

Ingegneria del Software

Lombardi Lorenzo

January 2024

1 Introduzione

L'economia delle nazioni sviluppate si basa sui software. L'ingegneria del software ha a che fare con teorie, metodi e strumenti utilizzati per lo sviluppo di un software. Nei PC, i costi del software sono molto più elevati rispetto ai costi dei componenti hardware; inoltre, per i software è vero che il costo di mantenimento supera di molte volte il costo di sviluppo. Un progetto per lo sviluppo di un software può fallire se si sottovaluta la complessità del sistema richiesto, se non si riesce a stare al passo con la domanda del mercato o se non si utilizzano metodi e tecniche proprie dell'ingegneria del software.

1.1 Alcune definizioni

- **Software:** Programmi per computer e relativa documentazione, destinati a un cliente specifico o a un mercato ampio.
- **Attributi del buon software:** Manutenibilità, affidabilità e facilità di utilizzo; rispetta inoltre i requisiti richiesti di funzionalità e prestazioni.
- **Ingegneria del software:** Disciplina ingegneristica che si occupa degli aspetti di produzione di un software.
- **Attività fondamentali dell'ingegneria del software:** Specifica, Sviluppo, Validazione, Evoluzione.
- **Differenza tra ingegneria del software e informatica:** L'informatica si occupa dei fondamenti teorici, mentre l'ingegneria del software si focalizza sull'aspetto pratico dello sviluppo e consegna di software utili.
- **Differenza tra ingegneria del software e ingegneria dei sistemi:** L'ingegneria dei sistemi ha a che fare con tutti gli aspetti dello sviluppo dei sistemi basati su computer, inclusi hardware, software e ingegneria dei processi. L'ingegneria del software è parte di questo più generale processo.
- **Quali sono le sfide principali dell'ingegneria del software:** Affrontare la crescente diversità, la domanda di tempi di consegna ridotti e lo sviluppo di software affidabili.
- **Quali sono i costi?:** Il 60% sono costi di sviluppo, il 40% sono costi di testing. Per il software personalizzato, i costi di evoluzione sono maggiori dei costi di sviluppo.

- ***Quali sono i metodi e le tecniche migliori per l'ingegneria del software:*** Diversi a seconda della richiesta, ad esempio, un gioco richiede più prototipi, mentre un sistema di sicurezza richiede una specifica più chiara. Hanno in comune l'attenzione alla gestione del progetto.
- ***Il web come ha influito sull'ingegneria del software:*** Il web ha reso disponibile la possibilità di sviluppare sistemi largamente distribuiti e ha portato avanzamenti importanti nell'ambito del riutilizzo del software e dei linguaggi di programmazione.

1.2 Attributi del buon software

- **Manutenibilità** - Scritto in virtù di una possibile evoluzione futura.
- **Affidabilità** - Tale da essere pressoché inviolabile e incapace di fare danni in caso di system failure.
- **Efficienza** - Tale da non sprecare risorse del sistema; ad alta reattività, veloce e leggero.
- **Accettabilità** - Capace di essere comprensibile, utilizzabile e compatibile con gli altri sistemi in uso.

1.3 Attività nella creazione di un software

- **Specificità** - Dove clienti e ingegneri definiscono cosa farà il software e quali saranno i vincoli operativi.
- **Sviluppo** - Disegno e programmazione.
- **Validazione** - Dove si verifica che il software soddisfi i requisiti.
- **Evoluzione** - Dove il software viene modificato per riflettere i cambiamenti voluti dal cliente e dal mercato.

1.4 Problemi che affliggono il software

- **Eterogeneità** - Ai sistemi è richiesto di lavorare con i tipi più disparati di dispositivi.
- **Cambiamento sociale** - La società e il business si evolvono velocemente e nuove tecnologie vengono create; bisogna aggiornarsi velocemente.
- **Sicurezza e fiducia** - Ci si deve fidare di un sistema che è intrecciato così a fondo con le nostre vite.
- **Scalabilità** - Il sistema sviluppato potrebbe essere impiegato nei sistemi integrati molto piccoli oppure usato accessoriamente su sistemi cloud-based che servono una comunità globale.

1.5 Tipi di applicazione

- **Stand-alone** - Viene eseguito su un host, possiede ogni risorsa necessaria e non ha bisogno di connessione.
- **Interattive e basate su transazioni** - Viene eseguita su un computer remoto, gli utenti vi accedono dai loro dispositivi (es. e-commerce).

- **Sistemi di controllo integrati** - Controllano e gestiscono la parte hardware, sono i più diffusi.
- **Batch processing** - Sistemi business che processano dati su larga scala.
- **Sistemi di intrattenimento** - Per uso personale.
- **Sistemi di modellazione e simulazione** - Sviluppati da scienziati e ingegneri per modellare processi fisici e interazioni.
- **Sistemi di raccolta di dati** - Collezionano dati mediante sensori e li inviano a sistemi di elaborazione.
- **Sistemi di sistemi** - Sistemi composti da sistemi.

Le applicazioni web-based si suddividono nella stessa maniera; inoltre, per queste, è importante il riutilizzo di moduli se già esistenti e si può scegliere un approccio *agile* oppure *incremental* (discussi più avanti). Nel campo delle applicazioni web, sono diffusi i sistemi orientati ai servizi e l'utilizzo di tecnologie di sviluppo delle interfacce (AJAX, HTML5).

1.6 Codice etico

Oltre al rispetto delle leggi vigenti e del cliente, l'ingegnere deve prestare attenzione alla **confidenzialità** delle informazioni sui clienti e sui dipendenti; alla **competenza** rifiutando lavori se fuori dalla propria portata; ai **diritti della proprietà intellettuale** come copyright e brevetti; all'**uso improprio dei computer**, inteso dal giocare sul posto di lavoro alla disseminazione di virus.

1.6.1 ACM/IEEE Code of Ethics

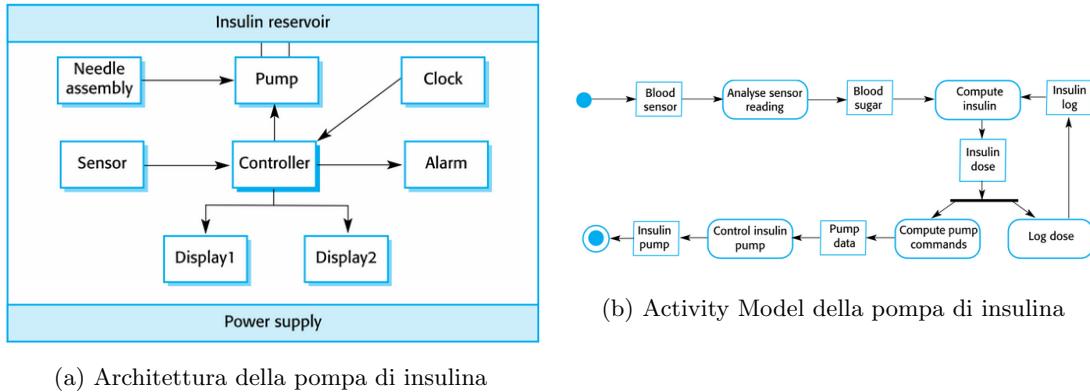
Otto regole pensate da ingegneri, manager, educatori, studenti e decisori politici con lo scopo di rendere l'ingegneria del software un lavoro rispettabile e benefico. Vengono di seguito riassunte:

1. **PUBLIC** - Si agisce nel pubblico interesse.
2. **CLIENT AND EMPLOYER** - Si agisce a vantaggio di clienti e datore di lavoro.
3. **PRODUCT** - Prodotti e modifiche di essi rispettano gli standard.
4. **JUDGEMENT** - Si mantiene l'integrità nel giudizio.
5. **MANAGEMENT** - Manager e leader promuovono un approccio etico alla gestione.
6. **PROFESSION** - Si coltiva l'integrità e la reputazione della professione.
7. **COLLEAGUES** - Si dimostra supporto e onestà verso i colleghi.
8. **SELF** - Si partecipa a un processo di apprendimento lungo tutta la vita nella pratica della professione e si promuove un approccio etico.

1.7 Casi studio

1.7.1 Sistema di controllo di una pompa di insulina

Collezione dati da un sensore di zucchero nel sangue e calcola la quantità di insulina da iniettare. Di seguito delle foto illustrate per l'architettura hardware e l'activity model (discussi nei prossimi capitoli).



(a) Architettura della pompa di insulina

(b) Activity Model della pompa di insulina

I requisiti essenziali di alto livello sono la disponibilità al rilascio di insulina quando richiesto e l'affidabilità nei calcoli.

1.7.2 Sistema informativo di un istituto di sanità mentale

Un sistema informativo per registrare pazienti che soffrono di problemi mentali e le relative cure ricevute. Il sistema realizzato prende il nome di *"Mentcare"* e permette di organizzare le cure in maniera efficiente, permettendo inoltre il download locale delle cartelle cliniche.

Key Features

- I medici possono creare, consultare e modificare le cartelle cliniche. Il sistema propone una panoramica breve di problemi e trattamenti per ogni paziente.
- Patient Monitoring
- Report mensili auto-generati su pazienti presenti nella clinica, quelli che l'hanno abbandonata, le prescrizioni mediche e i loro costi.

System Concerns

- Privacy** - Le informazioni sono confidenziali e mai divulgare se non allo staff medico autorizzato e al paziente stesso.
- Safety** - Per via della gravità dei problemi di alcuni pazienti, i medici curanti vengono avvisati nel caso di pericolo per la loro vita o del paziente. Il sistema deve essere sempre disponibile per prescrivere le cure giuste ai pazienti.

2 Software Process

Set di attività richieste per sviluppare un sistema software. Processi differenti, ma tutti coinvolgono specifica, sviluppo, validazione ed evoluzione. Un modello di processo del software è una rappresentazione astratta di un processo che lo descrive da una certa prospettiva. Si parla delle attività di un processo come: specificare un data model, disegnare un'interfaccia, etc. Le descrizioni di un processo includono:

- **Prodotti** - Risultato di un'attività di processo
- **Ruoli** - Responsabilità di chi è coinvolto
- **Pre e post condizioni** - Ciò che è vero prima e dopo un'attività di processo

Due principali modalità di processi:

- **Plan-driven** - Le attività sono pianificate in anticipo e sono usate per misurare i progressi
- **Agile** - Planning incrementale per adattarsi al cambiamento dei requisiti del cliente

Di seguito i principali modelli di processo del software:

- **Modello Waterfall**
- **Sviluppo Incrementale**
- **Integrazione e Configurazione**

2.0.1 Modello Waterfall

Modello plan-driven con le fasi di specifica e sviluppo separate.

Pro - Veloce per sistemi piccoli e senza cambiamenti, comodo per coordinare sistemi grandi in sviluppo su più siti.

Contro - Poco flessibile ai cambiamenti, step troppo separati l'uno dall'altro.

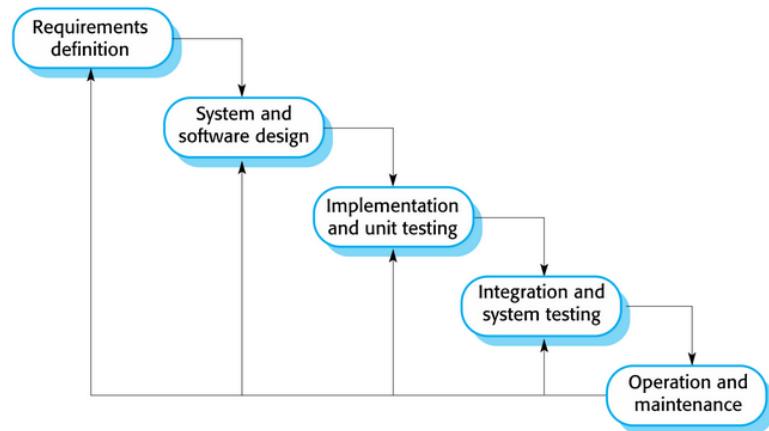


Figura 2: Schema del Modello Waterfall

2.0.2 Sviluppo Incrementale

Specifico, sviluppo e validazione sono interlacciate. Può essere plan-driven o agile.

Pro - Più flessibile per clienti che cambiano requisiti, c'è la possibilità di rilasciare demo e quindi di avere un feedback dai clienti anche a lavoro non terminato. Il prodotto è pronto in tempo ridotto.

Contro - Il sistema degrada velocemente e richiede più cura nello sviluppo. Il processo non è visibile da subito e non si ha modo di misurarla.

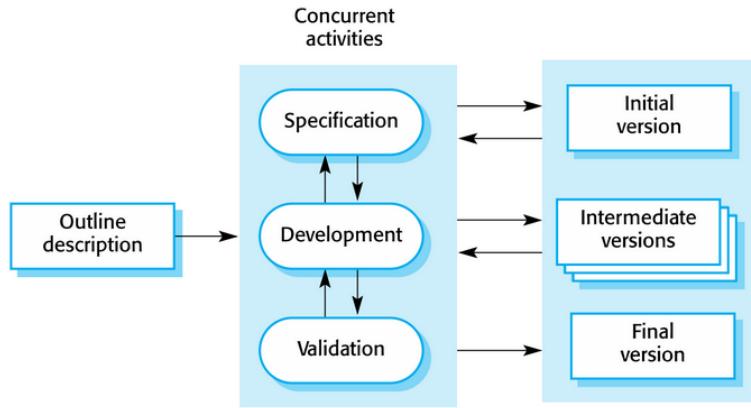


Figura 3: Schema dello Sviluppo Incrementale

2.0.3 Integrazione e Configurazione

Il sistema è assemblato da componenti esistenti. Può essere plan-driven o agile. I componenti esistenti possono essere sistemi applicativi stand-alone (conosciuti come **COTS** ovvero commercial-off-the-shelf), collezioni di oggetti da integrare in framework come .NET e J2EE, servizi web standard da utilizzare attraverso remote invocation.

Pro - Meno costi e meno rischi, consegna veloce.

Contro - Sono necessari dei compromessi per i requisiti e si perde facilmente il controllo dei componenti quando ci si evolve.

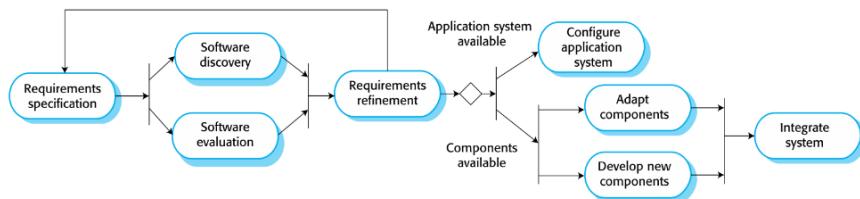


Figura 4: Schema del Modello a Integrazione e Configurazione

2.1 Attività dei Processi

Specifiche del Software - Per capire cosa vogliono gli stakeholders da questo software, quali sono i requisiti in dettaglio (e se sono validi) e quali sono i vincoli operativi e di sviluppo.

Design e Implementazione - Per convertire la specifica in un sistema eseguibile dopo un'attenta progettazione, sono due attività interlacciate.

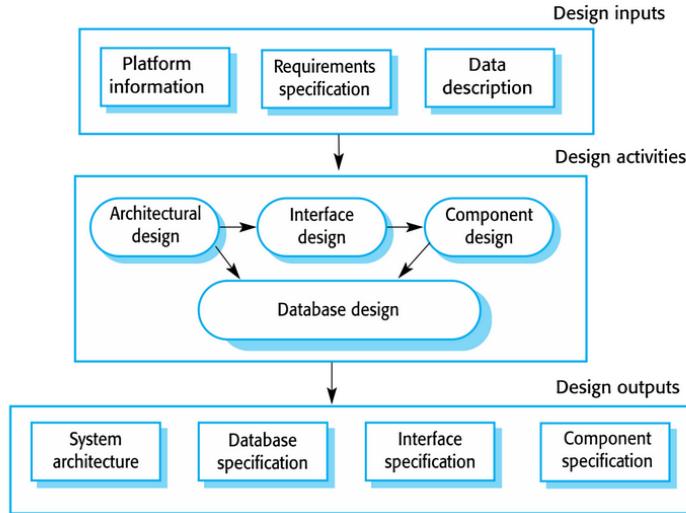


Figura 5: Modello generale del processo di Design

Di seguito le attività di design spiegate:

- **Design Architetturale** - Si identifica la struttura del sistema e le relazioni tra i principali moduli
- **Design del Database** - Si identificano le strutture e le rappresentazioni nel database
- **Design dell'Interfaccia** - Si definiscono le interfacce tra i componenti
- **Design dei Componenti** - Si cercano componenti adeguati già esistenti o si progettano

Si implementa e si inizia la fase di debug.

Validazione - Per mostrare che il programma è conforme alla specifica e rispetta i requisiti. Involge il testing eseguito con blocchi realistici di dati da processare. Le fasi sono:

1. **Test dei Componenti** - Test individuali su componenti (funzioni e/o oggetti)
2. **Test del Sistema** - Test del sistema completo con un focus sulle proprietà olistiche che emergono
3. **Test del Cliente** - Test con dati del cliente per vedere se vengono rispettate le richieste iniziali

Evoluzione - I software sono flessibili e devono supportare aggiornamenti.

2.2 Affrontare i Cambiamenti

Nei grandi progetti, il cambiamento è inevitabile; cambiamenti del business importano nuovi requisiti nel sistema, nuove tecnologie suggeriscono nuove implementazioni e nuove piattaforme richiedono nuove applicazioni. Come affrontare il cambiamento? Prototipi e rilascio incrementale.

2.2.1 Prototipo

Un prototipo è una versione del software rilasciata così da avere un feedback dal cliente, viene successivamente scartata perché è inutile come base di lavoro, facilmente degradabile e non documentata. Vengono chiaramente trascurate alcune funzionalità che non si vogliono mostrare e alcuni aspetti della sicurezza.

2.2.2 Rilascio Incrementale

Sviluppo e rilascio sono spezzettati in diversi frammenti nei quali si sviluppa a piccoli passi ogni funzionalità. I primi incrementi hanno le funzionalità ad alta priorità così da chiarire immediatamente con il cliente le cose che vanno aggiustate; è la prassi nello sviluppo *agile*.

Pro - I primi incrementi funzionano come i prototipi, si ha un feedback molto rapido e utile, si abbassa il rischio di fallimento del progetto, i primi incrementi subiscono più testing essendo presenti da più tempo nel sistema in sviluppo.

Contro - Molti sistemi utilizzano strutture utili in diverse parti del loro codice e con questo tipo di rilascio non si ha una reale concezione di ciò che è utile modularizzare. In molte aziende la specifica iniziale è parte del contratto e un lavoro che ne richiede una non finita e in continuo aggiornamento può causare conflitti.

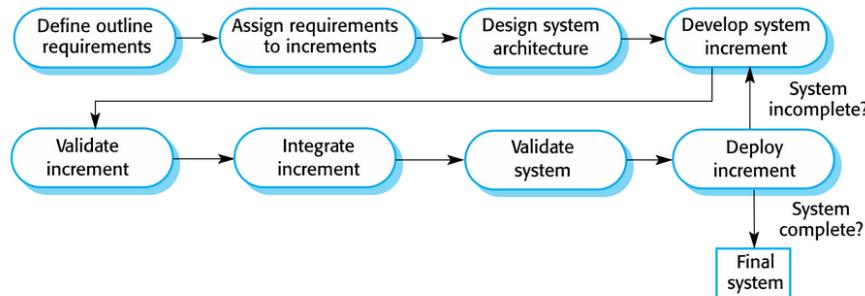


Figura 6: Schema dell'Incremental Delivery

2.3 Miglioramento dei Processi

Spesso si utilizza l'approccio *process maturity* in cui si migliora il management e le pratiche utilizzate; altre volte invece si usa un approccio *agile* in cui si riducono gli overheads (risorse accessorie del sistema). Il ciclo di miglioramento opera in 3 fasi: **MISURA**, dove si verifica se il miglioramento precedente ha funzionato, **ANALISI**, dove si trovano le debolezze e si pensa a un modello che le risolva, **CAMBIAMENTO**, dove si introducono le migliorie. Bisogna sempre raccogliere dati su tempi per completare le attività, sulle risorse (persone) coinvolte, sui difetti trovati, sugli errori ripetuti.

2.4 Capability Maturity Model - SEI

Un approccio al miglioramento dei processi al fine di ottimizzare la produzione e l'utilizzo di risorse. Gli stadi rilevati sono:

- **Initial** - Processi fuori controllo e poco reattivi
- **Repeatable** - Processi caratterizzati per i progetti
- **Defined** - Processi caratterizzati per l'organizzazione
- **Managed** - Processi misurati e controllati
- **Optimising** - Focus sul miglioramento dei processi

3 Sviluppo Software - Agile

Il requisito più importante in questo settore è la rapidità nello sviluppo e nella consegna dei sistemi. Lo sviluppo *agile*, che meglio risponde alle esigenze di un mercato in rapido cambiamento, è nato alla fine degli anni '90 e ha le seguenti caratteristiche:

- Specifica, design e implementazione interlacciati
- Il lavoro procede per incrementi ed è sottoposto al vaglio degli *stakeholders*; è importante la collaborazione e la poca complessità
- La documentazione è ridotta e ci si concentra sul codice, mirando ad aumentare la flessibilità del sistema
- Vengono forniti strumenti di test automatizzati per supportare lo sviluppo

3.1 Metodi Agile

Si cambia approccio alla programmazione e si riducono gli overheads (funzionalità accessorie) per rispondere più velocemente al cambiamento delle esigenze degli stakeholders, focalizzandosi sul codice e non sul design, adottando un approccio iterativo allo sviluppo.

3.1.1 Manifesto dello sviluppo agile

*Individui e interazioni piuttosto che processi e strumenti.
Software funzionante piuttosto che documentazione completa.
Collaborazione con il cliente piuttosto che negoziazione del contratto.
Rispondere al cambiamento piuttosto che seguire un piano.*

3.1.2 Principi dei Metodi Agile

- **Coinvolgimento del Cliente** - I clienti comunicano i nuovi requisiti e giudicano esternamente le iterazioni del sistema
- **Rilascio Incrementale** - Sviluppo incrementale con funzionalità dettate dai clienti
- **Persone, Non Processi** - Bisogna valorizzare le persone coinvolte nel team, lasciandole libere di esprimersi
- **Abbracciare il Cambiamento** - Prevedere i futuri cambiamenti e disegnare il sistema per accomodarli
- **Semplicità** - Nel software e nel processo di sviluppo

Si applicano a sistemi medio-piccoli e in ambienti in cui i clienti vogliono essere coinvolti e ci sono poche regole imposte dall'esterno.

3.2 Tecniche Agile

3.2.1 Extreme Programming (XP)

Vengono fatte build più volte al giorno, gli incrementi sono rilasciati una volta ogni due settimane se passano tutti i test.

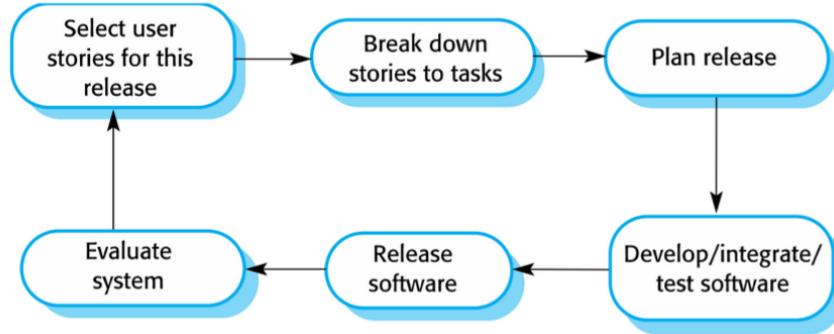


Figura 7: Schema dell'Extreme Programming (XP)

Pratiche Comuni

- Funzionalità riassunte in *story cards* e messe in ordine secondo priorità
- Un cliente (o un suo rappresentante) è presente sul sito di sviluppo e fa attivamente parte del team
- Vengono pensati i test e poi implementate le funzionalità
- Refactoring continuo appena si trova una miglioria applicabile
- Piccole release
- Lavoro in coppie, ogni coppia lavora su tutti i campi di sviluppo
- Integrazione continua
- Lavoro rapido
- Design semplice

Risulta complesso attuare questo tipo di approccio, quindi si preferisce attuarne solo alcune pratiche. Vengono utilizzate le *story cards*, il refactoring, i test prima dello sviluppo e la programmazione in coppia.

Refactoring Il codice viene ispezionato e migliorato anche se sembra che non ci sia un bisogno immediato; ad esempio, vengono rinominati attributi poco chiari o vengono modularizzate alcune azioni.

Test-driven Development (TDD) Scrivere blocchi di test come programmi da integrare in futuro una volta implementate le funzioni. Questa pratica non è condivisa perché è difficile giudicare la completezza di un blocco di test, le interfacce grafiche non possono essere testate facilmente e, inoltre, è cattiva abitudine cercare *workaround* quando si scrivono test reputati troppo complessi.

3.3 Gestione di un Progetto Agile

Bisogna rispettare principalmente tempo e budget; il che risulta difficile non disponendo di un piano e di una specifica chiari e completi sin dall'inizio (come nello sviluppo plan-driven).

3.3.1 Scrum

Un metodo agile che si focalizza sulla gestione di uno sviluppo iterativo invece che sulle pratiche agile. Organizzato in tre fasi:

1. Si stabiliscono gli obiettivi e si disegna l'architettura
2. Serie di cicli *sprint*, dove ogni ciclo è un incremento
3. Si chiude il progetto, si finisce la documentazione e il manuale, e si valutano gli insegnamenti del lavoro

Il lavoro è eseguito da un team autogestito e auto-organizzato di massimo 7 persone che non comunica con l'esterno. Il lavoro permetterebbe il rilascio di incrementi autonomi già pronti al mercato e supervisionati da un proprietario (cliente, *stakeholder* oppure capo del progetto).

Scrum - Un incontro giornaliero tra membri del team che comunicano gli obiettivi del giorno.

ScrumMaster - Un responsabile delle comunicazioni con l'esterno, si elegge a portavoce del team scrum per interracciarsi col resto della compagnia.

Sprint - Un'iterazione di sviluppo, di solito lunga 2-4 settimane.

Velocity - Una stima di quanto backlog (to-do list) può sopportare uno sprint.

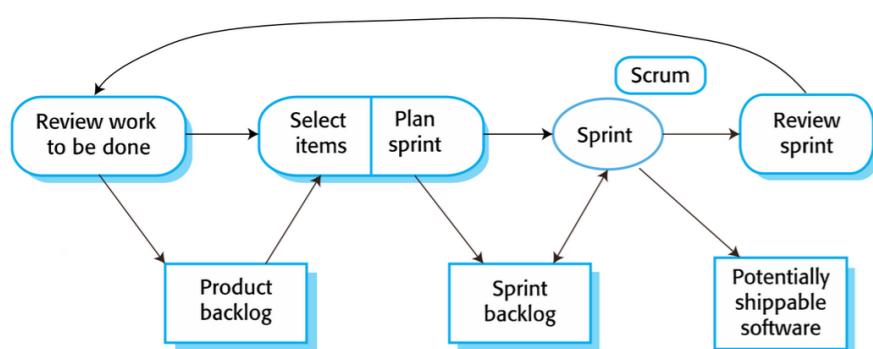


Figura 8: Schema del ciclo di sprint scrum

Quali benefici porta questo tipo di gestione? Il prodotto è spezzettato in una serie di chunk comprensibili e maggiormente gestibili; tutto il team vede tutto e la comunicazione funziona meglio; i clienti vedono incrementi continui e aumenta la loro fiducia nel progetto.

3.4 Ridimensionare i Metodi Agile

- **Scaling Up** - Usare metodi agile per sistemi più grandi
- **Scaling Out** - Usare metodi agile all'interno di aziende molto grandi e con molta esperienza

Mentre si opera *scaling*, è importante mantenere una pianificazione flessibile, delle release incrementalì e una buona comunicazione. Il problema principale rimane l'approccio informale e che è svincolato dall'idea di "contratto" che invece è radicata presso molte aziende. Si consiglia quindi di stare attenti con questo tipo di approccio e di assicurarsi di poter disporre il più a lungo possibile di un team composto dalle stesse persone; inoltre, forzare una release in un ambiente che fa fatica ad ingranare non presuppone un buon operato.

3.5 Conclusioni

L'idea è quindi quella di scegliere l'approccio in base all'azienda, al progetto che si ha in mente, alle persone coinvolte e al budget. I fattori comuni (e quindi la natura dei problemi) dei due approcci sono:

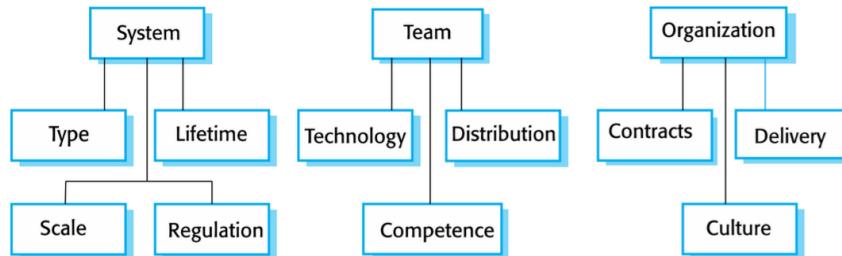


Figura 9: Fattori comuni di sviluppo plan-driven e agile

Le aziende tradizionali hanno una cultura basata sull'approccio plan-driven. Spesso i sistemi grandi sono collezioni di sistemi più piccoli (poco flessibili) in comunicazione tra loro; sono inoltre soggetti a regolamentazioni esterne e dotati di un team instabile oltre che a un set eterogeneo di *stakeholders*.

3.6 Metodi Agile per i *Large Systems*

Come si può passare da sistemi piccoli a grandi in maniera sicura con i metodi agile? Bisogna trovare un equilibrio nel rilascio degli incrementi e bisogna istituire sistemi di comunicazione cross-team. È possibile utilizzare anche lo scrum multi-team ma bisogna predisporre un proprietario e uno ScrumMaster per ogni team, una data comune di rilascio degli incrementi (così da assemblare il sistema), e uno scrum di scrum (quindi assemblee di rappresentanti di rappresentanti); ovvero si esegue *scaling* sulla gerarchia totale.

4 Ingegneria dei Requisiti

Processo per stabilire vincoli e requisiti di un sistema.

Requisito - Astrazione di un servizio richiesto oppure specifica funzionale matematica dettagliata. Nell'interpretazione di Davis il requisito deve essere inizialmente astratto, così che diverse aziende possano concorrere e interpretare a loro modo il contratto; solo successivamente viene definito in dettaglio.

Esistono diversi tipi di requisiti:

- **Requisiti dell'utente** - Espressi in linguaggio naturale e accompagnati da diagrammi dei servizi offerti. Scritti per i clienti
- **Requisiti di sistema** - Documento dettagliato di funzioni, servizi e vincoli operativi del sistema. Definiscono cosa deve essere implementato, sono la base di un contratto.

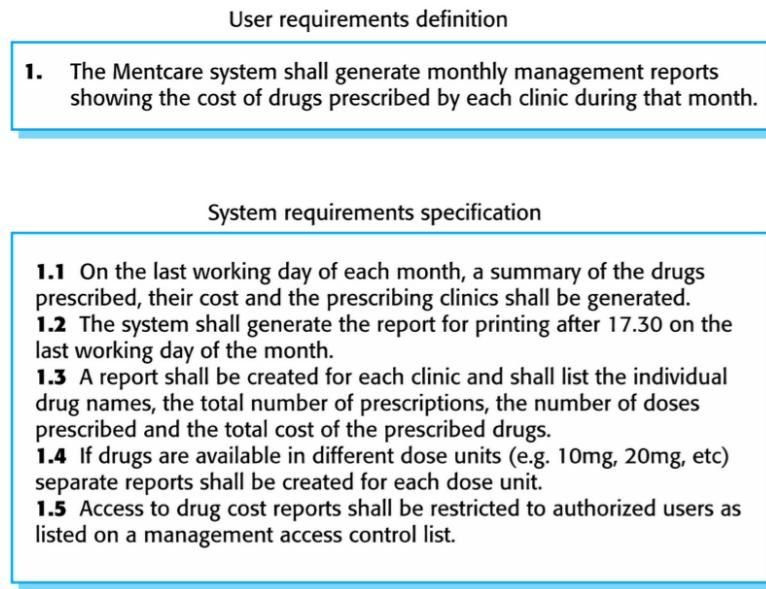


Figura 10: Differenza tra requisiti dell'utente e di sistema

Secondo i metodi agile questa specifica è inutile perché cambia continuamente, viene allora espressa attraverso le *user stories*, quest'ultime sono quindi adatte per sistemi di business ma non per sistemi critici come quelli di sicurezza.

- **Requisiti funzionali** - Servizi offerti, come il sistema reagisce agli input e che output sono prodotti. A volte può indicare cosa il sistema non fa. Devono essere completi (con descrizione) e consistenti (non devono generare ambiguità)
- **Requisiti non funzionali** - Vincoli dei servizi e delle funzioni. Affliggono l'intero sistema e ognuno di essi può generare diversi. Possono riguardare il tipo di IDE da utilizzare o il metodo di programmazione, la sicurezza (presenza di backup), la velocità, la facilità di utilizzo (finestre di aiuto), la portabilità, gli standard e le leggi da rispettare, etc.

4.1 I Processi dell'Ingegneria dei Requisiti

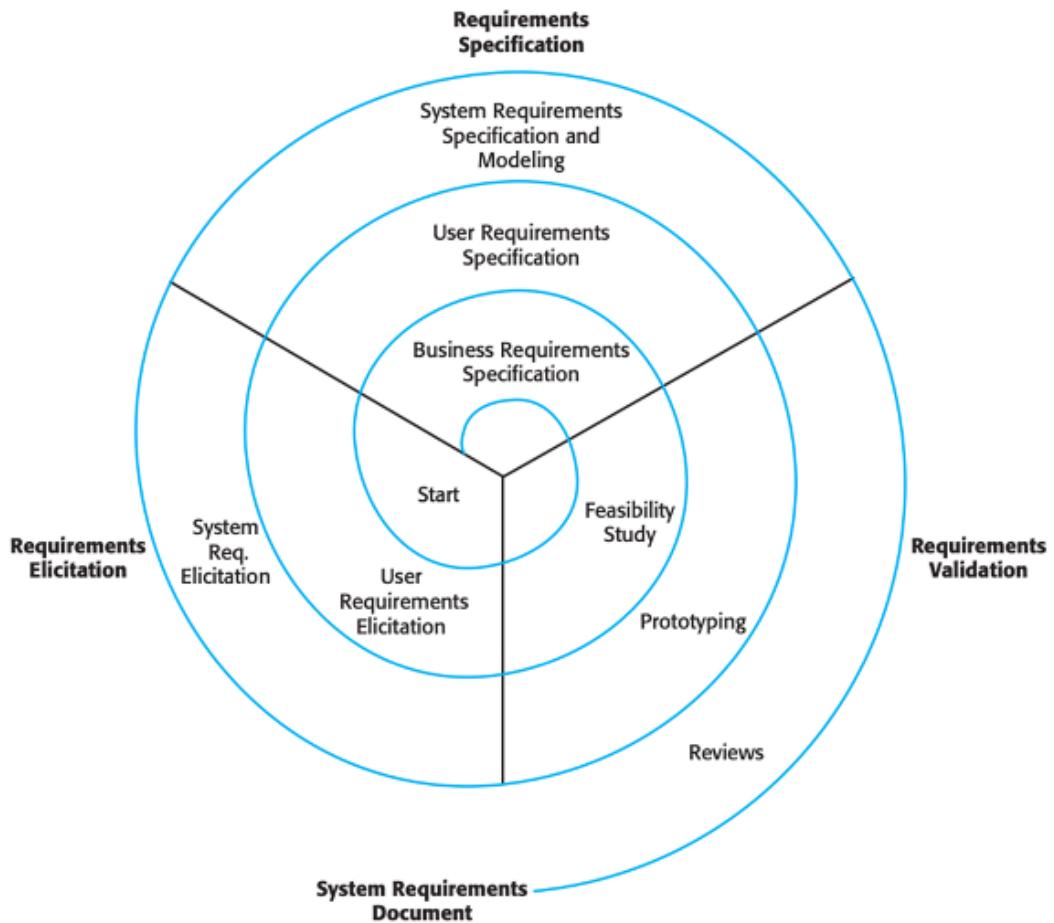


Figura 11: La spirale dei processi della RE

4.1.1 Elicitazione e Analisi

Coinvolge lo staff tecnico e i clienti per capire il dominio di applicazione del sistema, i servizi che deve offrire e i vincoli. Può coinvolgere anche tutti gli altri *stakeholders*. Il problema è che non tutti hanno le idee chiare su cosa vogliono, e quelli che lo sanno potrebbero trovarsi in disaccordo, o potrebbero cambiare idea in seguito. Gli *stages* sono quattro:

1. **Scoperta** - Dialogo con gli *stakeholders*
2. **Classificazione e Organizzazione** - Si raggruppano i requisiti in cluster
3. **Prioritizzazione e Negoziazione** - Si ordinano per priorità
4. **Specificazione** - Si documentano i requisiti e si passa al prossimo step della spirale

Colloqui - Strumento utilizzato per risolvere problemi con gli *stakeholders*. Ne esistono di chiusi (lista predeterminata di domande da porre) o di aperti (di discussione). Vengono usati impropriamente se non si è disposti a collaborare o a capire le richieste, o se il dominio non viene discusso perché ritenuto banale.

Si usano scenari d'utilizzo per studiare il sistema in costruzione. Dall'inizio a tutto il flusso di eventi fino a cosa può andare storto e allo stato del sistema al momento finale; gli *stakeholders* riescono meglio ad empatizzare con questo sistema perché riguarda più da vicino la vita reale.

4.1.2 Specifica

Come funziona la specifica? Si documentano i requisiti dell'utente e del sistema in maniera completa e dettagliata.

Specifiche - Linguaggio Naturale Ogni frase rappresenta un requisito, è comprensibile da utenti e clienti; si deve adottare uno standard e si deve evitare il gergo dei computer. Il problema principale è la mancanza di precisione, e il miscuglio tra requisiti funzionali e non funzionali

Specifiche - Form-based/Strutturata Per ogni funzione definire gli input, gli output, le informazioni necessarie a comprenderne il funzionamento, le pre e post condizioni, gli effetti collaterali e le azioni da intraprendere.

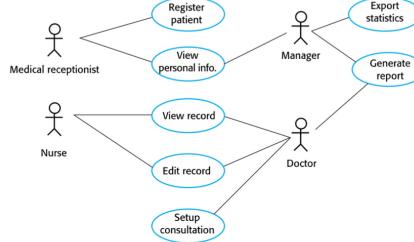
ECLIPSE/W orkstation/Tools/DE/FS/3.5.1	
Function	Add node
Description	Adds a node to an existing design. The user selects the type of node, and its position. When added to the design, the node becomes the current selection. The user chooses the node position by moving the cursor to the area where the 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 addition of a node of the specified type at the given position.

Definition: ECLIPSE/W orkstation/Tools/DE/RD/3.5.1

Specifiche - Tabulata Usata a supporto del linguaggio naturale, utile per definire un numero di alternative legate ad un'azione.

Condition	Action
Sugar level falling ($r2 < r1$)	CompDose = 0
Sugar level stable ($r2 = r1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing $((r2 - r1) < (r1 - r0))$	CompDose = 0
Sugar level increasing and rate of increase stable or increasing $((r2 - r1) \geq (r1 - r0))$	CompDose = round(($(r2 - r1)/4$) If rounded result = 0 then CompDose = MinimumDose

Specifiche - Use Case Scenari inclusi nell'UML, dovrebbero descrivere tutte le possibili interazioni di ogni agente che compare nel sistema.



Considerazioni La specifica dovrebbe contenere ciò che il sistema deve fare senza spiegare il come. Alcuni standard sono stati definiti dall'IEEE e si è osservato che più un sistema cresce, meno dettagliato tende ad essere la sua specifica. Una buona specifica contiene una prefazione, una introduzione al sistema, l'evoluzione che ha subito nel tempo, la dichiarazione dei requisiti dell'utente e del sistema, i modelli utilizzati, i grafici che mostrano le interazioni tra gli agenti e un indice.

4.1.3 Validazione

Momento in cui si dimostra che i requisiti definiscono l'esatto sistema che vuole il cliente. Gli errori in questa fase sono costosi da riparare quindi è un momento delicato. Si effettuano controlli su:

- **Validità** - Provvede le funzioni richieste?
- **Consistenza** - Ci sono conflitti tra requisiti?
- **Completezza** - Sono incluse tutte le funzioni?
- **Realismo** - Il budget copre le spese di implementazione?
- **Verificabilità** - I requisiti possono essere controllati?

Che tecniche vengono utilizzate durante la validazione?

- Review sistematiche dei requisiti (sono coinvolti sia i tecnici che il cliente) e quindi se è possibile testare, modificare, introdurre, eliminare elementi senza spese ingenti.
- Prototipi
- Generazione test

4.1.4 Gestione

Dopo l'installazione può esserci nuovo hardware, nuove interfacce con nuovi sistemi, nuove leggi e gli utenti utilizzatori potrebbero essere diversi dal cliente. Bisogna allora cambiare i requisiti dopo aver compreso esattamente cosa è cambiato nel mondo intorno a noi. Per operare una buona gestione è **necessario** conoscere come i requisiti sono collegati e quale di questi ne influenza altri; e quanti fondi si hanno a disposizione, così come quanto il cambiamento sia necessario.

5 System Modeling

Il processo mediante il quale vengono creati modelli astratti secondo prospettive diverse con lo scopo di comunicare con il cliente e analizzare le funzionalità del sistema. I modelli vengono utilizzati per presentare proposte agli *stakeholders* e documentare il sistema in vista dell'implementazione.

Prospettive

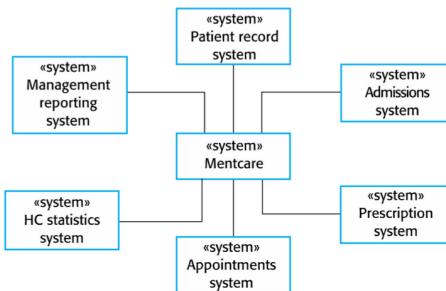
- External
- Interaction
- Structural
- Behavioral

Tipi di diagrammi UML

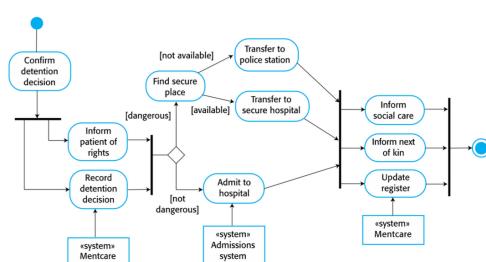
- Activity diagram
- Use case diagram
- Sequence diagram
- Class diagram
- State diagram

5.1 Context Models (external)

Mostrano cosa avviene fuori dai confini del sistema e nei rapporti con gli altri sistemi. La scelta dei confini è politica e si ripercuote fortemente sui requisiti. Per capire come viene utilizzato un sistema in un contesto si utilizzano *process models* accompagnati da *activity diagrams*.



(a) Esempio di context model



(b) Esempio di process model

5.2 Interaction Models

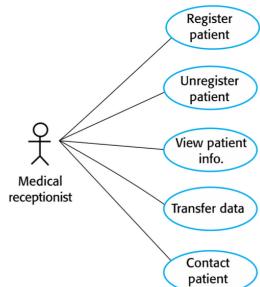
Modellare le interazioni dell'utente aiuta a capire i requisiti; modellare le interazioni tra sistemi aiuta a prevenire problemi comunicativi; modellare le interazioni tra componenti permette di misurare se un sistema sarà efficiente. Si utilizzano *use case diagrams* e *sequence diagrams*.

5.2.1 Uso degli use case diagrams

Gli agenti possono essere persone o altri sistemi; sono rappresentazioni grafiche accompagnate da testo scritto.

5.2.2 Uso dei sequence diagrams

Mostrano le sequenze di interazioni coinvolte in uno use case. Oggetti e attori sono riportati in cima mentre le frecce indicano le interazioni.



(a) Esempio di use case diagram

MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcase system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

(b) Descrizione testuale di uno use case diagram

Medical Receptionist

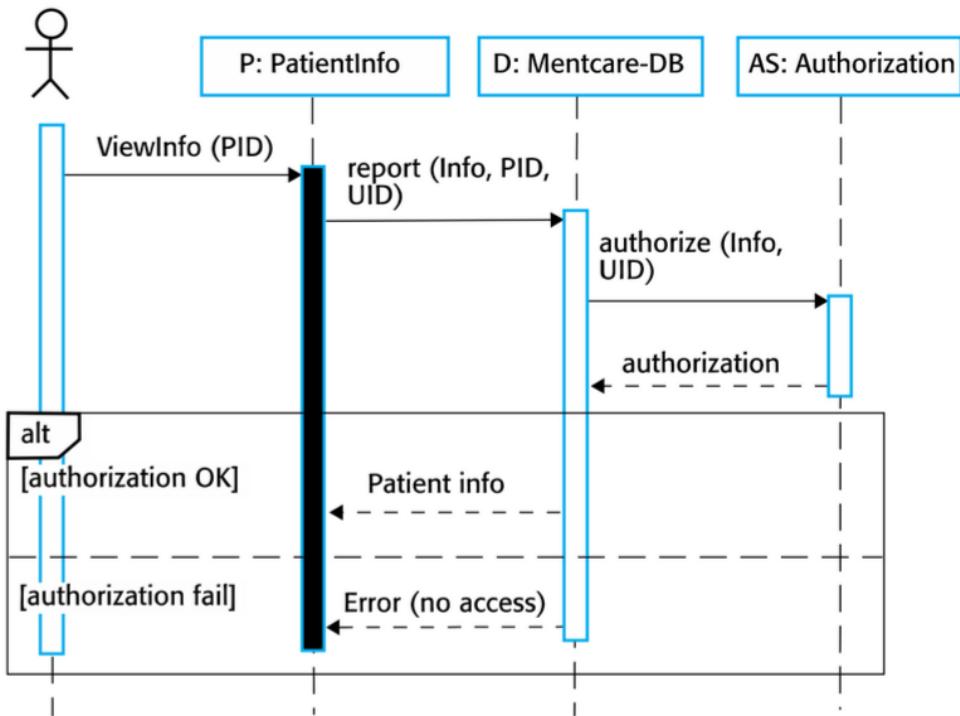


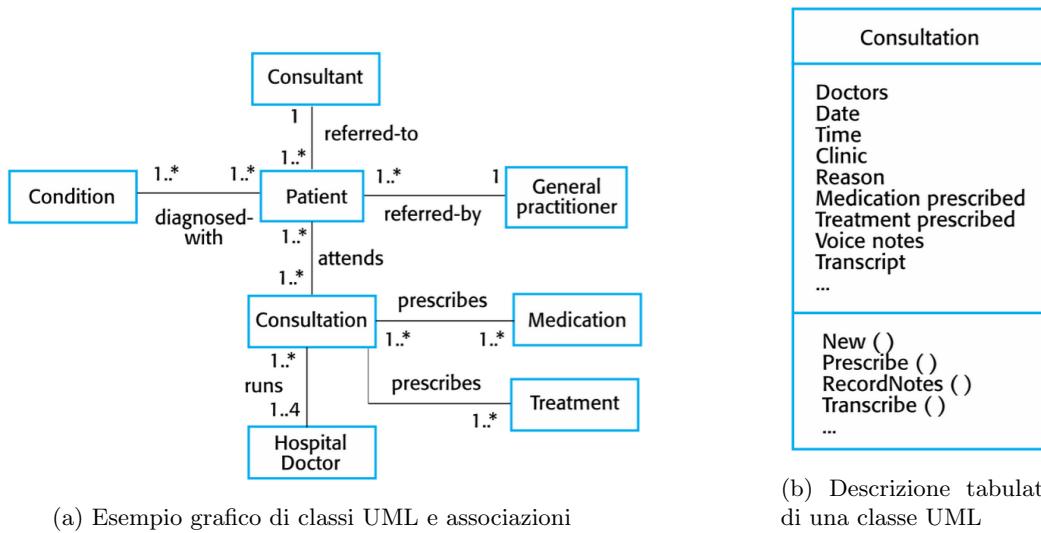
Figura 14: Esempio di sequence diagram

5.3 Structural Models

Mostra le relazioni tra i componenti di un sistema. Esistono modelli **statici**, che mostrano il design, e **dinamici**, che mostrano l'organizzazione del sistema quando è in esecuzione. Si utilizzano quando si discute l'architettura del sistema.

5.3.1 Uso dei class diagrams

Sono perfetti quando si utilizza la programmazione orientata agli oggetti perché mostrano le associazioni tra le classi. Quindi gli oggetti del sistema sono istanze delle classi e un'associazione indica una possibile interazione.



Generalizzazione Processo mediante il quale un oggetto viene associato a una classe nota per gestirne la complessità; in programmazione, si utilizza l'ereditarietà. Le sottoclassi (figlie) ereditano attributi e operazioni dalle classi superiori (più si scende, più si è specifici).

5.4 Behavioral Models

Mostra come il sistema reagisce agli stimoli dell'ambiente. Gli stimoli sono di due tipi:

- **Dati** - Informazioni da elaborare.
 - **Eventi** - Situazioni che innescano l'elaborazione del sistema. Possono essere accompagnate da dati.

5.5 Data-driven Modeling

Alcuni sistemi sono quasi esclusivamente di elaborazione dati; è necessario disporre di un *activity diagram* che mostri come avviene questa elaborazione dall'inizio alla fine.

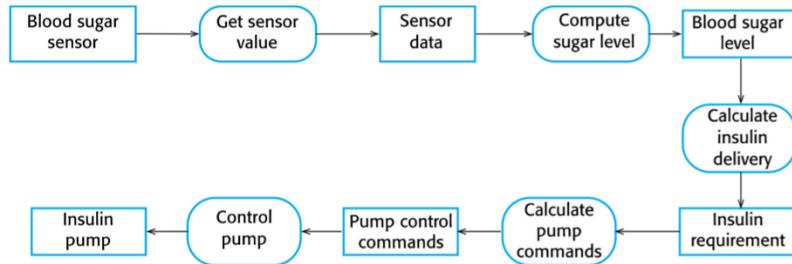


Figura 16: Esempio di un activity diagram per una pompa di insulina

5.6 Event-driven Modeling

Alcuni sistemi dotati di sensori sono costruiti come delle macchine a stati; serve allora uno *state diagram* che metta in evidenza la risposta ad eventi interni ed esterni.

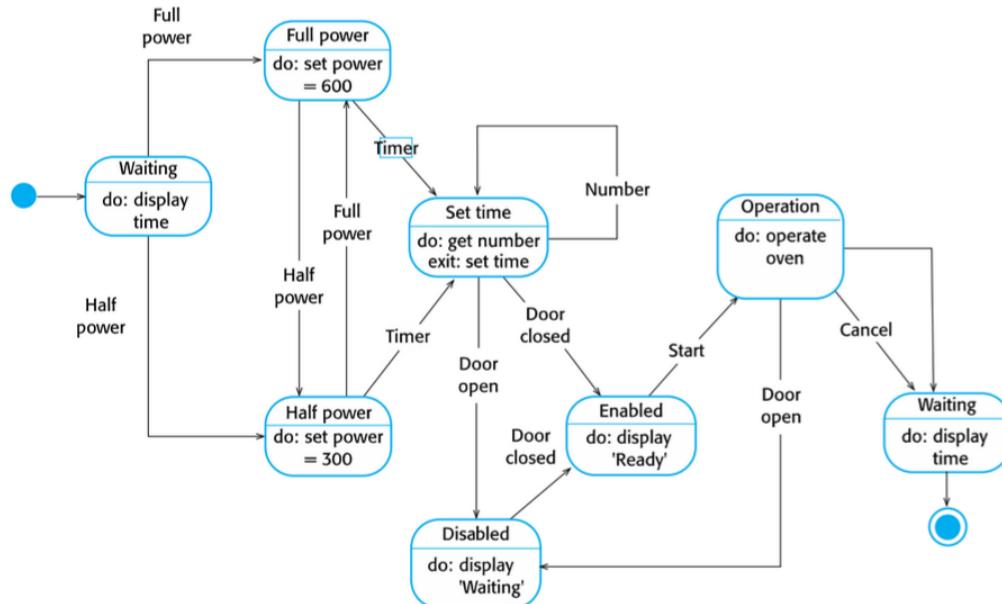
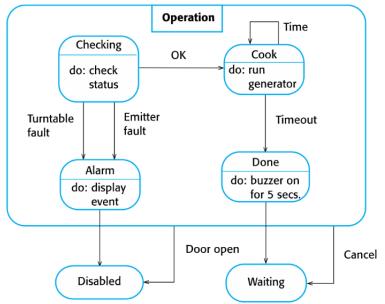


Figura 17: Esempio di state diagram per un forno a microonde



(a) Ulteriore esempio di state diagram

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

(b) Specifica di stati e stimoli per uno state diagram

5.7 Model Driven Engineering

Approccio all'ingegneria del software in cui l'output non è un programma, ma un modello capace di generarlo. I sostenitori trovano positiva la maggiore astrazione e libertà concessa agli ingegneri, mentre i detrattori ritengono che i costi di adattamento (e traduzione) per le nuove piattaforme siano troppo alti; resta un'idea in fase di sviluppo.

5.8 Model Driven Architecture

Il precursore dell'MDE utilizza modelli grafici a diversi livelli di astrazione per descrivere un sistema; questi modelli dovrebbero, in linea di principio, essere sufficienti per generare un programma funzionante. Che tipi di modelli usa?

- **CIM (Computation Independent Models)** - Modellano l'astrazione dei domini.
- **PIM (Platform Independent Models)** - Modella le operazioni del sistema senza citare l'implementazione e descrive come il sistema reagisce staticamente agli stimoli.
- **PSM (Platform Specific Models)** - Collezione di PIM per piattaforme diverse.

5.8.1 MDA e metodi agile

I sostenitori dell'MDA dichiarano che è stato ideato pensando alle iterazioni; tuttavia, avere un'idea compatta e statica di un progetto sin dall'inizio sembra andare in disaccordo con il manifesto agile. Idealmente, la combinazione di PIM e MDA potrebbe automatizzare la produzione di un sistema.

5.9 Adozione di MDA/MDE

Cosa ha rallentato l'adozione di questi approcci?

- Il focus potrebbe essere sulla sicurezza o sull'affidabilità piuttosto che sull'implementazione.
- I tool specifici per la traduzione tra modelli dovrebbero essere sviluppati da un'azienda che può garantire un supporto a lungo termine, attualmente non ovvio.
- L'astrazione nella discussione non può essere la stessa che per i modelli.

6 Progettazione Architetturale

La progettazione architetturale riguarda l'organizzazione di un sistema e la struttura complessiva; è il collegamento che c'è tra il design e l'ingegneria dei requisiti poiché identifica i componenti principali e le relazioni tra loro. Il lavoro viene iniziato nei primi stadi dell'agile così da evitare un refactor costoso.

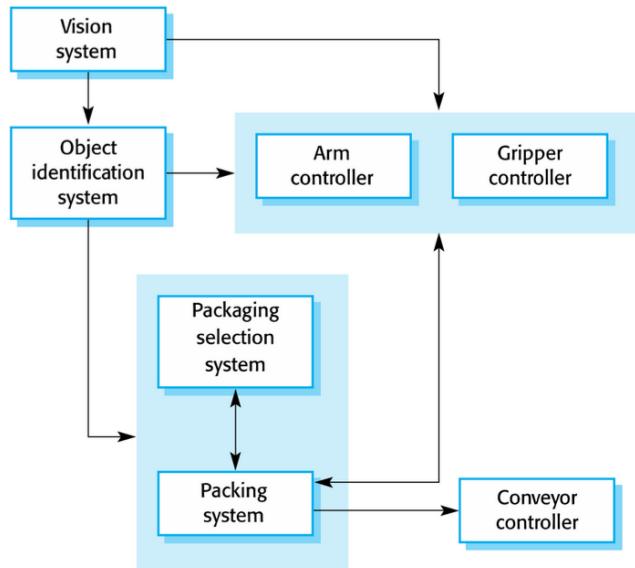


Figura 19: Esempio di architettura di un sistema di controllo di un robot

Ci occupiamo dell'architettura del singolo programma (come si scomponga in pezzi) e di un'azienda (come i programmi comunicano). Quali sono i punti di forza dell'architettura esplicitata?

- **Comunicazione con *stakeholders***
- **Analisi del sistema** - Se sono rispettati i requisiti non funzionali
- **Riutilizzo su larga scala**

Esistono diversi tipi di modelli, alcuni utili per mostrare a grandi linee il progetto, altri utili per la documentazione (e quindi molto più dettagliati).

Decisioni In un processo creativo ma delicato come la progettazione architetturale le scelte operate possono influire e modificare i requisiti non funzionali. All'interno dello stesso dominio si riutilizzano ad esempio gli stessi tipi di architettura, nascono allora i *pattern*.

Pattern Rappresentati da tavole e grafici, sono l'insieme delle buone pratiche per il riuso dei componenti.

6.1 Architectural Views

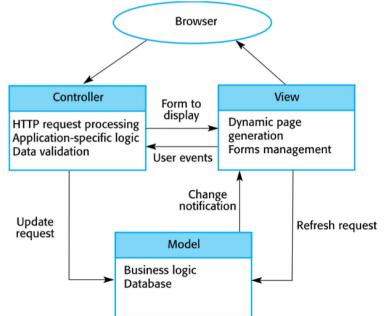
Ogni visualizzazione/prospettiva mette a fuoco un solo aspetto del sistema. Utilizziamo il modello 4+1.

6.1.1 4+1 View Model

- **Logical View** - Visualizza oggetti e classi
- **Process View** - Mostra come in run-time il sistema è composto da processi che interagiscono
- **Development View** - Mostra come è spezzettato il software per lo sviluppo
- **Physical View** - Mostra l'hardware
- Il +1 sono gli scenari/use case che legano queste prospettive

6.2 Pattern

6.2.1 Architettura Model-View-Controller



(a) Esempio di web application che usa l'architettura MVC

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

(b) Pattern dell'architettura MVC

6.2.2 Architettura a strati

Usato per modellare le interfacce dei sottosistemi. Il sistema è organizzato a strati, ogni strato offre una serie di servizi; se uno stato cambiano solo quelli adiacenti a lui.

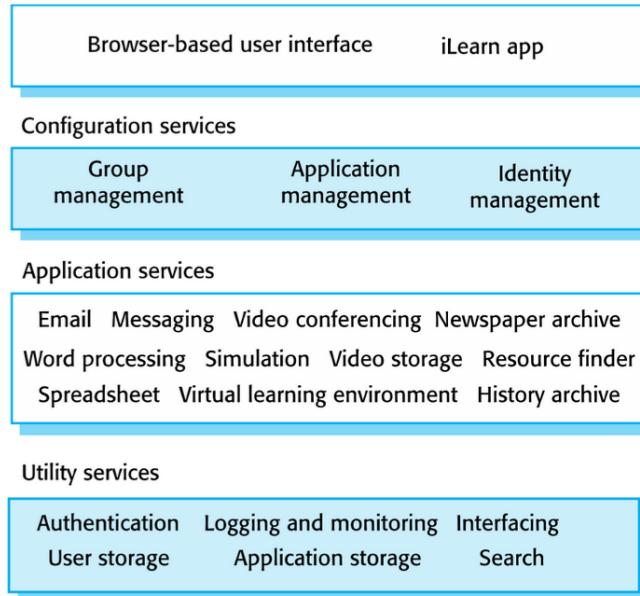


Figura 21: Esempio di architettura a strati in un sistema di insegnamento online

6.2.3 Architettura a repository

I sottosistemi scambiano dati in due modi:

- I dati condivisi sono tenuti in un database centralizzato e tutti i sottosistemi possono accedervi
- Ogni sotto-sistema ha il proprio database e lo passa quando viene richiesto

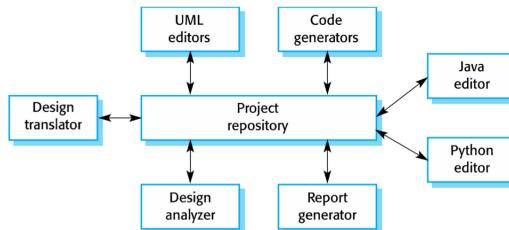


Figura 22: Esempio dell'architettura a repository per un IDE

6.2.4 Architettura client-server

Servono un modello di sistema distribuito per verificare dove e come avviene l'elaborazione, dei server specifici che provvedono dei servizi singoli, dei client che chiedono questi servizi, una rete che permette ai client l'accesso ai server.

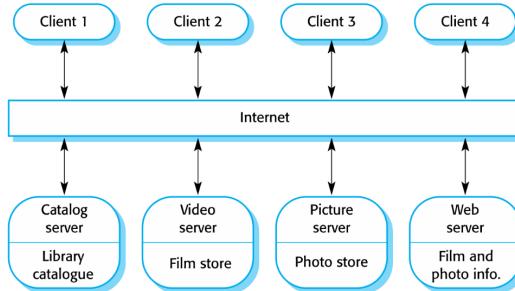


Figura 23: Esempio dell'architettura client server per una cineteca

6.2.5 Architettura pipe and filter

Usata per applicazioni di data processing e non per sistemi interattivi. Concezione sequenziale delle operazioni.

6.3 Tipi di applicazioni

Per elaborazione di:

- Dati
- Transazioni (richieste degli utenti e aggiornamento database)
- Eventi (esterni)
- Lingua (mediante interpreti e compilatori)

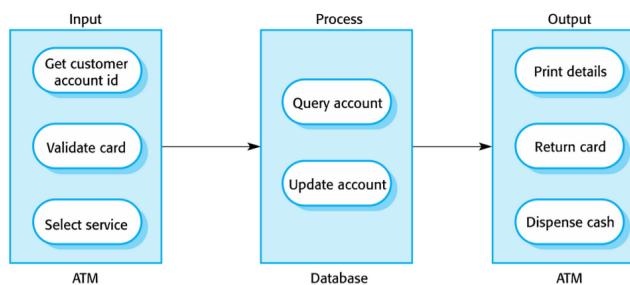


Figura 24: Architettura di un sistema ATM

Tra i più diffusi ci sono i sistemi informativi web-based come i siti di e-commerce con tutte le loro funzionalità interne e servizi offerti; e i sistemi di elaborazione linguistica che accettano una lingua naturale o artificiale per generarne un'altra rappresentazione mediante un interprete e un traduttore (controllo sintassi e semantica).

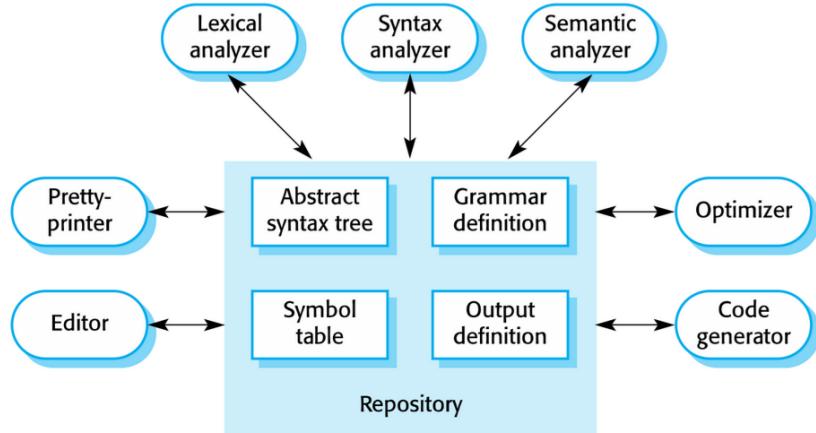


Figura 25: Architettura a repository per un sistema di elaborazione linguistica

7 Che fine ha fatto il capitolo 7?

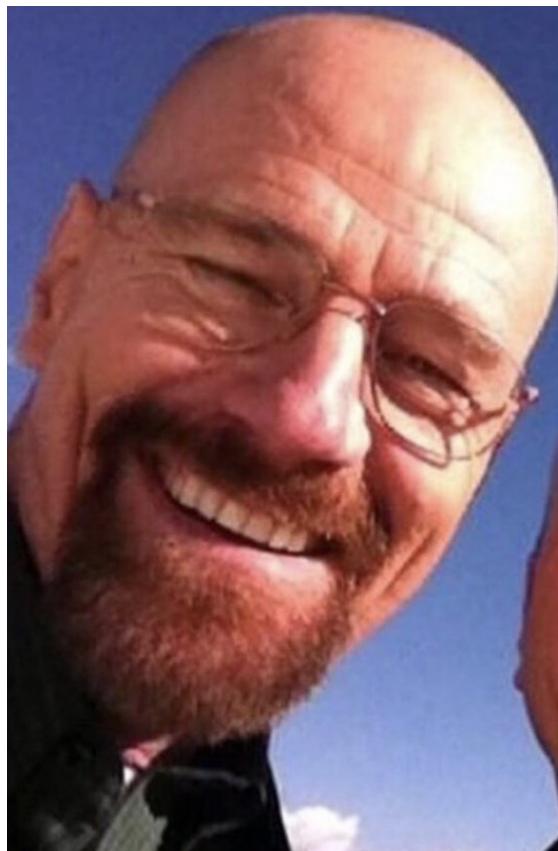


Figura 26: Non esiste un capitolo 7

8 Testing



Figura 27: Il capitolo 8 invece c'è e picchia fortissimo

La fase di testing fa parte di un processo molto più ampio di verifica e validazione; i test operati su un software sono molteplici e servono a capire eventuali comportamenti indesiderati ed errori. Le **ispezioni** vengono effettuate sulla rappresentazione statica del sistema mentre il **testing** sul comportamento dinamico.

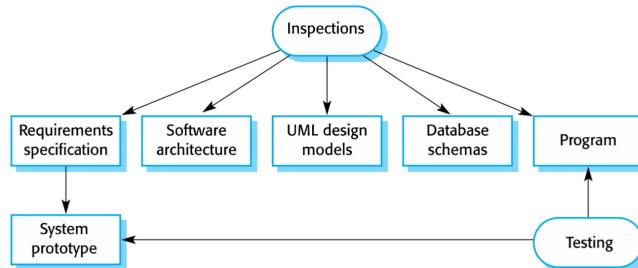


Figura 28: Schema di interazione tra ispezione e testing

Tipi di test

- **Test di validazione** - Per dimostrare allo sviluppatore e al cliente che il sistema rispetta i requisiti.
- **Test di difetti** - Per mostrare se ci sono risposte incorrecte o non conformi alla specifica.

Quanto fidarsi di V & V? Dipende dall'obiettivo del sistema.

- Software importante per un'azienda? Aumentare l'affidabilità.
- Software in forte domanda? Rilasciarlo anche se ha ancora dei difetti.

Definizioni

- **Validazione** - È il programma corretto? Rispetta i requisiti del cliente?
- **Verifica** - Il programma funziona? Rispetta la specifica?

8.1 Ispezioni

Avvengono prima dell'implementazione e come mostrato dalla Figura 28 possono riguardare i requisiti, il design e anche i test. L'ispezione aiuta a considerare anche se un prodotto è standard, se è portatile e se è manutenibile; inoltre, poiché si svolge su un sistema statico, mostra anche tutti quegli errori che in fase di testing potrebbero essere mascherati da altri bug. Bisogna però considerare che le ispezioni e il testing sono strumenti complementari e devono essere utilizzati entrambi, questo perché le ispezioni da sole non permettono di valutare le performance o di avere un feedback dal cliente.

8.2 Testing

Il testing di un software avviene in tre fasi:

1. **Development Testing** - Per evidenziare bug e difetti del codice.
2. **Release Testing** - Operato da un team separato da quello di sviluppo poco prima del rilascio al pubblico.
3. **User Testing** - Dove gli utenti provano il sistema nei loro ambienti.

8.2.1 Development Testing

Composto da tre attività degli sviluppatori che sono

- **Unit Testing** - Test delle funzionalità, delle classi e quindi dei metodi e degli oggetti presi singolarmente (con particolare attenzione agli oggetti coinvolti nell'ereditarietà).
- **Component Testing** - Si uniscono diverse unità e quindi si testano le interfacce.
- **System Testing** - Si uniscono i componenti per formare un sistema e quindi si testano le interazioni tra componenti.

8.2.1.1 Unit Testing - Quali sono i concetti e le procedure principali?

Object-Class Testing Dove si testano gli attributi di una classe e i metodi per controllare che sia sempre possibile accedervi.

Automated Testing Negli *automation frameworks* come JUnit si testano le unità con test basilari e personalizzabili. I componenti principali sono la struttura di setup dove si settano gli input, la struttura di chiamata dove si chiamano i metodi e quella di asserzione dove si comparano i risultati ottenuti con quelli attesi.

Partition Testing Si considerano i gruppi di input con caratteristiche simili (e che generano un output analogo) per ragionare secondo partizioni (e quindi velocizzare il processo di testing).

Guidelines Testing Basata sugli errori commessi solitamente dai programmatore. Si utilizzano input di lunghezza diversa o composti da caratteri insoliti per testare i filtri sugli input. L'obiettivo principale è generare tutti i possibili messaggi di errore.

8.2.1.2 Component Testing - Lo scopo è verificare che le interfacce tra le unità funzionino correttamente.

Interfacce per gestire

- **Parametri**
- **Memoria condivisa**
- **Procedure**
- **Messaggi**

Errori tipici

- **Utilizzo errato** (es. parametri nell'ordine scorretto in una chiamata di funzione)
- **Incomprensione** (es. assunzioni sul tipo tornato da una funzione)
- **Timing** (es. chiamante e funzione chiamata operano in tempo diverso e si finisce per accedere a dati scaduti)

Consigli per il testing:

- Passare alla funzione parametri il cui valore è all'estremo del range.
- Usare puntatori nulli.
- Forzare il crash del componente o stressarlo a livello di risorse e tempo.
- Per i sistemi che condividono memoria, variare il momento di accesso alle risorse.

8.2.1.3 System Testing - L'obiettivo è testare l'interazione tra i componenti, il comportamento di un sistema, e la corretta condivisione dei dati. Questa fase è collettiva (in alcune compagnie coinvolge un team specifico). Quali strumenti vengono utilizzati?

Use-case Testing Accompagnato da use-case e *sequence diagrams* per capire l'ordine delle interazioni.

Policies Poiché il sistema è spesso molto grande bisogna accordarsi sulla copertura dei test. Ad esempio, si potrebbe decidere di testare tutti i menù che l'utente può raggiungere, le combinazioni di funzioni all'interno dello stesso menù e i filtri degli input, decidendo di tralasciare altri aspetti o di rimandare il testing per le altre funzionalità.

8.2.1.4 Test-driven Development - Scrrittura del codice e testing sono interlacciati. È stato introdotto come metodo agile (parte dell'extreme programming) e successivamente utilizzato anche nello sviluppo plan-driven. Non si procede con il prossimo incremento finché quello attuale non ha passato tutti i test automatizzati.

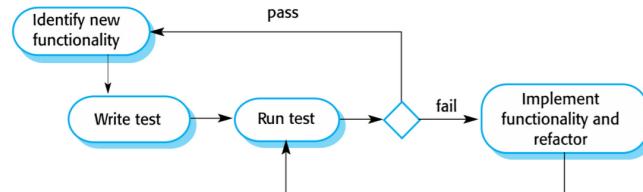


Figura 29: Schema del TDD

Quali sono i vantaggi del TDD?

- **Code coverage** - Ogni segmento di codice ha almeno un test associato.
- **Regression testing**.
- **Debugging semplificato**.
- **Documentazione del sistema** - I test stessi sono documenti.

Regression Testing - Testare il sistema per verificare che le nuove aggiunte non lo abbiano "rotto". Solitamente è un processo costoso, ma se viene automatizzato come con l'approccio che stiamo descrivendo è intuitivo ed economico.

8.2.2 Release Testing

Lo scopo è dimostrare che il prodotto è pronto, i test associati derivano esclusivamente dalla specifica. Quale è la differenza con il system testing? Il system testing si focalizza sulla ricerca dei bug (defect testing) mentre il release testing coinvolge un team esterno allo sviluppo e si focalizza sul rispetto dei requisiti richiesti (validation).

Requirements-based Testing - I test sono costruiti a partire dai requisiti; ad esempio un requisito per un ospedale potrebbe essere "Mostrare un messaggio di errore se si prova a prescrivere un farmaco ad un paziente che è allergico ad esso". Bisogna allora creare un utente senza allergie, uno con l'allergia al quale si vuole forzatamente prescrivere il medicinale, e allora specificare il perché della scelta, poi un paziente allergico a più medicinali, etc. Dobbiamo notare che questo presuppone allora anche la capacità di effettuare prescrizioni (emerge quindi che un requisito ne genera diversi).

Performance Testing - Overload volontario del sistema per testare le performance e l'affidabilità mediante input crescenti.

8.2.3 User Testing

Momento essenziale in cui i test vengono eseguiti sulle macchine dei futuri utilizzatori. Esistono diversi tipi di user testing:

- **Alpha Testing** - Gli utenti collaborano con il development team e svolgono i test presso il luogo di sviluppo del sistema.
- **Beta Testing** - Release usata per sperimentare e testare con gli sviluppatori.
- **Acceptance Testing** - Per sistemi personalizzati; i clienti comunicano al team se il sistema è pronto o no (l'ultimo momento prima del rilascio sul mercato).

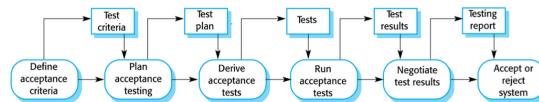


Figura 30: Schema delle fasi dell'acceptance test

9 Considerazioni finali

Questo documento non sostituisce le lezioni in aula, ma cerca di ordinare i discorsi affrontati per renderli più comprensibili anche grazie all'utilizzo di schemi, foto e riassunti.

Tutto ciò che è riportato in questo documento è una rielaborazione di appunti personali, slide del prof. Enrico Tronci e fonti miste.

Ogni sezione presente in questo documento è stata sottoposta allo sguardo critico di ChatGPT che ne ha corretto la sintassi il più possibile. Questa è una prima versione terminata la sera del 19 gennaio 2024.

Lorenzo