1. Processi di sviluppo

1.1. Software

Il software è un qualcosa di complesso (sia per dimensione, sia per numero di interconnessioni), costruito da più persone, versionato nel tempo, in uso per un lungo periodo e che cambia nella struttura, nel funzionamento e nei requisiti nel corso del tempo.

Distinguiamo due tipi di progetti software:

- di routine, cioè quelli che vanno a risolvere well-known problems e che nella loro implementazione fanno utilizzo di soluzioni parzialmente già esistenti
- innovativi, cioè quelli che si propongono come soluzioni nuove a problemi nuovi.

Generalmente si lavora a progetti innovativi: riuscire a trasformare progetti innovativi in progetti di routine è una tecnica che richiede grossa abilità.

Il Sommerville propone anche un altra classificazione:

- progetti generici, cioè che producono software vendibili al mondo intero. La software house controlla le specifiche.
- progetti personalizzati, cioè costruiti a partire dalle necessità di un particolare cliente. Il cliente controlla le specifiche.

Un software deve avere proprietà (cioè una caratteristica misurabile, ad esempio il tempo di esecuzione), ed in base a queste proprietà è possibile andare a definire le qualità del SW (cioè i vincoli che le proprietà devono rispettare, ad esempio il tempo di esecuzione deve essere minore di X secondi)

Il SW deve inoltre:

- fornire le funzionalità e le prestazioni richieste dal committente
- essere manutenibile, cioè deve essere scritto in modo che possa essere modficato in caso dei cambiamenti di necessità del cliente
- essere affidabile, cioè deve avere una bassa probabilità di malfunzionamento
- essere efficiente, cioè non deve usare in modo sproporzionato le risorse
- essere usabile, cioè non deve avere una difficoltà di utilizzo sproporzionata

1.2. Ingegneria del software

L'ingegneria del software è una disciplina molto nuova. Pur sembrando simile ad ogni altro tipo di ingegneria è semplice osservare delle enormi differenze che la rendono una disciplina separata dalle altre. In particolare:

- le specifiche di un progetto sono completamente variabili, sia nel corso del suo sviluppo che al completamento di esso
- un software è quindi soggetto a cambiamenti ed evoluzioni continue (patch, bugfix, perfezionamenti)
- è una disciplina che esiste da pochissimo tempo, molto più giovane dei software
- essendo molto giovane ha pochissima storia

L'ing. del SW è necessaria al giorno d'oggi: più e più persone usano e necessitano di nuovo software ogni giorno, la necessità di produrre software affidabile e funzionante è fondamentale. Inoltre nel lungo termine è molto più efficace ed economico applicare i fondamenti dell'ing. del SW al posto che scrivere codice direttamente.

1.3. Processo di sviluppo

Il processo di sviluppo è l'insieme di attività che portano alla sviluppo di un software, cioè le fasi del ciclo di vita di un software, che lo portano da essere una semplice idea ad un prodotto completo.

Distinguiamo 5 fasi del processo di sviluppo:

- analisi dei requisiti, si capisce cosa deve fare il software che si vuole realizzare
- progettazione, si progetta come il nostro software deve essere implementato
- sviluppo, si programma il software
- convalida, si verifica che il SW rispetti i requisiti
- evoluzione, si modifica il software in base alle necessità del cliente o del mercato

1.4. Modelli di processo

Un processo di sviluppo è basato su un modello di processo, una rappresentazione semplificata di un processo di sviluppo: questi modelli non sono soluzione definitive, ma semplici astrazioni utili a spiegare le differenti modalità di approcciare il processo di sviluppo.

1.4.1. Modello a cascata

Il modello a cascata prende le varie fasi del processo di sviluppo e le mette una di seguito all'altra. Ogni fase non può cominciare se la precedente non è ancora iniziata.

Il modello a cascata è stato introdotto da Royce nel 1970. E' un processo basato sulla pianificazione: tutto deve essere pianificato prima di iniziare a lavorare. Il passaggio da una fase all'altra produce uno o più documenti che la fase utilizzzerà per completare il proprio lavoro.

Guarda ad esempio: https://en.wikipedia.org/wiki/Cleanroom_software_engineering

Viene difficile cambiare le specifiche in caso di necessità: molte volte è addirittura necessario rifare da capo alcune fasi del processo.

Un processo di sviluppo basato sul modello a cascata è applicabile solo quando i requisiti sono ben noti e difficilmente modificabili.

Il modello a cascate riflette il processo di sviluppo di altre ingegnerie diverse da quelle del software. E' applicabile tranquillamente ad esempio nella produzione industriale meccanina, elettronica o di qualsiasi altro tipo.

1.4.2. Modello a sviluppo incr ementale

Questo approccio alterna le attività di specifica, sviluppo e validazione. Il sistema è sviluppato come una serie di versioni (incrementi), ad ogni incremento si aggiungono funzionalità alla versione precedente.

Lo sviluppo incrementale rappresenta il modo in cui risolviamo problemi: difficilmente affrontiamo un problema nella sua interezza e complessità, piuttosto ci muoviamo verso la soluzione a piccoli step, facendo backtracking solo quando ci accorgiamo di aver fatto errori.

Ogni versione introduce nuove funzionalità, dalle quali si può ricevere feedback dal cliente per progettare e pianificare i cicli di sviluppo successivi. Le funzionalità core vengono sviluppate prima, in questo modo sono più testate e facilmente modificabili in caso i requisiti non fossero precisi.

Tra una versione/ciclo e l'altra è impossibile cambiare i requisiti: saranno modicabili solo all'inizio del ciclo successivo.

Il modello di sviluppo incrementale produce processi di sviluppo iterativi ed evolutivi.

Lo sviluppo incrementale ha come vantaggi:

- minor costo in termine di tempo e di risorse in caso di cambiamento di requisiti
- maggior semplicità nella raccolta di feedback dal cliente, può vedere il software da subito
- maggior velocità di delivery e deployment del SW al cliente, che lo può usare anche quando non è ancora completato
- minor rischio di fallimento
- abbracciare il cambiamento

Svantaggi:

- può essere difficile identificare le "common facilities" necessarie da tutti gli incrementi, visto che i requisiti non sono definiti fino a che un incremento non è completato.
- è difficile sviluppare in modo agile un rimpiazzamento, gli utenti vogliono tutta la funzionalità del sistema vecchio e non vogliono sperimentare con il sistema nuovo.
- molte volte i contratti sono scritti con tutte le specifiche spiegate e complete. Questo non è possibile con uno sviluppo iterativo: può diventare difficile scrivere contratti per clienti di grandi dimensioni come multinazionali o agenzie governative.

Il modello iterativo non si adatta bene ovunque: ad esempio è difficile applicarlo su team che lavorano in locazioni differenti, su progetti che necessitano di un hardware completo prima di scrivere il software e su alcuni sistemi particolarmente critici dove la safety e la secutrity sono fondamentali e dove è quindi necessario controllare a priori l'interazione di tutte le specifiche.

Modello a spirale

Una specializzazione del modello a sviluppo incrementale la si ha con il modello a spirale, formulato da Boehm nel 1988. Il processo di sviluppo è rappresentato con una spirale. Ogni quarto della spirale rappsenta una fase dello sviluppo. Più la spirale si allarga più il progetto prosegue nella sua esecuzione.

In particolare le 4 fasi che si ripetono sono:

- determinazione di obbiettivi, alternative e vincoli
- valutazione di alternative e rischi
- sviluppo e test
- pianificazione (per la fase successiva)

Si differenzia da altri modelli per la esplicita analisi dei rischi. Una volta che i rischi sono stati analizzati e definiti, lo sviluppo prosegue, seguita da una pianificazione. Rischio = qualcosa che può andare storto.

Modello agile

Basato su:

- sviluppo incrementale ed iterativo
- iterazioni brevi (mentre nel modello a spirale non c'è limite) e timeboxed (tempi fissi, si misura il ciclo totale, non la singola attività)
- usa le pratiche e i principi agili:
    - comunicazione, tutti sanno tutto di tutti

    - semplicità, test driven development
    - feedback da parte degli utenti
    - coraggio, di buttare via il lavoro già fatto ed innovare

Manifesto agile

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Agile principles

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity--the art of maximizing the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Modello RUP

Metodologia simil-agile prima del manifesto agile. E' un processo basato sul linguaggi di modellazione UML. E' particolarmente utile per progetti non innovativi, e non è applicabile ad esempio per startup.

Caratteristiche:

- Iterativo ed incrementale
- Basato sull'UML e i suoi modelli e non sul linguaggio naturale
- Incentrato sull'architettura

Fasi RUP:

- Avviamento
    - stabilire "business rationale" del progetto
    - stabilire obiettivi e stime dei costi e dei tempi
    - analisi di fattibilità
    - produce: Live-Cycle Objective Milestone
    - stabilire le entità e le interazioni che hanno all'interno del sistema
- Elaborazione
    - raccoglie requisiti in modo dettagliato
    - analisi ad alto livello per stabilire l'architettura di base

    - analizzare i rischi principali
    - produrre stime affidabili
    - produce: Life-Cycle Architecture Milestone
    - capire il dominio del problema, definire un architettura ed piano
- Costruzone
    - costruisce parte del sistema
    - test ed integrazione
    - Sulle slide di esercitazione questa è una fase a parte
    - preparazione al rilascio
    - produce: initial operational capacity milestone
    - il software è documentato e funziona
- Transizione
    - transizione dagli sviluppatori agli utenti
    - fine-tuning delle funzionalità, prestazione, qualità, ottimizzazione
    - produce: produt release milestone
    - il software documentato e funzionante e completamente inserito nel suo ambiente di esecuzione

Modello Scrum

Basato sui principi agili di

- Comunicazione
    - Meeting giornalieri
- Semplicità
    - Il codice fa solo quello che deve fare
- Feedback
    - Il cliente da continuamente il suo feedback
- Coraggio
    - Coraggio di prendere parti di codice e riscriverle

Caratteristiche:

- Release brevi
    - Il tempo tra una release e l'altra deve essere minimo
- Progetto semplice
    - In ogni istante tutti i test devono essere eseguibili
    - Non esiste logica duplicata
    - Ci sono tutte e sole le funzionalità minime richieste
- Refactoring
    - Miglioramento continuo del codice
    - Si riscrivono parti del codice tra una release e l'altra a favore dell'atomicità e semplicità dei metodi
- Test Driven Development
    - Prima si scrivono i test e poi si implementa il codice
- Pair Programming
    - Si programma a coppie
    - In questo modo si evitano gli errori più comuni
    - Ha la stessa efficienza del singolo programmatore
- Collective Ownership
    - Il codice è di tutti
    - Si evitano situazioni dove non si trova il responsabile di una determinata parte del programma
    - Si evitano situazioni dove si incolpa un singolo programmatore per un bug critico
- Continuous Integration
    - Daily Builds del codice
    - Check-in/Integrazioni frequenti tra tutti i componenti

- Presenza del cliente on-site
    - Dove non è possibile il cliente, presenza del Product Owner
        - Si interpone tra il team di sviluppo e il cliente
        - Fa da intermediario per lo sviluppo di feature e per la raccolta di feedback

Ruoli:

- Team
    - Il team di sviluppo
- Scrum Master
    - Colui che è responsabile di mandare avanti il workflow Scrum
- Product Owner
    - Gestisce l'interazione tra team e cliente
- Stakeholder
    - Ogni interessato (per un qualsiasi motivo) al progetto

Fasi:

Tutte le fasi sono supervisionate da uno Scrum master, una persona con esperienza nello sviluppo Scrum che guida tutte le parti coinvolte nel progetto verso il rispetto del ciclo Scrum.

- Startup
    - Raccolta requisiti
    - Analisi eventuale backlog
    - Pianificazione Sprint (e backlog sprint)
    - PO+Team
- Sprint
    - Daily Meetings
    - Durata di 30 giorni
    - Partecipa il team
- Review
    - Revisione dello sprint
    - Parrecipano PO+Team+Stakeholders

2. Tipi di Sistema e Proprietà

Sistema = collezione significativa di componenti che lavorano per un obiettivo comune.

Distringuiamo i sistemi in due categorie:

- tecnico-informatici
  - hardware e software. Non includono ne gli operatori ne i processi operazionali
  - il sistema non conosce lo scopo del suo utilizzo
- socio-tecnici
  - includo sia sistemi tecnici che processi operazionali ed operatori
  - sono fortemente condizionati da politiche e regole aziendali

2.1. Proprietà

Categorie:

- non emergenti
  - misurabili sui singoli componenti
  - LOC
- emergenti
  - proprietà del sistema finale, dipende da tutti i componenti
  - le posso misurare solo alla fine
  - volume, affidabilit, protezione, riparabilità, usabilità
- Non deve accadere
  - Misure di prestazione e di affidabilià
  - Downtime NON superiore a XXX secondi
  - Page load NON superiore a XXX secondi
  - Oppure comportamenti che non decono accadere
  - Sono difficili da gestire e da garantire
- Altre proprietà
  - Riparabilità (facilita di essere riparato)
  - Manutenibilità (facilità con cui un sistema può essere modificato per nuovi requisiti)
  - Sopravvivenza (capacità di fornire servizio anche sotto attacco)
  - Tolleranza all'errore (quanti errori/problemi può evitare)

2.2. Sistemi critici

- Safety Critical
  - il fallimento implica la perdita di vite umane, danni o infortuni a cose e persone
- Mission Critical
  - il fallimento implica il non completamento un obiettivo
- Business Critical
  - il fallimento implica la perdia di denaro

2.2.1. Sistemi Critici Socio-Tecnici

- Fallimenti Hardware
  - E' l'HW che fallisce
- Fallimenti Software

    - Il SW contiene un bug che fa fallire
- Fallimenti operativi
    - L'utente sbaglia ad usare SW/HW

2.2.2. Fidatezza (dependability)

Dato un sistema critico misuro la sua fidatezza, riflette il livello di confidenza dell'utente nei confronti del sistema, un sistema può essere utile anche se gli utenti non hanno confidenza nel suo buon funzionamento.

Si misura combinando disponibilità, affidabilità, sicurezza e protezione.

3. Analisi dei Requisiti

3.1. Requisiti

I requisiti sono una descrizione di ciò che il sistema deve fare: i servizi che deve fornire e vincoli operativi a cui deve sottostare.

3.1.1. Ingegnerizzazione dei r equisiti

E' il processo di ricerca, analisi, documentazione e verifica dei servizi richiesti dal cliente e i vincoli entro i quali i servizi devono operare.

3.2. Requisiti utente e di sistema

- **Requisiti utenti**, espressi in linguaggio naturale (e completati con diagrammi), ci dicono quello che il sistema deve offire e i vincoli operazionali. Sono scritti per i clienti e devono quindi essere comprensibili dagli utenti non tecnici.
- **Requisiti di sistema**, definiscono precisamente e dettagliatamente quello che deve essere implementato. E' il punto di contatto tra il compratore e gli sviluppatori software. Stanno alla base per il progetto della soluzione.

E' importante scrivere i requisiti a diversi livelli: tutte le parti coinvolte nel progetto devono essere in grado di leggerli!

3.3. Requisiti funzionali e non funzionali

- **Funzionali**, i servizi che il sistema deve fornire, cioè come il sistema deve reagire agli input e come il sistema deve comportarsi in particolari situazioni.
    - In alcuni casi i requisiti funzionali possono specificare quello che il sistema *NON* deve fare.
- **Non funzionali**, i vincoli imposti alle funzioni offerte dal sistema. Ad esempio vincoli temporali, sul processo di sviluppo e vincoli imposti da standard da rispettare. Sono generalmente applicati al sistema "as a whole" e non alle singole funzionalità.
    - Sono requisiti molto spesso critici: se non li si rispetta il sistema può essere inutilizzabile.

Sfortunatamente, nella realtà, questa divisione non è sempre così netta.

E' possiible che i requisiti siano tra di loro dipendenti, una funzionalità può dipendere da un altrà, oppure una caratteristica del sistema può richiedere l'implementazione di una o più funzionalità.

3.3.1. Requisiti non funzionali

I requisiti non funzionali li suddividiamo a sua volta in altre categorie:

- **di prodotto**, che specificano come il software deve comportarsi.
    - Si classificano a loro volta in requisiti di usabilità, efficienza (a sua volta in prestazionali e di spazio), affidabilità e portabilità.
    - Ad esempio: "L'interfaccia sarà implementata in HTML"
    - Usabilità, efficienza, prestazioni, spazio, affidabilità, portabilità

- **organizzativi**, che derivano dalle policies e dalle procedure adattate all'interno dell'organizzazione che richiede il software.
    - Possono anche includere requisiti su come il software deve essere sviluppato (modello di processo, linguaggi, tecnologie)
    - Consegna, standard, implementazione
- **esterni**, che derivano da fattori esterni che influenzano il sistema.
    - Leggi e regolamenti.
    - Interoperabilità, etici, sicurezza, riservatezza

3.4. Problemi nella raccolta dei requisiti

- **Ambiguità**, data dal fatto che i requisiti utente sono troppo generici.
    - L'utilizzo di aggettivi come "appropriato" o "adatto" rende il requisito ambiguo: come capisco cosa vuol dire l'utente con "adatto"?
- **Incompletezza**, i requisiti non riescono a descrivere tutto quello che c'è nel sistema.
    - In pratica, nella SW. Eng., è impossibile descrivere tutto il sistema
    - E' necessario essere pronti a gestire i cambiamenti
- **Inconsistenza**, le descrizioni contengono conflitti e contraddizione.
    - E' raro avere consistenza.
    - Ad esempio "l'utente riceve tutte le news pubblicate dall'accesso; le news dopo 7 giorni si cancellano", quindi deve riceve tutte le news o solo quelle degli ultimi 7 giorni?

Questi problemi non sono facilmente risolvibili, anzi sono spesso intrinsechi alal natura del progetto software. Una raccolta di requisiti potrebbe essere incompleta perchè il requisito di oggi è diverso dal requisito di domani.

3.5. Come scrivere i requisiti

Scrivere i requisiti in linguaggio naturale è molto pericoloso. Il linguaggio naturale, per sfortuna nostra, porta con se diverse problematiche:

- Ambiguità
    - una cosa si può fraintendere
- Mancanza di chiarezza
    - E' difficile essere precisi senza scrivere troppo testo
- Eccesso di flessibilità
    - una stessa cosa si può dire in molti modi diversi

Questo genera:

- Confusione tra requisiti
- Amalgama dei requsiti
- Mancanza di struttura

3.5.1. Alternative al linguaggio natuale

Informali:

- Linguaggio naturale strutturato
- Modello visuale informale

Formali:

- Specifica testuale formale

- Modello visuale formale

3.5.2. Scrittura dei requisiti

- Attraverso un formato standard utilizzato per tutti i requisiti
  - Un po' come adottare delle convenzioni di codice
- Usare il linguaggio in modo consistente
  - Usare termini come DEVE e DOVREBBE
- Evidenziare i concetti importanti
- Evitare il gergo informatico

Si potrebbe scrivere anche seguendo lo standard IEEE830 (Software Requirements Specifications).

Modello MoSCoW

Si classificano i requisiti in:

- Must
  - Una feature che il sistema DEVE avere
  - Priorità massima
- Should
  - Una feature che il sistema DOVREBBE avere
  - Priorità Alta
- Could
  - Una feature che il sistema POTREBBE avere
  - Priorità Media-Bassa
- Want/Wishlist
  - Una feature che si VORREBBE il sistema avesse, prima o poi
  - Priorità Bassa

4. Casi d'uso

Un caso d'uso è una storia testuale, utilizzata per scoprire e catalogare i requisiti.

4.1. Elementi di un caso d'uso

- Attore, qualcosa che interagisce con il sistema
- Scenario, una specifica sequenza di azioni tra gli attori e il sistema
    - Detta anche **istanza di un caso d'uso**
- Goal, obiettivo che lo scenario vuole portare a raggiungere

Un caso d'uso è una collezione di scenari *successfull* e *failure* che descrivono un attore usante un sistema in supporto ad un goal.

4.2. Tipi di attore

- **Attore primario**: raggiunge degli obiettivi utente usando i servizi del sistema
- **Attore di supporto**: offre un servizio, spesso è un sistema informatico
- **Attore fuori scena**: ha un interesse nel caso d'uso, ma non rientra nelle prime due categorie

4.3. Tipo di caso d'uso

- **brief**, un paragrafo scarso
    - generalmente descrive un solo scenario di successo
- **informale**, più paragrafi che coprono vari scenari sia di successoc che di fallimento
- **full**, tutti gli step e le variazioni del caso d'uso sono descritte in dettaglio
    - Nome del caso d'uso
        - Inizia con un verbo
    - Portata
        - Il sistema che si sta progettando
    - Livello
        - Obiettivo utente o sottofunzione
    - Attore primario
    - Parti interessate e interessi
    - Pre-condizioni
    - Garanzia di successo
        - Cosa deve essere vero se il caso d'uso viene completato con successo
    - Scenario principale di successo
    - Estensioni
        - Scenari alternativi di successo e di fallimento
    - Requisiti speciali
        - Non funzionali
    - Elenco delle variabili tecnologiche e dei dati
        - "Varianti nei metodi I/O e nel formato dei dati"
        -
    - Frequenza di ripetizione
    - Altre note

4.4. Come scrivere casi d'uso

Concentrarsi su:

- lo scopo dell'attore
- cosa deve fare il sistema

Ignorare:

- l'interfaccia utente
- come il sistema fa le azioni richieste

4.5. Verificare un caso d'uso

- Test del Capo
  - Se rispondendo con il nome del caso d'uso alla domanda del capo "Cosa hai fatto tutto il giorno?", il capo non si arrabbierà
- Test EBP
  - Ci si chiede se il caso d'uso aggiunge valore di business e lascia i dati in uno stato coerente
- Tese della dimensione
  - Il caso d'uso deve essere tra le 3 e le 10 pagine nel formato dettagliato

5. Modelli di dominio

Un modello di dominio è una rappresentazione visuale di classi concettuali o di oggetti del mondo reale di un dominio.

NON rappresenta oggetti software.

Astrazione visuale della terminologia e del contenuto informativo del dominio.

Applicando UML, il modello di dominio è un insieme di diagramma delle classi in cui non sono definite le operazioni, ma sono descritte solamente le classi concettuali, le associazioni tra classi e gli attributi.

6. Diagrammi di Sequenza e Contratti

Fanno parte dei diagrammi di interazione.

Un diagramma di sequenza di sistema è un elaborato che illustra eventi di input ed output relativi ai sistemi in discussione.

Un diagramma di sequenza di sistema è una figura che mostra, per un particolare scenario di un caso d'uso, gli eventi generati da attori esterni, il loro ordine e gli eventi tra i vari sistemi.

6.1. Contratti

I contratti delle operazioni usano pre-condizioni e post-condizione per descrivere i cambiamenti agli oggetti in un modello di dominio.

6.1.1. Esempio di Contratto

Operazione: enterItem(itemId: ItemID, quantity: Integer) Riferimenti: caso d'uso Process Sale Pre-Condizioni: E' in corso una vendita Post-Condizioni: E' stata creata una istanza di SalesItem, associata con la Sale corrente. SalesItem.quantity è diventata quantity, SalesItem è stata associata con una ProductDescription, in base alla corrispondenza con itemID

6.1.2. Sezioni del contratto

- Operazioni: nome e parametri dell'operazione
- Riferimenti: casi d'uso in cui può verificarsi questa operazione
- Pre-condizioni: ipotesi significati sullo stato del sistema
- Post-condizioni: lo stato degli oggetti dopo il completamente dell'operazione (istanze, associazioni, modifica attributi)