto dall’utente (per verificare disponibilità inventario, carta di credito etc..) e confermare la spedizione e addebitare la carta del cliente. Sappiamo che i diagrammi di casi d’uso sono utilizzabili a diversi livelli di astrazione, qui alto livello e infatti mancano le frecce tra le associazioni (implicite).

Nell’activity diagram come sappiamo nodo iniziale e finale, l’iniziale corrisponde all’evento di attivazione da parte dell’attore e inizia la specifica di caso d’uso per portarlo a buon fine: nell’esempio la specifica del caso d’uso per sfogliare il catalogo di prodotti; dalla richiesta del

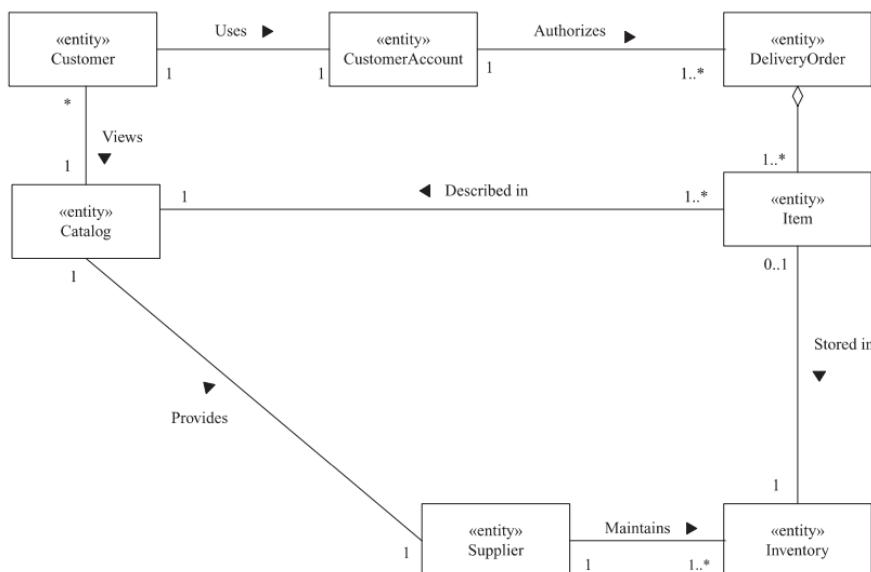
catalogo e i suoi item al mostrarli a schermo con il relativo prezzo. Facciamo la specifica per ognuno dei casi d'uso individuati.

In questo caso per il caso d'uso di richiesta ordine tornano i nodi decisionali di branch, dove seguiamo uno specifico arco solo se la condizione che lo identifica si verifica.

Partiamo da un class diagram chiamato context class diagram, dove sfruttando l'approccio gerarchico permesso dalle classi strutturate (UML 2) definiamo l'intera applicazione con una singola classe strutturata che viene annotata con lo stereotipo <<software system>>. Si sfrutta quindi la classe Online Shopping System per rappresentare l'intero sistema. Questo approccio gerarchico come già accennato l'altra volta ricorda molto i DFD Data Flow diagram, dove si partiva da una rappresentazione del software dal punto di vista del flusso dei dati usando un solo processo, che poi man mano veniva descritto in modo più dettagliato.

Facciamo lo stesso in questo caso: partendo dal context class diagram vediamo sempre più nel dettaglio il blocco Online Shopping System per capire cosa contiene.

Per farlo, in modo similare a quanto fatto nel primo modulo, analizziamo i requisiti per determinare anzitutto le classi entity, le associazioni e cardinalità.



Queste entità sono ricavate sia secondo il metodo dei sostantivi (approccio noun-phrase) dai requisiti sia dallo user case diagram, per cui sappiamo che i due attori supplier e customer devono essere inseriti come classi. Per le associazioni osserviamo come si espliciti anche un verso, ciò serve a facilitare la lettura del diagramma. Poi vabbé le cardinalità, fissato un cliente questo ha associato un solo catalogo, fissato un catalogo ha associati 1...N (*) clienti etc... Poi associazione 1 a 1 tra cliente e account cliente, ricordiamo che le associazioni 1:1 potrebbero essere anche fuse in una singola classe (una singola istanza è legata a una singola istanza in entrambe le direzioni). Poi a destra in alto DeliveryOrder vediamo che viene usato il simbolo del rombo vuoto, esso ricordiamo rappresentare una relazione di contenimento aggregation (composition rombo pieno per cui existence dependency se cancello il contenitore cancello anche il contenuto, aggregation rombo vuoto per cui semantica di riferimento, l'oggetto contenuto continua a esistere perché se cancello contenitore cancello solo il riferimento).

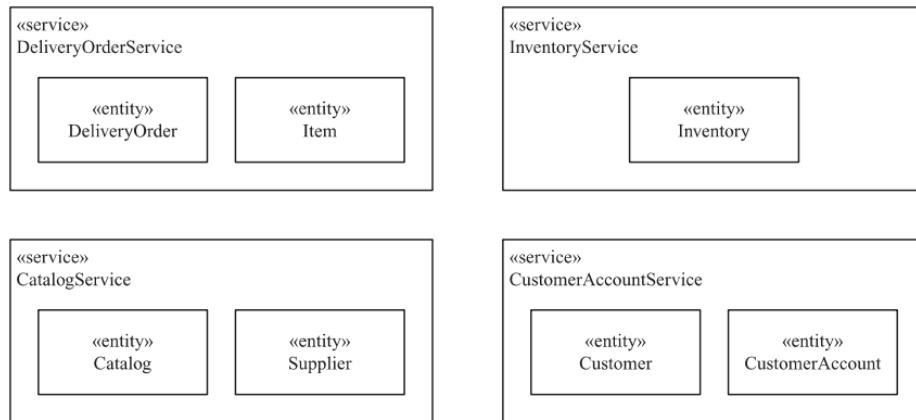
Perché in questo caso deliveryorder contenitore di item come aggregation? Perché

ovviamente i prodotti ordinati se l'ordine è cancellato potrebbero essere ordinati da altri clienti.

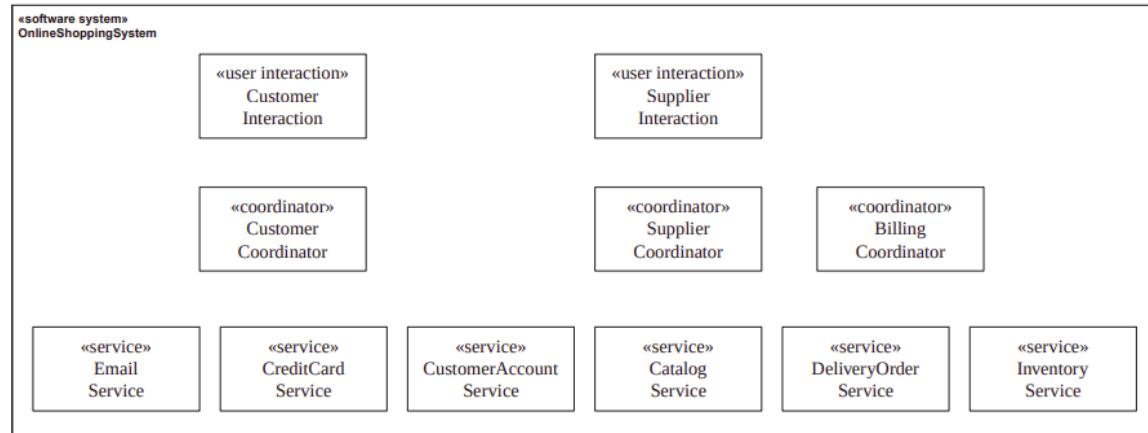
Per ogni classe entity definiamo ora i relativi attributi. Da qui in poi iniziamo a differenza della prima parte del corso a introdurre il concetto di classe strutturata grazie all'uso di UML 2.

Essendo un'applicazione service oriented, le entity classes sono integrate nell'architettura utilizzando classi note come service classes.

Si delineano 4 service classes:



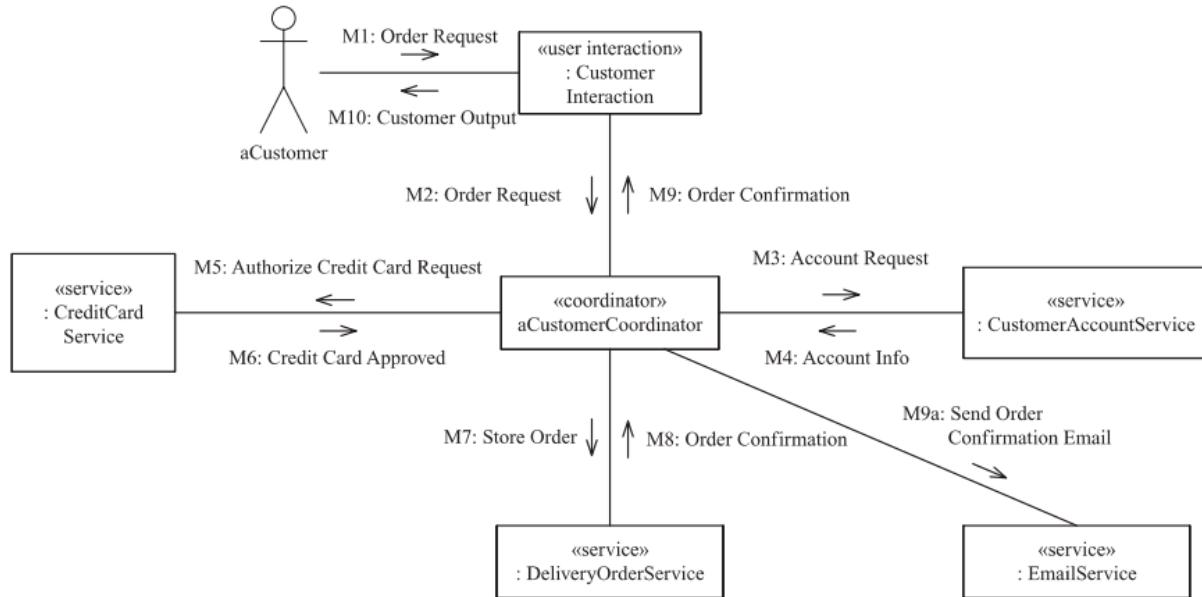
Esse permettono l'accesso alle entity classes. Tuttavia le classi service non sono sufficienti per la nostra applicazione, in quanto esse permettono l'accesso alle classi entity ma non avremo solo entity, ma anche boundary e control.



Si parla quindi di altre classi: User Interaction Classes (per Boundary) necessarie per l'interazione con gli utenti esterni (in particolare customer interaction e supplier interaction, quindi una classe per attore) e Coordinator Classes per gestire il coordinamento per l'invocazione delle service classes (in questo caso customer coordinator per gestire le richieste del cliente e supplier coordinator per le azioni del supplier, infine un billing coordinator per coordinare l'attività di pagamento per l'ordine da parte del cliente). A questo punto la classe strutturata identificata inizialmente come Online Shopping System, che rappresentava l'intero software, è descritta in modo molto più dettagliato.

Due classi aggiuntive: una per il pagamento via carta di credito e una per inviare i messaggi via mail (ciò suggerisce come si utilizzi due servizi esterni per integrare ciò, i nostri 4 dovremmo implementarli mentre gli altri due li useremo come servizi esterni blackbox tipo gmail e paypal, senza che li implementi ex novo).

A questo punto torniamo al modello dei dati, bisogna infatti specificare come le funzionalità descritte nel diagramma delle attività vengono realizzate in termini di interazione tra oggetti. Si usa poi il Dynamic Modeling, dove invece che usare i sequence diagram come fatto nella prima parte del corso utilizzeremo i communication diagrams (ex collaboration diagrams). Non vi è in questo caso la lifeline tipica dei sequence diagram, non è rappresentata in modo esplicito la sequenza temporale di come sono scambiati i messaggi ma si può tranquillamente ricostruire visto che ad ogni messaggio scambiato è associato un sequence number.



Si noti comunque come l'approccio BCE sia sempre rispettato: l'attore interagisce con una user interaction class che interagisce con il coordinator che si occupa di prendere i dati dalle service classes.

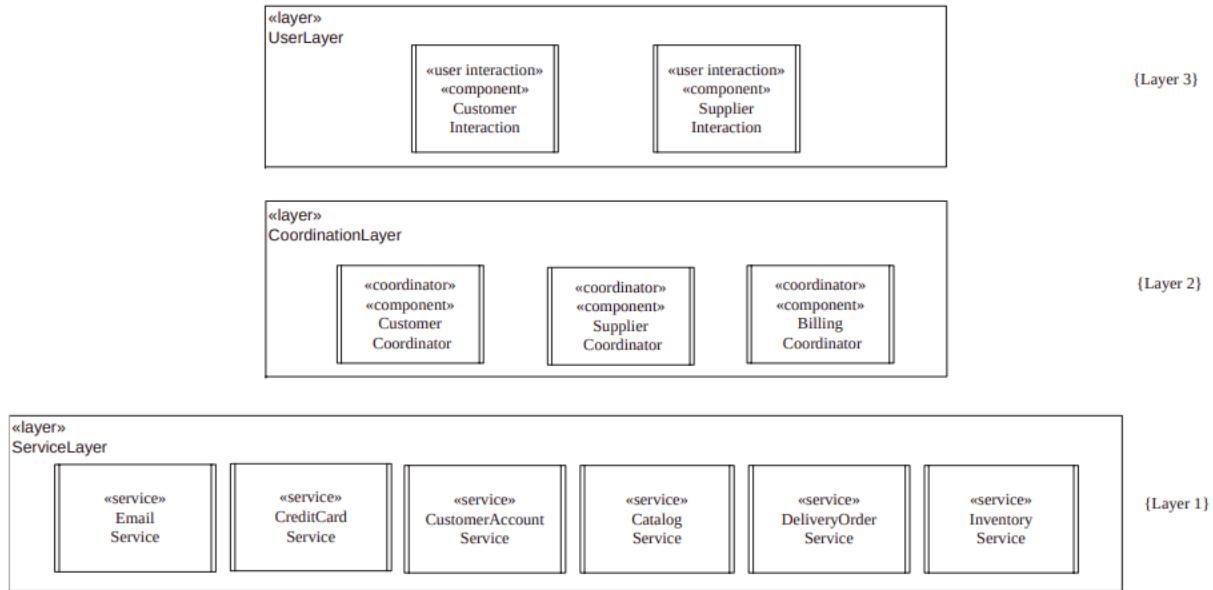
Ricordiamoci come CreditCard Service ed Email Service sono classi che non verranno ulteriormente sviluppate in quanto servizi sviluppati da altri, sarà definita solo l'interfaccia necessaria per poterli utilizzare.

Si ha nelle slide un communication diagram per ogni caso d'uso, in ognuno si rispetta la stratificazione dettata da BCE (boundary può interagire solo con boundary o control, control solo con control o entity, entity solo con entity o control).

A questo punto passiamo alla parte di progettazione (Design Modeling), dove andremo più nel dettaglio relativamente agli elementi che dobbiamo progettare ed implementare.

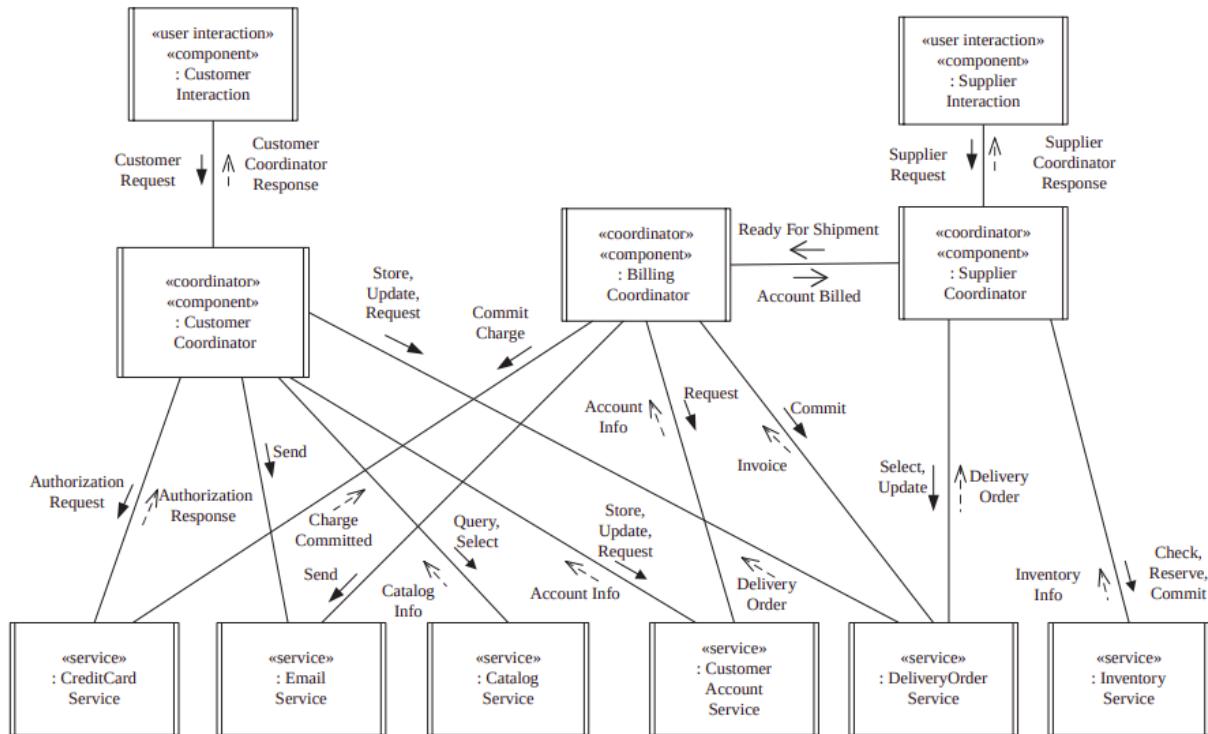
Dobbiamo anzitutto identificare i vari componenti, da strutturare in modo opportuno, per progettare l'applicazione service oriented. Un componente sappiamo essere un elemento basato su un principio di netta separazione tra interfaccia e implementazione -> per ogni componente da definire l'interfaccia e per quelli che effettivamente implementeremo da definire dettagli di progettazione e implementazione.

Partiamo dalla architettura stratificata, dove vediamo i tre strati rappresentati in modo esplicito (User, Coordination e Service, corrispondono al BCE).



Quando si fa uso di architetture Service Oriented, sappiamo che facciamo uso di alcuni Pattern per garantire la comunicazione tra oggetti. I pattern che definiremo a livello architetturale in questo caso saranno in particolare, il Broker Handle Pattern (per cui ogni richiesta che va dal cliente al fornitore passerà attraverso il broker per cui per ogni interazione quattro messaggi), poi due meccanismi di comunicazione, uno sincrono e uno asincrono (nel primo chi fa la richiesta si blocca in attesa della risposta), Service Discovery per scoprire quali servizi forniscono funzionalità di mail o addebitamento per carte di credito e Two Phase Commit per quel che riguarda eventuali transazioni.

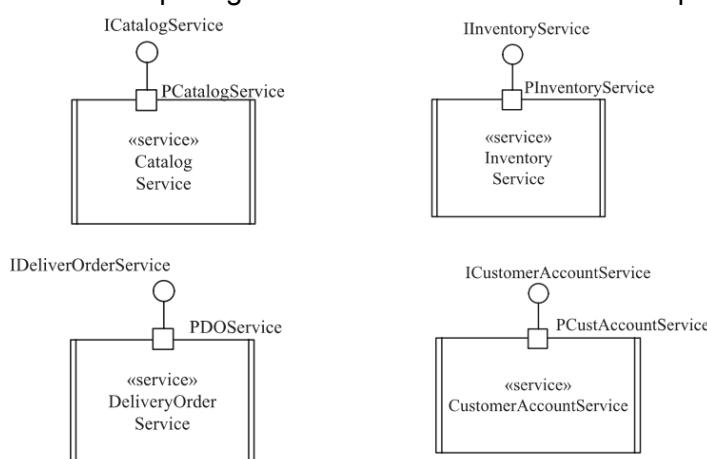
Per questo caso di studio si farà maggiormente uso della comunicazione sincrona, con svantaggio il fatto che il client sarà sospeso nell'attesa di risposta dal servizio. Un'alternativa è utilizzare la comunicazione asincrona con callback da parte del servizio quando egli è pronto a inviargli la risposta, e questa scelta sarà utilizzata per la progettazione sia del Supplier Coordinator che del Billing Coordinator, mentre per il Customer Coordinator le interazioni saranno gestite con comunicazione sincrona.



Viene rappresentato il communication diagram, definito “concurrent” in quanto i rettangoli hanno barre laterali che indicano come ogni istanza eseguita dal communication diagram è eseguita indipendentemente dalle altre.

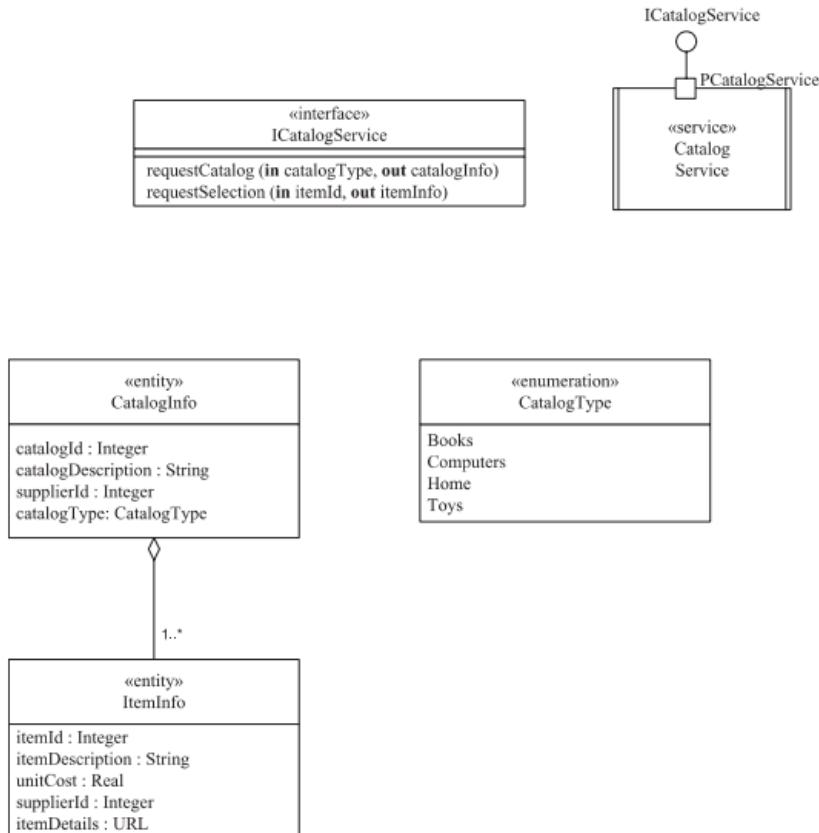
In UML ricordiamo che call sincroni e signal messaggi asincroni. In UML 2 questa differenza è resa esplicita usando frecce piene per chiamate sincrone, normali per asincrone. Si nota come anticipato che nel nostro esempio quasi tutte sincrone salvo la comunicazione con il Billing coordinator che è asincrona.

Progettare una componente significa definirne interfaccia e implementazione. Questo quindi lo facciamo per ognuno dei servizi che dobbiamo implementare:



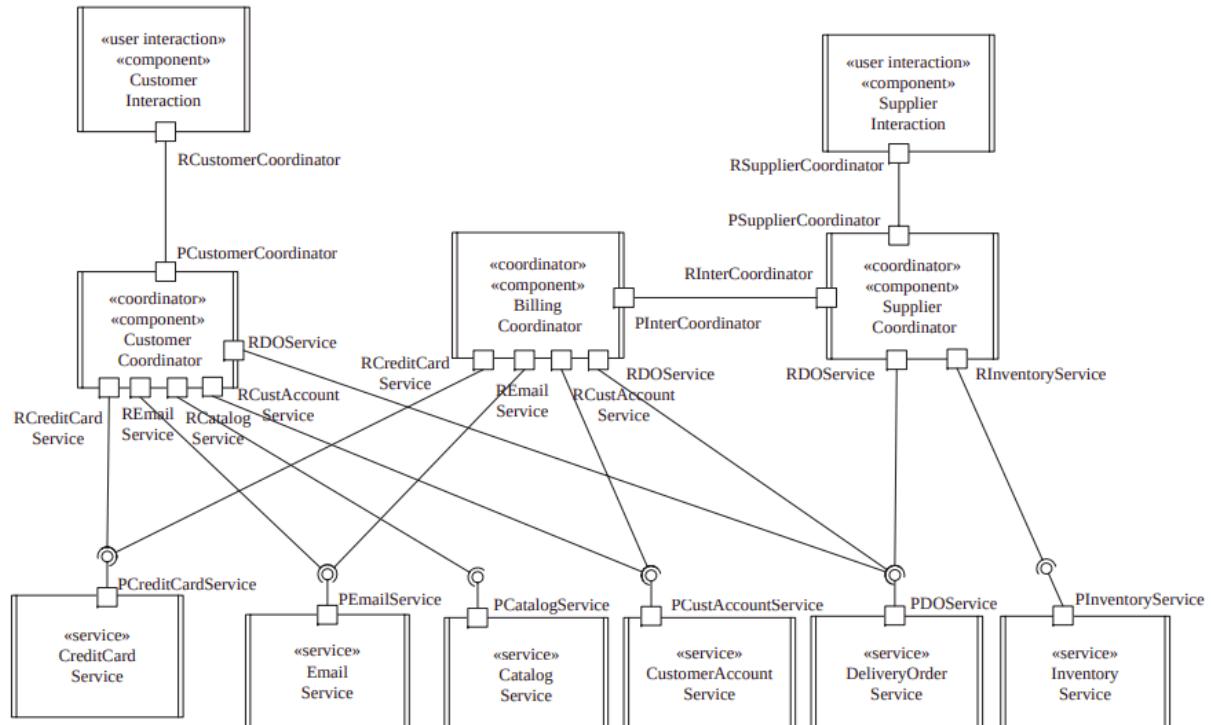
UML permette di rappresentare l’interfaccia fornita da ciascun componente usando la notazione PALLINO, collegato alla classe attraverso una porta (coerentemente alla rappresentazione concettuale di servizio visto come black box alla quale accedo attraverso la porta) (IcatalogService interfaccia servizio catalogo, Pcatalogservice porta servizio catalogo).

A questo punto dobbiamo definire le interfacce del servizio.



L'interfaccia di Catalog Service è costituita da due operazioni: richiedicatalogo e richiediselezione (richiedere la selezione di un certo insieme di elementi). Ogni operazione è fornita con la lista di parametri formali, e per ognuno di essi si esplicita se sia di input o output. Dal punto di vista progettuale CatalogService permette di accedere alle classi CatalogInfo e ItemInfo, si rappresentano quindi anche gli attributi delle classi e la relativa associazione di aggregazione. Anche la classe enumeration CatalogType per indicare i tipi di catalogo messi a disposizione dal fornitore.

Si fa la stessa operazione (non solo interfaccia, ma anche classe di origine e classi con cui interagirà attraverso le operazioni, ossia classi contenute!) anche per customeraccountservice, il deliveryorderservice e l'inventoryservice. Le altre due classi per la posta elettronica e per i pagamenti non li progettiamo noi, quindi ci limitiamo a descriverne l'interfaccia.



La nostra architettura è quindi costituita da una serie di componenti, ognuna con la propria interfaccia. Le componenti sono collegate dalle porte e si effettuano richieste definite secondo un prefisso R (required, ossia il componente richiede una certa funzionalità) o un prefisso P (provided). Chiaro come R per boundary e P per controllo come risposta in quanto per l'interfaccia si richiede il servizio al coordinator.

In basso notazione Lollipop, si utilizza per rendere ancora più esplicito il fornitore e il <richiedente. Se sopra la palla del lollipop richiedente, se stecco del lollipop fornitore (vedi tra Pinventoryservice e RinventoryService, c'è una roba che sembra un lollipop).

Ci si aspetta quindi che tutte le componenti User Interaction non vi sia alcuna funzionalità offerta (ha solo bisogno della componente coordinator). Le "forchette" indicano che questi servizi necessitano solo funzionalità e non ne offrono. Sul lato coordinator invece entrambe le cose: sopra si forniscono funzionalità alle user interaction, sotto si richiedono funzionalità alle componenti di servizio. Lato service invece solo servizi messi a disposizione (pallino). A questo punto bisognerà andare nel dettaglio per ogni componente di modo da descrivere come essa deve essere realizzata limitatamente a solo quelli che dobbiamo realizzare. Questo è importante perché in ottica di riusabilità dell'applicazione, infatti una volta implementati i nostri servizi come il catalog service o il delivery order service possiamo metterli a disposizione su rete e diventare noi stessi fornitori di servizi.

Quindi per applicazioni service oriented vale lo stesso discorso fatto per le component based: se trovo le componenti già fatte che fanno al caso mio allora le prendo, altrimenti le implemento e poi le metto a disposizione per eventuali clienti futuri.

DESIGN PATTERN

I Design Pattern sono artefatti riutilizzabili introdotti per incrementare la riusabilità nella fase di progettazione software, riducendo tempi e costi e aumentando l'affidabilità.

Si vuole quindi definire il modello dei dati (diagramma delle classi) non solo correttamente, ma anche in modo quanto possibile riusabile!

Sappiamo che in fase di analisi dei requisiti si rappresentano come classi entità che hanno una vera e propria controparte nel mondo reale, in fase di progetto emergono anche classi che invece non hanno alcuna controparte nel mondo reale (vedi classi boundary o di controllo), e le astrazioni che emergono in fase di progetto in questo senso sono fondamentali per rendere il progetto riusabile.

Ciò che fanno i Design Pattern è aiutare ad identificare queste astrazioni insieme alle classi che possono rappresentare, e l'idea di base è individuare soluzioni a problemi che sono ricorrenti in fase di progettazione, evitando al progettista di pensare a soluzioni che già sono state trovate da altri, semplificando la manutenzione. Ovviamente l'uso di design pattern non risolve tutti i problemi.

Sono stati introdotti vari tipi di Design Pattern nel tempo, per capire quale Design Pattern fa al nostro caso, è necessario classificarli, e per farlo si sfruttano vari criteri:

Il primo criterio è l'obiettivo (Purpose) del design pattern, distinguiamo tre classi:

- Creazionali: pattern utilizzati per facilitare operazioni di creazione oggetti
- Strutturali: utilizzati, sfruttando le caratteristiche OO come ereditarietà e polimorfismo, per definire la struttura del sistema in termini di composizione di classi e oggetti.
- Comportamentali: si concentrano sul modellare il comportamento del sistema definendo le responsabilità delle componenti e le modalità di interazione.

Il secondo criterio riguarda il raggio di azione (scope), ossia su cosa si applicano i pattern.

Due possibilità:

- Classi: pattern che definiscono le relazioni statiche tra classi e sottoclassi (basate sul concetto di ereditarietà e quindi relazioni statiche, ossia definite a tempo di compilazione)
- Oggetti: pattern che definiscono le relazioni dinamiche tra oggetti (relazioni possono cambiare durante l'esecuzione).

Ci sono più di 23 pattern. Ciascun pattern viene descritto usando un formato standard, che presenta un totale di 10 campi:

- Nome e Classificazione (nome tipicamente significativo per descrivere l'essenza del pattern, classificazione scope e purpose)
- Motivazione: descrive il perché il pattern è stato introdotto e quindi il problema che deve risolvere.
- Applicabilità: descrive le situazioni in cui il pattern può essere applicato
- Struttura: descrive graficamente la configurazione di elementi che risolvono il problema. È uno schema di soluzione non una soluzione per un progetto specifico.
- Partecipanti: classi e oggetti che fanno parte del pattern con relative responsabilità
- Conseguenze: risultati che si ottengono applicando il pattern
- Implementazioni: tecniche e suggerimenti utili all'implementazione del pattern
- Codice di esempio: frammenti di codice che illustrano come implementare il pattern in un certo linguaggio di programmazione (java o c++ tipicamente)
- Usi conosciuti: esempi di applicazione in sistemi reali

- Pattern Correlati: tipicamente non si applica un solo pattern ma spesso è possibile combinarne l'uso.

Prima di passare ai pattern ricordiamo un concetto utile, i Framework.

Un Framework non è una semplice libreria, ma un design riutilizzabile di un sistema, definito da un insieme di classi astratte. Rappresenta lo scheletro di un'applicazione che viene personalizzata dallo sviluppatore implementando interfacce e classi astratte.

I framework permettono quindi di definire lo scopo e la struttura statica di un sistema e sono un buon esempio di progettazione orientata agli oggetti. Permettono di raggiungere due obiettivi: il riuso di design e il riuso di codice.

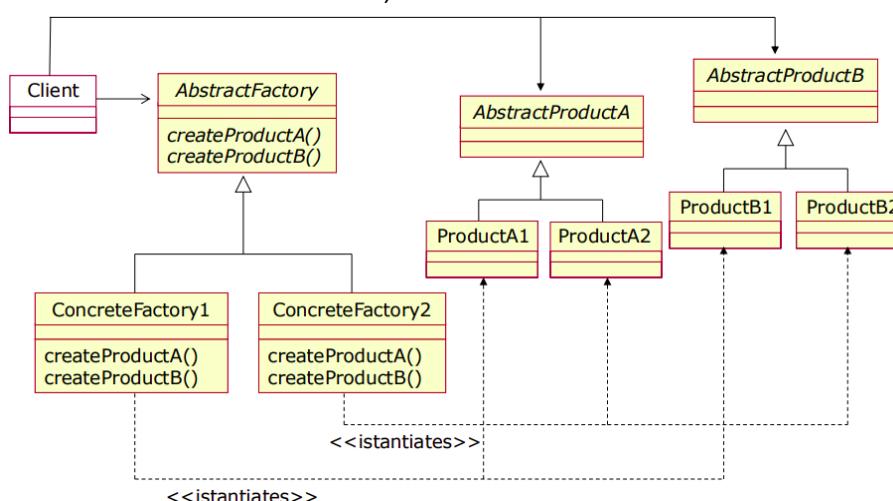
I framework sono basati sul concetto di Classe Astratta, una classe con almeno un metodo non implementato ("astratto"). Questa classe funge da template per le sottoclassi.

Un framework è rappresentato da un'insieme di classi astratte e dalle loro interrelazioni.

I Pattern vengono considerati come mattoni per la costruzione di framework.

Abstract Factory Pattern

- Classificazione: creazionale basato su oggetti
- Scopo: fornire un'interfaccia per la creazione di famiglie di oggetti tra loro correlati.
- Motivazione: realizzazione strumenti di (UI) in grado di sopportare diversi tipi di look & feel (ossia gli elementi che fanno parte dell'interfaccia come finestre, menù etc.. sono sempre gli stessi, ma si vogliono avere diversi modi di visualizzarli es. versione chiara/scura). Per garantire la portabilità di un'applicazione tra look & feel diversi, gli oggetti non devono essere cablati nel codice.
- Struttura: L'applicazione client non istanzia direttamente gli oggetti, ma fa riferimento a una fabbrica astratta (AbstractFactory) responsabile della loro creazione. Questa fabbrica viene implementata da fabbriche concrete che sanno come istanziare correttamente le combinazioni di oggetti per un dato look&feel. (chiaramente in generale n prodotti e altrettante fabbriche concrete)

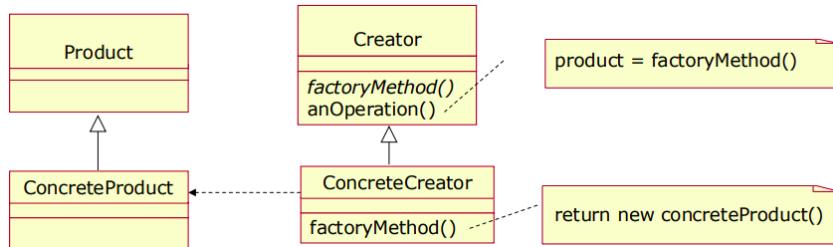


- Applicabilità: si fa riferimento a software indipendente dalle modalità di creazione dei prodotti con cui opera (il client non deve sapere come vengono creati gli oggetti), in particolare è utile quando il sistema è configurabile con famiglie di prodotti diverse ed il client non è legato a una specifica famiglia (ossia può sceglierne una come un'altra).

- Partecipanti: elementi partecipanti nella struttura -> AbstractFactory, ConcreteFactory, AbstractProduct, ConcreteProduct, Applicazione Client
- Conseguenze: grazie all'abstract factory si isolano le classi astratte dalle concrete, permette di cambiare facilmente le famiglie di prodotti (la factory completa compare in un unico punto), ma l'aggiunta di nuove famiglie richiede la ricompilazione del codice (l'insieme di prodotti è legato all'interfaccia della factory).

Factory Method Pattern

- Classificazione: creazionale basato su classi
- Scopo: definire un'interfaccia per la creazione di un oggetto, che consenta di decidere a tempo di esecuzione quale specifico oggetto istanziare (mentre quindi nell'abstract factory l'ereditarietà è utilizzata in modo statico (aggiungere una nuova factory richiede ricompilazione), qui si delega invece il compito runtime (dinamico) a una sottoclasse).
- Motivazione: ampiamente utilizzato nei framework, dove si hanno classi astratte che definiscono le relazioni tra elementi del dominio e sono poi responsabili per la creazione degli oggetti concreti.
- Struttura: L'operazione factoryMethod() (parte della classe astratta Creator) delega la creazione dell'oggetto del prodotto specifico a una sottoclasse (ConcreteCreator) implementata dagli utilizzatori del framework. Il prodotto concreto (ConcreteProduct) è istanziato dall'implementazione concreta del factoryMethod().

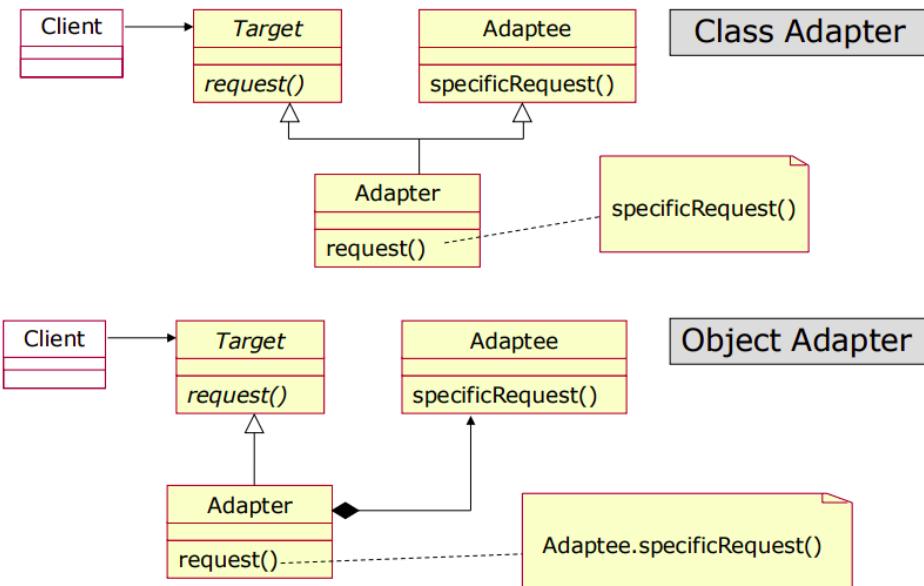


- Applicabilità: si applica quando una classe non è in grado di sapere in anticipo le classi di oggetti che deve creare, quando la classe vuole che siano sottoclassi specifica a scegliere gli oggetti da creare e quando le classi delegano la responsabilità di creazione.
- Partecipanti: Product, ConcreteProduct, Creator e ConcreteCreator
- Conseguenze: elimina la necessità di riferirsi a classi dipendenti dall'applicazione all'interno del codice.

Adapter Pattern

- Classificazione: strutturale basato classi/oggetti
- Scope: Convertire l'interfaccia di una classe esistente incompatibile con quanto necessario al client, in una versione compatibile.
- Motivazione: Consideriamo un editor che consente di disegnare e comporre elementi grafici. L'astrazione chiave è un singolo oggetto grafico. Si suppone di voler integrare un nuovo elemento ma che questo non ha interfaccia compatibile con l'editor.
- Struttura: Target (interfaccia a cui adattarsi), Adapter (classe adattatore), Adaptee (classe da adattare). Con Class Adapter si fa uso dell'ereditarietà (si chiama il metodo della

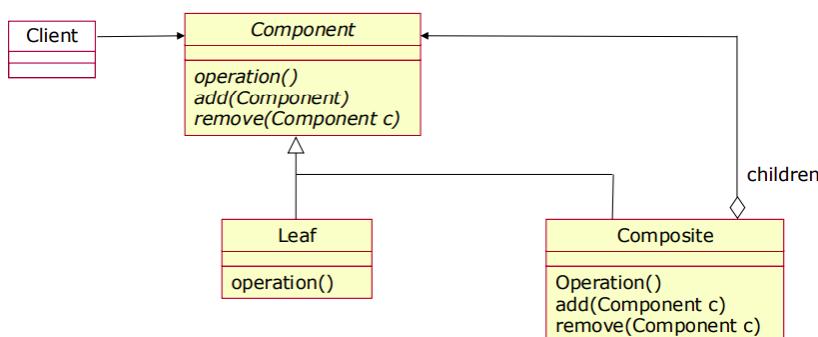
superclasse), con Object Adapter composizione.



- Adattabilità: si usa quando siamo interessati a riusare una classe esistente ma con interfaccia incompatibile a quella desiderata
- Partecipanti: Client, Target, Adapter e Adaptee
- Conseguenze: è necessario prendere in considerazione l'effort necessario all'adattamento.

Composite Pattern

- Classificazione: strutturale basato su oggetti
- Scopo: comporre oggetti in strutture gerarchiche che consentano di trattare i singoli elementi e la composizione in modo uniforme (allo stesso modo).
- Motivazione: Le applicazioni grafiche consentono di trattare in modo uniforme sia le forme geometriche di base (linee, cerchi etc...) sia oggetti complessi che si creano a partire da questi elementi semplici. Molti editor grafici hanno ad esempio la funzione raggruppa.
- Struttura: Il Client interagisce con il Component avente una certa interfaccia con i metodi **add** e **remove**, classi elementari (**Leaf**) che costituiscono direttamente i Component che posso creare (possono essere n) e Composite ricordandomi di usare non solo il costrutto di aggregazione o composizione ma anche ereditarietà. Se si usa composition se si cancella la figura composita si cancellano anche gli elementi contenuti all'interno, se usiamo aggregation posso cancellare il raggruppamento senza cancellare gli oggetti raggruppati.



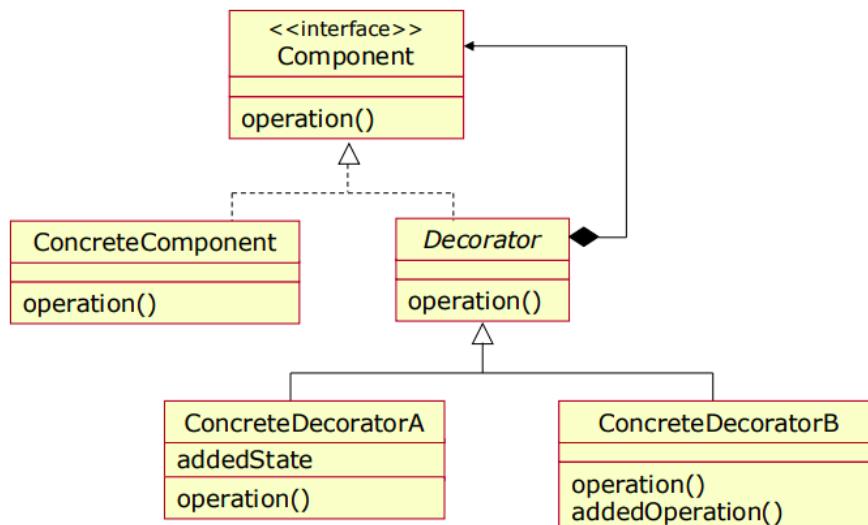
- Applicabilità: si usa quando si vogliono rappresentare le gerarchie di oggetti in modo che oggetti semplici e composti siano trattati allo stesso modo
- Partecipanti: Component e Composite, Leaf, Client

- Conseguenze: i client sono semplificati perché gli oggetti semplici e composti sono trattati allo stesso modo, l'aggiunta di nuovi oggetti Leaf o Composite è relativamente semplice sfruttando il codice dell'applicazione già esistente. Tuttavia può rendere il sistema troppo generico e poco flessibile, con questa soluzione infatti non posso creare un oggetto composito che contenga esclusivamente un certo tipo di oggetti.

Decorator Pattern

Classificazione: strutturale basato su oggetti

- Scopo: Aggiungere dinamicamente funzionalità (responsabilità) ad un oggetto. Il subclassing è un'un'alternativa statica (compile time, non runtime), il cui scope è a livello di classe e non di singolo oggetto.
- Motivazione: uno scenario classico è la realizzazione di un'interfaccia utente (testo scorrevole, bordo).



- Struttura: L'interfaccia Component descrive un componente generico, poi classe astratta Decorator implementata da tutte le possibili decorazioni (A, B, ...). Ogni volta che creo un oggetto ConcreteDecorator dovrò specificare come parametro del costruttore l'oggetto che dovrà essere decorato.
- Applicabilità: si applica se necessario aggiungere responsabilità a oggetti in modo trasparente e dinamico, quindi quando in particolare il subclassing (statico) non è adatto.
- Partecipanti: Component e ConcreteComponent, Decorator e ConcreteDecorators
- Conseguenze: maggior flessibilità rispetto all'approccio statico, evita quindi di definire strutture gerarchiche complesse.

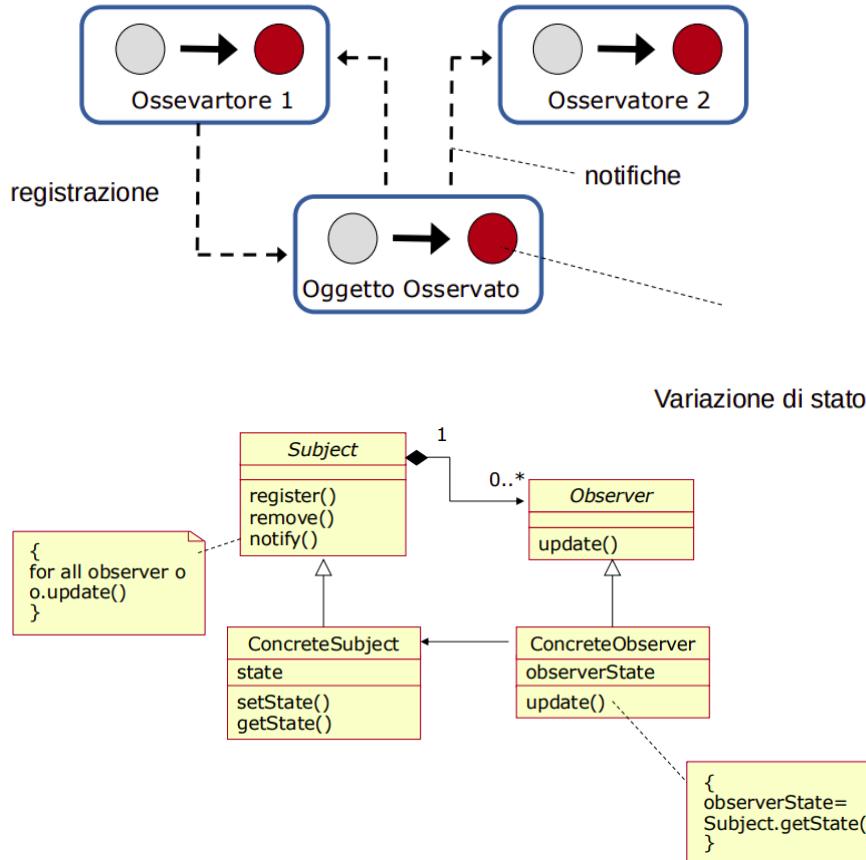
Alcune note aggiuntive: il Decorator è molto usato in Java, in particolare nella definizione degli stream Input/Output:

Il decorator sembra simile al pattern Composite ma la finalità è diversa. Decorator sembra simile anche al pattern Adapter, ma quest'ultimo si limita ad un adattamento di un'interfaccia. Tra le principali limitazioni del Decorator il fatto che le decorazioni possono essere applicate più e più volte, ciò che però non è vietato è applicare la stessa funzionalità più e più volte. Ciò non ha senso dal punto di vista della funzionalità ma è consentita dal pattern.

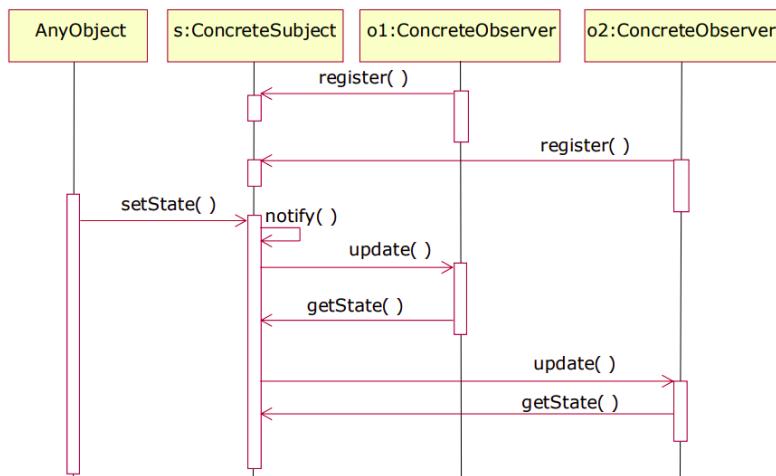
Observer Pattern

- Classificazione: comportamentale basato su oggetti

- Scopo: definire una dipendenza uno a molti tra oggetti, mantenendo basso il grado di coupling. La variazione dello stato di un oggetto deve essere osservata da altri oggetti, in modo che possano aggiornarsi automaticamente.
 - Motivazione: Lo scenario classico è quello con GUI ossia applicazioni con interfaccia grafica secondo il paradigma Model-View-Control (simile a BCE), per cui ad esempio se si vuole mostrare via interfaccia il cambiamento dei dati se oggetti Model (entity) cambiano, allora gli oggetti che implementano la View (boundary) devono aggiornarsi.
soluzione (naive): si potrebbero utilizzare nell'oggetto osservato attributi pubblici o metodi pubblici (getters) che leggono il valore di un attributo protetto -> periodicamente gli osservatori controllano se vi sono stati aggiornamenti. Non si tratta tuttavia di una buona soluzione, in quanto non è scalabile, gli osservatori dovrebbero continuamente interrogare l'oggetto osservato e variazioni rapide potrebbero non essere comunque rilevate da qualche osservatore.
- Il pattern Observer prevede che gli osservatori si registrino presso l'oggetto osservato, in questo modo sarà lui a notificare ogni cambiamento di stato agli osservatori.



- Struttura: Subject è l'oggetto da osservare, l'Observer è l'oggetto interessato ai cambiamenti di stato. Entrambe sono classi astratte che mettono a servizio i metodi register() e remove() (Subject) che utilizzano gli observer per mostrare o meno interesse verso un subject, notify() è invece un metodo usato dallo stesso Subject per notificare gli oggetti interessati di cambiamenti di stato.
- Il metodo update() è invece il metodo eseguito dall'Observer una volta ricevuta la notifica. Dire che avviene un cambiamento di stato significa dire che il ConcreteSubject (ossia colui che effettivamente implementa la classe astratta) esegue setState(), e dopo averlo eseguito esegue notify(). Dopodiché sarà l'Observer in questione che nel metodo update() chiamerà il metodo getState() dal subject per recuperare il nuovo valore dello stato.



Il workflow appena descritto è chiaro guardando il seguente Sequence Diagram:

Ogni altro oggetto dell'applicazione chiama setState() sull'oggetto osservato facendogli quindi cambiare stato. Quindi il ConcreteSubject invoca su se stesso il metodo notify() che poi invoca l'update su tutti gli oggetti che si erano registrati come observer. Dopodiché l'osservatore, se interessato, effettuerà il getState() sull'oggetto osservato.

- Applicabilità: Si applica quando una azione può essere scomposta in due ambiti, ciascuno encapsulato in oggetti separati per mantenere basso il livello di coupling. Utile quindi per gestire le modifiche di oggetti conseguenti la variazione dello stato di un oggetto.

- Partecipanti: Subject, ConcreteSubject, Observer, ConcreteObserver

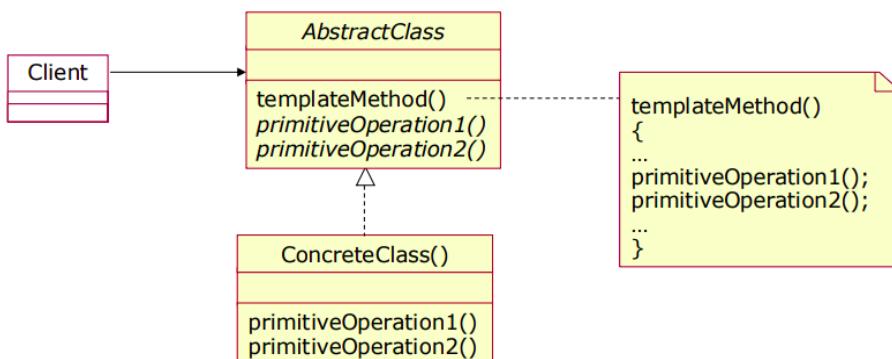
- Conseguenze: L'accoppiamento tra Subject e Observer è astratto (il Subject conosce solo la lista degli osservatori). La notifica è di tipo broadcast. Tuttavia, ogni modifica al Subject (setState()) comporta una serie di messaggi e modifiche a tutti gli osservatori.

Template Method Pattern

- Classificazione: comportamentale basato su classi

- Scopo: Definire la struttura di un algoritmo interno di un metodo, delegando alcuni dei suoi passi alle sottoclassi.

- Motivazione: Un framework per costruire applicazioni che gestiscono diversi documenti. Il Template Method definisce un algoritmo in base ad operazioni astratte che saranno definite nelle sottoclassi specifiche



- Struttura: Il client invoca il templateMethod() (nella classe astratta AbstractClass) che include operazioni, alcune delle quali sono astratte e implementate dalle sottoclassi (ConcreteClass).

- Applicabilità: utilizzato per implementare la parte invariante di un algoritmo, lasciando alle

sottoclassi la definizione degli step variabili. È utile quindi quando ci sono comportamenti comuni che possono esser inseriti nel template.

- Partecipanti: AbstractClass, ConcreteClass, Client.
- Conseguenze: permettono ovviamente riuso del codice, creano struttura di controllo invertito dove è la classe padre a chiamare le operazioni ridefinite dai figli e non viceversa (come solito nell'ereditarietà).

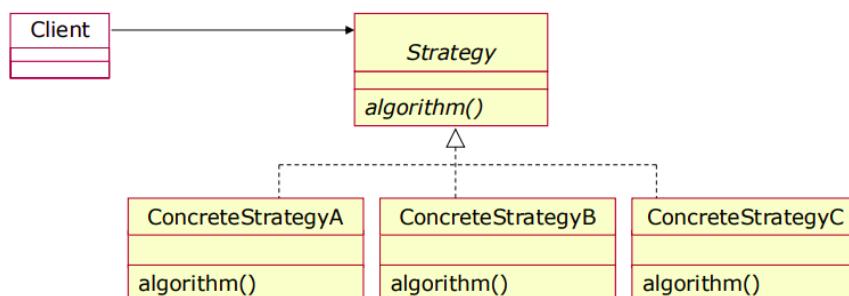
I metodi richiamati dalla superclasse sono detti metodi gancio (hook), offrono un comportamento standard che la sottoclasse volendo può ridefinire.

TemplateMethod e FactoryMethod sono simili tra loro nell'approccio utilizzato, la differenza principale sta nello scopo:

il Factory Method è metodo astratto che deve creare e restituire l'istanza di classe concreta, al fine di "deresponsabilizzare" il client dalla scelta del tipo specifico. Il Template Method d'altra parte è un metodo che invoca metodi astratti al fine di generalizzare un algoritmo.

Strategy Method Pattern

- Classificazione: comportamentale basato su oggetti.
- Scopo: permette di definire famiglie di algoritmi in modo da renderli intercambiabili indipendentemente dal client che li usa.
- Motivazione: Consideriamo la famiglia degli algoritmi di ordinamento (es. QuickSort, HeapSort, etc..). Si vuole costruire un'applicazione che li supporti tutti e che sia facilmente estendibile, e che inoltre permetta una scelta rapida del miglior algoritmo da usare.



- Struttura: Si implementa un'interfaccia Strategy (es. SortingStrategy) che viene poi implementata da un insieme di sottoclassi concrete (ConcreteStrategy come QuickSortStrategy, HeapSortStrategy) con il proprio algoritmo. L'idea di fondo è disaccoppiare quindi il client dall'implementazione degli algoritmi e dal definire una logica di scelta dell'algoritmo da utilizzare in base al caso.
- Applicabilità: molte classi correlate differiscono solo per il comportamento, quando è necessaria un'interfaccia comune per diverse varianti di uno stesso algoritmo.
- Partecipanti: Strategy, ConcreteStrategy, Client
- Conseguenze: il Pattern separa l'implementazione degli algoritmi dal contesto dell'applicazioni (evitando di implementare algoritmi direttamente come sottoclassi del client). Inoltre in questo modo si eliminano i blocchi condizionali che sarebbero altrimenti necessari inserendo tutti i diversi comportamenti in un'unica classe. Lo svantaggio principale sta nel fatto che i Client devono conoscere le diverse strategie.

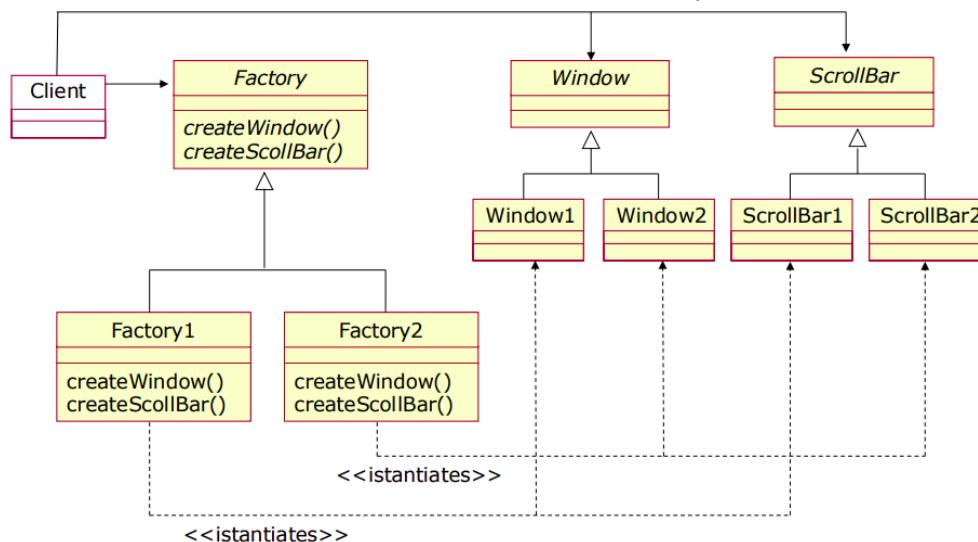
(ES SALTARE fino a pag 137)

ES ABSTRACT METHOD

Esempio: Si considera uno strumento per realizzare UI dove si vuole garantire compatibilità con due look&feel. Il primo prevede l'utilizzo di finestre create con la classe Window1 con relativa scrollbar ScrollBar1, e il secondo lo stesso ma per Window2 e ScrollBar2.

Chiaramente l'applicazione deve realizzare una UI che rispetti le relazioni tra oggetti e sia portabile da un Look&Feel a un altro -> bisogna rispettare le relazioni tra oggetti, evitando che il client accoppi (sbagliando) ad es. Window1 e Scrollbar2.

Per riuscire a far ciò si utilizza quindi l'Abstract Factory.



L'applicazione client ha a che fare con le 4 classi Window1, ... ScrollBar 2. Affinché le classi siano associate in modo corretto si evita che sia il client a istanziare oggetti a partire da esse. Si utilizza quindi un'abstract factory (nome corsivo in UML per astrazione) responsabile della creazione di windows e scrollbar, a cui il client farà riferimento!

La fabbrica astratta è implementata attraverso la Factory1 e 2 (necessarie per esprimere le 2 combinazioni possibili in questo caso). Quindi se il client volesse usare Look&Feel1, si riferisce alla Factory1 che saprà come istanziare correttamente gli oggetti.

Se non si utilizzasse l'abstract factory, il client dovrebbe avere la responsabilità di accoppiare correttamente gli oggetti, mentre se usa abstract factory creerà un'oggetto da quella classe astratta a cui sarà quindi demandata la responsabilità.

Il rispetto delle relazioni è cablato nel codice e deve essere noto al client.

```
Window w = new Window1();
...
ScrollBar s = new ScrollBar1();
```

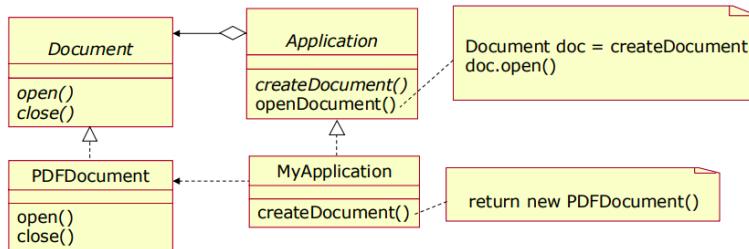
Con l'Abstract Factory la responsabilità è demandata alla Factory.

```
Factory f = new Factory1();
Window w = f.createWindow();
...
ScrollBar s = f.createScrollBar();
```

ES FACTORY METHOD

Esempio: Consideriamo un framework di gestione di documenti di tipo diverso. Le due astrazioni chiave del framework sono Application (applicazione principale) e Document (classe astratta che rappresenta il concetto di documento, ma non il tipo specifico). Sono gli utilizzatori a dover definire delle sottoclassi per ottenere implementazioni adatte al loro caso specifico. Application sa quando creare un documento (può crearlo in qualsiasi momento) ma non sa esattamente quale deve creare.

Il pattern Factory incapsula la conoscenza della specifica classe da creare al di fuori del framework.

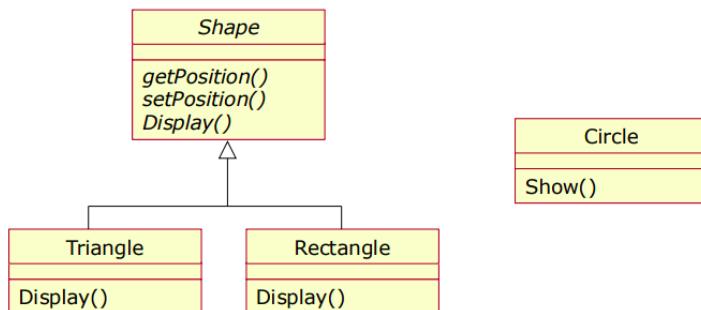


Il `Document` è classe astratta in questo caso implementata da un particolare tipo di documento (PDF), l'`Application` è invece responsabile della creazione del documento. In essa è presente un metodo astratto `createDocument()` e uno concreto `openDocument()` implementato usando la nota in alto a dx (invoca `createDocument` e poi lo apre). Il metodo `createDocument()` essendo astratto è implementato dalla sottoclasse specifica che sa esattamente quale tipo di documento deve esser creato (in questo caso PDF).

Si chiama Factory Method perché si utilizza piuttosto che una classe astratta un Metodo “Fabbrica”, metodo astratto che dovrà essere implementato dalle specifiche sottoclassi poi responsabili della creazione dello specifico tipo di documento nel nostro caso.

ES ADAPTER

Esempio: Supponiamo di voler integrare il componente Circle nell'editor che già supporta le forme Triangel e Rectangle.

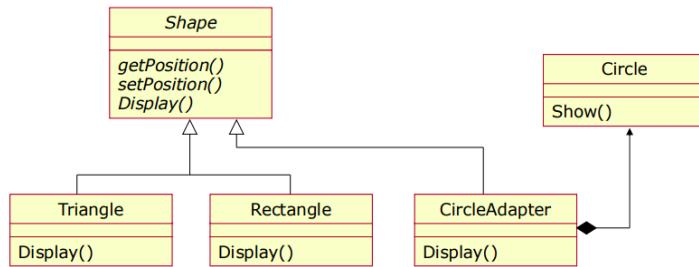


La classe astratta Forma è poi implementata da `Triangle` e `Rectangle`, che in particolare implementano il metodo `display`. Si suppone poi di avere a disposizione una classe Circonferenza che però è stata implementata con interfaccia differente (non ho `Display` ma `Show`). Ora o implemento una nuova classe `Circle` con metodo `Display` o adatto il metodo `Show()` che già ho a disposizione.

Prima soluzione: Adapter con scope a Oggetti

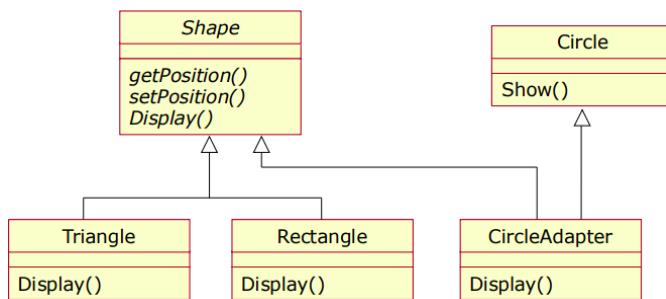
Si realizza una classe chiamata non `Circle` ma `CircleAdapter` che crea l'oggetto `Circle` (riusando la classe `Circle`), e quando viene invocato il metodo `Display()` nell'adattatore sarà invocato il relativo metodo `Show()` per `Circle`.

Si utilizza la Composition UML (rombo pieno) in quanto appunto l'oggetto Circle è creato all'interno dell'oggetto CircleAdapter -> è encapsulato.



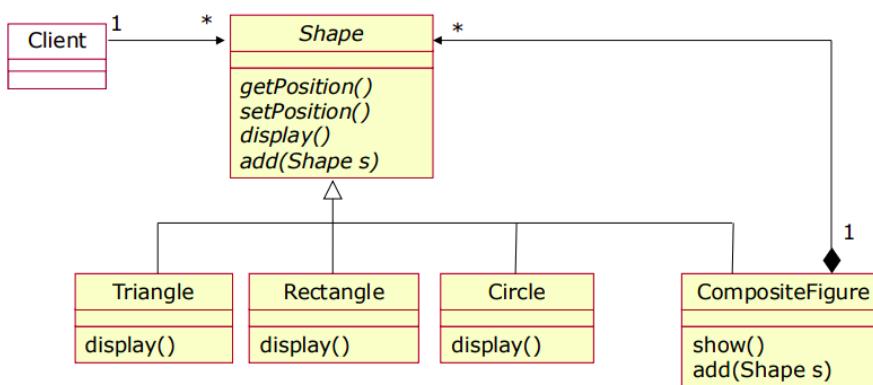
Seconda soluzione: Adapter con Class Scope

Invece di usare la composition si usa l'ereditarietà. CircleAdapter eredita sia da Shape che da Circle, il metodo `Display()` ridefinisce il metodo `Show()` facente parte della classe Circle.



ES COMPOSITE

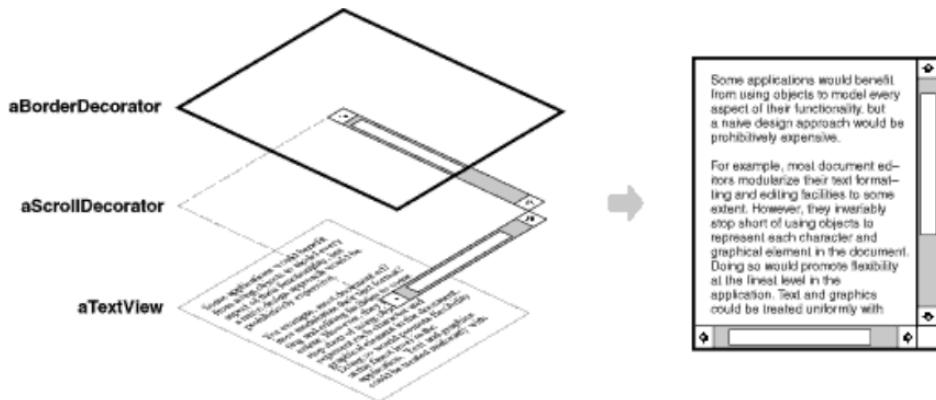
Esempio: Si considera come esempio una app grafica in grado di gestire gli oggetti elementari Triangle, Circle e Rectangle. Si ha come requisito che l'applicazione permetta il raggruppamento dinamico di oggetti elementari in oggetti composti. I due tipi di oggetti devono essere trattati allo stesso modo (in modo uniforme).



Il client interagisce con la classe Forma. Rispetto a prima abbiamo anche il metodo `add` che permette di aggiungere a un elemento altri elementi. Oltre agli oggetti elementari, abbiamo un'altra classe FiguraComposita che può essere costituita da 1 o più. Costrutto di Composizione in quanto la FiguraComposita è costituita da 1 o più Figure, e se muore l'oggetto da **CompositeFigure** muoiono anche gli oggetti che fanno parte di lui. È necessario introdurre ereditarietà anche per **CompositeFigure** in quanto senza non era possibile creare altre **FigureComposite** a partire da una **FiguraComposita** non rispettando quindi i requisiti,

così facendo viene considerata come se fosse una figura geometrica a se stante.

ES DECORATOR

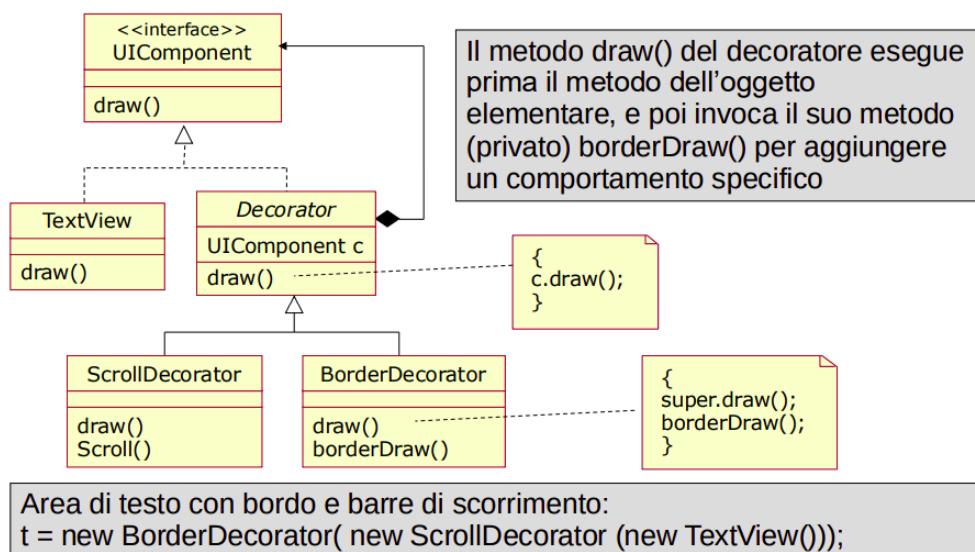


Si parte dall'oggetto base, una finestra di testo (TextView) al quale vogliamo aggiungere delle responsabilità ("decorazioni") come ad esempio la barra di scorrimento ScrollDecorator e un bordo BorderDecorator.

Ma come raggruppare questi tre oggetti al fine di raggiungere l'obiettivo di dinamicità (cioè a tempo di esecuzione poter decorare la finestra con gli altri due oggetti)?

L'approccio classico che scartiamo è quello basato su subclassing, dal momento che come detto prima è approccio statico (una volta creato BorderTextView) non posso più cambiarlo, devo creare una sottoclasse per ogni esigenza, inoltre se volessi creare una finestra che abbia entrambe le funzionalità dovrei creare una specifica sottoclasse.

Un approccio più flessibile è quello di racchiudere un oggetto elementare in un altro, che aggiunge una responsabilità particolare, chiamato Decorator. Il Decorator ha un'interfaccia conforme all'oggetto da decorare e trasferisce le richieste all'oggetto decorato ma può svolgere funzioni aggiuntive (come appunto aggiungere un bordo o la scrollbar) prima o dopo il trasferimento della richiesta.



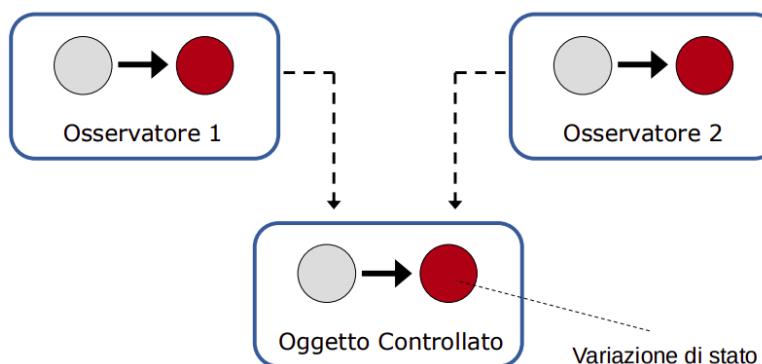
Si ha un'interfaccia che è il nostro componente generico di User Interface, essa ha il metodo `draw()` per visualizzare il componente a livello grafico. `TextView` e `Decorator` implementano il

metodo generico di interfaccia (fanno draw()) e Decorator in più contiene UIComponent -> può contenere un oggetto TextView.

Tuttavia Decorator rappresenta ancora una classe astratta -> da essa eredita ScrollDecorator e BorderDecorator, in particolare ognuna di esse aggiunge il metodo necessario per estendere le funzionalità base dell'oggetto elementare (TextView). In questo caso il metodo draw() esegue prima il metodo dell'oggetto elementare, e poi chiama eventualmente il suo metodo specifico (es. borderDraw()) per aggiungere una funzionalità specifica. Da borderDraw() si evoca il draw della superclasse (Decorator) che a sua volta invoca il draw del componente che incapsula (TextView) -> viene prima disegnata la finestra di testo e poi si aggiunge il bordo.

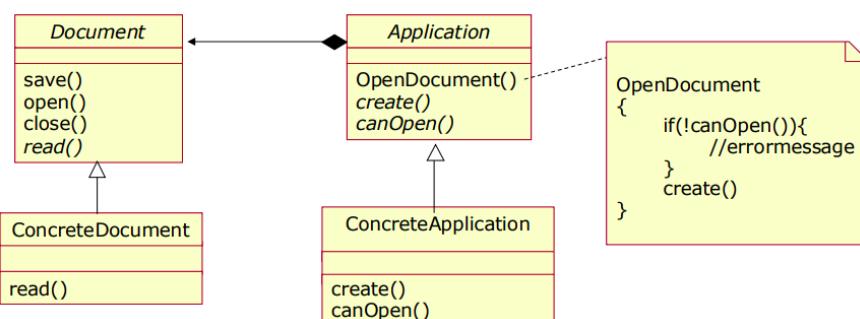
Questo meccanismo è molto flessibile, in quanto se voglio creare un oggetto che ha sia bordo che barra di scorrimento faccio `t = new BorderDecorator(new ScrollDecorator(new TextView()))` (TextView oggetto base è incapsulato in ScrollDecorator, e l'oggetto così ottenuto viene ulteriormente incapsulato in BorderDecorator aggiungendo quindi l'ulteriore responsabilità del bordo. In questo modo posso aggiungere tutte le funzionalità che voglio senza implementare in modo statico tutte le combinazioni di funzionalità come sottoclassi.

OBSERVER



Es. software che mostra i risultati di una partita di calcio, gli oggetti legati al risultato possono cambiare dinamicamente stato in tempi anche molto ristretti, si vuole fare in modo che quando lo stato cambia gli osservatori (come le semplici interfacce grafiche che mostrano all'utente questi risultati) siano modificati aggiornando il valore.

TEMPLATE METHOD



Document e Application classi astratte, il primo prevede i metodi salva, apri e chiudi mentre `read()` che cambia in base al tipo di documento -> se devo leggere Word devo implementare l'algoritmo necessario a leggerlo etc... In Application (composto da un insieme di documenti) invece metodi `create`, `canOpen` e `OpenDocument()` che definisce l'algoritmo necessario per creare e leggere il documento tramite i primi due (il corpo prevede che se non si può aprire

allora errore altrimenti crealo. Tuttavia i metodi `create()` e `canOpen()` sono astratti e implementati in `ConcreteApplication` in base al tipo di documento (struttura di controllo invertito!).

Esercizio 1: realizzare un'applicazione per gestire il download dai siti http e ftp. La selezione avviene in base all'utilizzo dell'url (ftp o http). Ci viene suggerito di utilizzare una factory, per creare l'algoritmo di download e togliere la responsabilità dal client di scegliere quale algoritmo utilizzare in base al caso, e uno strategy per implementare il download secondo i due protocolli (si combinano quindi pattern creazionale con comportamentale!). L'idea è scrivere un'applicazione che sia estendibile senza problemi (coerentemente con i concetti di manutenibilità e riusabilità). L'interfaccia `ProtocolStrategy` fornisce un metodo `getFile` dove si specifica URL e destinazione del file, essendo limitata la possibilità di download a HTTP e FTP lo strategy è implementato da `FTPStrategy` e `HTTPStrategy` Il client è in questo caso colui che, come si vede nel codice a destra del metodo `null`, si occupa di "decidere" quale metodo usare per `getFile` (FTP o HTTP, lo capisce tramite `if, else if`). Per aggiungere un eventuale nuovo protocollo si deve modificare la classe `DownloadApp` e il metodo `download`. Inoltre `DownloadApp` in questo scenario deve conoscere dettagli implementativi come sapere esattamente i protocolli supportati. Per evitare ciò si introduce una Factory. Si utilizza secondo la logica vista per la Factory una classe astratta creator che fornisce un metodo astratto di creazione implementato dalla classe `ProtocolFactory`. Stavolta è la classe `ProtocolFactory` a contenere il metodo `create` contenente la logica prima associata al Client di dover scegliere che download usare in base al protocollo: infatti ora la Factory prende in input l'URL, se inizia per HTTP usa la strategia HTTP implementata da `HTTPStrategy` altrimenti `FTPStrategy` se inizia con FTP. Ancora `HTTPStrategy` e `FTPStrategy` implementano il `getFile` in base al protocollo, ciò che veramente cambia è la classe `Download`, molto più compatta: si deresponsabilizza il client dal dover conoscere e "scegliere" quale metodo utilizzare grazie all'uso della Factory -> il client istanzia la `ProtocolFactory`, le chiede di creare l'oggetto per effettuare il download in base solo all'URL che prende in input e poi chiede di scaricare il file (`ps.getFile`) -> se volessi introdurre nuovi protocolli e quindi algoritmi di download mi basta agire sulle classi implementative di `Strategy` e ridefinire il metodo `create` in `ProtocolFactory`

Esercizio 2: Si vuole descrivere un modello a oggetti che rappresenti gli impiegati di un'azienda. Essi devono avere un metodo `chiSono()` per visualizzarne il nome e la responsabilità. Ogni impiegato può avere responsabilità aggiuntive come CapoArea e/o CapoProgetto, ed una particolare categoria di impiegati che ci interessa sono gli Sviluppatori. Viene suggerito di utilizzare il Decorator Pattern per assegnare dinamicamente le responsabilità (es. se uno sviluppatore diventa capoprogetto, devo poterlo rendere evidente in runtime) ed inoltre l'ereditarietà per contraddistinguere Sviluppatori da impiegati generici. Dall'esame dei requisiti si può provare a disegnare una struttura con queste caratteristiche: deve sicuramente esistere una classe `Impiegato` che definisca un'interfaccia comune (metodo `chiSono()`), deve esistere una classe `Sviluppatore` che eredita da `Impiegato` e si devono definire via Decorator due classi ancora più specifiche: `CapoArea` e `CapoProgetto`. (L'idea naive poteva essere creare superclasse `impiegato` con sottoclassi `sviluppatore`, `sviluppatore capoarea`, `sviluppatore capoprogetto`, `sviluppatore capoarea capoprogetto...` non scalabile e tutti i problemi visti quando abbiamo parlato di Decorator). Si ha l'interfaccia principale di impiegato implementata dalla classe concreta `sviluppatore` e dalla classe astratta `decorator`, che a sua volta è implementata da `capoprogetto` e `cupoarea`.

Ricordiamo l'importanza dell'associazione di composizione tra RespDecorator e Impiegato: il decorator infatti funziona creando un oggetto decorato con l'oggetto da decorare; in fase di creazione di un oggetto CapoProgetto, lo "decoreremo" facendogli contenere un oggetto Sviluppatore (si ricorda anche il problema di decorator per cui se si usa in modo inappropriato si potrebbe creare uno sviluppatore con responsabilità due volte capoprogetto e così via...). Vediamo come prima anche l'implementazione vera e propria. La classe Sviluppatore implementa l'interfaccia Impiegato -> costruttore per cui alla creazione dello sviluppatore si deve inserire nome e metodo chiSono che oltre a nome restituisce la responsabilità che in questo caso è semplice sviluppatore. Vediamo ora come rendere lo sviluppatore anche capoprogetto e/o capoarea. La classe astratta Decoratore implementa Impiegato, essendo chiaramente classe astratta non posso istanziare oggetti direttamente da questa classe. L'attributo responsabile viene assegnato all'impiegato su cui viene applicata la decorazione, il metodo getName() ritorna l'utilizzo del metodo getName() sul responsabile in questione e chiSono() sarà utilizzato sull'implementazione concreta dell'impiegato. Vediamo cosa succede per la decorazione concreta: CapoProgetto estende RespDecorator (extends perché classe astratta), invoca il costruttore della superclasse ed l'implementazione del suo metodo chiSono() consiste nell'invocare il metodo della superclasse aggiungendo la frase "e sono anche CapoProgetto" delineando la responsabilità aggiuntiva! Quando si utilizza ciò l'utente crea l'oggetto Impiegato nominandolo come sviluppatore. Se poi lo si vuole render capoprogetto (decoro l'oggetto pippo come capoprogetto), per farlo creo l'oggetto CapoProgetto passando nel costruttore l'impiegato base. Posso chiaramente continuare a decorare ad esempio l'oggetto pipp come CapoArea e risponderà adeguatamente (sono anche un Capoprogetto e sono anche un CapoArea)

Esercizio 3: si considera un oggetto Subject (da controllare) che riceve valori dall'esterno e in modo casuale cambia il suo stato interno accettando o meno il valore ricevuto. Due oggetti Watcher e Psychologist devono monitorare il cambiamento di subject. Si suggerisce di utilizzare le interfacce Observer e Observable. In Java sono presenti package che forniscono l'interfaccia Observer (con metodi standard per il pattern come update) e la classe astratta Observable che fornisce i metodi per registrare, notificare etc... (sarebbe il nostro subject) Il subject implementa l'observable semplicemente usando un attributo che rappresenta lo stato (in questo caso un semplice int). ReceiveValue per ottenere eventuali valori e decidere se cambiare stato, getValue necessario per l'Observer affinché recuperi eventuali nuovi valori di fronte a una notifica. In questo caso il modo per vedere se si cambia o meno lo stato di fronte ad un valore è che se un numero random generato tra 0 e 1 è < 0.5 -> cambio lo stato con il numero che è stato passato altrimenti no. Dopodiché in ogni caso avviene la notifica verso gli observer. (NB invece di value nel codice dovrebbe essere state, invece di public int returnValue() dovrebbe essere public int getValue() e poi return state, piccole inconsistenze). Si implementa per l'Observer il metodo update dove si ritorna lo stato del subject, si usa l'attributo changes per contare il numero cambiamenti dell'oggetto osservato. Qui l'esempio di come deve essere utilizzato il tutto lato client: si istanzia il subject, si istanziano i due osservatori watcher e psycho, poi si simulano dieci chiamate (via for) il metodo receiveValue ed ogni volta che si fa l'invocazione si stampa lo stato. Poi alla fine si utilizza l'observedChanges() per capire quante volte il subject ha cambiato lo stato.

Esercizio 4: si vuole realizzare un Editor in grado di gestire elementi grafici costituiti di elementi elementari come frame, testi e immagini. L'editor deve supportare due diversi algoritmi di formattazione e inoltre degli elementi grafici complessi come barre di scorrimento

o bordi grafici. Si suggerisce l'utilizzo di pattern Composite (per gestire elementi grafici sia elementari che compositi), Strategy (per i due algoritmi di formattazione), Decorator (per decorare gli elementi grafici con elementi aggiuntivi come barre di scorrimento/bordi). Il frame è componente grafico che raggruppa altri componenti e eventualmente anche altri frame -> usiamo Composite usando l'associazione di composizione/applicazione (nel nostro caso applicazione perché rombo vuoto) che permette di raggruppare elementi sia atomici che compositi. Si ha una classe astratta ElementoGrafico ereditata da Immagine, Testo e Frame con metodo draw() per rappresentare graficamente e insert() per aggiungere un nuovo elemento grafico, chiaramente insert da implementare anche per Immagine e Testo che sono base -> se si utilizza insert su un elemento atomico viene generata un'eccezione. Usiamo strategy per implementare due algoritmi di formattazione: RightAlign e Justify. Quindi poi quando si prende in considerazione un elemento grafico Composto (Composition) secondo quanto visto con Composite l'algoritmo che si utilizza per la formattazione è uno dei due illustrati in figura. Infine il Pattern Decorator permette di aggiungere Border o ScrollBar all'elemento grafico. Nella soluzione complessiva abbiamo l'elemento grafico da cui siamo partiti come classe astratta implementata da Immagine, Testo e Frame che può a sua volta contenere immagine, testo o frame (Composite), Decorator che ha come sottoclassi concrete tutti gli elementi che possono decorare l'elemento grafico e infine Strategy per utilizzare uno dei due algoritmi di formattazione.

Ora vediamo tramite alcuni esempi di applicazione come tradurre i Pattern in codice Java. Partiamo dall'Adapter, pattern che sappiamo viene usato nel caso in cui si vogliano riusare classi di cui già si dispone ma incompatibili a interfacce che si necessita usare. Si immagina in particolare di voler gestire gli oggetti tramite un'interfaccia Polygon e si ha a disposizione una classe Rectangle che si vorrebbe riutilizzare, ma che ha interfaccia diversa che non si vuole modificare. (si osserva come alcuni metodi di Rectangle conformi all'interfaccia mentre altri no come getSurface() non disponibile in Rectangle). setShape() come costruttore di Rectangle. Sappiamo che l'adapter può essere usato sia per oggetti che per classi. Caso 1) Class Adapter: viene creata una RectangleClassAdapter che estende Rectangle e implementa l'interfaccia Polygon. Essa per implementare l'interfaccia adatterà quindi quanto fornito dalla superclasse. Caso 2) Object Adapter: qui invece di estendere la classe da adattare la incapsuliamo in un oggetto. La costruzione dell'Object Adapter si basa quindi sulla creazione di una nuova classe RectangleObjectAdapter che avrà al suo interno un oggetto della classe Rectangle e che implementa l'interfaccia Polygon. Si sfrutta l'oggetto adaptee (che è istanza di Rectangle) per adattare i metodi dell'interfaccia richiesti similmente a prima. Vediamo ora un esempio di Composite. Pensiamo a un FileSystem, struttura tipicamente gerarchica ad albero che può presentare elementi semplici (files) e contenitori (cartelle). Si vuole permettere al client di accedere e navigare il File System senza conoscere la natura degli elementi che lo compongono in modo da trattarli tutti allo stesso modo. Per far ciò quindi il client userà la stessa interfaccia per l'accesso a entrambi gli elementi, mentre l'implementazione nasconderà la gestione degli oggetti a seconda della loro reale natura. Il client usa l'interfaccia File tramite cui può stampare il contenuto dell'elemento. Se si parla di un singolo file allora l'implementazione si limita a stampare il singolo file, nel caso delle directory posso stampare l'elenco dei suoi elementi, posso aggiungerci file, rimuoverli, ottenere l'elemento in una certa posizione o ancora ottenere il numero totale di elementi contenuti. (poi è esplosa la rec vabbé) Vediamo un esempio di Decorator, dove consideriamo stavolta uno scenario diverso: abbiamo l'esigenza di monitorare l'invocazione di un metodo senza possibilità di modificare il codice. Si utilizza

quindi il pattern Decorator per esigenze di debug; "wrappiamo" un metodo con delle semplici istruzioni print-screen. L'interfaccia Component fornisce una certa operazione implementata da ConcreteComponent (questa operazione la dobbiamo monitorare), a questo punto si introduce l'interfaccia MyDecorator che estende Component ed è implementata dal LoggingDecorator, che aggiunge l'operazione LoggingDecorator e attraverso la quale viene creato l'oggetto decoratore che contiene l'oggetto da decorare. Ipotizzando di non poter usare il codice di ConcreteComponent per monitorare il metodo operation, utilizziamo il Decoratore definendo l'interfaccia Decorator e la classe LoggingDecorator. Quando si crea l'oggetto decoratore si assegna l'attributo myComponent ed ogni volta che si esegue l'operazione stampo a video dei messaggi che monitorano l'esecuzione del metodo.

Partendo dall'esempio possiamo creare diverse classi concrete Decorator che aggiungano nuove funzionalità, come una classe WaitingDecorator che preveda una pausa durante l'esecuzione. Vediamo come diventa il Class Diagram in seguito all'inserimento di questa classe. Ora vediamo un esempio di Factory Method. Si suppone di avere un'applicazione che legge dati da un file di testo contenente informazioni relative a rilevazioni di letture di contatori per acqua e gas. Abbiamo nel nostro codice una classe dedicata a ciò che legge i vari formati di file: AcquisizioneLettura. Questa implementazione è corretta ma poco flessibile: immaginiamo ad es. che il client inizia a vendere anche energia elettrica e quindi acquisire anche i file con le relative letture. Sarà necessario modificare la classe per gestire questo tipo aggiuntivo di file. Conviene quindi usare il pattern Factory Method, che insieme all'Abstract Factory è Pattern creazionale. L'obiettivo è separare il codice soggetto a modifiche dalla restante parte che resta sempre uguale, come fare? Un'idea è incapsulare la creazione di FileLetturaReader in una nuova classe FileReaderFactory: Si modifica di conseguenza AcquisizioneLettura affinché sia responsabilità della factory gestire la logica di lettura del documento e non del client. In questo modo la classe AcquisizioneLettura non dovrà più esser modificata in caso di nuovi tipi di file da leggere. Ora ipotizziamo che acquisiamo due importanti clienti, che vendono acqua e gas utilizzando però formati XML differenti. Adeguiamo quindi il codice alle nuove esigenze scrivendo due nuove factory Cliente1ReaderFactory e Cliente2ReaderFactory che derivano dalla nostra ReaderFactory. L'unica differenza tra i due codici è che si usano in uno il lettore specifico per il file XML del client 1 e il secondo per il client 2. Possiamo quindi migliorare il codice portando il Factory direttamente in AcquisizioneLettura per rendere la classe autosufficiente, senza che però perda la flessibilità ottenuta fino ad ora. CHE SCHIFO Vediamo infine un esempio per l'observer. Due osservatori sono interessati al cambio di stato di un soggetto osservato. Successivamente uno dei due cancella la sottoscrizione e non riceve più notifiche. In questo caso dobbiamo creare il Subject e l'Observer e le classi concrete rispettive.

METRICHE DI STRUTTURA

Le metriche di struttura aiutano a quantificare la qualità di un software.

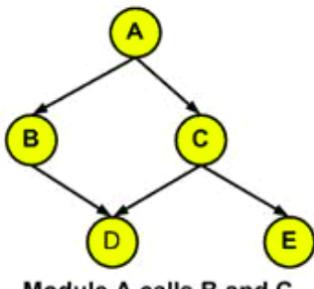
Le misure intermodulari permettono di quantificare le dipendenze tra moduli in base all'architettura software determinata in fase di progettazione preliminare

Un modulo come sappiamo è una sequenza contigua di istruzioni, delimitata da alcuni elementi e che ha un certo identificatore (quando si pensa a un modulo si pensa a una parte di software che può esser compilata indipendentemente).

È importante saper misurare la soluzione architettonale in quanto tutte le decisioni che prendiamo in questa fase hanno impatto significativo sul software risultante, in particolare su attributi di qualità come facilità di implementazione, affidabilità, manutenibilità e riusabilità (nb affidabilità importante non solo per software critico!). Le misure offrono feedback per capire se le caratteristiche del software soddisfano i requisiti.

La relazione tra progettazione preliminare e codice include relazioni 1 a 1 tra i seguenti argomenti indicati nel design e nel codice: moduli, connessioni intermodulari e interfacce data intermodulari.

Un'architettura software sappiamo essere un insieme di componenti che hanno tra loro relazioni di dipendenza, si può quindi concettualizzare usando un grafo (structure chart), dove i nodi sono i moduli e gli archi le relazioni di dipendenza. Le relazioni possono rappresentare diverse cose (es. chiamata di procedura, flusso di dati etc...).



Module A calls B and C
Module B calls D
Module C calls D and E

Quando si definisce l'architettura dei moduli è importante definire il valore della modularità, cioè il grado con cui un software è definito da componenti discrete tale che il cambiamento di una di esse comporta minimo impatto sulle altre componenti.

Si ricorda quindi che alta modularità è desiderabile, in quanto se si ha bassa modularità allora è più facile fare errori -> difficile manutenzione, meno riusabili, meno affidabili etc...

La modularità può essere misurata tramite i seguenti sotto-attributi: cohesion, coupling, morfologia e information flow.

- Coesione: grado con cui un modulo individualmente realizza un task ben definito.

- Coupling: grado di interdipendenza tra moduli

Si voleva massimizzare coesione e minimizzare coupling per ottenere elevato livello di modularità.

- Morfologia: misura la forma della structure chart.

- Information Flow: considera il flusso di informazioni tra moduli -> interconnessione tra moduli non solo dal punto di vista di scambio dati ma anche dal punto di vista del flusso di controllo.

Iniziamo dalla morfologia.

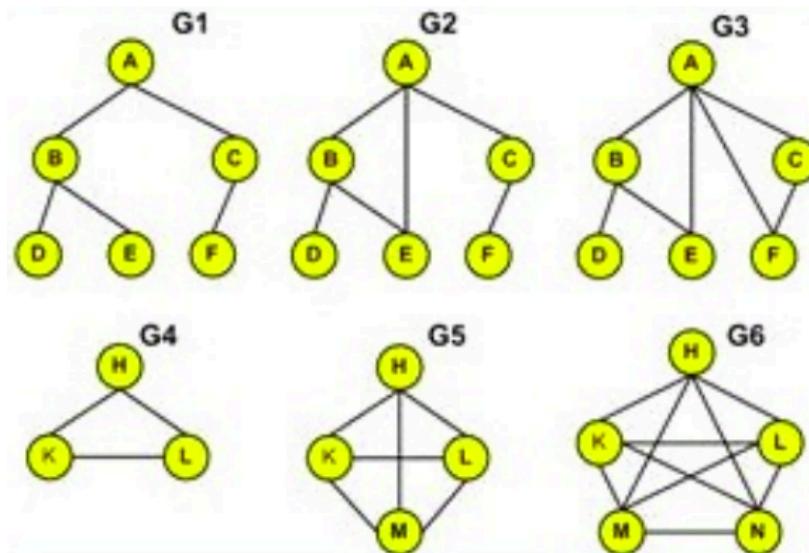
Essa è caratterizzata da:

- Size (numero di nodi e archi),
- Depth: distanza radice-nodo (si vuole un grafo che sia quanto più simile a un albero possibile),
- Width massimo numero di nodi tra tutti i livelli,
- Edge-to-Node Ratio: misura di connettività nel grafo (ossia quanto è denso -> rapporto archi/nodi).

Un concetto importante per misurare la Morfologia è il concetto di Tree Impurity: esso misura quanto il grafo dell'architettura è lontano da un albero.

Sia $m(G)$ quanto il grafo è diverso da un albero -> minore $m(G)$, migliore la morfologia.

Più sono i cicli, peggiore è la morfologia, in particolare non si vogliono ovviamente grafi fortemente connessi o clique. (Si vogliono evitare situazioni come G5 e G6 e avvicinarsi quanto più possibile a G1.)



Come misurare quantitativamente $m(G)$?

$$m(G) = \frac{\text{number of edges more than spanning tree}}{\text{maximum number of edges more than spanning tree}}$$

$$m(G) = \frac{2(e - n + 1)}{(n - 1)(n - 2)}$$

dove e è il numero di archi e n il numero di nodi. Varia tra 0 e 1.

Spanning tree: albero minimo (sottografo di un grafo connesso) che connette tutti i nodi. Il numero massimo di archi per un grafo in più rispetto allo spanning tree è la differenza tra numero di archi di quel grafo con n nodi completo e quel grafo ma come spanning tree -> $n-1$ archi spanning tree e $(n(n-1))/2$ per clique, la differenza porta a $(n-1)(n-2)$ (da qui il denominatore).

Sopra invece numero di archi tra spanning tree e grafo effettivo.

Un altro per misurare sempre la Morfologia è l'Internal Reuse (Yin and Winchester): Essa indica il grado con cui i moduli sono riusati all'interno dello stesso prodotto.

Per misurarlo si usa $r(G) = e-n+1$.

Minore $r(G)$ significa meno riuso -> migliore morfologia in quanto minor interdipendenza
Questa tecnica di misurazione tuttavia non tiene conto del numero di chiamate fatte da un modulo ad un altro e non si tiene conto della dimensione dei moduli -> bisogna utilizzare oltre che questa misura anche Tree Impurity per capire la morfologia.

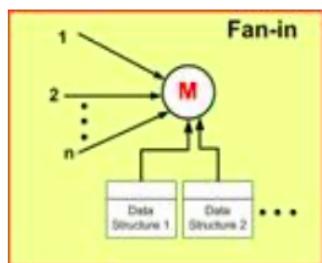
Per approfondire come i moduli dipendano effettivamente uno dall'altro dobbiamo misurare l'Information Flow. Questa misura assume che la complessità di un modulo dipenda da due fattori principali:

- la complessità intrinseca del modulo (il suo codice)
- la complessità della sua interfaccia (ossia quanto è aperto all'ambiente circostante, quanto dipende/influenza altri moduli).

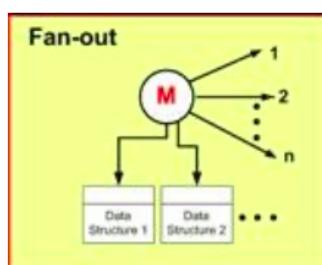
Il livello totale di information flow in un sistema è attributo intermodulare, tuttavia è possibile anche misurare l'information flow tra un singolo modulo e il resto del sistema come quindi attributo intramodulare.

Per misurare l'information flow si devono contare le connessioni tra un modulo con il resto dei moduli del sistema (fan-in e fan-out di un modulo), inoltre si assume che le misure di information flow sono basate su flussi di informazioni sia locali (ossia legati a un modulo che chiama un altro modulo se diretto o il valore di ritorno di un metodo invocato se indiretto) che globali (quando l'informazione tra moduli è condivisa e accessibile a tutti).

L'obiettivo dell'information flow è misurare quanto i moduli dipendano dagli altri, quindi può essere utilizzato per individuare le parti critiche del sistema e capire se la nostra progettazione può essere pericolosa in termini di affidabilità, manutenibilità, riusabilità etc...



- Fan-In di un modulo M: rappresenta il numero di flussi locali (diretti e indiretti) che terminano nel modulo M in aggiunta al numero di flussi globali (strutture dati globali utilizzate dal modulo M) -> tutto ciò che entra nel modulo sia in termini di valori di ritorno che in termini di dati presi da strutture dati globali fa parte del Fan-in del modulo.



- Fan-Out: rappresenta viceversa tutto ciò che "esce del modulo": il numero di flussi locali (diretti o indiretti) che partono dal modulo M più il numero di aggiornamenti che M fa su strutture dati globali.

È importante misurarli in quanto se un modulo ha alto fan-out allora esso probabilmente influenza molti altri moduli, viceversa elevato fan-in indica che il modulo dipende da molti altri moduli. Un modulo con elevato fan in e fan out è sicuramente un modulo che è al centro del sistema, mentre uno con basso fan in e fan out alla periferia del sistema.

L'obiettivo è ovviamente ridurre il fan in e fan out per ogni modulo in quanto se elevato allora indica un modulo complesso che potrebbe portare a più errori in quanto:

- se devo modificarlo devo modificare molti altri moduli e inoltre
- se voglio modificare una funzione devo guardare in più punti.

Si introduce quindi la misura dell'Information Flow (Henry & Kafura).

$$IF(M_i) = [fan-in(M_i) \times fan-out(M_i)]^2$$

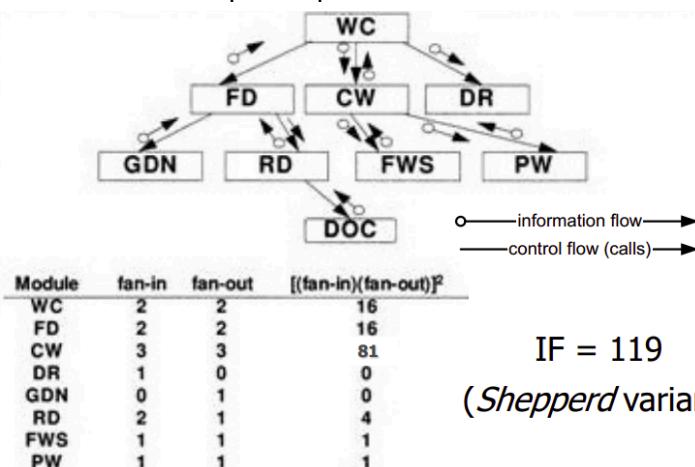
IF for a system with n modules is:

$$IF = \sum_{i=1}^n IF(M_i)$$

Questa misura tiene conto dell'information flow sia come flusso dati che come flusso di controllo (quindi si contano anche le chiamate di un modulo a un altro senza effettivo scambio di dati).

Esiste una variante di Information Flow Measure, la Shepperd, che si limita al flusso di dati.

Vediamo un esempio di questa misura con la variante Shepperd.



Abbiamo una buona structure chart (in quanto ad albero). Ogni arco rappresenta chiamate fatte tra moduli (flusso di controllo) mentre le frecce con pallino flusso di dati.
(autoesplicativo, vedi frecce con pallino fuori e dentro per contare fan in e out per poi calcolare la shepperd measure dove si conta solo flusso di dati).

Structural Measurements

Quando si parla di misure strutturali si fa riferimento alla struttura del modulo, tre principali componenti:

- Struttura del flusso di controllo: sequenza di istruzioni del programma
- Flusso di dati: si tiene traccia dei dati creati e gestiti dal modulo
- Struttura dei dati: organizzazione dei dati indipendenti dal modulo

È importante usare queste misure perché importanti in fase di reverse engineering (manutenibilità), testing (un passaggio importante è path coverage in questo senso, ossia cercare di percorrere tutti i possibili percorsi di esecuzione per rilevare eventuali errori NB se si fanno tutti non vuol dire che il software sia privo di difetti in quanto magari non restituisce errore ma non si comporta comunque adeguatamente), ristrutturazione codice, data flow analysis (si ricorda in questo senso l'importanza di modelli come il data flow diagram)...

Come rappresentare la struttura del singolo modulo? Attraverso un grafo del flusso di controllo. Una volta rappresentata la struttura potremo definirne la complessità usando la complessità ciclomatica.

Basic Control Structure

Si definiscono delle strutture di controllo di base (BCS Basic Control Structure), che sono null'altro che i meccanismi essenziali di control flow usati per costruire la struttura logica del programma.

Tre tipi di BCS:

- Sequenza (serie di istruzioni senza che vi siano altre BCS ad influire su di esse),
- Selezione (if, then, else...),
- Iterazione (while, repeat until ...).

Esistono anche strutture di controllo avanzate ACS che permettono di introdurre nuovi concetti come Chiamata di funzione/procedura, ricorsione, interrupt e concorrenza.

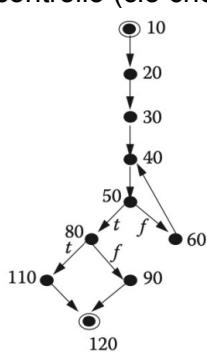
Flowgraph

Mentre nella fase preliminare per utilizzare le metriche abbiamo visto l'architettura come un grafo dove ogni nodo è un modulo, similmente in fase di progettazione dettagliata il flow chart è rappresentato da un grafo che stavolta fa riferimento a un singolo modulo.

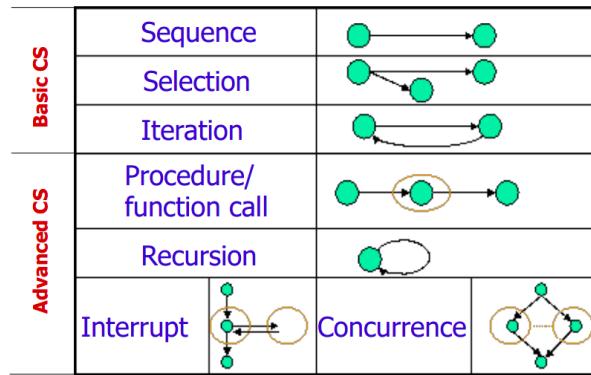
Un flow chart è quindi un grafo diretto dove ogni nodo corrisponde a una istruzione. Esistono diversi tipi di nodi:

- Nodo Procedurale: nodo con un solo arco uscente (significa che quando il nodo termina la propria esecuzione esiste uno e un solo nodo successivo)
- Nodo Predicato: nodi con numero di archi uscenti $\neq 1$ (es. istruzioni decisionali, terminata l'esecuzione del nodo si "sceglie" uno e un solo nodo dei tanti possibili, non è concorrente è decisionale)
- Nodo Start: nodi con numero di archi entranti = 0
- Nodo End: nodi che ha archi in ingresso ma non in uscita

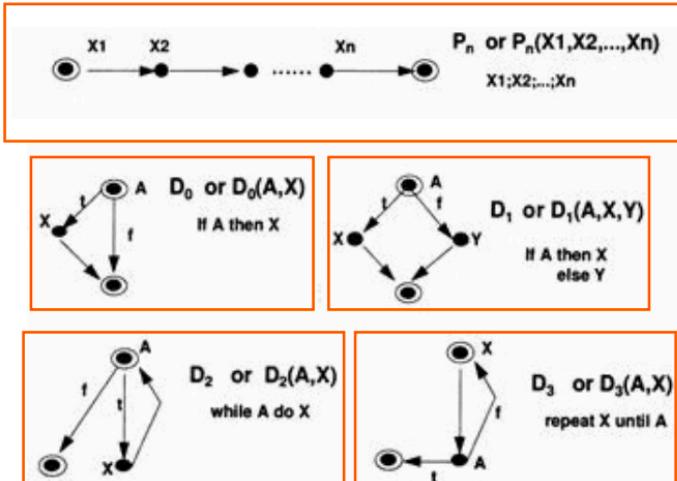
Mentre il nodo rappresenta i vari statement, gli archi rappresentano il vero e proprio flusso di controllo (ciò che avviene nell'eseguire le istruzioni).



Se il codice è ben strutturato, il corrispondente grafo di flusso esibisce dei costrutti di base (sequenza, iterazione, selezione). Ci sono anche strutture avanzate.



Limitandoci alle strutture base, è possibile individuare dei grafi di flusso comuni a tutti i programmi strutturati. (sequenza di statemente, if, if else, while e while do)



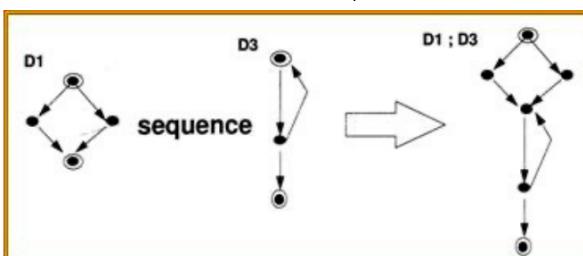
P_n come sequenza di n statement procedurali, D_0 o $D_0(A,X)$ l'if (dove A condizione e X codice da eseguire se la condizione è vera), poi D_1 o $D_1(A,X,Y)$ if then else, while do come D_2 e repeat until D_3 .

While do è il solito while do e repeat sarebbe il do while.

Un grafo di flusso ben strutturato a livello di sottofase di progettazione dettagliata è costituito solo ed esclusivamente da questi sottografi di base, opportunamente combinati.

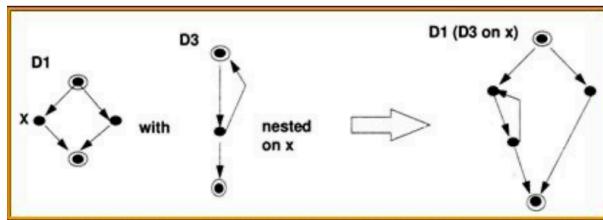
Due possibili operazioni per combinarle:

- Sequenziamento: dati due flow graph F_1 e F_2 . Mettiamo i due flow graph in sequenza rendendo il nodo finale di F_1 il nodo iniziale di F_2 , indichiamo il tutto con il simbolo ; (F_1, F_2).



- Nesting: in questo caso, dati sempre F_1 e F_2 flowgraphs, il nesting di F_2 in F_1 rispetto a X

rappresenta il fatto che l'intero "codice" descritto da F_2 sostituisce la condizione X di $F_1 \rightarrow$ si sostituisce l'arco uscente da X con F_2 . $F_1(F_2)$.

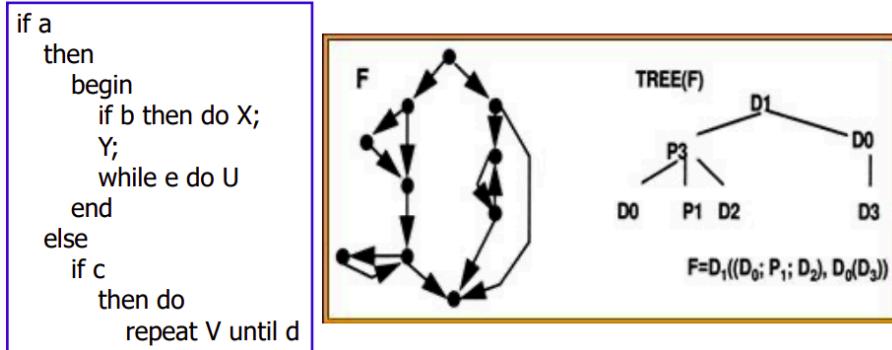


In generale, un Prime Flowgraph rappresenta un flowgraph che non può essere ulteriormente decomposto secondo il sequencing e il nesting.

Quindi P_1 (non P_n !), D_0 , D_1 , D_2 , D_3 sono tutti Prime Flowgraphs e sono dette D-structures (D da Dijkstra).

Ciò che vogliamo fare a questo punto è verificare quanto è D-strutturato il grafo di flusso identificato in fase di progettazione dettagliata, per farlo utilizzeremo il Prime Decomposition Theorem (Fenton-Willy).

Il teorema afferma che ogni flow graph ha un'unica decomposizione in una gerarchia di flow graph primitivi (prime), detta "albero di decomposizione".



Determinare l'albero di decomposizione di un grafo di flusso permette di definire le nostre metriche per capire se il codice risulta ben strutturato.

Daremo esempio di due metriche: la Depth of Nesting e la D-structureness. Esse si definiscono per le primitive di base e poi per applicazioni di sequencing e nesting.

La Depth of Nesting $n(F)$ di un grafo di flusso F è:

- Per Primitive di Base (Primes):

$$n(P_1) = 0; n(P_2) = n(P_3) = \dots = n(P_k) = 1;$$

$$n(D_0) = \dots = n(D_3) = 1$$

- Sequencing: quando si fa sequencing non si aggiunge alcun annidamento, quindi la depth of nesting di una sequenza corrisponde al max tra i singoli sequenziati

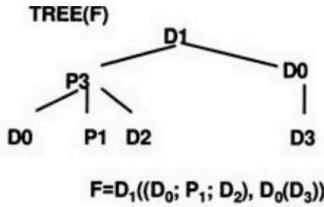
$$n(F_1; F_2; \dots; F_k) = \max\{n(F_1), n(F_2), \dots, n(F_k)\}$$

- Nesting: quando invece si fa nesting allora si introduce un ulteriore livello di annidamento quindi aggiungo 1: $n(F(F_1, \dots, F_k)) = 1 + \max\{n(F_1), \dots, n(F_k)\}$

Più la depth of nesting è grande più il codice è complesso.

- **Example:**

$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$



$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$

$$n(F) = n(D_1((D_0; P_1; D_2), D_0(D_3))) =$$

$$= 1 + \max\{ n(D_0; P_1; D_2), n(D_0(D_3)) \} =$$

$$= 1 + \max\{ \max\{ n(D_0), n(P_1), n(D_2) \}, 1 + n(D_3) \} =$$

$$= 1 + \max\{ \max\{ 1, 0, 1 \}, 2 \} = 1 + \max\{ 1, 2 \} = 3$$

Diciamo in particolare che un programma è strutturato se è D-strutturato $d(F)$. Come prima:

- Primitive di base:

$$d(P_1) = 1; d(D_0) = \dots = d(D_3) = 1, \text{ 0 altrimenti}$$

- Sequencing:

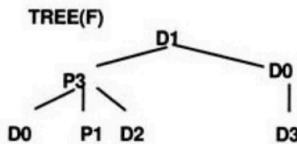
$$d(F_1; \dots; F_k) = \min\{d(F_1), \dots, d(F_k)\}$$

- Nesting:

$$d(F(F_1, F_2, \dots, F_k)) = \min\{d(F), d(F_1), d(F_2), \dots, d(F_k)\}$$

- **Example:**

$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$



$$d(F) = d(D_1((D_0; P_1; D_2), D_0(D_3))) =$$

$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$

$$= \min\{ d(D_1), d(D_0; P_1; D_2), d(D_0(D_3)) \} =$$

$$= \min\{ d(D_1), \min\{ d(D_0), d(P_1), d(D_2) \},$$

$$\min\{ d(D_0), d(D_3) \} =$$

$$= \min\{ 1, \min\{ 1, 1, 1 \}, \min\{ 1, 1 \} \} = \min\{ 1, 1, 1 \} = 1$$

Se $d(F) = 1 \rightarrow$ il grafo di flusso è D-strutturato, ossia costruito partendo solo da primitive di base!

Un'altra misura che permette di valutare la complessità del codice è la Complessità Ciclomatica, essa può essere calcolata o sul flowgraph o sul codice.

Dato un flow graph F, la sua complessità ciclomatica $v(F)$ è pari al numero di archi meno il numero di nodi più 2: $v(F) = e - n + 2$

dove e rappresenta il numero degli archi e n il numero dei nodi.

Questo valore misura quindi il numero di percorsi linearmente indipendenti di F (ossia che tale percorso non è insieme (combinazione lineare) di altri percorsi).

Riguardo il metodo basato non sul flow graph ma su codice, $v(F) = 1 + d$ ossia il numero di nodi predicatori (decisionali) + 1.

- La complessità delle primitive è quindi

$$v(F) = 1 + d.$$

- Se invece si introduce il sequenziamento allora

$$v(F_1; \dots; F_n) = \sum_{i=1}^n (v(F_i) - n + 1)$$

- La complessità ciclomatica in caso di nesting è invece misurata come segue:

$$v(F(F_1; \dots; F_n)) = v(F) + \sum_{i=1}^n (v(F_i) - n)$$

$$v(F) = e - n + 2$$

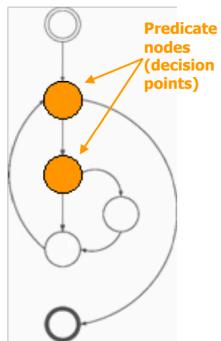
$$v(F) = 7 - 6 + 2$$

$$v(F) = 3$$

or

$$v(F) = 1 + d$$

$$v(F) = 1 + 2 = 3$$



```
#include <stdio.h>
main()
{
    int a ;
    scanf ("%d", &a);
    if ( a >= 10 )
        if ( a < 20 )      printf ("10 < a< 20 %d\n" , a);
        else                printf ("a >= 20    %d\n" , a);
    else                    printf ("a <= 10    %d\n" , a);
}
```

$$v(F) = 1 + d = 1 + 2 = 3$$

La misurazione della complessità ciclomatica rappresenta una misura generale di complessità del nostro codice, più è alto più il codice risulta tipicamente difficile da mantenere e da testare.

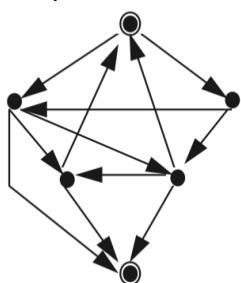
McCabe suggerisce che quando questo numero inizia ad esser superiore a 10 allora il modulo inizia ad esser problematico.

Esiste un'altra misura introdotta proprio da McCabe: la Complessità Essenziale.

Essa è $v_e(F) = v(F) - m$

dove m è il numero di sottografi di F di tipo D_0 , D_1 , D_2 o D_3 (si esclude P).

Deve essere 1 se il grafo è d-strutturato (ha $d(F) = 1$). La complessità essenziale rappresenta quindi il grado con cui il grafo di flusso può essere ridotto attraverso decomposizioni di sottografi D_0 , D_1 , D_2 e D_3 .



Essential complexity = 6

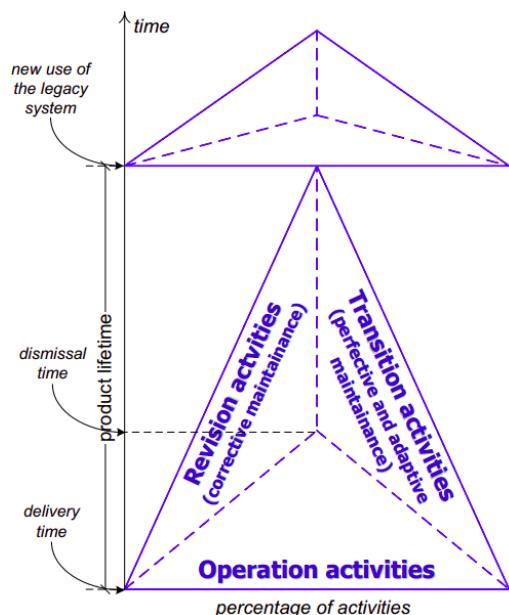
La complessità ciclomatica ha il vantaggio di essere una misura oggettiva di complessità del programma, tuttavia ha gli svantaggi di poter essere usata solo a livello di singola componente, due programmi con stessa complessità ciclomatica potrebbero essere diversi a livello di complessità di programmazione.

QUALITÀ DEL SOFTWARE

La qualità del software (standard IEEE) rappresenta per definizione il grado per cui il software possiede una serie di attributi "desiderabili", può essere analizzata da vari punti di vista:

- Trascendentale: prodotto eccellente. (l'obiettivo però non deve essere sviluppare un prodotto eccellente quanto un prodotto che faccia correttamente quanto deve fare per questo, questo punto non ci interessa).
- Utente: quanto il prodotto software contribuisca a raggiungere gli obiettivi dell'utente.
- Prodotto: si valutano le caratteristiche del prodotto software (correttezza, affidabilità etc..)
- Organizzazione: ci si focalizza sui benefici aziendali -> costi e profitti (se il prodotto è facile da mettere sul mercato, quanto il prodotto permetta di incrementare i profitti/ridurre i costi etc...).

Il concetto di qualità è un insieme di caratteristiche facilmente inquadrabili in modo soggettivo, ma si devono valutare in modo oggettivo per poter essere misurate e dare valutazione quantitativa. Si usa il Quality Model introdotto da McCall: il Quality Triangle.



Sulle ascisse la percentuale di attività che vengono svolte sul prodotto a partire dal momento in cui questo è immesso sul mercato.

Sulle ordinate il tempo, che parte dal momento di delivery (rilascio, momento in cui il software passa dallo stadio di sviluppo a quello di uso/manutenzione).

Dal momento del rilascio il software sarà usato da diversi attori in modi diversi: vi saranno gli utenti che svolgeranno le cosiddette Operation Activities. Mentre gli utenti usano il software gli sviluppatori avvieranno delle attività di monitoraggio e manutenzione pressoché continue per molti motivi.

Man mano che il software viene utilizzato iniziano anche le Revision Activities (manutenzione correttiva) e Transition Activities (manutenzione perfettiva e adattiva).

Ad un certo punto, quando gli utenti terminano di usare il software (Operation Activities), il software entra nello stadio di dismissione.

Come sappiamo questo stadio non è istantaneo e non rappresenta necessariamente la cancellazione del software, ma magari una sua rielaborazione (quindi ulteriori attività di

sviluppo e manutenzione) per arrivare eventualmente a un nuovo utilizzo del sistema legacy (triangolo in alto).

Ciò che fa McCall, per ogni possibile attività di Revision, Transition o Operation, è definire degli attributi di qualità del software (detti nel caso di McCall Indici di Qualità). Quindi qualità come combinazione di attributi desiderabili, nel caso di McCall come combinazione di 12 indici differenziati in base al tipo di attività.

Operation:

- i_1 Correttezza: rappresenta il grado con cui il prodotto soddisfa le specifiche e gli obiettivi degli utenti (quanto il software fa quello che l'utente vuole?)
- i_2 Affidabilità: grado con cui il software esegue le sue funzioni con la precisione richiesta (diverso dall'affidabilità classica che avevamo definito come la probabilità che il software funzioni correttamente in un certo intervallo di tempo)
- i_3 Efficienza: fa riferimento alle risorse di calcolo, quanto potenza mi serve per eseguire il software
- i_4 Integrità: il software è sicuro? Si fa riferimento alla sicurezza del software in termini di accesso esterno indesiderato
- i_5 Usabilità: impegno richiesto per riuscire ad utilizzare il software da parte dell'utente

Revision:

- i_6 Manutenibilità: impegno necessario per individuare e correggere un difetto del programma
- i_7 Testabilità: impegno necessario per testare il prodotto e verificare che funzioni come desidero
- i_8 Flessibilità: impegno richiesto per modificare il prodotto (legato alla complessità del software stesso).

Transition:

- i_9 Portabilità: impegno richiesto per trasportare il prodotto da un ambiente operativo (SO) a un altro
- i_{10} Riusabilità: il grado con cui il prodotto o sue parti possono essere riusate in applicazioni differenti
- i_{11} Interoperabilità: quanto il software può essere accoppiato con altri prodotti
- i_{12} Evolabilità: effort richiesto per aggiornare il prodotto al fine di aggiornare eventuali nuovi requisiti (manutenzione perfettiva anche detta evolutiva)

Secondo McCall in particolare la qualità del software è definita da un vettore contenente i 12 indici di qualità appena visti

$$q = (i_1, \dots, i_{12}).$$

Ciascun indice può essere misurato facendo riferimento ad un insieme di attributi di qualità. $i_j = (a_1, a_2, \dots, a_n)$.

Un attributo può impattare anche più indici di qualità. Gli attributi sono 10:

- Complessità: come livello di comprensibilità,
- Accuratezza: come precisione nel calcolo,
- Completezza: come quanto il software ha implementato in modo completo le funzionalità richieste,
- Consistenza: come utilizzo di approcci di design uniforme,
- Error Tolerance: come quanto il software riesce a "sopravvivere" ad eventuali malfunzionamenti e continuare a funzionare,
- Tracciabilità: grado di relazione tra due o più prodotti nel processo di sviluppo (es. tra

codice e documento requisiti etc..),

- Espandibilità: come quanto il software può essere esteso in termini di storage e funzioni,
- Generalità: come ampiezza di applicazioni potenziali,
- Modularità: come grado di indipendenza tra moduli
- Auto-documentation: quanto il software è in grado di aiutare l'utente attraverso un help in linea.

Quindi McCall definisce alcuni di questi attributi per ogni indice e misura tutti questi attributi secondo specifiche metriche. Vediamo un esempio per 4 indici.

(+/- denotes positive/negative impact)

i₁ **Correctness**

- + Completeness
- + Consistency
- + Traceability

i₇ **Flexibility**

- + Traceability
- + Consistency
- Complexity
- + Modularity
- + Generality
- + Auto-documentation

i₂ **Reliability**

- + Error Tolerance
- + Consistency
- + Accuracy
- Complexity

i₁₂ **Evolubility**

- + Consistency
- Complexity
- + Modularity
- + Expandability
- + Generality
- + Auto-documentation

$$i_1 = (a_3, a_4, a_6)$$

$$i_2 = (a_1, a_2, a_4, a_5)$$

$$i_7 = (a_1, a_4, a_6, a_8, a_9, a_{10}) \quad i_{12} = (a_1, a_4, a_7, a_8, a_9, a_{10})$$

Per misurare gli attributi a loro volta sono necessari tipicamente dei sottoattributi per arrivare ad un'effettiva misura: es. per la modularità a_9 per misurarla si utilizzano tipicamente i sottoattributi come abbiamo visto nelle lezioni precedenti di coesione e coupling, morfologia e information flow (poi esistono anche altri modi, ogni modello usa il proprio e può far quindi riferimento a diverse metriche).

Per questi calcoli, che quindi possono avvenire usando diverse metriche, si introduce il Checklist Method, che mette a disposizione a chi deve valutare la qualità del software delle checklist che tipicamente pongono delle domande alle quali semplicemente l'utilizzatore deve porre una risposta.

Sarà a carico del metodo checklist, una volta fornite le risposte, calcolare il valore complessivo dell'attributo. Una volta calcolati i valori dei vari attributi questi vengono raggruppati per ottenere il livello del corrispondente indice (tenendo ovviamente conto dell'impatto, se positivo o negativo).

Una checklist rappresenta un insieme di domande, e ad ogni domanda sono associate quattro possibili risposte con un valore numerico. Alcune domande potranno essere identificate come Non Applicabili se non si vuole prendere in considerazione la domanda per il calcolo del valore di un attributo, Non Valutabili se invece si vuole scegliere lo score più basso possibile tra quelli elencati nelle quattro risposte alla domanda.

Colui che si prende carico di rispondere a queste domande è il Checklist Evaluation Team, composto da almeno quattro persone che svolgono ruoli differenti e sono competenti in termini di qualità del software.

Si raccomanda che il team sia costituito da: un Quality Assurance Specialist, un Project Leader, un System Engineer, un Software Analyst.

In particolare il metodo per calcolare il valore numerico di ciascun attributo (eccetto che per la modularità) è basato sulla seguente formula:

$$V_{attribute} = \frac{\sum_{i=1}^{\# questions} V_{answer_i}}{\sum_{i=1}^{\# questions} \max(V_{answer_i})}$$

Calcolati i valori per ogni attributo, si procede al calcolo del valore dell'indice.

$$V_{index} = \frac{\sum V_{attribute(+)} + \sum (1 - V_{attribute(-)})}{\text{number of attr. associated to index}}$$

Chiaramente, essendo valori normalizzati, i valori degli attributi sono compresi tra 0 e 1 e lo stesso vale per i valori associati agli indici.

Una volta ottenuti i valori degli indici, vengono definiti i così detti “livelli di accettazione” degli indici di qualità (Acceptance Level Scale)

V_{index}	<i>Quality Level</i>
$0.66 < V \leq 1$	High
$0.33 < V \leq 0.66$	Medium
$0 < V \leq 0.33$	Low

(saltare fino a 167)

Vediamo ora degli esempi di checklist per capire meglio il funzionamento di questa valutazione. Immaginiamo di trovarci nella fase di progettazione architettonale (preliminare), dopo la quale abbiamo prodotto l'architettura del nostro sistema software. Tra i 10 attributi consideriamo da calcolare: a1 Complessità, a8 Generalità e a9 Modularità (per la quale vedremo un metodo di calcolo diverso da quello visto).

Iniziamo dalla Complessità, ecco la checklist:

General aspects

- 1) Are modules, procedures and structure names significant or conform to a standard, if any.
 - 0 : Yes, almost always
 - 1 : Quite often
 - 2 : Rarely
 - 3 : No
- 2) Is the formalism utilised to describe the system architecture only one, or multiple ones are present.
 - 0 : Unique formalism, standard and properly chosen
 - 1 : Few formalisms, standard and properly chosen
 - 2 : Few formalisms and someone out of standard
 - 3 : A lot of formalisms, someone out of standard
- 3) Is the global system design properly structured and easy understandable by people without specific knowledge about the system.
 - 0 : Yes
 - 1 : Almost positive
 - 2 : Not properly structured
 - 3 : Bad structured, and hardly understandable
- 4) Is it possible to deduce the class of information of each single data present into the logical model. Are all the utilisation of this data declared in the documentation and realised with the purpose to read or write that class of information.
 - 0 : No
 - 1 : Rarely
 - 2 : Quite often
 - 3 : Approximately always.

Metrics applications

- 5) If is possible to perform measurements about the program complexity, based on the program calling graph, then use the metrics below defined.
 - 5.1) *Hierarchical* complexity: is the average number of modules per calling-tree level, that is the total number of modules divided by the number of tree levels
 - 0 : 0 up to 4
 - 1 : 4 up to 8
 - 2 : 8 up to 12
 - 3 : Less than 2 or greater than 12
 - 5.2) *Structural* complexity: is the average number of call for each module, that is the number of inter-module calls divided by the number modules.
 - 0 : Less than 2
 - 1 : 2 up to 4
 - 2 : 4 up to 8
 - 3 : Greater than 8

Chiaro come complessità abbia impatto negativo sugli indici -> caso migliore quello per cui si scelgono tutte risposte con 0 (si vuole ridurre la complessità).

Chiaramente le checklist cambiano in base anche al tipo di documentazione che si utilizza (ogni documentazione fornisce informazioni diverse, un conto è lavorare con il documento di specifica requisiti un conto con l'architettura software).

Per quanto riguarda invece l'attributo generalità questo impatta positivamente perciò si

vogliono valori più alti.

- 1) Are all functions offered by modules (and their interfaces) properly planned so that is possible to reuse them in some implementations that wasn't planned at the start time of the project. (i.e. : a specific percentage [18%], or any percentage of any value)

0 : No
1 : Rarely
2 : Quite often
3 : Always

- 2) Concerning modules that haven't enough generality, is it possible to make them more generic with low effort (near 10% of global effort to obtain each of them).

0 : No
1 : Rarely
2 : Quite often
3 : Always

- 3) Are all the data structure manipulations combined into modules specifically dedicated to this.

0 : No
1 : Rarely
2 : Quite often
3 : Always

Vediamo ora come si calcola invece l'attributo Modularità.

Cohesion

- 1) Which is the percentage distribution of system modules according the following four types:

1A modules with coincidental cohesion
1B modules with logical or temporal cohesion
1C modules with procedural or communicational cohesion
1D modules with informational or functional cohesion

Coupling

- 2) Which is the percentage distribution of communication modes (coupling) between modules according to the following four types:

2A pairs of modules with content coupling
2B pairs of modules with common coupling
2C pairs of modules with control coupling
2D pairs of modules with stamp or data coupling

General

- 3) Is the system decomposition organised in a hierarchical way

0 : No
1 : Rarely
2 : Yes, enough
3 : Yes

- 4) Which is the percentage of hierarchical structures (each program contains a comparable number of modules)

0 : less than 70%
1 : within 70% and 80%
2 : within 80% and 90%
3 : more than 90%

Sappiamo che un modo per misurarla riguarda l'utilizzo dei concetti e le misurazioni di Cohesion e Coupling, ma anche di Morfologia e Information Flow.

In questo caso si fa riferimento solo alla coesione e al coupling.

Alle risposte non sono associati valori numerici, ma delle etichette A, B C, o D. Viene richiesto di fornire la percentuale di moduli in base al tipo (coesione coincidentale, logica o temporale, procedurale o comunicativa, informational o functional). Invece di separare i 7 li si raggruppa quindi in gruppi per un totale di 4 risposte. Lo stesso vale per il coupling, dove viene richiesto di definire la distribuzione percentuale di coppie di moduli in base al tipo di coupling (content, common, control, stamp o data). Successivamente si passa sugli attributi generali di modularità e si torna quindi alle classiche checklist. Anche in questo caso, essendo modularità un attributo con impatto positivo sull'indice, le risposte migliori sono quelle tali per cui il valore è più alto.

Vediamo ora come calcolare il valore complessivo di modularità.

$$V_{\text{answer}_1} = \frac{(0 \times \%1A) + (1 \times \%1B) + (2 \times \%1C) + (3 \times \%1D)}{50}$$

$$V_{\text{answer}_2} = \frac{(0 \times \%2A) + (1 \times \%2B) + (2 \times \%2C) + (3 \times \%2D)}{50}$$

where $\%X$ is the percentage of modules (for answer 1) or pairs_of_modules (for answer 2) of type X with respect to the whole set of modules/pairs_of_modules of the system

$$V_{\text{Modularity}} = \frac{V_{\text{answer}_1} + V_{\text{answer}_2} + V_{\text{answer}_3} + V_{\text{answer}_4}}{\max(V_{\text{answer}_1}) + \max(V_{\text{answer}_2}) + \max(V_{\text{answer}_3}) + \max(V_{\text{answer}_4})}$$

Le prime due domande sono calcolate a parte come si vede sopra: in particolare si pesa zero la coesione e il coupling peggiore (coincidentale e content) mentre si pesa a 3 quelle migliori. Dividendo tutto per 50 si normalizza ottenendo quindi ancora un valore tra 0 e 1.

Torniamo a vedere come si effettua la valutazione di questi attributi.

La documentazione come anticipato deve essere analizzata in dettaglio dal Checklist Evaluation Team per rispondere a ciascuna domanda, (tramite tecniche come Walkthrough o Ispezioni).

Ogni membro del team deve rispondere alle domande in modo indipendente senza confrontarsi subito con gli altri.

Durante le riunioni di Walkthrough o i meeting di ispezione i membri del team finalmente si confrontano per verificare di aver dato le stesse risposte alle domande e il perché. Se sono diverse allora bisogna discuterne per arrivare a un'unica risposta comune, per poter finalmente calcolare il valore dell'attributo.

Si dà ad ogni attributo un template contenente la lista di domande della checklist, se la domanda non è valutabile o non applicabile, e il punteggio. Si fa lo stesso per gli indici con la lista degli attributi, se hanno peso negativo o positivo, il punteggio di ogni attributo, il calcolo dell'index e il quality level (low, high o avg).

Software Quality Assurance

L'utilizzo delle checklist (ognuna diversa per ogni attributo e in base al documento di specifica) e la valutazione di qualità fa parte di un'attività più generale molto importante nel progetto software: la Software Quality Assurance. Questa fase del progetto rappresenta l'approccio sistematico per assicurare che sia il processo software (sviluppo) che il prodotto software in sé siano conformi agli standard, processi e procedure stabiliti.

Gli obiettivi della fase SQA sono in particolare migliorare la qualità del software monitorando sia il software che il processo di sviluppo assicurandosi di star seguendo gli standard prestabiliti.

Fare SQA è un'attività molto costosa che richiede personale esperto, tempo effort... quindi l'attività incide sui costi del progetto. Per introdurre un programma SQA è necessario che i top manager che gestiscono il progetto siano d'accordo sul goal e supportino SQA con budget adeguato. A quel punto si dovrà identificare un piano di SQA stabilendo quali sono gli standard di riferimento, per poi realizzarlo al fine di valutare la qualità del software prodotto.

Il ruolo del team di SQA è garantire ai manager responsabili che quanto si sta facendo appunto è conforme agli standard e alle procedure. Si garantisce l'uso di un'appropriata metodologia per lo sviluppo, che si proceda secondo standard e procedure, che vengano realizzate opportune review, che si producano documenti per supportare la manutenzione e il miglioramento del software, che si utilizzi un sistema di software configuration management per garantire la tracciabilità di quanto prodotto, che venga effettuato il testing e che questo venga superato correttamente, che si identifichino deviazioni ed eventuali problemi (se rilevati in modo tempestivo si possono affrontare in modo più semplice e meno costoso).

Gli obiettivi principali di SQA sono quindi ridurre rischi monitorando in modo appropriato il software e il processo di sviluppo, assicurandosi la conformità con standard e procedure e assicurandosi anche che le inadeguatezze e gli eventuali problemi del prodotto software, processo software o standard siano portati immediatamente all'attenzione del management affinché si risolvano velocemente.

È importante dire che SQA non è responsabile della produzione di software di qualità, ma di verificare che chi lavora al prodotto lo faccia correttamente seguendo gli standard e le procedure cercando di capire se vi sono problemi nel mentre (auditing, monitoraggio su chi produce il software per accorgersi se c'è qualcosa che non va).

Parlando di SQA abbiamo detto spesso i termini standard e procedure.

Gli standard in questo contesto rappresentano delle linee guida alle quali il progetto software dovrebbe esser comparato per assicurarsi di procedere bene (gli standard definiscono QUELLO che in generale dovrebbe essere fatto).

Il minimo standard indispensabile da seguire include:

- Documentation Standard (es. che si utilizzi un template per il documento di specifica dei requisiti piuttosto che inventarsene la struttura di sana pianta)
- Design Standard (standard di progettazione, si vuole cioè trasformare i requisiti software in progettazione software utilizzando un'adeguata documentazione progettuale come linguaggi di modellazione, specifiche di definizione degli algoritmi...)
- Code Standard (standard di codifica per ottenere codice che sia quanto più possibile uniforme)

Quando si utilizza il termine "Procedure" invece si fa riferimento alle linee guida che mi suggeriscono COME procedere effettivamente nello sviluppo (nella specifica, progettazione etc...) anche suggerendo chi deve farla, quando e cosa fare con i risultati.

Uno di questi standard per l'SQA Plan è l'IEEE che specifica gli obiettivi, le attività da svolgere, gli standard e procedure da seguire etc...

TESTING

All'inizio abbiamo visto come nel ciclo di vita del software, nella macrofase di sviluppo, non appare una fase dedicata al testing. Questo perché l'attività è così importante da non dover essere trattata come fase a sé stante ma deve essere integrata con ogni altra fase.

Il testing si articola in Verifica e Validazione:

- Verifica: controllo di correttezza del prodotto, che sia conforme con la sua specifica (alla fine di ogni fase verifico che quanto ho fatto sia corretto con ciò che ho ricevuto in input dalla fase precedente), "are we building the product right"
- Validation: "are we building the right product", ciò che sto realizzando deve essere conforme alle aspettative dell'utente.

È possibile quindi realizzare un software corretto (mai al 100% perché nessuno strumento di testing potrà mai garantire che il software sia privo di difetti, obiettivo pulire quanto più possibile da difetti latenti) ma non valido.

Uno dei documenti più importanti dello sviluppo software, il Test Plan, contiene proprio le istruzioni da seguire per poter adottare le tecniche V&V. Possiamo avere:

Testing dinamico: viene usato spesso per identificare le attività condotte in modo dinamico (ossia su artefatti che posso eseguire per effettuarvi sopra controlli), come le Software Inspections

Testing statico quando devo eseguire il testing su artefatti non eseguibili (analisi documentale), come il Software Testing

Esistono diversi tipi di testing:

- Validation Testing: è il testing utilizzato per dimostrare che il software soddisfa i requisiti utente. Normalmente il validation testing viene fatto alla fine dello sviluppo, quando abbiamo a disposizione il codice da testare. Fare validation durante lo sviluppo è molto più difficile, se non si ha a disposizione il codice si deve infatti lavorare con modelli di simulazione del software.

- Statistical Testing: dal punto di vista dell'affidabilità il software si comporta in modo diverso dall'hardware. Questi sono test fatti per riflettere la frequenza degli input dell'utente.

Validare un requisito di affidabilità è ben più difficile che validare un requisito di efficienza.

Chiaramente non si può validare il requisito di affidabilità aspettando ad esempio 3 anni, si devono fare quindi analisi probabilistiche -> testing statistico che si basa molto sull'Operational Profile (variano in base alle classi di utenti) grazie al quale si possono localizzare e risolvere diversi difetti.

Una volta ottenuta la lista di queste istanze dopo il periodo di testing, questa viene data in input a un modello di stima di affidabilità del software che potrà finalmente fornire la stima di affidabilità per capire se il requisito è soddisfatto o meno.

Due obiettivi: pulire il software dai difetti e ottenere la stima di affidabilità per convalidare i requisiti di affidabilità.

Tuttavia questo testing è molto costoso e tipicamente utilizzato solo per quei software di tipo critico (dove i requisiti di affidabilità sono molto stringenti).

- Defect Testing: Racchiude tutte le attività di testing improntate soltanto alla scoperta di difetti latenti. Un Defect Testing di successo è uno che rileva la presenza di difetti. Sappiamo infatti che il software può presentare difetti causati da errori umani che possono portare a guasti.

Defect Testing

Si inquadra in due fasi principali di testing:

- Component Testing: svolta dallo stesso sviluppatore software e fa riferimento al testing di unità e modulo. Il programmatore a cui viene detto di codificare l'unità/modulo, deve verificare che funzioni correttamente basandosi sulle sue competenze.

Ma i moduli fanno parte di un'architettura e interagiscono tra loro secondo specifiche relazioni di dipendenza, a questo punto quindi interviene l'Integration Testing.

- Integration Testing: svolto a livello di sistema e sottosistema da parte di un testing team indipendente. Si occupa quindi di verificare che l'integrazione tra moduli è corretta o meno. Infine si ha un'altra fase, che però non fa parte del Defect Testing, che è lo User Testing (per verificare che il software fa quanto atteso).

Si ricordi ancora una volta che il testing esaustivo è impossibile, non è possibile avere la certezza che non vi siano difetti, perciò bisogna massimizzare il risultato delle attività di testing, testare situazioni tipiche è più importante che testare casi specifici.

Per questa ragione il testing deve esser basato su un sottinsieme ben identificato di possibili test cases, in accordo con le politiche dichiarate dal team di testing indipendente (e non il team di sviluppatori).

Come anticipato le attività di testing sono basate sui Test Cases e sui Test Data.

- Test Cases: gli input che devono essere forniti dal sistema assieme all'output atteso in base al requisito che si vuole soddisfare. I Test Cases sono tipicamente generati manualmente (costoso!) dato che è difficile generare automaticamente gli output in base a specifiche informali (i requisiti possono essere informali)

- Test Data: gli input generati per cercare di identificare quanti più difetti possibile. I test data sono generabili automaticamente.

Sia i Test Cases che i Test Data sono utilizzati nel processo di Defect Testing.

Black Box Testing

Nel black box testing il software è preso in considerazione semplicemente dal punto di vista di esecuzione, senza vedere cosa avviene al suo interno.

In questo caso i test case sono derivati dalla specifica del sistema e i tester semplicemente forniscono l'input alla componente/sistema e confrontano l'output con quello atteso del test case. Se non corrispondono -> test ha avuto successo e ho riscontrato un difetto.

Questo testing black box è anche detto Testing Funzionale in quanto fa riferimento alle funzionalità del software e non alla sua implementazione.

Quindi il black box testing mette in evidenza solo il comportamento anomalo, che dovrà essere successivamente investigato per trovare i difetti.

Equivalence Partitioning.

Questa tecnica si basa sul fatto che i dati di input e i corrispondenti output possono essere divisi in classi di dati, all'interno delle quali i valori portano il programma a comportarsi in modo simile. Ognuna di queste classi è chiamata partizione d'equivalenza, e il programma si comporterà in modo simile per ogni class member.

L'idea è di scegliere alcuni valori da ognuna di queste partizioni per definire i Test Cases.

La tecnica suggerisce di testare il programma anche per input diversi da quelle che si devono testare (valori non validi).

In generale quindi l'equivalence partitioning è una tecnica che ha senso utilizzare in caso si faccia riferimento a requisiti numerici o in casi in cui le pre-condizioni e post-condizioni della funzione ci permettono di capire i possibili scenari e quindi identificare le partizioni.

Test Guidelines

Quando si effettua l'attività di testing possiamo farci guidare da delle Testing Guidelines per limitare il numero di input da testare, soprattutto per quel che riguarda il testing di sequenze (array, liste etc...). (NB le Testing Guidelines derivano proprio dall'identificazione di partizioni via equivalence partitioning).

Esempi di queste linee guida: fare il test con sequenze che hanno solo un singolo valore, usare sequenze di dimensioni differenti, utilizzare sequenze dove il primo, l'ultimo e l'elemento in mezzo siano utilizzati (acceduti) e testare liste vuote.

Structural Testing (anche detto White Box Testing).

In questo caso durante il testing si ha accesso all'intero codice del programma e i relativi test cases sono derivati dal programma stesso, e non dalla specifica.

Mentre con il black box l'obiettivo era trovare input con output anomali, con la white box l'obiettivo è testare quanti possibili program statements (non tutti i possibili percorsi di esecuzione).

Si vogliono quindi identificare i test cases affinché ogni program statement sia stato testato almeno una volta.

Si identifica in particolare come obiettivo un certo numero, detto Testing Coverage, che rappresenta la percentuale di program statement (istruzioni) che si vogliono raggiungere (in base ai tempi, costi, budget a disposizione il numero cambia).

Così come nel black box, dove le linee guida ci suggerivano come la ricerca di un elemento in un array, nel white box è possibile utilizzare la tecnica di equivalence partitioning.

Si identificano un insieme di test case nei quali il valore da cercare sta al confine di ognuna di queste partizioni (si assume facendo ciò, come per black box, che tali valori siano altamente rappresentativi). (ES: binary search)

In definitiva anche in questo caso, come per black box ma in generale per defect testing, non si testa la funzione (ossia si derivano test cases) al fine di valutare il soddisfacimento di uno o più requisiti ma soltanto la correttezza della funzione in sé.

L'obiettivo è in questo caso a differenza del black box però è esercitare tutte le istruzioni del programma.

Un approccio diverso, sempre per white box (quindi che richiede la conoscenza della struttura in dettaglio del programma), è quello orientato al testare non tutte le possibili istruzioni ma tutti i possibili percorsi. In questo caso implicitamente si effettuerrebbe anche il

white box testing classico in quanto se si effettuano tutti i percorsi è ovvio che si eseguano anche almeno una volta tutte le istruzioni del programma.

Path Testing

L'obiettivo chiaramente è cercare di determinare un insieme di test cases per cui ogni percorso del programma sia eseguito almeno una volta (è più costoso).

Infatti il numero di possibili percorsi presenti in un programma/flow graph sappiamo essere rappresentato dalla metrica di Complessità ciclomatica. Maggiore la dimensione del programma -> maggiore la complessità ciclomatica -> difficile gestione del path testing.

Il punto di partenza per effettuare Path Testing è proprio il flowgraph del programma, che rappresenta tutti i suoi possibili percorsi.

L'obiettivo del path testing è in realtà individuare ed eseguire tutti i possibili percorsi indipendenti, dove con indipendenti si intende il fatto che nel percorso che ho individuato esiste almeno una freccia che non faceva parte di un altro percorso.

Quindi per prima cosa dobbiamo individuare questi percorsi indipendenti.

Una volta individuati, l'obiettivo è identificare test cases affinché tutti i percorsi in questione siano eseguiti almeno una volta.

Il numero di percorsi indipendenti equivale alla complessità ciclomatica

Tipicamente i flow graph sono molto più grandi e complessi -> si utilizzano Dynamic Program Analysers o Profilers che sono strumenti di testing che aiutano i tester a capire quante istruzioni sono state eseguite e quale percorso.

Questo per quanto riguardava il testing dal punto di vista di componenti. Sappiamo tuttavia che nello sviluppo di software medio-grandi il numero di queste componenti, questi moduli dovranno essere integrati nella fase di integrazione. Esiste quindi un testing da fare per convalidare l'integrazione di tutti i moduli.

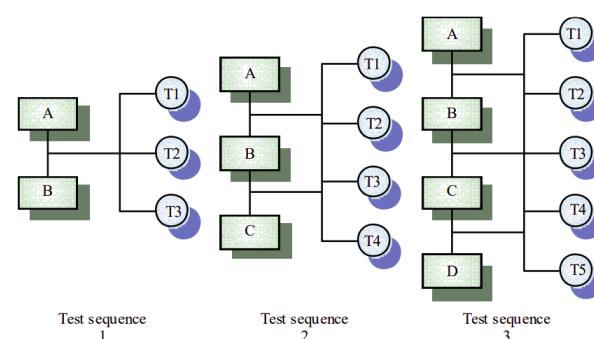
Integration Testing

Per il testing d'integrazione non ci interessa più entrare in dettaglio nei singoli moduli.

Testing di tipo black box con test cases derivati direttamente dal Documento di Specifica.

La difficoltà principale in questo tipo di testing è identificare gli errori, in quanto possono esservi interazioni complesse tra le componenti del sistema e quando si scopre un output anomalo può esser complesso capire il perché.

Per tale ragione si suggerisce di adottare un approccio incrementale nell'Integration Testing. Per iniziare la fase di Integration Testing non è necessario che tutte le componenti siano disponibili, ma solo la parte d'interesse. Per questo è possibile procedere con il testing incrementale:



Sapendo che le componenti possono essere integrate a vari livelli come visto (livello gerarchico, dalle componenti a basso lvl fino alle più importanti ad alto lvl) possiamo utilizzare due approcci:

- Top-down testing: si parte dai livelli più alti di gerarchia fino ad arrivare alla radice per componenti "meno rilevanti". Poiché si parte da un insieme vasto di componenti è possibile che queste non siano presenti, in tal caso sono sostituite con degli stubs, ossia delle funzioni che emulano il comportamento della componente.
- Bottom-up testing: si procede dall'integrare le componenti individuali a basso livello di gerarchia fino al sistema completo.

Non esiste un approccio migliore, tipicamente anzi queste strategie sono combinate in base alle situazioni (es. disponibilità di componenti, stub etc...).

Come detto nell'approccio top-down le componenti ad alto livello sono sviluppate prima di quelle a basso livello -> si devono usare gli stub che hanno la stessa interfaccia del componente ma funzionalità limitate (un emulatore della funzione che sarà integrata dal modulo).

Si parte quindi dalle componenti a livello più alto sostituendo quelle di basso livello non ancora disponibili con stubs. Quando le componenti saranno disponibili, si sostituiranno e così via.

Riguardo il Bottom-up testing invece si parte da componenti più in basso nella scala gerarchica iterativamente fino a testare l'intero sistema. Poiché le componenti sono disponibili per il test si utilizzano dei Test Drivers, codice che eserciterà il componente per valutarne la capacità di integrazione con i livelli successivi.

Bottom-up e Top-down possono essere confrontati da 4 punti di vista:

- Architectural Validation: con top-down è più facile identificare gli errori.
- System Demonstration: top-down permette di costruire demo con funzionalità limitate (perché devo includere gli stubs) che può essere usata come strumento di convalida fin dall'inizio del testing.
- Test Implementation: è più implementare tramite bottom-up.
- Test Observation: ossia valutare i risultati prodotti dall'attività di testing, in entrambi i casi è difficile capire in base agli output dove si trova l'errore e possono essere necessari ulteriori test.

Interface Testing

Due componenti che interagiscono tra loro conoscono solo la loro rispettiva interfaccia, non i dettagli implementativi.

Ogni modulo o sottosistema ha un interfaccia ben definita, utilizzata per chiamare quel modulo/sottosistema da altri componenti.

L'obiettivo di questo testing dell'interfaccia è scoprire difetti introdotti a causa di errori d'interfaccia o assunzioni non valide relative all'interfaccia.

Questo testing è molto importante nel mondo object oriented perché gli oggetti sono definiti secondo la loro interfaccia.

L'obiettivo dell'interface testing è quindi identificare dei test cases che testino l'interfaccia del componente e non la sua struttura interna.

Esistono 4 tipi di interfaccia che possono essere utilizzati per realizzare l'interazione tra componenti/sottosistemi:

- Interfaccia basata su Parametri: le componenti si passano dei parametri per interagire
- Interfaccia basata su Memoria Condivisa: le componenti interagiscono accedendo in modalità rw ad un blocco di memoria condiviso tra loro
- Interfaccia Procedurale: la componente incapsula un insieme di procedure che possono essere chiamate da altre componenti. (usate in sistemi basati su tipo di dato astratti)
- Interfaccia basata su scambio di Messaggi: una componente richiede un servizio da un'altra componente inviandogli un messaggio e attendendo una risposta. (usate in architetture C/S)

Gli errori legati all'interfaccia sono i seguenti:

- Utilizzo Scorretto dell'Interfaccia: es. una componente ne chiama un'altra scrivendo i parametri in modo sbagliato.
- Incomprensione dell'Interfaccia: il componente chiamante potrebbe fare delle assunzioni errate nei confronti del comportamento del componente chiamante.
- Timing Error: le componenti lavorano a velocità differenti e quindi non sincronizzate.

Per ottimizzare le attività di interface testing si possono fornire linee guida, che includono: progettare test case che facciano fallire il componente (es. usando parametri sbagliati), usare stress testing (progettare test case che generano molti più messaggi rispetto a quelli generati normalmente, per testare la robustezza), modificare l'ordine con cui i componenti vengono attivati.

In generale sono le tecniche statiche quelle più adatte all'interface testing.

Ora di queste guidelines approfondiamo lo Stress Testing.

L'obiettivo dello stress testing è pianificare una serie di test dove il carico viene incrementato di volta in volta finché le prestazioni del sistema diventano inaccettabili.

Ovviamente lo stress testing deve essere calibrato affinché il sistema non fallisca in modo catastrofico, inoltre il fallimento non deve causare una perdita eccessiva di servizi e dati. Questo testing è particolarmente importante in sistemi distribuiti, dove si possono osservare cali prestazionali a causa di sovraccarichi di rete.

Altro aspetto importante è l'Object Oriented Testing, dove le componenti da testare sono classi che si istanziano in fase di esecuzione come oggetti.

In questo caso l'unità di codice è tipicamente superiore rispetto alla singola funzione (infatti la classe tipicamente incapsula un insieme di funzioni) -> si devono estendere in questo senso gli approcci whitebox.

Inoltre un sistema OO è costituito da un insieme di oggetti che si scambiano messaggi -> non è un sistema organizzato gerarchicamente come visto in precedenza e per questo sistemi di questo tipo non particolarmente adatti ad approcci top-down o bottom-up.

In generale quindi l'obiettivo deve essere testare il sistema OO per livelli, testando anzitutto i metodi degli oggetti (qui posso usare approcci black e whitebox), poi gli oggetti/classi nella loro interezza, poi cluster di oggetti che interagiscono tra loro, fino al sistema OO completo.

Vediamo cosa significa fare testing di classi/oggetti:

Fare un test coverage completo di un oggetto significa testare tutte le operazioni ad esso

associate, assegnare valori a tutti i suoi attributi e testare l'oggetto in tutti i suoi possibili stati
L'ereditarietà rende più difficile il testing di oggetti: se la mia classe fornisce un insieme di operazioni ereditate da sottoclassi -> tutte le sottoclassi in questione dovrebbero a loro volta essere testate.

L'integration Testing per sistemi OO riguarda non più un insieme di componenti ma un insieme di classi, l'idea è di analizzare un cluster di oggetti che interagiscono tra di loro. Si parla quindi di Cluster Testing, ossia testare le classi insieme.

Per il Cluster Testing esistono altri 3 approcci:

- Use-case o scenario testing: il testing è basato sulle interazioni con il sistema (a tutti gli effetti si testano i casi d'uso). Ha il vantaggio di testare le funzionalità come se fossero usate dall'user.
- Thread Testing: Consiste nel verificare che il sistema risponda correttamente a una specifica sequenza di eventi.
- Object interaction testing: viene testata una sequenza di messaggi tra oggetti.

Parlando del testing basato su scenario, gli scenari sono come detto identificati da casi d'uso, che però non potrebbero includere abbastanza informazioni per derivare un insieme appropriato di test cases. Quindi il caso d'uso può essere affiancato dal diagramma d'interazione sequence diagram definito in fase di specifica dei requisiti, che può aiutare in questo senso.

Per gestire l'attività di testing per software di dimensioni medio-grandi si utilizzano diversi strumenti, tra cui i Testing Workbenches. Si tratta di un insieme organizzato di tool che permettono di organizzare correttamente l'attività di testing anche ottimizzando l'uso di risorse in termini di tempi/costi.

Spesso questi workbenches sono specifici per l'organizzazione, alcuni degli strumenti sono realizzati proprio dall'organizzazione stessa.

BPMN

1. Gestione dei Processi Aziendali (BPM - Business Process Management)

Le organizzazioni svolgono attività per fornire prodotti e servizi ai propri clienti, il cui risultato è l'esito di diverse attività eseguite manualmente dai dipendenti, con l'ausilio di sistemi informativi, o automaticamente dai sistemi informativi. Questi prodotti e servizi sono forniti in base a obiettivi aziendali come l'aumento della soddisfazione del cliente, la riduzione dei costi, il miglioramento dell'efficienza, la riduzione dei tempi di esecuzione e la riduzione dei tassi di errore. Per raggiungere tali obiettivi, le risorse dell'impresa (dipendenti, sistemi informativi) devono collaborare.

La soluzione consiste nel definire, analizzare, attuare e monitorare i cosiddetti processi aziendali.

- Processo Aziendale (BP): È un insieme di attività logicamente correlate che vengono eseguite in modo coordinato per raggiungere congiuntamente un obiettivo aziendale. Un BP è composto da eventi, attività e punti decisionali, e il suo risultato ha valore per almeno un cliente del BP.

- Workflow o Processo Aziendale Automatizzato: Un processo che è automatizzato, in tutto o in parte, da un sistema software che trasferisce informazioni tra i partecipanti secondo le dipendenze temporali e logiche del modello di processo.

- Interazioni dei Processi Aziendali:

Orchestrazione dei Processi Aziendali: BP eseguiti all'interno dell'organizzazione e controllati in modo centralizzato, simili a un'orchestra con il suo direttore.

Coreografia dei Processi Aziendali: BP che richiedono interazione con BP di altre organizzazioni, sincronizzate tramite scambio di messaggi, simili a ballerini che eseguono una propria danza secondo una coreografia comune.

Concetti Chiave del Processo Aziendale: Evento, Attività, Decisione, Risultato, Cliente, Partecipante, Oggetto.

Business Process Management (BPM): È un approccio incentrato sul processo per migliorare le prestazioni, che combina le tecnologie dell'informazione con metodologie di processo e governance. Include concetti, metodi, tecniche e strumenti per supportare tutte le fasi del ciclo di vita del BP.

Discipline Correlate al BPM: Il BPM trae spunto da:

Total Quality Management (TQM): Miglioramento continuo della qualità di prodotti e servizi, ma focalizzato su prodotti/servizi, non sui processi.

Lean Production: Eliminazione delle attività che non aggiungono valore al cliente.

Six Sigma: Minimizzazione dei difetti (errori) utilizzando misurazioni dell'output. Il BPM combina l'approccio di miglioramento continuo del TQM, i principi e le tecniche di Lean e Six Sigma, sfruttando le capacità della tecnologia dell'informazione.

Stakeholder del BPM:

- Chief Process Officer: Ruolo di alta direzione, gestisce globalmente i BP dell'impresa.
- Business Engineer: Esperto di dominio, definisce gli obiettivi strategici dei BP.
- Process Designer: Modella i BP interagendo con gli altri stakeholder.
- Process Responsible: Incaricato di garantire la corretta esecuzione del BP.
- System Architect: Imposta i sistemi informativi per attuare il BP.
- Developers: Professionisti IT che implementano il BP nei sistemi.
- Process participants and workers: Eseguono le attività assegnate nell'istanza del BP.

Livelli dei Processi Aziendali:

- Processi Aziendali Organizzativi: Processi di alto livello, definiscono le funzionalità aziendali senza dettagli tecnici.
- Processi Aziendali Operativi: Definizione dei modelli BP, specifica di attività e relazioni.
- Processi Aziendali Implementati: Includono informazioni tecniche per attuare il BP su una piattaforma specifica.

Ciclo di Vita del Processo Aziendale: Consiste in fasi correlate secondo un modello di sviluppo a spirale, con possibili approcci incrementali ed evolutivi. Le fasi principali sono:

1. Identificazione del Processo: Primo passo per il team BPM, consiste nell'identificare i processi rilevanti e le loro relazioni (architettura del processo). Richiede la definizione di misure di performance (costo, tempo, qualità). Include due fasi:

- Designazione: Comprendere i processi dell'organizzazione e le loro interrelazioni, trovando un compromesso sul numero di processi da identificare (pochi con ampio scopo vs. molti con piccolo scopo).
- Valutazione: Priorizzare i processi in base a importanza, disfunzione e fattibilità.

2. Architettura del Processo: Un modello concettuale che mostra i processi e le relazioni tra di essi (tipicamente produttore-consumatore). Può essere definita considerando il tipo di caso (classificazione dei casi gestiti) e la funzione (scomposizione dell'organizzazione). Una matrice può essere usata per selezionare combinazioni di funzioni e tipi di caso che formano un processo aziendale.

3. Modellazione del Processo (o Discovery/Design): Mira a comprendere in dettaglio il processo aziendale. Il risultato sono uno o più modelli "as-is" che documentano lo stato attuale dei processi rilevanti, spesso sotto forma di diagrammi (flowchart) per ridurre l'ambiguità. Fasi della Modellazione: Costituzione del team, Raccolta delle informazioni, Modellazione, Garanzia della qualità.

Ruoli del Team di Modellazione:

- Analisti di Processo: Familiarità con i linguaggi di modellazione, comprensione limitata del processo, dipendono dagli esperti di dominio.
- Esperti di Dominio: Conoscenza approfondita del processo, non familiarità con i linguaggi di modellazione. I ruoli sono complementari.

Tecniche di Raccolta Informazioni: Basate su evidenze (analisi documenti, osservazione), basate su interviste, basate su workshop. Differiscono per obiettività, ricchezza, tempo e immediatezza del feedback.

Metodo di Modellazione (5 stadi): Identificare i confini del processo, Identificare attività ed eventi, Identificare risorse e loro trasferimenti, Identificare il flusso di controllo, Identificare elementi aggiuntivi.

MOF (Meta Object Facility) e BPMN: La complessità delle istanze nella modellazione è gestita con livelli di astrazione (MOF a 4 livelli: M3 meta-metamodello, M2 metamodello, M1 modello, M0 istanza). BPMN è un metamodello M2 definito da OMG utilizzando MOF.

4. Analisi del Processo: Identificazione e analisi dei problemi dei processi. Il risultato è una raccolta strutturata di problemi, quantificati utilizzando misure di performance (KPI - Key Performance Indicator).

Misure di Performance:

- Tempo: Cycle time (o throughput time), Processing time (o service time), Waiting time (tempo di inattività).
- Costo: Costi fissi (non influenzati dall'intensità di elaborazione) e costi variabili (correlati alla quantità).
- Qualità: Qualità esterna (soddisfazione del cliente) e qualità interna (qualità dal punto di vista del partecipante).
- Tecniche di Misura delle Performance: Analisi del flusso, Teoria delle code, Simulazione (esegue istanze virtuali del processo per statistiche).

5. Riprogettazione del Processo (Redesign): Identificazione e analisi di rimedi per i problemi. Propone una versione riprogettata ("to-be") del processo, confrontando diverse opzioni di cambiamento in termini di misure di performance. Cambiare un processo è difficile a causa della resistenza dei lavoratori e dei costi di modifica dei sistemi informativi.

6. Implementazione del Processo: Vengono eseguite le modifiche necessarie per passare dal processo "as-is" al processo "to-be". Include:

- Gestione del Cambiamento Organizzativo: Attività per cambiare il modo di lavorare dei partecipanti (spiegare i benefici, piano di gestione, formazione, monitoraggio).

- Automazione del Processo: Sviluppo e implementazione di sistemi IT per eseguire il processo "to-be".

7. Monitoraggio del Processo: Monitoraggio e analisi dei dati rilevanti raccolti sul processo per identificare possibili aggiustamenti (stato, eccezioni, tempi di esecuzione, utilizzo delle risorse). È uno sforzo continuo per rilevare colli di bottiglia, errori ricorrenti o deviazioni dal comportamento atteso e identificare rimedi.

Business Process Model and Notation (BPMN)

BPMN è uno standard dell'Object Management Group (OMG) che fornisce una notazione per descrivere i Processi Aziendali, comprensibile da analisti, sviluppatori, lavoratori e manager. La sua importanza risiede nella formalizzazione della semantica di esecuzione e nella definizione formale del metamodello.

Componenti di un Linguaggio di Modellazione:

- Sintassi: Insieme di elementi di modellazione e regole per combinarli (es. attività, eventi, gateway, flussi di sequenza).
- Semantica: Associa elementi sintattici a descrizioni testuali per un significato preciso (comportamento degli elementi). Il concetto di token è usato per descrivere come gli elementi BPMN si comportano, simulando il flusso attraverso il processo.

Notazione: Definisce i simboli grafici per gli elementi.

Elementi Base BPMN:

- Flusso di Sequenza (Sequence Flow): Connette gli elementi del modello, mostrando il loro ordine di esecuzione.
- Evento di Inizio (Start Event): Punto di inizio del processo.
- Evento di Fine (End Event): Punto in cui termina il flusso del processo.
- Task (Attività Atomica): Unità di lavoro che non può essere ulteriormente scomposta.
- Gateway: Usati per controllare come i flussi di sequenza interagiscono mentre convergono e divergono all'interno di un processo.
- Exclusive Gateway (XOR): Crea percorsi alternativi, solo uno dei quali può essere intrapreso. Converge percorsi alternativi senza sincronizzazione.
- Parallel Gateway (AND): Crea percorsi paralleli senza condizioni. Converge percorsi paralleli, aspettando che tutti i flussi in ingresso siano completati prima di proseguire.
- Inclusive Gateway (OR): Crea percorsi alternativi ma anche paralleli. Tutte le espressioni di condizione vere sono attraversate da un token. Converge una combinazione di percorsi alternativi e paralleli, sincronizzando i token attesi.

Modellazione delle Risorse:

- Pools: Rappresentano entità organizzative indipendenti che non condividono sistemi comuni per la comunicazione implicita (es. Cliente e Fornitore).
- Lanes: Rappresentano classi di risorse multiple all'interno della stessa organizzazione che condividono sistemi comuni (es. Dipartimento Vendite e Dipartimento Marketing).

Flusso di Messaggi (Message Flow): Mostra il flusso di messaggi tra due partecipanti (pools separate). Non può connettere oggetti all'interno dello stesso pool.

Artefatti: Informazioni aggiuntive non direttamente legate ai flussi (Associazioni, Gruppi, Annotazioni di testo).

Oggetti Dati (Data Objects): Rappresentano dati e documenti scambiati nel processo (Data object, Collection data object, Data input, Data output, Data store).

Tipi di Processo BPMN:

- Orchestrazione: BP interno a una specifica organizzazione.
- Collaborazioni: Mostrano le interazioni tra due o più partecipanti (pools) tramite flussi di messaggi.
- Coreografie: Modella l'interazione tra processi.

Task Specializzati: Task di invio/ricezione messaggi, User activity (umana con software), Manual activity (senza software), Service calling (usa servizio esterno), Script (eseguito da motore BP).

Definizione degli Eventi: Un evento è una composizione di:

- Posizione: Start (inizio), Intermediate (intermedio), End (fine).
- Natura: Catch (cattura, genera token) o Throw (lancia, può essere catturato).
- Impatto: Interrupting (interrompe) o Non-Interrupting (non interrompe).
- Tipo: Messaggio, Timer, ecc..

Event-Based Gateway: I percorsi alternativi si basano su eventi che si verificano; il primo evento attivato determina il percorso.

Introduzione alla Modellazione e Simulazione (M&S - Modeling & Simulation)

Simulare, in senso scientifico, significa riprodurre il comportamento di un dato sistema, processo o fenomeno tramite l'uso di un modello. Implica imitare o emulare l'esecuzione, la sperimentazione o l'esercizio di un modello per un obiettivo specifico (es. risoluzione problemi, formazione).

Sistema Sotto Indagine: Può essere un sistema esistente (attuale) o un sistema ipotetico (da realizzare).

Modello: Una rappresentazione astratta del sistema, basata su ipotesi/approssimazioni logiche e matematiche su come funziona.

Perché la Simulazione è Richiesta:

Il modello matematico è difficile o impossibile da definire/risolvere analiticamente, o non considera la complessità del problema.

La sperimentazione non è possibile (sistema non esiste), non è sicura (rischio di perdite), o non è conveniente (troppo costosa). Il concetto scientifico di simulazione risale agli anni '40 con la "Monte Carlo analysis".

Modeling and Simulation (M&S): La simulazione deve sempre avere un modello, e la modellazione è una parte essenziale. M&S è al centro del nostro processo di pensiero ("costruiamo modelli mentali e li simuliamo per trarre conclusioni").

Benefici della Simulazione: Aiuta a contenere i costi, ridurre gli errori (aumentare la qualità), risparmiare tempo e fatica. Oggi integra i pilastri tradizionali della scienza: teoria e sperimentazione/osservazione.

Simulazione al Computer: Uno sforzo di simulazione realizzato tramite una piattaforma basata su computer. Implica progettare un modello, implementarlo come software eseguibile, eseguirlo su una piattaforma e analizzare l'output.

Aree di Applicazione: Progettazione di sistemi manifatturieri, valutazione di sistemi militari, requisiti hardware/software per reti/sistemi informatici, progettazione di sistemi di trasporto, valutazione di organizzazioni di servizi, analisi di processi aziendali, sistemi di inventario, sistemi finanziari/economici.

Usi Intesi: Risoluzione dei Problemi: Valutazione, Comparazione, Predizione, Analisi di Sensibilità, Ottimizzazione, Ranking e Selezione.

Tassonomia LVC (Tipi di Simulazione):

- Live: Persone reali che operano attrezzature reali.
- Virtual: Persone reali che operano attrezzature simulate (es. simulatore di volo).
- Constructive: Persone simulate che operano attrezzature simulate.

Piattaforme di Esecuzione della Simulazione:

- Simulazione Locale (o Sequenziale, Centralizzata): Software eseguito su un singolo processore.
- Simulazione Parallela: Software eseguito su un host multi-processore.
- Simulazione Distribuita: Software eseguito su più host connessi tramite rete (es. Internet-based DS).

Approcci M&S: Classificati in base a:

- Rappresentazione del modello: Simulazione a eventi discreti, continua, Monte Carlo, basata su agenti, dinamica dei sistemi.
- Esecuzione del modello: Sequenziale, parallela, distribuita/web-based.
- "In-the-loop": Hardware-in-the-loop, Software-in-the-loop, Human-in-the-loop.

Monte Carlo M&S: Basata sulla generazione di numeri (pseudo-)casuali. Utilizzata per risolvere problemi complessi attraverso ripetizioni casuali, come stimare un'area irregolare.

Agent-based M&S: Utilizza un modello che rappresenta più entità autonome e le loro interazioni. Gli agenti reagiscono agli eventi nel loro ambiente, sono autonomi, adattivi, reattivi, proattivi, "sociali" e possono apprendere. Esempi: evacuazione edifici, simulazione folle, diffusione virus.

Simulazione a Eventi Discreti (DES - Discrete Event Simulation): Modella un sistema che evolve nel tempo. Componenti:

- Variabili di Stato: Descrivono lo stato del sistema fisico.
- Rappresentazione del Tempo: Attraverso la simulazione temporale (simulation time), un'astrazione del tempo fisico.
- Evento: Un'occorrenza istantanea che può cambiare lo stato del sistema. Lo stato cambia solo in un numero discreto di punti nel tempo, quando si verificano eventi.

Nozioni di Tempo:

- **Tempo Fisico:** Tempo nel sistema reale.
- **Tempo di Simulazione:** Astrazione usata dal simulatore.
- **Tempo Reale (Wallclock time):** Tempo durante l'esecuzione del programma di simulazione.

Meccanismi di Flusso del Tempo: In una simulazione discreta, il modello vede il sistema cambiare stato solo in punti discreti nel tempo di simulazione, "saltando" da uno stato all'altro.

Esecuzione Basata su Eventi (Event-driven execution): Le variabili di stato vengono aggiornate solo quando si verifica "qualcosa di interessante" (un evento). Ogni evento ha un timestamp e causa cambiamenti nelle variabili di stato.

Esempio DES "Manuale" (Coda a singolo server): Obiettivo (stimare il ritardo medio in coda), Variabili di stato (stato del server, lunghezza della coda, tempo di arrivo dei clienti), Eventi (arrivo del cliente, completamento del servizio). Dimostra l'avanzamento del tempo basato sull'evento successivo e il calcolo delle misure di performance.

Programmi DES: Tipicamente utilizzano variabili di stato, una lista di eventi e una variabile di orologio globale.

Pianificazione Eventi (Event Scheduling): Meccanismo per creare nuovi eventi nella simulazione, implementato allocando memoria per un nuovo evento, compilando i campi (timestamp, tipo, parametri) e aggiungendolo alla lista eventi. Modella le relazioni causali nel sistema fisico.

Ciclo Principale dell'Esecutivo di Simulazione: L'handler degli eventi modifica le variabili di stato e pianifica nuovi eventi nel futuro simulato. L'esecutivo processa sempre l'evento con il timestamp più piccolo successivo, garantendo la riproduzione fedele delle relazioni causali.

Implementazione dei Programmi DES: Diverse tipologie con flessibilità e costi variabili: linguaggi di programmazione generici (es. Java, C++), pacchetti di simulazione (es. Simjava, CSIM), strumenti di simulazione di alto livello (es. Arena, SIMUL8).