Analisi e Progettazione del Software

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

Capitolo 1

Introduzione

Questi appunti sono presi durante le lezioni in aula. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Grazie mille e buono studio!

Capitolo 2

Introduzione all'Ingegneria del software

Durante il corso di Analisi e Progettazione del software si analizzano i modelli e i principi per lo sviluppo di un software mantenibile, andando anche ad analizzare, in maniera sommaria, anche tutti gli strumenti di ingegneria del software, necessari per lo sviluppo ottimale di un software.

In questo corso studieremo in dettaglio i seguenti argomenti:

- introduzione all'ingegneria del software
- progettazione sistemi orientati ad oggetti
- modellazione a dominio
- UML e analisi dei casi d'uso
- design pattern
- sviluppo test-driven(cenni)
- code smell e refactoring(cenni)

Il software, è l'insieme delle componenti modificabili e non fisiche di un calcolatore, viene diviso in due categorie:

generici , per un ampio range di clienti, come ad esempio gli elaboratori di testi

custom , per un singolo cliente, come ad esempio i gestionali specifici di un impresa

Questa differenza si sta sempre più assottiliando in quanto recentemente varie aziende stanno sviluppando un software generico, che viene poi adattato in base alle esigenze del singolo cliente, come ad esempio i software Oracle e Sas.

Nello sviluppo di software si utlizza spesso una base pre-esistente, infatti l'ingegneria del software si occupa di tutti gli aspetti per lo sviluppo del software, sfruttando di solito una base preesistente.
Di solito quando si sviluppa un software si presuppone che abbia una durata di alcuni anni, con la predispozione al cambiamento e all'introduzione di nuove features infatti un software per essere utile deve essere continuamente cambiato.

Si hanno due tipologie di progetti:

1. **progetti di routine**, con soluzione di problemi e riuso di vecchio codice

2. **progetti innovativi**, con soluzioni nuovi

e solitamente l'ingegneria del software si occupa principalmente di progetti innovativi con specifiche del progetto variabili, con la presenza di cambiamenti continui e ovviamente non è uguale nè similare con l'ingegneria tradizionale.

Un programmatore normalmente lavora da solo, su un programma completo con specifiche note mentre un ingegnere del software lavora in gruppo, progetta componenti e l'architettura e identifica requisiti e specifiche. Il costo del software spesso supera quello hardware

Nello sviluppo di un software si hanno le seguenti fasi di sviluppo:

- **analisi dei requisiti**, che indica cosa deve fare il sistema
- **progettazione**, progetto del sistema d implementare
- **sviluppo**, produzione del sistema software
- **convalida**, verifica dei requisiti del cliente
- **evoluzione**, evoluzione al cambiare di requisiti del cliente

Esistono dei sistemi software per l'automazione delle attività svolte nel progetto software, i cosiddetti **CASE** (Computer-Aided Software Engineering) che si dividono in:

CASE di alto livello per il supporto alle prime attività di processo come la raccolta requisiti e la progettazione

CASE di basso livello per il supporto alle ultime attività di processo come la programmazione, il debugging, testing e reverse engineering

Nella figure X1 e X2 si hanno delle risposte alle comuni domande sull'ingegneria del software e le caratteristiche di un ottimo software, necessario per mantenerlo facilmente modificabile nel tempo.

I

specifiche del software

sviluppo del software

convalida del software

evoluzione del software

Il software, oltre a fornire quanto richiesto dal cliente, deve essere:

- **mantenibile**, quindi semplice da mantenere ed aggiornare
- **affidabile**, quindi con bassa possibilità di malfunzionamenti
- **efficiente**, quindi deve fare buon uso delle risorse
- **facilmente usabile**

Il software deve essere accettato dagli utenti per i quali è stato sviluppato per cui deve essere comprensibile, usabile e compatibile con altri sistemi. Esiste un codice etico, **ACM/IEEE**, con otto principi legati al comportamento degli ingegneri del software ed è fondamentale seguirlo dato che la Un **sistema** è una collezione significativa di componenti interrelati che lavorano assieme per realizzare un obiettivo comune quindi include software e parti meccaniche o elettriche. Si ha inoltre che i vari componenti possono dipendere da altri componenti e che le proprietà e il comportamento dei vari componenti sono intrinsecamente correlati, quindi si hanno due macro categorie:

1. **sistemi tecnico-informatici** con le seguenti caratteristiche:
    - includono hardware e software
    - non includono gli operatori e i processi operazionali
    - il sistema non conosce lo scopo del suo utilizzo
    - si hanno **proprietà emergenti:**
        - le proprietà del sistema finale dipendono dalle sue componenti e dalle relazioni tra le componenti

  - queste proprietà possono essere misurate solamente sul sistema finale
- si ha il **non-determinismo**, ovvero il sistema non risponde sempre con lo stesso output dato lo stesso input perché il risultato è spesso dipendente dal comportamento degli operatori umani
- si ha un complesso legame tra i sistemi e gli obbiettivi aziendali, ovvero l'efficacia nel supportare gli obiettivi aziendali non dipende solo dal sistema stesso

2. **sistemi socio-tecnici** con le seguenti caratteristiche:
   - includono uno o più sistemi tecnici, ma anche i processi operazionali e gli operatori
   - sono fortemente condizionati da politiche aziendali e regole

Vediamo qualche proprietà emergente:

- **volume:** il volume di un sistema (lo spazio totale occupato) varia a seconda di come sono disposti e collegati i componenti che lo formano
- **affidabilità:** l'affidabilità del sistema dipende dall'affidabilità dei componenti, ma interazioni impreviste possono produrre nuovi fallimenti e quindi influenzare l'affidabilità dell'intero sistema
- **protezione:** la protezione del sistema (la sua abilità di resistere agli attacchi) è una proprietà complessa che non può essere facilmente misurata. Possono essere inventati nuovi attacchi che non erano stati previsti dai progettisti del sistema, in modo da abbattere le difese integrate
- **riparabilità:** questa proprietà riflette la facilità con cui è possibile correggere un problema del sistema quando questo viene scoperto; dipende dalla possibilità di diagnosticare il problema, di accedere ai componenti difettosi e di modificarli o sostituirli
- **usabilità:** questa proprietà mostra la facilità duso del sistema e dipende dai componenti del sistema tecnico, dai suoi operatori e dal suo ambiente operativo

Si hanno poi le proprietà del tipo *"non deve accadere"*:

- prestazioni o affidabilità possono essere misurate
- altre proprietà sono solo comportamenti che **non devono accadere**; si hanno due tematiche:
    1. **sicurezza:** il sistema non deve causare danni a persone o all'ambiente
    2. **protezione:** il sistema non deve permettere utilizzi non autorizzati

I sistemi Socio-tecnici sono sistemi pensati per raggiungere obiettivi aziendali o organizzativi e bisogna comprendere a fondo l'ambiente organizzatico nel quale un certo sistema è usato

2.0.1 Sistemi Critici

Abbiamo tre esempi di sistemi critici:

1. **sistemi safety-critical** dove i fallimenti comportano rischi ambientali o perdite di vite umane (per esempio un sistema di controllo per un impianto chimico)

2. **sistemi mission-critical** dove i fallimenti possono causare il fallimento di attività a obiettivi diretti (per esempio un sistema di navigazione di un veicolo spaziale)

3. **sistemi business-critical** dove i fallimenti possono risultare in perdite di denaro sostenute (per esempio un sistema bancario)

Ci possono essere vari fallimenti:

- **fallimenti hardware**, errori di progetto, produzione o "consumo"
- **fallimenti software**, errori di specifica, progetto, implementazione
- **errori operativi**, errori commessi da operatori umani (forse una delle maggiori cause di fallimenti)

Nei sistemi critici, generalmente la fidatezza è la più importante proprietà del sistema in quanto si ha un alto costo dei fallimenti. La fidatezza riflette il livello di confidenza che l'utente ha verso il sistema. **Utilità e fidatezza non sono la stessa cosa**. Possiamo elencare delle proprietà della fidatezza.

- **disponibilità**, ovvero la capacità del sistema di fornire servizi quando richiesto
- **affidabilità**, ovvero la capacità del sistema di fornire servizi come specificato
- **sicurezza**, ovvero la capacità del sistema di lavorare senza rischiare fallimenti catastrofici
- **protezione**, ovvero la capacità del sistema di proteggersi contro intrusioni accidentali o volontarie
- **riparabilità**, ovvero la facilità con cui un sistema può essere riparato in caso di fallimenti
- **mantenibilità**, ovvero la facilità con cui un sistema può essere adattato a nuovi requisiti
- **sopravvivenza**, ovvero la capacità del sistema di fornire servizi anche se sotto attacco

- **tolleranza dell'errore**, ovvero fino a che punto il sistema riesce a tollerare o evitare errori di immissione

Il costo dei fallimenti nei sistemi critici è così alto che i metodi di ingegneria del software non sono cost-effective per i sistemi non critici. D'altro canto i costi della fidatezza incrementano esponenzialmente all'aumentare del livello di fidatezza richiesto, per due ragioni principali:

1. l'uso di tecniche di sviluppo più costose ed hardware di alta affidabilità
2. l'aumentare del tempo dedicato al testing ed analisi per convalidare la raggiunta fidatezza.

A causa del costo di un sistema dotato di elevata fidatezza, potrebbe essere conveniente sviluppare un sistema non fidato e pagare per i costi dei fallimenti; questa scelta dipende da fattori sociali e politici. La scelta finale dipende sempre dal sistema, per un normale sistema di business una fidatezza moderata può essere sufficiente

2.0.2 Modelli di Processo

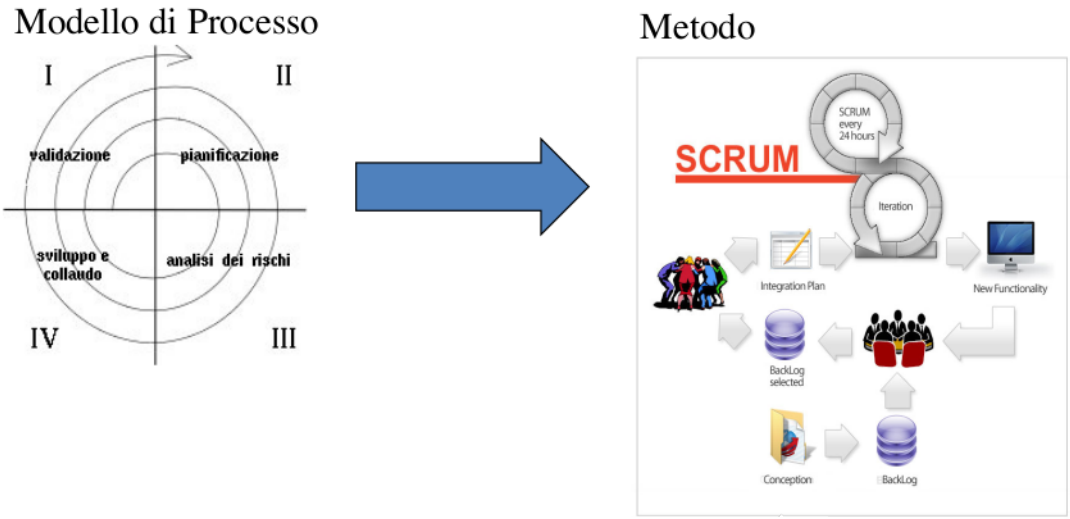
Il modello di un processo software è una rappresentazione semplificata del processo. Questa rappresentazione è basata su un aspetto specifico:

- **modello a flusso di lavoro (Workflow)** per una sequenza di attività
- **modello a flusso di dati (Data-flow)** per il flusso delle informazioni
- **modello ruolo/azione (role/action)** per decidere i vari ruoli

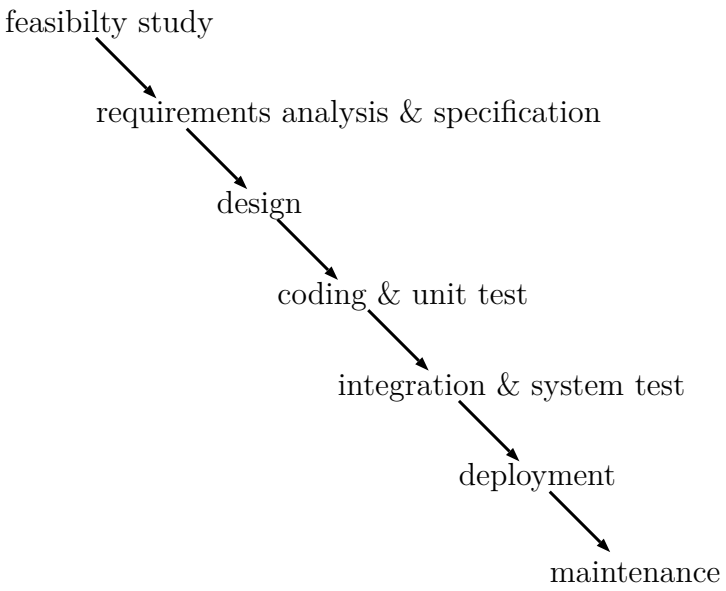
si hanno poi tre modelli generici:

1. **modello a cascata (waterfall)**
2. **modello ciclico (Iterative development)**
3. **ingegneria del Software basata sui componenti (CBSE)**

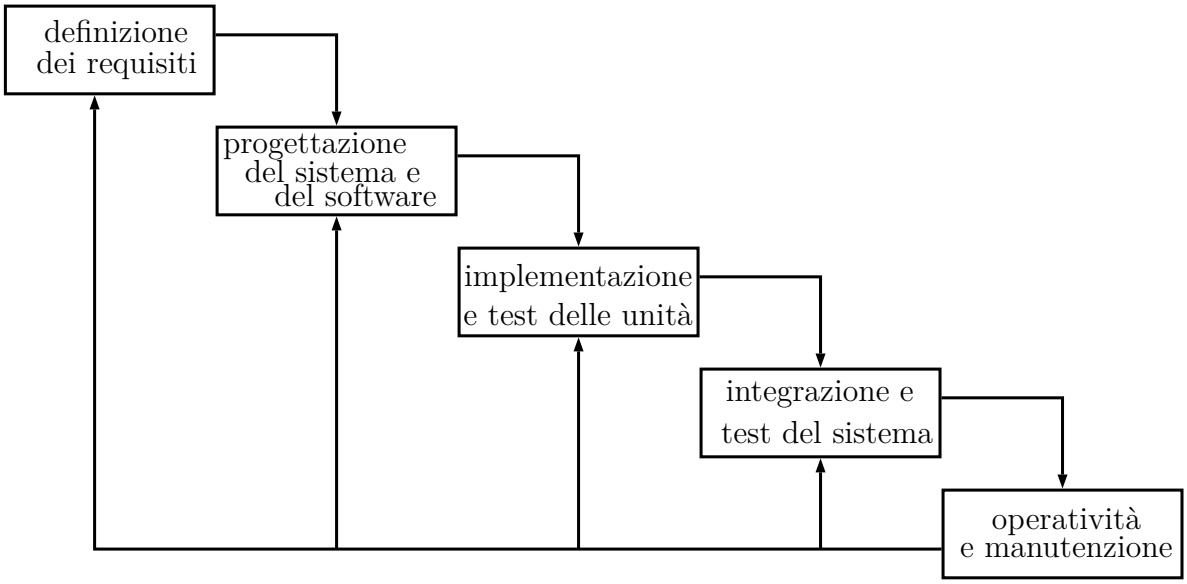
Con i metodi di ingegneria del software su ha un approccio strutturato allo sviluppo software (quali modelli utilizzare e quando, convenzioni sulla notazione, vincoli sui modelli. per esempio id univoco alle entità raccomandazioni, o conformità ad un certo stile progettuale guida al processo). Si passa quindi da un modello di processo ad un metodo:



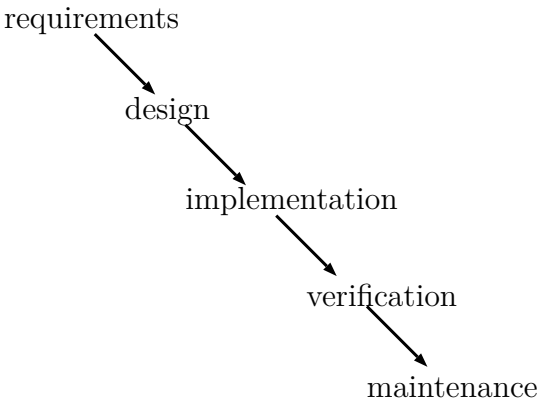
Si ha il modello a cascata, inventato da Royce:



si ha il seguente feedback del modello a cascata:

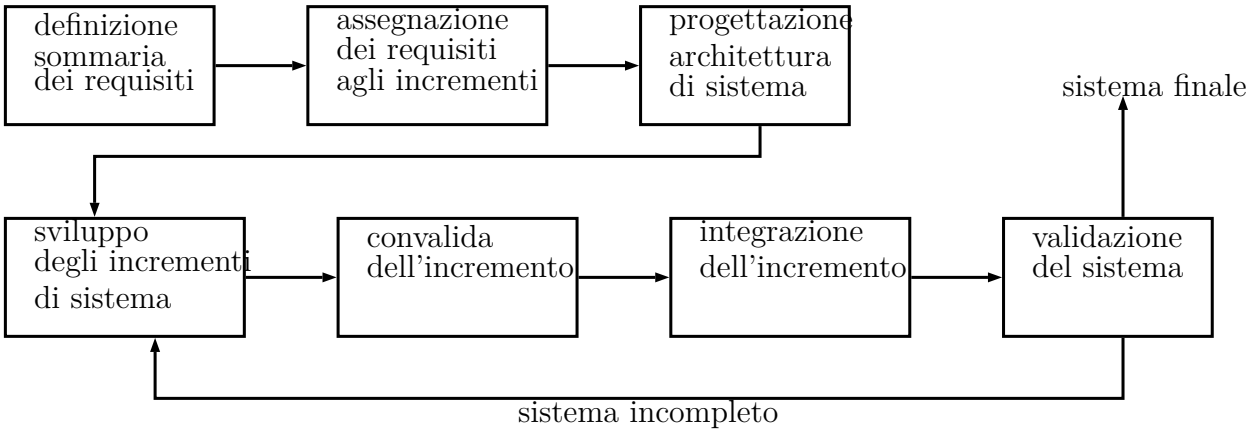


I requisiti di sistema evolvono **sempre** durante il corso del progetto.
Quindi il modello a cascata è così ben rappresentabile:



Invece di rilasciare il sistema in una singola consegna, sviluppo e consegna sono strutturati in una sequenza di incrementi, ognuno dei quali corrispondenti a parte delle funzionalità richieste. Questa è la **consegna incrementale**. requisiti utente sono ordinati per priorità, i requisiti ad alta priorità sono inclusi nei primi incrementi.

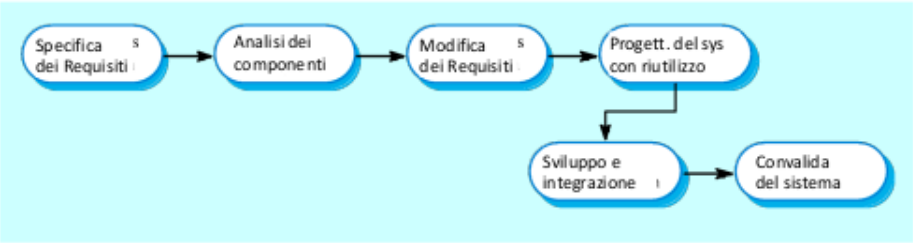
Una volta che lo sviluppo di un incremento inizia, i requisiti sono congelati; invece possono evolvere i requisiti per gli incrementi successivi:



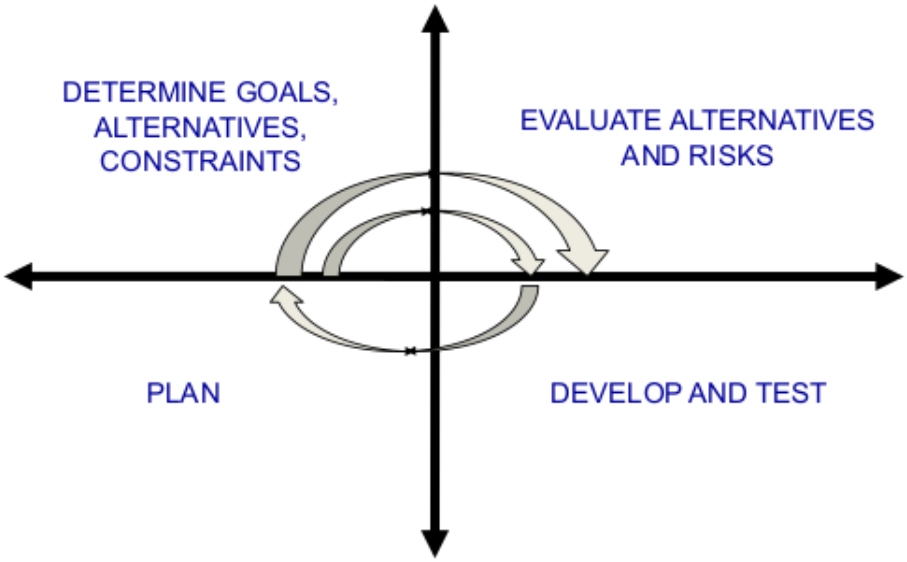
Con la consegna incrementale si hanno i seguenti vantaggi:

- funzioni utili per il cliente possono essere rilasciate ad ogni incremento, quindi alcune funzionalità di sistema sono disponibili sin dai primi incrementi
- i primi incrementi rappresentano dei prototipi che supportano la scoperta dei requisiti per i successivi incrementi
- i rischi di fallimento si abbassano
- i servizi a priorità più alta tendono ad essere collaudati più a fondo

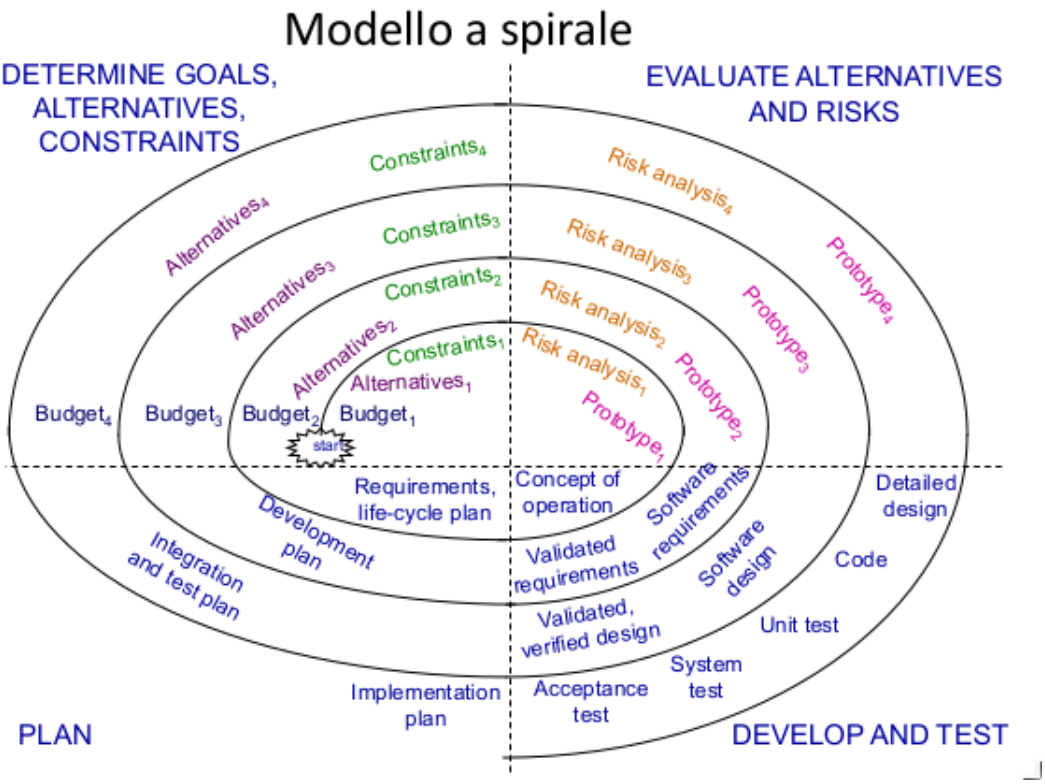
Se si ha che i requisiti vengono scoperti attraverso lo sviluppo si ha lo **sviluppo evoluzionistico**. Si usano prototipi che poi non verranno utilizzati nel corso del progetto vero e proprio



Si ha ingegneria del software basata su componenti se si hanno già librerie su cui lavorare.
Passiamo al primo modello iterativo, quello di B. Boehm.



si parte dal quadrante in alto a sinistra e si arriva al prodotto finale mediante una serie di iterazioni. Si sceglie ovviamente la soluzione più sicura. I vari passaggi sono qui rappresentati:

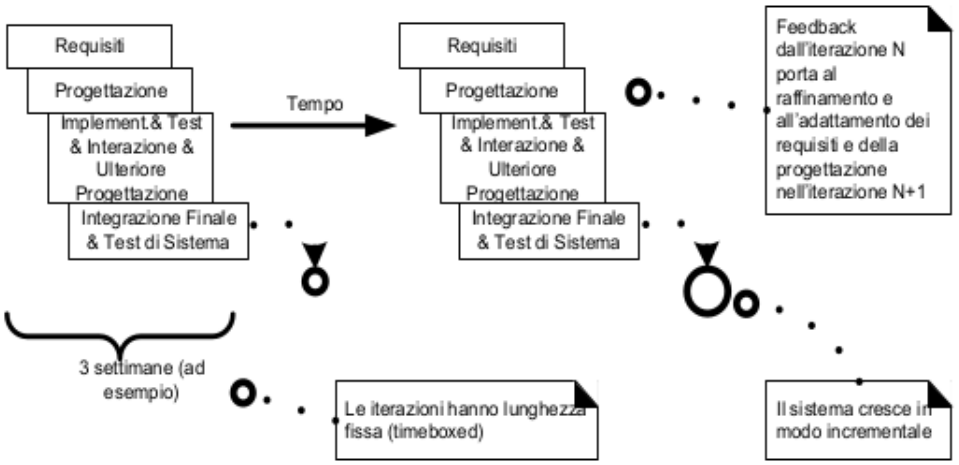


2.1 Modelli agili

Si hanno i modelli agili, che si basano sulla **consegna incrementale**. Si hanno iterazioni brevi e timeboxed, con un raffinamento evolutivo di piani, dei requisiti e del progetto. Questi metodi favoriscono la collaborazione nei gruppi e la riduzione dei costi.

2.1.1 Modello UP o RUP

Il metodo UP, *inified process*, conosciuto anche come RUP, *Rational Unified Process*, usa la notazione UML, *unified modeling language* ed è uno dei più importanti modelli agili. È è un processo iterativo e incrementale. Si basa sulla suddivisone di un grande processo in iterazioni controllate. Si hanno passi piccoli, timeboxed da 2 a 6 settimane, feedback rapito e adattamento, di requisiti, modelli, stime di sviluppo e costi e priorità.



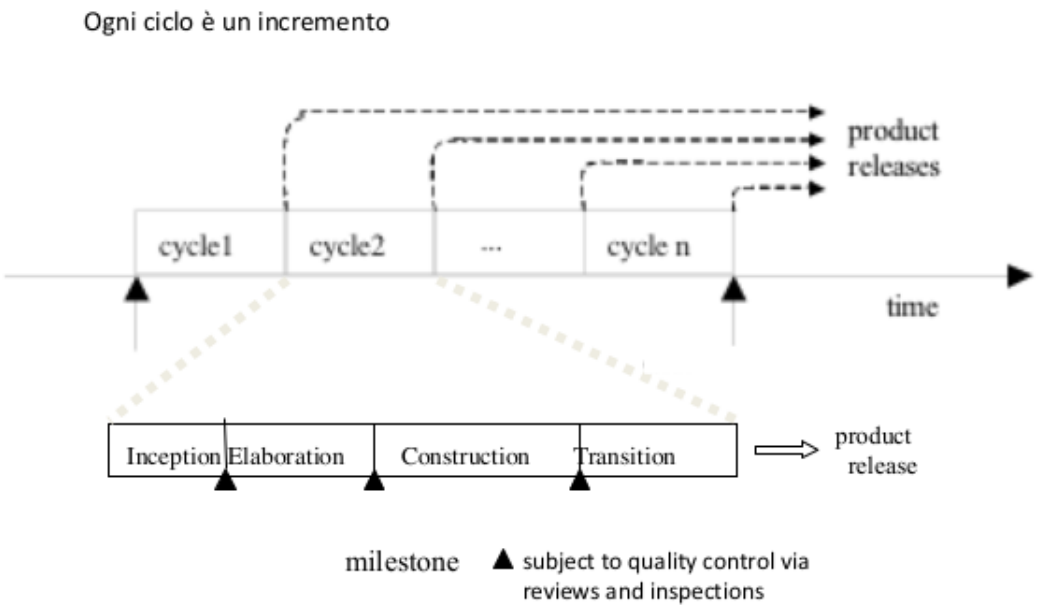
Si hanno 4 fasi nel modello RUP:

1. **avviamento:** dove viene stabilita la bussines rationale del progetto e stabiliti gli obiettivi, stime dei costi e dei tempi. Si ha uno studio di fattibilità, *Life-Cycle Objective Milestone*

2. **elaborazione**: dove si raccolgono i requisiti in modo più dettagliato, si procede con l'analisi ad alto livello per stabilire l'architettura di base e vengono analizzati i rischi principali. Si hanno stime più affidabili: *Life-Cycle Architecture Milestone*

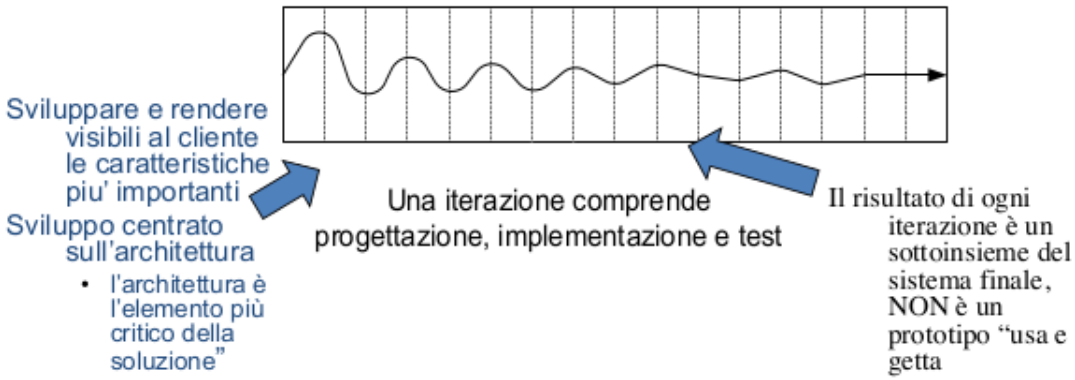
3. **costruzione** che consiste di molte iterazioni, e ad ogni iterazione viene costruita una parte del sistema che soddisfa un sottoinsieme di requisiti. Viene effettuato il testing e l'integrazione. Si ha la preparazione al rilascio: *Initial Operational Capability Milestone*

4. **transizione:** dove vengono affrontati tutti gli aspetti legati al fine-tuning delle funzionalità, prestazioni, qualità, beta-testing, ottimizzazione, formazione degli utenti,... È la *Product Release Milestone*

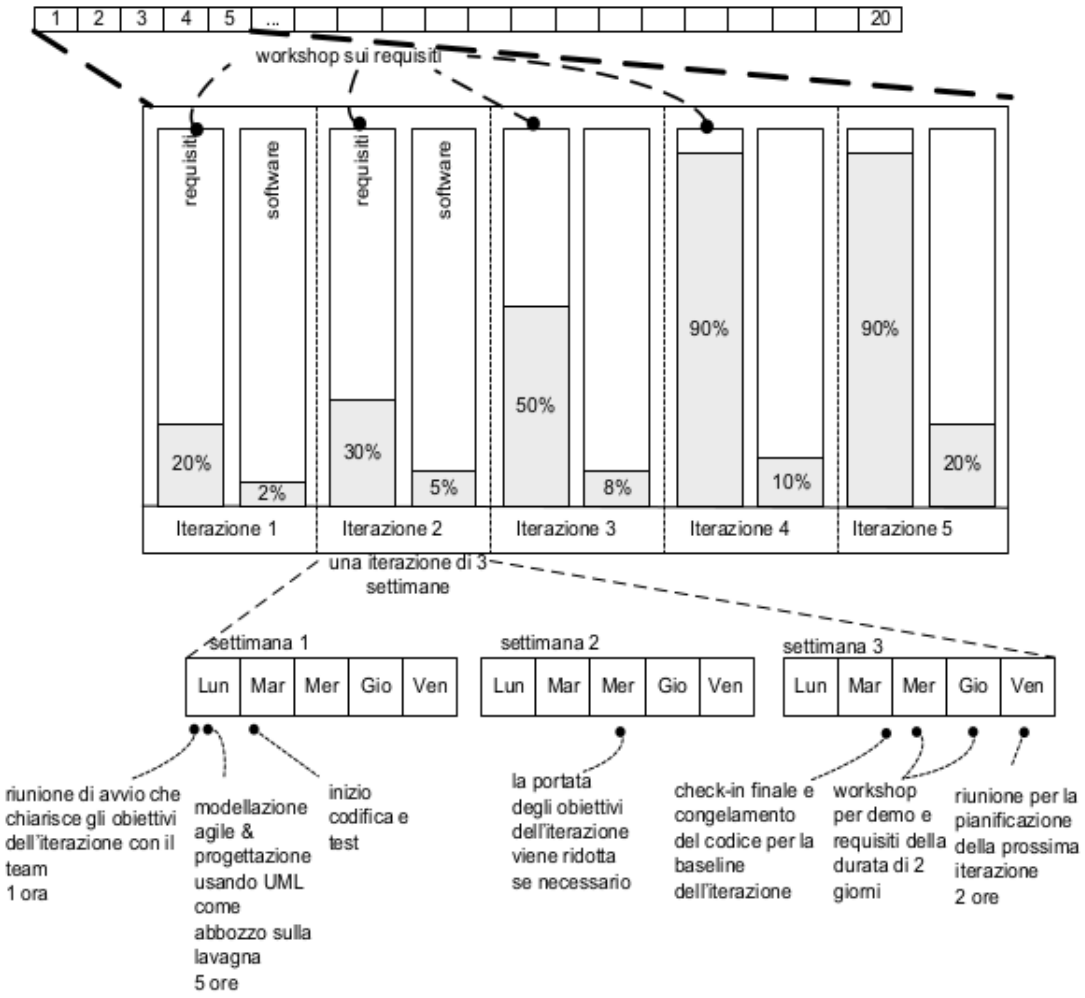


Si hanno i seguenti vantaggi dello sviluppo iterativo:

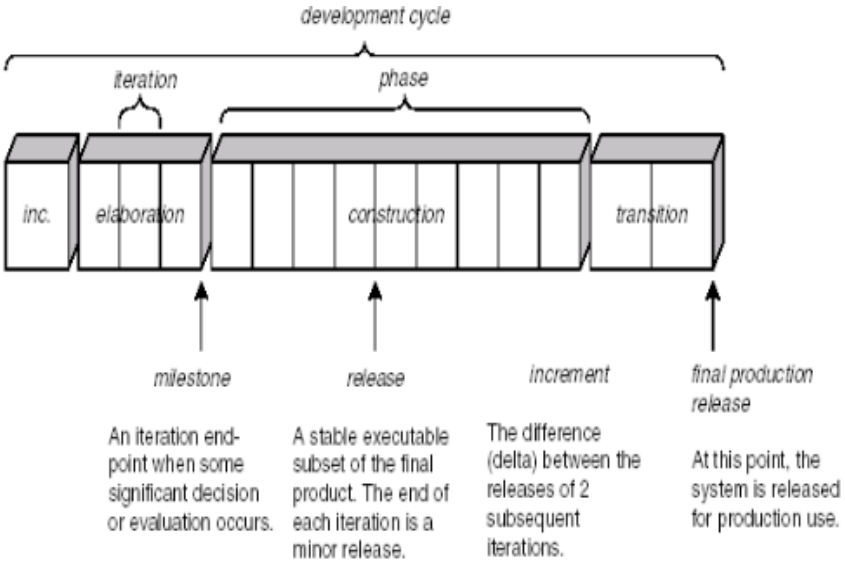
- minore chance di fallimento del progetto
- riduzione precoce dei rischi
- progresso visibile
- feedback precoce e coinvolgimento dell'utente
- "abbracciare il cambiamento"



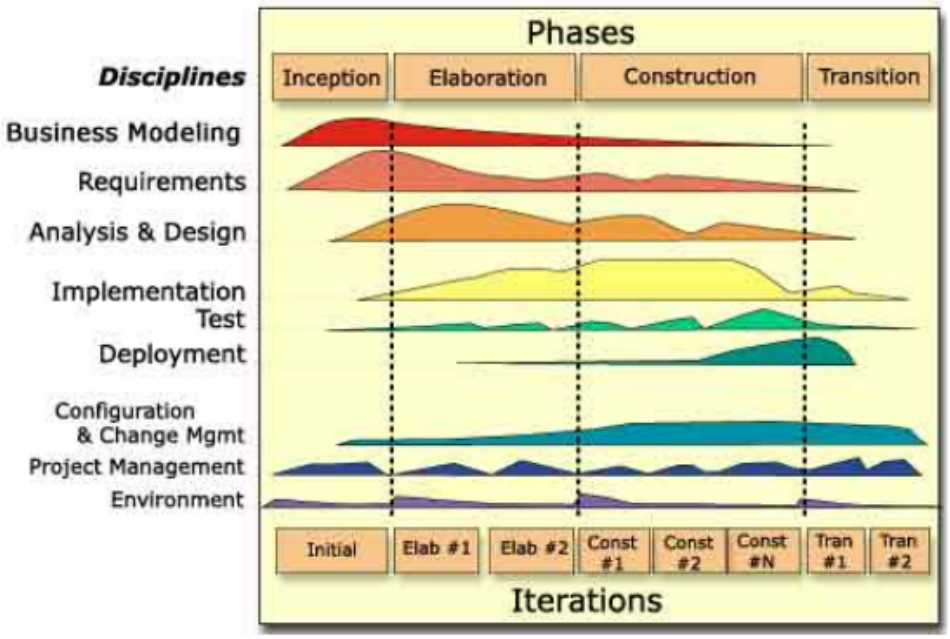
Vediamo un esempio con 5 iterazioni:



vediamo un'altra rappresentazione del ciclo di sviluppo:



vediamo anche un'immagine per rappresentare l'organizzazione del processo, con fasi, iterazioni e "discipline":

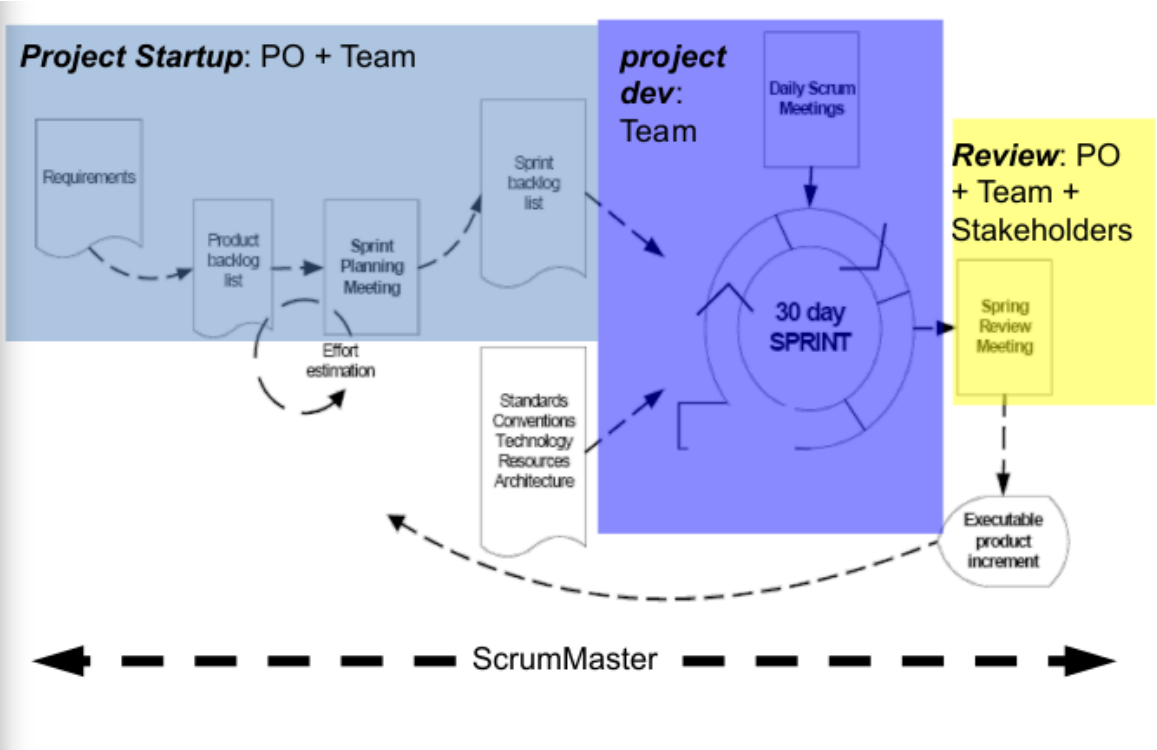


tra le discipline troviamo analisi e progettazione Si hanno le seguenti caratteristiche principlai per l?UP:

- è iterativo e incrementale
- si enfasi sul modello invece che sul linguaggio naturale
- è centrato sull'architettura

2.1.2 Processo Scrum

Iniziamo con un'immagine che rappresenta questo metodo agile:



È un processo iterativo basato sull controllo dello stato di avanzamento. Si hanno i seguenti principi fondamentali:

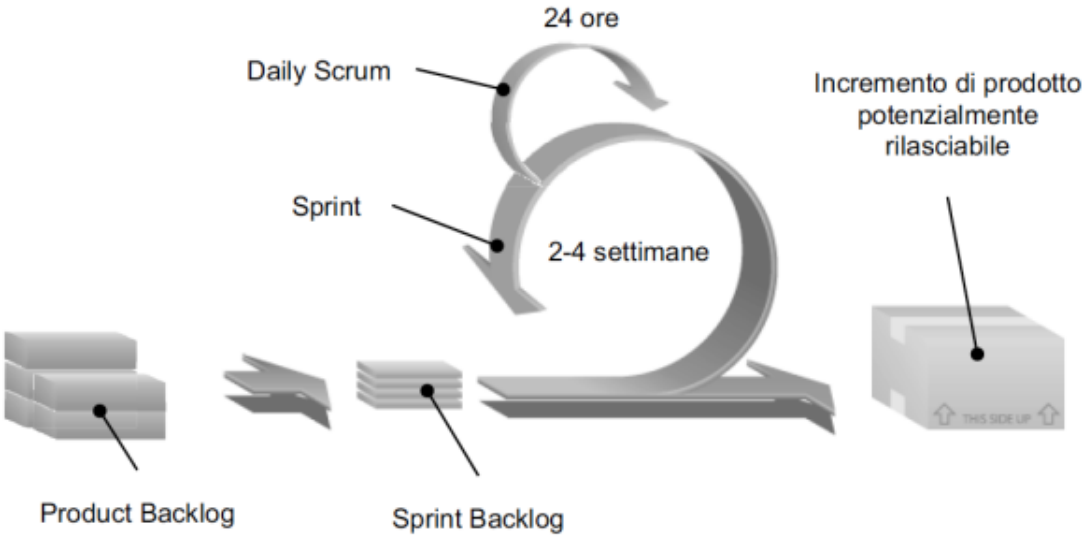
- **visibilità:** gli aspetti significativi del processo di sviluppo devono essere visibili a tutti e tutti gli aspetti devono essere chiari
- **ispezione:** poter ispezionare frequentemente il lavoro fatto per verificare che si sta procedendo verso gli obiettivi posti

- **adattamento**, se si sta deviando dagli obiettivi, occorre un adattamento per minimizzare altre deviazioni. Deve essere svolto nel minor tempo possibile in caso di necessità

Si hanno i seguenti ruoli:

- **product owner** che si occupa della definizione delle caratteristiche del progetto da sviluppare (1 sola persona, e.g. committente, o altri...); gestisce il Product Backlog. Lavora costantemente col team
- **team:** dedicato allo sviluppo e rilascio del prodotto attraverso incrementi successivi (da 3 a 7/8 membri)
- **scrum master:** responsabile che lo SCRUM venga applicato correttamente. Non è un project manager, fa da intermezzo tra team e product owner, collabora col team etc

Ecco un'immagine che rappresenta lo sviluppo con Scrum:



SI hanno quindi:

- releases brevi con sottoinsiemi consegnati velocemente
- in ogni istante si hanno test eseguibili, non si ha logica duplicata e si ha un numero minimo di features
- si ha *refactoring* con miglioramento continuo

- si un testo continuo e automatico, *testing first*
- si hanno due sviluppatori che lavorano insieme, *pair programming*
- il codice è proprietà del team e non del singolo dev, *collective owbership*
- si hanno check-in frequenti e integrazioni continue, *continuous integration*
- il cliente lavora col team

I **requisiti** sono una descrizione dei servizi del sistema e dei suoi vincoli operativi. Si hanno due tipi di requisiti:

- **requisiti utenti:** affermazioni in linguaggio naturale, corredate da tabelle e diagrammi, riguardanti i servizi che il sistema offre ed i vincoli operazionali. Sono scritti per i clienti e sono da loro comprensibili anche se non hanno conoscenze tecniche dettagliate.

- **requisiti di sistema:** un documento strutturato che definisce in modo dettagliato le funzioni del sistema, i servizi ed i vincoli operazionali. Definisce cosa deve essere implementato, quindi può essere parte del contratto tra acquirente e sviluppatore. È la base del progetto della soluzione e può essere illustrato utilizzando i **modelli di sistema**.

i requisiti possono avere 3 problemi:

- **ambiguità:** il sistema fornisce visualizzazioni appropriate per leggere i documenti
- **incompletezza:** i requisiti devono descrivere tutti i servizi forniti dal sistema (anche se in realtà tutto ciò è impossibile, per questo esistono i cambiamenti)
- **inconsistenza**: le descrizioni non devono contenere conflitti o contraddizioni (anche questa cosa è difficilissima da ottenere)

SI hanno anche i **requisiti funzionali** servizi che il sistema deve (o non deve) fornire. Si hanno anche i *requisiti non funzionali*, come vincoli sui servizi temporali, standard, sull'usabilità etc... I requisiti non funzionali possono essere più critici dei requisiti funzionali: **se non sono soddisfatti il sistema è spesso inutilizzabile**. Si hanno alcuni requisiti non funzionali:

- **prodotto:** specificano che il prodotto deve comportarsi in un modo particolare esempi: velocità di esecuzione, affidabilità.
  Linterfaccia utente sarà implementata con HTML semplice senza frames e applets

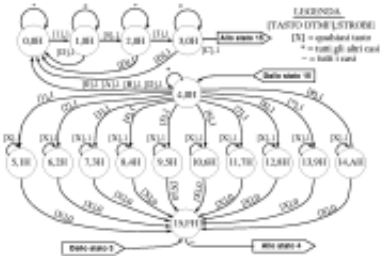
- **organizzativi:** conseguenza di politiche e procedure dellorganizzazione del cliente e dello sviluppatore. esempi: linguaggio di programmazione, metodo di sviluppo.
  Il processo di sviluppo e la relativa documentazione sarà conforme alle norme XYZCo-SP-STAN-95

- **esterni:** provengono da fattori esterni al sistema e al processo di sviluppo. esempi: interoperabilità, leggi e norme.
  Il sistema non deve rilevare agli operatori nessuna informazione personale sui clienti oltre al nome e al numero di riferimento

Tutto ciò perché il linguaggio naturale presenta mancanza di chiarezza, ambiguità, confusione etc... Esistono alternative:

Linguaggio naturale strutturato

Modello visuale informale



Modello visuale formale

Specifica testuale formale

I requisiti hanno un formato standard, usano il linguaggio in modo consistente, evitano il gergo tecnico e evidenziano delle porzioni di testo per identificare le parti più importanti dei requisiti. Un esempio di standard è quello IEEE830.
I casi d'uso possono essere visualizzati con l'UML:

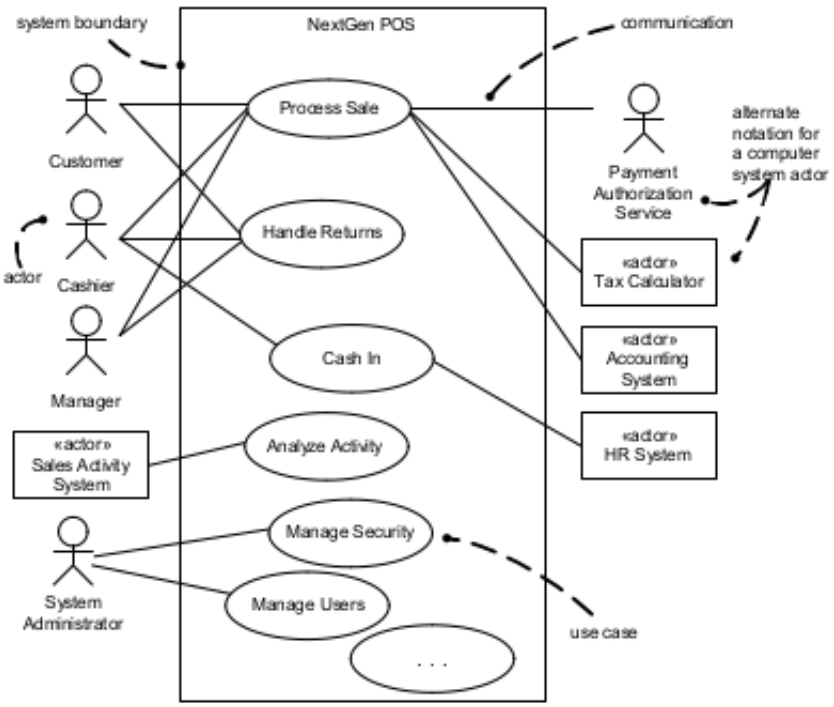
Si hanno i cosiddetti **casi d'uso**, che ci indicano una possibile situazione, infatti *documentano requisiti funzionali complementari alla lista dei requisiti non-funzionali punto di partenza per analisi e progettazione.* SI hanno i seguenti elementi fondamentali dei casi d'uso:

- **attori**, divisi in:
  - **attore primario:** raggiunge degli obiettivi utente utilizzando i servizi del sistema
  - **attore di supporto:** offre un servizio, spesso è un sistema informatico, ma può essere anche una organizzazione o una persona
  - **attore fuori scena:** ha un interesse nel caso d'uso, ma non è né un attore primario né di supporto
- **scenari**

qualora manchino scenari alternativi si ha il cosiddetto **formato informale**. Nello studio dei casi d'uso bisogna concentrasi sullo scopo dell'autore e su cosa deve fare effettivamente il sistema (*responsabilità*). Bisogna ignorare invece l'interfaccia utente e come effettivamente lo fa il sistema.
Si hanno i cosiddetti 3 test:

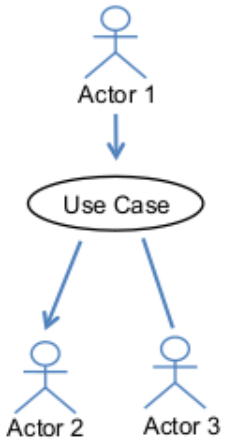
- **test del capo:** *Il capo chiede: "Cosa avete fatto tutto il giorno?" Voi rispondere con il nome del caso d'uso. Sarà soddisfatto?*
- **test EBP (*Elementary Business Process*:** *attività che aggiunge un valore di business misurabile e lascia i dati in uno stato coerente*
- **test della dimensione:** $>$ 1 azione (da 3 a 10 pagine nel formato dettagliato)

Attività secondari e piccoli passi possono diventare casi d'uso per evitare le ripetizioni del testo, in questo caso i tre test possono essere violati. I diagrammi, con il nome del caso d'uso, le relazioni e gli attori, sono secondari, **i casi d'uso sono documenti di testo**. Vediamo un esempio:



Le **associazioni** sono un canale di comunicazione tra attore e caso d'uso. Sono rappresentate:

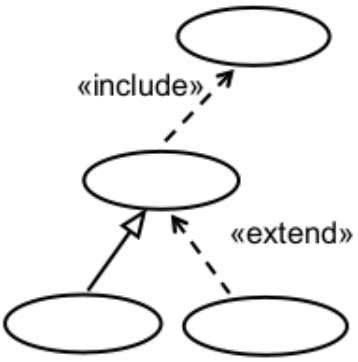
- **una linea direzionata** per specificare chi da inizio all'iterazione
- **una linea semplice** per indicare che entrambe le parti possono dare inizio all'iterazione



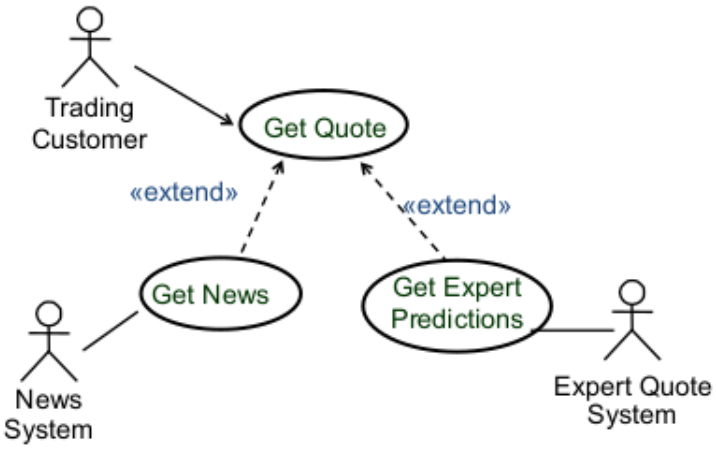
Include (Inclusione)

Extend (Estensione)

Generalization (Generalizzazione)



L'**include** è una relazione tra un caso d'uso base ed un caso d'uso incluso nel caso d'uso base il cui comportamento definito nel caso d'uso *inclusion* esplicitamente incluso nel caso d'uso base. Per l'*include* si ha che il caso d'uso è eseguito completamente quando viene raggiunto il punto di inclusione.
L'**extend** connette un caso d'uso esteso ad un caso d'uso base. Aggiunge varianti ad un caso d'uso base. Viene inserito solo se la condizione di estensione è vera. Estensione inserite solo in corrispondenza degli *extension point*.



Get Quote Use Case
Basic Flow:
1. Include "Identify Customer" to verify customer's identity.
2. Display options.
3. Customer selects "Get Quote."
4. Customer gets quote.
5. Customer gets other quotes.
6. Customer logs off.

A1. Quote System unavailable
…

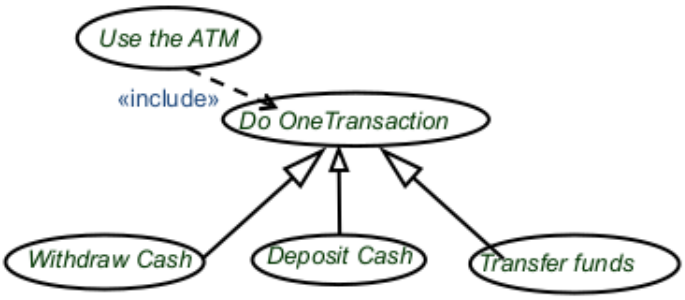
Extension Points:
The "Optional Services" extension point occurs at Step 3 in the Basic Flow and Step A1.7 in Quote System Unavailable alternative flow.

Get News Use Case
This use case extends the Get Quote Use Case, at extension point "Optional Services."

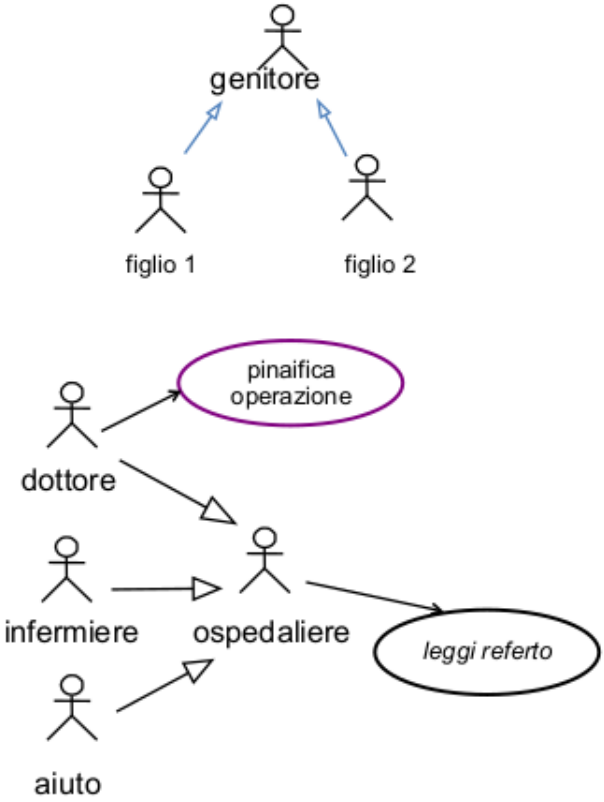
Basic Flow:
1. If Customer selects "Get News," the system asks customer for time period and number of news items.
2. Customer enters time period and number of items. The system sends stock trading symbol to News System, receives reply, and displays the news to the customer.
3. The Get Quote Use Case continues.

A1: News System Unavailable
A2: No News About This Stock
…

Per l'*include* si ha che caso d'uso esteso è eseguito quando il punto di estensione è raggiunto e la condizione di estensione è vera.
Un caso d'uso base è una **generalizzazione** di un caso d'uso child. Viene eseguito se la condizione di generalizzazione è vera. La generalizzazione è puramente concettuale, non ci sono regole da seguire, come ad esempio l'utilizzo di punti di estensione:



La *generalizzazione* collega un attore child ad un attore parent. Gli attori child partecipano a tutti i casi d'uso in cui partecipano il parent:


genitore
figlio 1
figlio 2
pinaifica
operazione
dottore
infermiere
ospedaliere
leggi referto
aiuto