Progettazione e Validazione di Sistemi Software

Corso tenuto dal Professor Mariano Ceccato

Università degli Studi di Verona

Alessio Gjergji

Indice

1 Processi software **4**
- 1.1 Introduzione . . . 4
  - 1.1.1 Descrizione del processo di sviluppo software . . . 5
- 1.2 Modelli del processo di sviluppo software . . . 6
  - 1.2.1 Code and fix . . . 6
  - 1.2.2 Waterfall . . . 6
  - 1.2.3 Sviluppo incrementale . . . 9
  - 1.2.4 Integrazione e configurazione . . . 10
- 1.3 Attività di sviluppo software . . . 12
  - 1.3.1 Specifica . . . 12
  - 1.3.2 Design e implementazione . . . 13
  - 1.3.3 Verifica e validazione . . . 14
  - 1.3.4 Evoluzione del software . . . 15
- 1.4 Gestione dei cambiamenti . . . 15
  - 1.4.1 Prototipazione . . . 16
  - 1.4.2 Vantaggi e Svantaggi dell'Approccio Incrementale . . . 16

2 Agile **17**
- 2.1 Agile . . . 17
  - 2.1.1 Approccio Agile . . . 17
  - 2.1.2 Manifesto Agile . . . 18
  - 2.1.3 Implementazione dell'Approccio Agile . . . 18
  - 2.1.4 Applicabilità dell'Approccio Agile . . . 18
- 2.2 Extreme Programming (`XP`) . . . 19
- 2.3 User Stories . . . 19
- 2.4 Test-Driven Development (`TDD`) . . . 19
  - 2.4.1 Refactoring . . . 20
  - 2.4.2 Pair Programming . . . 20
- 2.5 CI/CD nel Sviluppo Software Moderno . . . 20
  - 2.5.1 Continuous Integration (CI) . . . 20
  - 2.5.2 Continuous Delivery (CD) . . . 21
  - 2.5.3 Continuous Deployment . . . 21
- 2.6 Project Management Agile . . . 21

2.6.1 Scrum . . . 21
2.6.2 Terminologia . . . 21
2.6.3 Benefici . . . 22
2.7 Scrum Distribuito . . . 22
2.7.1 Scalabilità . . . 22
2.7.2 Problemi e Resistenze . . . 22

3 Ingegneria dei requisiti 23
3.1 Requisiti . . . 23
3.1.1 Requisiti utente e di sistema . . . 23
3.1.2 Requisiti funzionali e non funzionali . . . 24
3.1.3 Verifica dei requisiti non funzionali . . . 25
3.2 Processi di ingegneria dei requisiti . . . 25
3.2.1 Elicitazione dei requisiti . . . 26
3.2.2 Specifica dei requisiti . . . 28
3.2.3 Validazione dei requisiti . . . 29
3.2.4 Cambiamento dei requisiti . . . 29

4 Design architetturale 31
4.1 Design architetturale . . . 31
4.2 Pattern architetturale . . . 32
4.2.1 Model View Controller . . . 32
4.2.2 Architettura stratificata . . . 32
4.2.3 Architettura Repository . . . 33
4.2.4 Architettura Client-Server . . . 33
4.2.5 Architettura Peer-to-Peer . . . 34
4.2.6 Architettura Pipe and Filter . . . 34
4.3 Architetture eterogenee . . . 35

5 Testing 36
5.1 Introduzione . . . 36
5.1.1 Program testing . . . 36
5.1.2 Confidenza . . . 37
5.1.3 Code review . . . 37
5.1.4 Test driven development . . . 37
5.2 Testing . . . 38
5.2.1 Development testing . . . 38

6 Refactoring 43
6.1 Introduzione . . . 43
6.1.1 Code smell . . . 44
6.1.2 Classi di smell . . . 44
6.2 Software clone . . . 45
6.3 Refactoring . . . 45
6.3.1 Applicazione del refactoring . . . 45

6.3.2 Extract class 47
6.3.3 Replace inheritance with delegation 47
6.3.4 Replace conditional with polymorphism 47
6.3.5 Separate domain from presentation 48

Capitolo 1

Processi software

1.1 Introduzione

Il processo di sviluppo del software è un processo di ingegneristico, perciò è necessario che sia disciplinato. Lo sviluppo software non richiede solamente lo sviluppo dal punto di vista del codice, ma deve inoltre essere inserito in un ciclo strutturato e controllato, che permetta di produrre un prodotto software di qualità.

Anche il processo di sviluppo software va progettato, e deve essere soggetto a verifica e validazione. Dietro al processo di sviluppo software c'è quindi un processo di ingegneria del software, che mira a mantenerne la qualità nel tempo.

Se ci dotiamo di un processo di sviluppo software, possiamo ottenere dei vantaggi che rendono il processo software:

- **Ordinato**: si sa cosa fare e quando farlo;
- **Controllato**: il processo è controllato e misurabile, fornendoci quindi consapevolezza in merito al processo;
- **Ripetibile**: se vengono riscontrati dei problemi nel processo, sappiamo dove intervenire per risolverli, anche grazie alla consapevolezza acquisita dai progetti passati;

Il primo obiettivo è quello di migliorare la produttività degli sviluppatori, mettendo gli sviluppatori nella condizione di essere il più produttivi possibili. Il secondo obiettivo è quello di essere in grado di migliorare la qualità del prodotto software, in modo da ridurre i costi di manutenzione e di correzione degli errori (*banalmente dei requisiti scritti molto male possono portare delle ambiguità nel prodotto*).

> Solitamente un processo di qualità è un processo che porta ad un prodotto di qualità.

Quindi il costo pagato per avere un processo di qualità è ammortizzato dal fatto che il prodotto finale sarà di qualità. Inoltre, il costo del software è principalmente legato alla manutenzione,

quindi è importante che il processo di sviluppo sia di qualità, altrimenti si rischia di accumulare un debito tecnico che può essere molto costoso.

Processo di sviluppo software

Il processo di sviluppo software è un insieme di attività che devono essere concluse per lo sviluppo di un sistema software.

I processi di sviluppo software possono essere di diversi tipi, ma molti avranno dei tratti in comune. Le attività comuni che compongono il processo di sviluppo software sono:

- **Specifica**: ovvero la definizione di ciò che il sistema software dovrà avere;
- **Design e implementazione**: ovvero la progettazione e l'implementazione di ciò che è stato definito nella fase di specifica;
- **Validazione**: ovvero la verifica che il prodotto software sia conforme o allineato con le aspettative del cliente;
- **Evoluzione**: ovvero la manutenzione del software, che può essere in seguito a cambiamenti dei requisiti, aggiunta di funzionalità, a correzioni di errori o cambiamenti di normative.

1.1.1 Descrizione del processo di sviluppo software

Il processo software, solitamente, si definisce in termini di **attività** che vengono condotte, ovvero specifiche, design di interfacce grafiche, ma il processo include inoltre:

- **Prodotti**: che sono i risultati di ogni attività;
- **Ruoli**: ovvero le responsabilità che vengono assegnate alle persone che lavorano al progetto;
- **Pre/post-condizioni**: ovvero le condizioni che devono essere soddisfatte prima e dopo l'esecuzione di un'attività;

Tipicamente i processi software si dividono in due grandi categorie:

- **Plan-driven**: ovvero i processi che si basano su una pianificazione anticipata delle attività e dei prodotti, e che quindi sono più adatti per progetti di grandi dimensioni;
- **Agile**: ovvero i processi che si basano su una pianificazione meno dettagliata, dove le attività e i processi sono più adattabili, anche in linea con le esigenze del cliente (*composto da una pianificazione più generica nel lungo periodo e più dettagliata nel breve periodo*).

Nei processi plan-driven, la pianificazione è molto dettagliata perché si ha la consapevolezza del prodotto che si vuole sviluppare, mentre nei processi agili la pianificazione è più generica perché si ha una visione meno chiara del prodotto finale, vista la dinamicità del prodotto.

Nella realtà, molti processi software sono ibridi, ovvero sono una combinazione degli elementi dei processi plan-driven e agili.
Bisogna quindi ricordare che non ci sono processi software sbagliati.

1.2 Modelli del processo di sviluppo software

1.2.1 Code and fix

Tipicamente, quando si sviluppa un software, si tende a seguire un approccio **code and fix**, ovvero si scrive del codice e poi si correggono gli errori che si sono commessi. Questo approccio è molto semplice, ma è anche molto rischioso, perché non si ha una visione chiara del prodotto che si vuole sviluppare, e quindi si rischia di non soddisfare le aspettative del cliente.

Il codice viene implementato per tentativi, non è presente una fase di analisi dettagliata. Tipicamente questo approccio è efficace per progetti di piccole dimensioni, dove tipicamente il cliente è lo sviluppatore stesso. In progetti di grandi dimensioni o in progetti in cui è presente un la necessità di lavorare in team, questo approccio non è efficace. Di fatto questo **non è un processo di sviluppo software**, perché non è presente una pianificazione, e non è quindi possibile scalare il processo.

1.2.2 Waterfall

Il processo è nato in analogia con il processo di sviluppo dell'ingegneria meccanica, dove è necessario avere una visione chiara del prodotto che si vuole sviluppare, e dove è necessario avere una pianificazione dettagliata delle attività e dei prodotti.

Si tratta di un processo plan-driven, che è stato il primo processo di sviluppo software ad essere definito. Il processo è diviso in fasi, dove ogni fase produce dei prodotti che vengono utilizzati nella fase successiva, per questo motivo non è consentito passare alla fase successiva se non è stata completata la fase precedente.

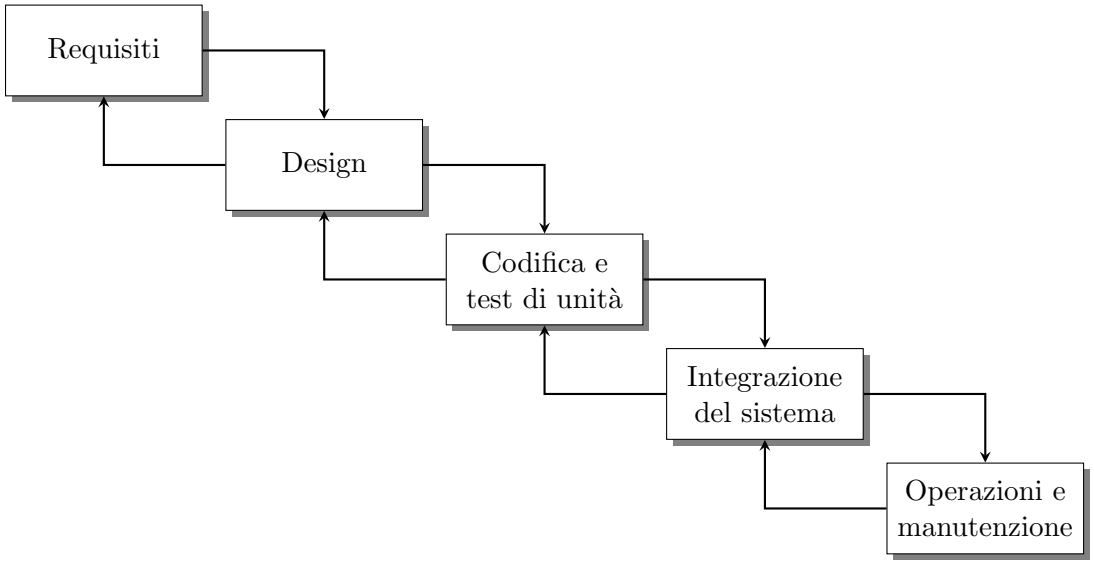


Figura 1.2.1: Processo waterfall

Definizione e analisi dei requisiti

Il processo inizia con la definizione e analisi dei requisiti. In questa fase, vengono determinati i requisiti del sistema software da sviluppare e gli obiettivi da raggiungere, anche in termini di funzionalità del sistema. Questi obiettivi sono stabiliti in collaborazione con gli stakeholders attraverso interviste e discussioni. Successivamente, si redige un documento dei requisiti, che elenca i requisiti specifici del software in sviluppo. Questo documento ha valore contrattuale: viene condiviso con il cliente e utilizzato per stabilire il contratto tra cliente e fornitore. La verifica di questo documento è essenziale per assicurare la coerenza dei dati e prevenire ambiguità o inconsistenze.

Progettazione del sistema

Una volta che i requisiti sono stati definiti, si passa alla progettazione del sistema. In questa fase, i requisiti vengono allocati a componenti hardware e software, e viene definita l'architettura del sistema. Questa fase è importante perché permette di identificare i componenti del sistema e le loro interazioni, e permette di definire una struttura di base del sistema. Ogni componente viene poi progettato nel dettaglio in modo da realizzare una funzionalità specifica del sistema.

Anche in questo caso, per alcuni sistemi critici, questa fase è soggetta a verifica formale e una volta approvata diventa permanente.

implementazione e test di unità

Una volta che la progettazione del sistema è stata completata, si passa all'implementazione del sistema. In questa fase, i componenti del sistema vengono implementati in un linguaggio

di programmazione, realizzando quindi il codice che realizza concretamente il design che è stato elaborato nella fase precedente.

In questa fase, è necessario anche testare i componenti del sistema, per assicurarsi che il codice implementato soddisfi i requisiti e che non ci siano errori. Questi test sono chiamati **test di unità**, e sono test che vengono eseguiti su ogni singolo componente del sistema.

Integrazione del sistema

Una volta che tutti i componenti del sistema sono stati implementati e testati, si passa alla fase di integrazione del sistema. In questa fase, i componenti del sistema vengono integrati per formare il sistema completo. Quando il sistema è stato integrato, è possibile mandare in esecuzione scenari che testano il sistema nel suo complesso, e che verificano che il sistema soddisfi i requisiti. Questi test sono chiamati **test di sistema**.

Quando il sistema è stato integrato, è possibile consegnare il sistema al cliente.

Operazioni e manutenzione

Quando il sistema è stato consegnato al cliente, il sistema viene utilizzato dal cliente. In questa fase, il sistema viene utilizzato per il suo scopo, e vengono identificati nuovi requisiti e nuove funzionalità che il sistema deve soddisfare. Questi nuovi requisiti vengono analizzati e aggiunti al sistema, e il sistema viene modificato per soddisfare i nuovi requisiti che possono essere requisiti legati alla natura del mercato o legati alla normativa. Questa fase è chiamata **manutenzione del sistema**. In questa fase potrebbero essere necessarie anche modifiche al sistema per correggere errori che sono stati scoperti durante l'utilizzo del sistema.

Fasi del processo waterfall

La caratteristica di questo processo di sviluppo software è che l'output di una fase è considerata *congelata* prima di passare alla fase successiva. Questa linea di principio è molto efficace per lo sviluppo di progetti hardware perché in quei casi è necessario avere una visione chiara del prodotto che si vuole sviluppare, poiché il costo relativo al cambiamento di un prodotto hardware è molto alto. Nel caso di progetti in cui è necessario avere un feedback continuo con il cliente, questo processo non è adatto, poiché il cliente non può vedere il prodotto fino alla fine del processo.

Inoltre prevede che i requisiti vengano congelati prima di iniziare la fase di progettazione, di fatto bloccando la possibilità di modificare i requisiti durante lo sviluppo del sistema. Il cambio di requisiti è possibile, ma implicherebbe un ritardo nello sviluppo del sistema, poiché bisognerebbe tornare alla fase di analisi dei requisiti e ripetere tutte le fasi successive.

Nei sistemi molto grandi (*anche gestiti da più compagnie*), sistemi critici, o sistemi che richiedono software-hardware integrati, questo processo è molto efficace, poiché permette di avere una visione chiara del sistema che si vuole sviluppare, e permette di avere un controllo preciso sulle fasi di sviluppo del sistema.

Vantaggi e svantaggi

I vantaggi del processo di sviluppo includono una forte enfasi sull'analisi dei requisiti e sulla progettazione del sistema. Questo approccio ritarda l'implementazione finché non viene effettuata un'analisi accurata delle esigenze degli utenti, rendendolo particolarmente adatto quando i requisiti sono chiari e stabili. Inoltre, introduce una pianificazione e uno sviluppo disciplinato, ideali per grandi progetti di ingegneria dei sistemi sviluppati in diversi siti. In tali contesti, la natura guidata dalla pianificazione del modello a cascata facilita il coordinamento del lavoro.

Tuttavia, esistono degli svantaggi significativi. Il processo richiede che ogni fase sia completata prima di procedere alla successiva, il che può creare inefficienze. Inoltre, è difficile accomodare i cambiamenti una volta che il processo è in corso, specialmente se emergono nuovi requisiti da parte del cliente. Questo lo rende adatto solo in circostanze dove i requisiti sono ben compresi e si prevede che i cambiamenti saranno limitati durante il processo di progettazione. Infine, pochi sistemi aziendali hanno requisiti stabili, il che limita l'applicabilità di questo processo in ambienti dinamici e in rapido cambiamento.

1.2.3 Sviluppo incrementale

Il processo incrementale è un processo plan-driven o agile, che si basa su una pianificazione anticipata delle attività e dei prodotti, ma che prevede che il prodotto software venga sviluppato in maniera incrementale, ovvero in più fasi, dove ogni fase produce un incremento del prodotto software. Ogni incremento è un prodotto software che può essere rilasciato al cliente, e che può essere utilizzato dal cliente.

Nel modello di sviluppo incrementale, l'implementazione iniziale del software viene esposta agli utenti fin dalle prime fasi. Questo approccio consente di evolvere il software attraverso diverse versioni, finché non si ottiene il sistema richiesto. Questo modello è generalmente preferito rispetto al modello a cascata, specialmente quando i requisiti si prevede possano evolvere nel corso dello sviluppo. A differenza del modello a cascata, le attività di specifica, sviluppo e validazione sono intrecciate e non separate. Ciò consente un feedback rapido e continuo tra le attività, promuovendo un affinamento progressivo del prodotto attraverso iterazioni successive.

Vantaggi e svantaggi

Il modello di sviluppo incrementale offre il vantaggio di ridurre i costi nel rispondere ai cambiamenti dei requisiti del cliente. A differenza del modello a cascata, richiede meno analisi e documentazione da rifare, facilitando l'ottenimento del feedback del cliente sul lavoro di sviluppo compiuto fino a quel momento. I clienti possono commentare le dimostrazioni e monitorare i progressi, consentendo la consegna e l'implementazione rapida delle parti più utili del software. In questo modo, i clienti sono in grado di utilizzare il software, seppur parziale, più precocemente rispetto a un processo a cascata.

Tuttavia, il processo presenta anche delle sfide. La visibilità del processo può non essere evidente, e i manager necessitano di consegne regolari per misurare i progressi. Se i sistemi

vengono sviluppati rapidamente, non è economicamente conveniente produrre documentazione che rifletta ogni versione del sistema. La struttura del sistema tende a degradarsi man mano che vengono aggiunti nuovi incrementi. A meno che non si investa tempo e denaro in attività di *refactoring* per migliorare il software, i cambiamenti regolari tendono a corrompere la sua struttura. Di conseguenza, incorporare ulteriori cambiamenti al software diventa sempre più difficile e costoso.

1.2.4 Integrazione e configurazione

Nel processo di sviluppo software di integrazione e configurazione, il sistema è assemblato da componenti sviluppati separatamente. Tale processo può essere sia plan-driven che agile.

Lo sviluppo di software basato sul riutilizzo è un approccio in cui i sistemi sono integrati a partire da componenti o sistemi già esistenti. Questo metodo sfrutta in particolare i componenti Commercial-off-the-shelf (`COTS`), che sono sistemi disponibili commercialmente e possono essere riutilizzati così come sono o configurati per adattarsi alle esigenze e ai requisiti specifici degli utenti. Gli elementi riutilizzati possono essere configurati per adattare il loro comportamento e le loro funzionalità, rendendo il riutilizzo l'approccio standard per la costruzione di molti tipi di sistemi aziendali.

Commonly used components - `COTS`

I sistemi autonomi (`COTS`), generalmente dotati di numerose funzionalità, vengono adattati o configurati per l'uso in un ambiente specifico. Si trovano anche collezioni di oggetti sviluppati come pacchetti da integrare con un framework di componenti, come ad esempio `Java Spring`. In aggiunta, vi sono i servizi web sviluppati secondo gli standard di servizio e che sono disponibili per l'invocazione remota. Questa pluralità di opzioni disponibili per il riutilizzo nel software fornisce una flessibilità senza precedenti nella progettazione e nello sviluppo di nuovi sistemi, consentendo alle aziende di risparmiare tempo e risorse.

Specifica dei requisiti

Nella fase iniziale di concezione di un sistema, vengono proposti i requisiti iniziali. Questi non necessitano di essere elaborati in dettaglio ma dovrebbero includere brevi descrizioni delle necessità essenziali e delle caratteristiche desiderabili del sistema. È fondamentale che questi requisiti riflettano una comprensione di base delle aspettative del cliente e della funzionalità chiave che il sistema dovrà avere, senza tuttavia immergersi in specifiche tecniche complesse. Questo approccio permette di stabilire una direzione generale per il progetto, lasciando spazio per l'affinamento progressivo dei requisiti man mano che si procede con lo sviluppo.

Selezione di Componenti e Sistemi per il Riutilizzo

Nel processo di sviluppo software orientato al riutilizzo, è cruciale effettuare una ricerca accurata di componenti e sistemi che forniscano la funzionalità richiesta. Una volta identificati potenziali candidati, questi componenti e sistemi vengono valutati attentamente per determinare se soddisfano i requisiti essenziali e se sono generalmente adatti al riutilizzo nel sistema in questione. Questa valutazione implica un'analisi approfondita delle caratteristiche tecniche,

della compatibilità con l'architettura esistente e della facilità di integrazione. La selezione accurata è fondamentale per garantire che il riutilizzo dei componenti non solo acceleri lo sviluppo ma contribuisca anche a una solida fondazione per il sistema finale.

Affinamento dei Requisiti e Componenti Riutilizzabili

Nel corso dello sviluppo software, i requisiti sono affinati utilizzando le informazioni sui componenti riutilizzabili scoperti. Questo processo di affinamento può comportare la modifica dei requisiti per riflettere i componenti disponibili, portando a una ridefinizione della specifica del sistema. È un'esercizio di bilanciamento tra le esigenze iniziali e le soluzioni pratiche offerte dai componenti esistenti. Laddove le modifiche ai requisiti si rivelino impossibili o impraticabili, l'attività di analisi dei componenti può essere ripetuta nella ricerca di alternative più adatte. Questa iterazione è vitale per assicurare che il sistema finale non solo soddisfi le necessità degli utenti ma sia anche costruito in modo efficiente e sostenibile.

Adozione di Sistemi Applicativi Preconfezionati

Nell'eventualità che sia disponibile un sistema applicativo preconfezionato che soddisfi i requisiti del progetto, è possibile procedere alla sua configurazione per l'uso al fine di creare il nuovo sistema. Questa operazione di configurazione si rivela spesso un'alternativa efficace allo sviluppo da zero, permettendo un notevole risparmio di tempo e risorse. La scelta di un'applicazione commerciale pronta all'uso può quindi trasformarsi in un vantaggio competitivo, specialmente quando si allinea con le necessità aziendali e si integra armoniosamente nell'infrastruttura `IT` esistente.

Sviluppo del Sistema in Assenza di Soluzioni Preconfezionate

Nel caso in cui non sia disponibile un sistema preconfezionato che soddisfi i requisiti specifici del progetto, è necessario intraprendere un approccio alternativo. In questa situazione, i componenti riutilizzabili individuali possono essere modificati per adattarli meglio alle esigenze del sistema. Inoltre, può essere necessario lo sviluppo di nuovi componenti per colmare le lacune funzionali o tecnologiche. Questi componenti, sia modificati che nuovi, sono poi integrati per creare il sistema finale. Questo processo richiede un'attenta pianificazione e coordinamento per assicurare che tutte le parti del sistema lavorino insieme in modo armonioso, risultando in una soluzione completa e funzionale.

Vantaggi e svantaggi

Lo sviluppo di software basato sul riutilizzo di componenti offre diversi vantaggi significativi. Tra questi, il più evidente è il **basso costo**, poiché meno software viene sviluppato da zero. Questo approccio riduce anche il **rischio** associato allo sviluppo di nuove soluzioni, rendendolo particolarmente attraente per progetti con budget e tempi limitati. Inoltre, permette una **consegna rapida** di un sistema funzionante, essendo gran parte del lavoro focalizzato sull'integrazione di componenti esistenti piuttosto che sulla creazione di nuovi.

Tuttavia, ci sono anche degli svantaggi. Uno dei principali è la potenziale **bassa qualità** dei componenti riutilizzati, che potrebbe non essere all'altezza degli standard richiesti. Inoltre,

spesso si verifica un **compromesso sui requisiti**, con il rischio di ottenere un sistema finale che non soddisfa completamente gli utenti. Un altro punto critico è la **perdita di controllo sull'evoluzione** dei componenti riutilizzati. Questo può portare a problemi nel lungo termine, specialmente se i componenti non sono più supportati o aggiornati dai loro fornitori originali.

1.3 Attività di sviluppo software

Le attività comuni che il processo di sviluppo software deve affrontare sono:

- **Specifica**: definizione dei requisiti del sistema;
- **Design e Implementazione**: definizione dell'organizzazione del sistema e la sua realizzazione;
- **Validazione**: verifica che il sistema sia conforme alle specifiche;
- **Evoluzione**: modifica del sistema in risposta al cambiamento dei bisogni degli utenti.

1.3.1 Specifica

La specifica delle funzionalità, nota anche come **ingegneria dei requisiti**, è il processo che mira a capire e a definire quali servizi sono richiesti e identificati per lo sviluppo del sistema. Questo processo è uno dei processi più critici del ciclo di vita del software, poiché un errore in questa fase comporterà sicuramente dei problemi nel sistema finale, potendone causare il fallimento.

Talvolta, è opportuno condurre un'attività preliminare di **studio di fattibilità** per verificare se il sistema proposto è fattibile dal punto di vista tecnico, economico e organizzativo. Di fatto si tratta di una versione in miniatura del progetto che permette di valutare se avviare o meno il progetto completo.

Tipicamente questa fare produce un **documento dei requisiti** che definisce formalmente i requisiti del sistema. Questo documento deve essere comprensibile sia per i clienti che per gli sviluppatori.

Produrre un documento dei requisiti

1. **Elicitazione dei requisiti**: per elicitazione, di fatto, si vuole intendere l'attività di trasferimento delle funzionalità richieste dal cliente verso gli ingegneri dei requisiti e/o del software. Consiste quindi in una descrizione del sistema che può essere fatta in vari modi (*interviste, questionari, osservazione di sistemi software precedenti, studio di documentazione, ecc...*). Tale fase potrebbe richiedere anche la realizzazione di prototipi per aiutare il cliente a capire meglio le sue esigenze.
2. **Specifica dei requisiti**: i requisiti, una volta raccolti, devono essere specificati, ovvero devono essere descritti in modo preciso e non ambiguo. La rappresentazione dei requisiti può essere fatta in vari modi (*scrittura di documenti, modelli grafici, modelli matematici, ecc...*), l'importante è che possa permettere il dialogo tra i clienti e gli sviluppatori.

I requisiti possono essere classificati in due categorie:

- **User Requirements**: descrivono le funzionalità del sistema dal punto di vista degli utenti, che sono interessanti per il cliente, poiché descrivono ciò che il cliente vuole dal sistema;
- **System Requirements**: descrivono le funzionalità del sistema dal punto di vista degli sviluppatori, che sono interessanti per gli sviluppatori, poiché descrivono ciò che il sistema deve fare.

3. **Validazione dei requisiti**: Una volta che i requisiti sono stati specificati, è necessario verificarli per assicurarsi che siano corretti, completi e realistici, catturando quindi tutte le funzionalità richieste dal cliente e che non ci siano ambiguità. L'attività prevede che vi sia anche un controllo di errori e omissioni. In caso di errori il documento dei requisiti deve essere corretto e aggiornato.

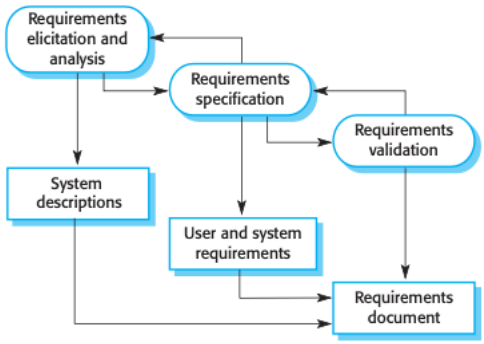


Figura 1.3.1: Processo di produzione di un documento dei requisiti

Ognuna di queste fasi può opzionalmente avere un *deliverable*, ovvero un documento che viene prodotto e utilizzo come input per la fase successiva. A seconda dell'approccio di sviluppo ci saranno step più o meno formali.

1.3.2 Design e implementazione

Questa fase consiste nel definire l'architettura del sistema e di fatto implementarlo mediante la scrittura del codice sorgente che rappresenta l'implementazione del sistema.

Per **design** si intende la definizione delle *strutture dati* che dovranno essere implementare, i *modelli* e le strutture utilizzate dal sistema, le *interfacce* che verranno utilizzate per comunicare con il sistema, gli *algoritmi* che verranno utilizzati per implementare le funzionalità del sistema e le *procedure di comunicazione* tra le varie componenti del sistema.

L'**implementazione** consiste nella scrittura del codice eseguibile che implementa le funzionalità del sistema.

Tipicamente il design del sistema è organizzato in quattro attività:

1. **Design architetturale**: ad alto livello si individuano le macro aree in cui il sistema verrà articolato. Definendo quindi i principali componenti del sistema e le loro interazioni.
2. **Design del database**: si definiscono le strutture dati utilizzate ad alto livello e come verranno rappresentate nel database.
3. **Design dell'interfacce**: si definiscono le interfacce che verranno utilizzate per comunicare con il sistema, poiché è importante definire come verranno scambiate le informazioni tra il sistema e l'ambiente esterno. Nel concreto, è importante definire l'interfaccia perché permette gli sviluppatori di essere guidati nella scrittura del codice.
4. **Selezione e design dei componenti**: una volta definite le interfacce e le loro funzionalità è possibile procedere per la ricerca di componenti disponibili sul mercato che soddisfano le caratteristiche di cui si ha bisogno. Nel caso in cui non si trovino componenti adatti, è necessario progettare i componenti che verranno utilizzati. I componenti non riutilizzabili vengono progettati e implementati.

La progettazione ha una struttura gerarchica, ovvero si parte da un livello più alto e si scende di livello fino ad arrivare al livello più basso. Questo è importante soprattutto per progetti software di grandi dimensioni, poiché permette di suddividere il lavoro in modo più semplice e di gestire meglio le risorse.

1.3.3 Verifica e validazione

Una volta che il sistema è stato implementato e anche durante lo sviluppo del software, è necessario verificare che il sistema sia allineato ai requisiti fissati. Per farlo è possibile utilizzare due approcci:

- **Verifica**: consiste nell'ispezionare il codice sorgente, ragionando sul codice scritto, per capire se il codice è corretto e se soddisfa i requisiti.
- **Testing**: consiste nell'eseguire il codice sorgente, per capire se il comportamento osservato è corretto e se soddisfa i requisiti (*anche mediante dati simulati*).

Anche il testing si divide in sottofasi:

1. **Testing del componente**: si testa il componente singolarmente, per capire se il componente se la responsabilità del componente è corretta.
2. **Testing del sistema**: si testa il sistema nel suo complesso, per capire se il sistema funziona correttamente. Si verifica che l'integrazione tra i vari componenti sia corretta.
3. **Accettazione del testing *customer testing***: si testa il sistema in un ambiente simile a quello in cui verrà utilizzato, anche con dati realistici, per capire se il sistema soddisfa i requisiti fissati.

1.3.4 Evoluzione del software

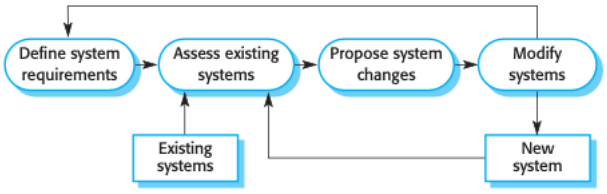


Figura 1.3.2: Processo di evoluzione del software

Quando si ritiene necessario apportare una modifica, può essere seguito uno schema di evoluzione del software, che prevede le seguenti fasi:

1. **Definizione dei requisiti di sistema**: si definisce il cambiamento che si vuole apportare al sistema, ovvero si definisce il requisito che si vuole aggiungere o modificare.
2. **Analisi dell'impatto**: si analizza l'impatto che il cambiamento avrà sul sistema, ovvero si analizza quali componenti del sistema verranno modificati e quali no. Per impatto si intende anche il costo che il cambiamento avrà sul sistema, e come tale dovrà essere negoziato con il cliente.
3. **Progettazione del cambiamento**: si ritorna dal cliente per unificare se il cambiamento è accettabile e se il costo è accettabile.
4. **Implementazione del cambiamento**: si implementa il cambiamento.

1.4 Gestione dei cambiamenti

Nel contesto del processo di sviluppo del software, è essenziale gestire efficacemente i cambiamenti per garantire una transizione fluida e un prodotto finale di alta qualità. Tre aspetti chiave della gestione dei cambiamenti sono:

1. **Cambiamenti che causano rilavoro e introduzione di nuove funzionalità:** Durante lo sviluppo, possono emergere richieste di modifiche o l'aggiunta di nuove funzionalità. È importante essere pronti ad affrontare queste modifiche, ma bisogna anche considerare l'impatto sul lavoro già svolto.
2. **Anticipazione dei cambiamenti:** Un approccio preventivo consiste nell'anticipare possibili modifiche prima che diventino necessari rilavori significativi. Ad esempio, condividere un prototipo con gli utenti finali per discutere e finalizzare i loro requisiti prima di impegnare notevoli risorse nell'implementazione definitiva.
3. **Tolleranza ai cambiamenti:** Il processo di sviluppo dovrebbe essere progettato in modo da consentire l'incorporazione di cambiamenti a un costo relativamente basso. Questo significa che le modifiche non dovrebbero comportare una revisione completa del lavoro svolto fino a quel momento.

1.4.1 Prototipazione

È anche importante menzionare il ruolo del prototipo nell'ambito dello sviluppo del software:

Prototipo

Una versione iniziale di un sistema software utilizzata per dimostrare concetti, esplorare opzioni di progettazione e approfondire la comprensione del problema e delle sue soluzioni possibili.

Lo sviluppo rapido e iterativo del prototipo aiuta a controllare i costi. Durante l'ingegneria dei requisiti, il prototipo aiuta nell'elicitation e nella validazione dei requisiti del sistema. Durante la progettazione del sistema, può essere utilizzato per esplorare soluzioni nell'ambito dell'interfaccia utente.

1.4.2 Vantaggi e Svantaggi dell'Approccio Incrementale

Lo sviluppo incrementale presenta vantaggi significativi. Permette di consegnare un prodotto funzionante al cliente in tempi brevi, consentendo così di ottenere un feedback tempestivo. Questo contribuisce a ridurre il rischio di fallimento del progetto, in quanto il cliente può verificare se il prodotto soddisfa i requisiti stabiliti. Inoltre, l'approccio incrementale facilita l'implementazione di modifiche rapide ed economiche al prodotto.

Tuttavia, vi sono anche svantaggi associati a questo approccio. Il prodotto finale potrebbe mancare di una struttura complessiva ben definita, poiché è stato sviluppato in modo incrementale, senza una pianificazione completa. Inoltre, il processo incrementale potrebbe comportare una minore copertura di test, in quanto lo sviluppo può concentrarsi sugli incrementi senza una visione completa del sistema.

Capitolo 2

Agile

2.1 Agile

2.1.1 Approccio Agile

Costruire sistemi safety-critical, governativi o gestire molti team distribuiti sul territorio richiede una coordinazione efficace. Verso la fine degli anni '90, il contesto di sviluppo software è cambiato radicalmente. I piani rigidi non erano più sempre efficaci, specialmente considerando le richieste di consegne sempre più rapide. Con team piccoli, l'overhead generato da processi pesanti sarebbe stato un costo eccessivo, portando così all'adozione di un approccio Agile.

Il requisito fondamentale dell'approccio Agile è rispondere rapidamente ai cambiamenti anziché seguire pedissequamente un piano preciso. Data l'incertezza sulle esigenze finali e la loro probabile evoluzione nel corso dello sviluppo, l'Agile propone una strategia per gestire tali cambiamenti. In questo contesto, si evita di congelare i requisiti iniziali, ma si cerca piuttosto di capire meglio cosa si vuole ottenere.

Nell'approccio Agile, la specifica, il design e l'implementazione procedono in parallelo e sono strettamente collegati. Il software finale viene gestito attraverso una serie di versioni, con ciascuna versione successiva che rappresenta un incremento del prodotto. È cruciale coinvolgere il cliente e gli stakeholder durante tutto il processo di sviluppo Agile in modo da assicurare un feedback costante e un allineamento continuo con le aspettative.

Di solito, si stabilisce un intervallo di tempo limitato per lo sviluppo di ciascuna versione, che va da 2 a 4 settimane. La documentazione è mantenuta al minimo per evitare discrepanze e inconsistenze tra la documentazione e il codice stesso. La comunicazione all'interno del team è spesso informale ma frequente, promuovendo un flusso costante di informazioni.

Per automatizzare il processo di testing, è fondamentale adottare strumenti per il testing automatico, in quanto il testing manuale risulterebbe troppo lento e inefficiente data la necessità di coprire numerosi scenari di test. Alcuni strumenti chiave per l'approccio Agile includono:

- Testing automatico

- Gestione della configurazione
- Integrazione continua
- Produzione automatica dell'interfaccia utente

L'obiettivo è rilevare e correggere gli errori il prima possibile, in modo da evitare che si propaghino attraverso il sistema.

2.1.2 Manifesto Agile

L'Agile si basa su alcuni principi chiave che sono riassunti nel Manifesto Agile:

- L'importanza è spostata dal processo formale all'individuo e alle interazioni.
- È più importante avere un software funzionante rispetto a una documentazione esaustiva.
- Si promuove la collaborazione con il cliente, che dovrebbe essere parte integrante del team di sviluppo piuttosto che un soggetto esterno con cui contrattare.
- L'attenzione è posta sulla capacità di rispondere al cambiamento in modo rapido e flessibile, piuttosto che sull'aderenza rigorosa a un piano prestabilito.

2.1.3 Implementazione dell'Approccio Agile

Per realizzare appieno i principi dell'Agile, è fondamentale coinvolgere attivamente il cliente nel processo di sviluppo. Il coinvolgimento del cliente deve andare oltre il semplice fornitore di specifiche, incoraggiando il cliente a partecipare attivamente alle discussioni e al processo decisionale. Questo assicura che le funzionalità sviluppate siano allineate con le aspettative del cliente e che i feedback siano integrati tempestivamente nel processo di sviluppo.

Un altro aspetto chiave è lo sviluppo incrementale, dove le funzionalità vengono fornite in maniera graduale e integrata nel sistema in crescita. In questo contesto, è fondamentale avere team con competenze solide e diversificate, che si fidino a vicenda per garantire un flusso di lavoro efficiente e un'efficace condivisione delle responsabilità.

Nell'approccio Agile, è importante prepararsi al cambiamento continuo dei requisiti nel corso del progetto. La capacità di adattarsi rapidamente ai nuovi requisiti e di integrarli nel processo di sviluppo è un fattore cruciale per il successo.

Il principio cardine per lo sviluppo Agile è quello di mantenere la semplicità. Ciò implica resistere alla tentazione di aggiungere funzionalità non essenziali che potrebbero introdurre una complessità eccessiva nel sistema. L'obiettivo è quello di adottare strategie che semplifichino il processo e riducano al minimo il rischio di complicazioni impreviste durante lo sviluppo.

2.1.4 Applicabilità dell'Approccio Agile

L'approccio Agile è particolarmente adatto in contesti in cui il cliente è disponibile e desideroso di partecipare attivamente al processo di sviluppo del software. È efficace in ambienti

in cui le normative sono meno restrittive e permettono una maggiore flessibilità nel processo di sviluppo. I processi di sviluppo Agile sono spesso implementati in combinazione con metodologie di gestione progetti agili, come ad esempio Scrum.

2.2 Extreme Programming (XP)

Spesso, nell'ambito dell'approccio Agile, si fa riferimento a **Extreme Programming**, che spinge all'estremo alcune delle caratteristiche fondamentali dell'Agile. Ad esempio, in Extreme Programming, sono comuni varie versioni del software consegnate al cliente quotidianamente, con incrementi che vengono immediatamente messi in produzione. Un principio chiave è che i casi di test devono superare con successo per ogni build, pertanto è necessario assicurarsi che tutti i casi di test siano funzionanti prima di procedere con una nuova build.

Nel contesto di Extreme Programming, viene dato ampio risalto alla pratica di scrivere i test prima di scrivere il codice effettivo. Il refactoring è una pratica costante con l'obiettivo di semplificare il codice sorgente per renderlo più leggibile e mantenibile nel tempo. Altro aspetto fondamentale di XP include il pair programming, in cui due sviluppatori lavorano insieme su un singolo codice, garantendo una maggiore qualità e condivisione delle conoscenze. Inoltre, si mira a mantenere un ritmo di sviluppo sostenibile nel tempo, evitando eccessive accelerazioni che potrebbero compromettere la qualità del software.

2.3 User Stories

Nel contesto dell'Agile, i requisiti vengono spesso raccolti in maniera informale tramite user stories, che sono narrazioni in prosa che descrivono le interazioni tra l'utente e il sistema. Questo metodo aiuta a mantenere i requisiti concreti e facilita l'elicitazione dei requisiti stessi poiché le user stories sono facili da scrivere e da comprendere.

Le user stories vengono successivamente suddivise in parti più piccole, comunemente chiamate task cards. Per ogni task, è necessario fornire una stima temporale per la sua realizzazione. L'obiettivo è quello di coinvolgere attivamente il cliente nella priorizzazione delle varie attività, poiché, in un contesto di sviluppo incrementale, non è possibile affrontare tutto contemporaneamente.

Tuttavia, con questo tipo di approccio, non si ha la certezza di implementare tutte le funzionalità richieste, poiché alcune di esse potrebbero non emergere in determinati scenari o iterazioni di sviluppo.

2.4 Test-Driven Development (TDD)

Nel Test-Driven Development (TDD), i test vengono scritti prima del codice stesso, e tali test possono essere eseguiti durante la fase di scrittura del codice. Questo approccio consente di individuare errori il prima possibile, valutando ogni singola microcomponente prima di passare a quella successiva. Il codice risultante è di alta qualità in quanto viene costantemente testato e non presenta bug evidenti.

I test, in questo contesto, rappresentano una documentazione degli scenari previsti e consentono di pensare al comportamento del sistema prima ancora di implementarlo effettivamente. Ciò porta a una maggiore consapevolezza del sistema e del suo funzionamento da parte del team di sviluppo.

Inoltre, è importante coinvolgere attivamente l'utente nella fase di verifica, in modo da sviluppare i cosiddetti acceptance test per le varie user stories. Questo è possibile solo grazie all'utilizzo di framework di test automatizzati. Di conseguenza, il numero di test generati è notevolmente elevato, garantendo la coerenza e la stabilità del sistema, specialmente in un contesto di sviluppo incrementale.

2.4.1 Refactoring

Il refactoring rappresenta il processo di riscrittura del codice al fine di renderlo più leggibile e mantenibile nel lungo periodo. Spesso, ogni singolo incremento potrebbe richiedere alcuni compromessi che, per essere integrati nel sistema, richiedono una sorta di "degradazione" del sistema. Il refactoring permette di affrontare tali problematiche e di mantenere l'integrità del sistema nel tempo, assicurandone la stabilità e la longevità. Il refactoring risulta essere una pratica altrettanto valida anche nel contesto di sviluppo pianificato (*plan-driven*).

2.4.2 Pair Programming

Il pair programming è una pratica in cui due persone lavorano insieme su un singolo computer. Mentre una persona scrive il codice, l'altra controlla attivamente il lavoro. I ruoli si alternano regolarmente, e le decisioni cruciali vengono prese in modo collaborativo. I vantaggi del pair programming sono numerosi, inclusa la condivisione della responsabilità, il controllo continuo e un feedback costante. Inoltre, il continuo scambio di conoscenze e la code review continua portano a un miglioramento delle abilità individuali e della qualità del codice. Il refactoring continuo contribuisce ulteriormente a migliorare la comunicazione all'interno del team. Contrariamente a ciò che si potrebbe pensare, il pair programming non causa una diminuzione significativa della produttività, ma piuttosto conduce a un miglioramento generale delle prestazioni del team.

2.5 CI/CD nel Sviluppo Software Moderno

CI/CD è un acronimo che sta per "Continuous Integration" (Integrazione Continua) e "Continuous Delivery" o "Continuous Deployment" (Consegna Continua o Distribuzione Continua). Questi sono concetti chiave nelle pratiche di sviluppo software moderno, in particolare nell'ambito della metodologia Agile e DevOps.

2.5.1 Continuous Integration (CI)

La Continuous Integration si riferisce alla pratica di integrare automaticamente i cambiamenti del codice nel repository principale di un progetto più volte al giorno. Il suo scopo è identificare e risolvere rapidamente i problemi di compatibilità o altri problemi introdotti da nuove modifiche. Questo processo coinvolge tipicamente l'uso di strumenti automatizzati per

compilare e testare il codice ogni volta che un cambiamento viene *committato*, assicurando che il codice nuovo funzioni correttamente con il codice esistente.

2.5.2 Continuous Delivery (CD)

La Continuous Delivery si estende dalla CI e implica la consegna automatica di cambiamenti del codice a un ambiente di test o di produzione dopo il processo di integrazione. L'obiettivo è rendere il processo di rilascio del software il più efficiente e prevedibile possibile, riducendo i tempi e i costi di distribuzione.

2.5.3 Continuous Deployment

Il Continuous Deployment è simile al Continuous Delivery. Tuttavia, mentre nel Continuous Delivery ogni rilascio richiede ancora un'intervento manuale per la distribuzione in produzione, nel Continuous Deployment ogni cambiamento che passa tutte le fasi di produzione viene rilasciato automaticamente senza intervento umano.

2.6 Project Management Agile

Nel contesto dell'approccio Agile, il project manager ha la responsabilità di assicurarsi che il software venga consegnato entro i tempi previsti, con tutte le funzionalità richieste e nel rispetto del budget stabilito. È responsabile del coordinamento del progetto e del monitoraggio dei progressi.

2.6.1 Scrum

Scrum è un metodo di project management agile che si basa sul lavoro per iterazioni, richiamando così il processo di sviluppo software agile. Le sue fasi principali includono:

1. Definizione di obiettivi generali e architettura ad alto livello del sistema.
2. Sprint planning, in cui si definiscono gli obiettivi per la prossima iterazione.
3. Fase conclusiva, che include la documentazione finale, la code review e la riflessione sulle lezioni apprese.

2.6.2 Terminologia

Alcuni termini chiave in Scrum includono:

- Team piccolo, con un massimo di 7 persone.
- Backlog, che rappresenta la lista di tutte le funzionalità richieste dal cliente.
- Product Owner, responsabile della priorizzazione delle funzionalità e delle decisioni su cosa è più necessario.
- Sprint, che rappresenta un periodo di lavoro di 2-4 settimane.
- Scrum Master, responsabile del monitoraggio e del supporto al processo di sviluppo.

- Velocità, che rappresenta la produttività del team e viene calcolata e aggiornata durante lo sviluppo per la pianificazione dei prossimi sprint.
- Sprint Review, una sessione per riflettere sull'iterazione e fornire input per migliorare i prossimi sprint.

2.6.3 Benefici

I benefici di Scrum includono la gestione efficiente di funzionalità nel breve termine, una conoscenza approfondita dei problemi e delle attività di ogni membro del team, nonché la puntualità delle consegne per il cliente. Scrum favorisce un clima di fiducia tra il cliente e il team di sviluppo, poiché il cliente ha la possibilità di visualizzare il sistema software funzionante e fornire un feedback continuo. Questo feedback aiuta a chiarire le esigenze del cliente e ad adattare il prodotto in modo efficace.

2.7 Scrum Distribuito

Lo Scrum distribuito si riferisce a situazioni in cui il team di sviluppo è distribuito in più sedi geografiche. In questo contesto, la comunicazione tra i membri del team avviene attraverso strumenti come chat in tempo reale, chiamate video e l'utilizzo di sistemi di continuous integration per garantire che il software sia sempre funzionante. Un piano di sviluppo comune aiuta a facilitare la comunicazione tra i vari team e sedi geografiche.

2.7.1 Scalabilità

La scalabilità di Scrum si riferisce alla sua capacità di adattarsi a progetti più grandi che coinvolgono più persone e team, o che richiedono un'implementazione distribuita. Alcuni punti chiave da considerare includono la necessità di mantenere le pratiche chiave come il Test Driven Development (TDD), la comunicazione e la consegna di incrementi funzionanti. La coordinazione tra lo sviluppo Agile e la manutenzione del sistema è altrettanto importante, specialmente quando le persone coinvolte potrebbero cambiare nel corso del progetto.

2.7.2 Problemi e Resistenze

Alcuni problemi e resistenze comuni nell'adozione di metodi Agile come Scrum includono la mancanza di esperienza del management nell'ambito della metodologia Agile, la necessità di adattarsi a procedure di controllo della qualità preesistenti e le resistenze culturali all'interno dell'organizzazione. Il successo della metodologia Agile può essere dimostrato attraverso casi di studio e una maggiore stabilità dei requisiti nel tempo.

Capitolo 3

Ingegneria dei requisiti

3.1 Requisiti

Un requisito è una descrizione di una funzionalità che il sistema deve fornire e le condizioni al contorno in cui tale sistema dovrebbe funzionare (*vincoli, operazioni, ...*). Riflette quelle che sono le necessità del cliente e degli utenti finali del sistema software.

Si tratta di un problema che all'origine sembra semplice, ma in realtà è molto complesso.

3.1.1 Requisiti utente e di sistema

User requirements

Si tratta di un requisito per come viene percepito dall'utente. Tipicamente sono frasi in lingua naturale che esprimono le proprietà del sistema ed eventualmente con qualche diagramma per rendere più chiara la descrizione.
Tipicamente sono frasi astratte, e quindi non prevedono una soluzione.

System requirements

System requirements, ovvero i requisiti di sistema, sono requisiti più specifici, strutturati e formali. Tipicamente sono espressi in un linguaggio formale, all'interno di un documento strutturato. Tipicamente il linguaggio è comprensibile anche da parte dell'utente, in modo che possa leggerlo e comprenderlo.

Tipicamente i gli user requirements sono scritti dall'utente per avere un'idea di come il sistema dovrebbe funzionare. Essi possono essere la base per la formulazione di un contratto, dando quindi modo agli sviluppatori di definire le risorse necessarie per lo sviluppo del sistema, abbozzando quindi un piano di lavoro.

I requisiti di sistema permettono di definire il sistema in modo più preciso, in modo che i clienti possano validare ciò che il sistema deve fare.

Stakeholders

Quando parliamo di requisiti, dobbiamo considerare gli stakeholder. Gli stakeholder sono tutte quelle persone che hanno un interesse nel sistema, ovvero tutti gli utenti e tutti quelli che verranno toccati anche non interagendo con il sistema.

Se venisse coinvolto un sottoinsieme di stakeholder, probabilmente alcune necessità verrebbero inespresse e di conseguenza implementate in modo parziale dal sistema software.

3.1.2 Requisiti funzionali e non funzionali

Requisiti funzionali

Intendiamo che catturano le funzionalità che il sistema deve fornire, ovvero i servizi che deve mettere a disposizione che possono essere usati dagli utenti, come il sistema deve reagire a particolari eventi e come deve funzionare in particolari scenari.

I requisiti funzionali sono espressi mediante diversi livelli di astrazione e possibili ambiguità possono portare a diverse implementazioni. In caso di ambiguità è opportuno marcare il requisito come ambiguo, in modo che venga chiarito in seguito.

Oltre all'ambiguità, in relazione ai requisiti funzionali, si possono avere diversi problemi. Prima di tutto bisogna cercare di raggiungere l'obiettivo legato alla **completezza**, ovvero che tutte le funzionalità siano state catturate dai requisiti. Il secondo obiettivo è quello della **consistenza**, ovvero che non ci siano requisiti che si contraddicono tra loro.

Nella pratica è difficile raggiungere questi obiettivi, perché stakeholder diversi potrebbero avere opinioni diverse su come il sistema dovrebbe funzionare, e quindi potrebbero esserci delle contraddizioni. Inoltre, potrebbero esserci imprecisioni e ambiguità tramite clienti e sviluppatori.

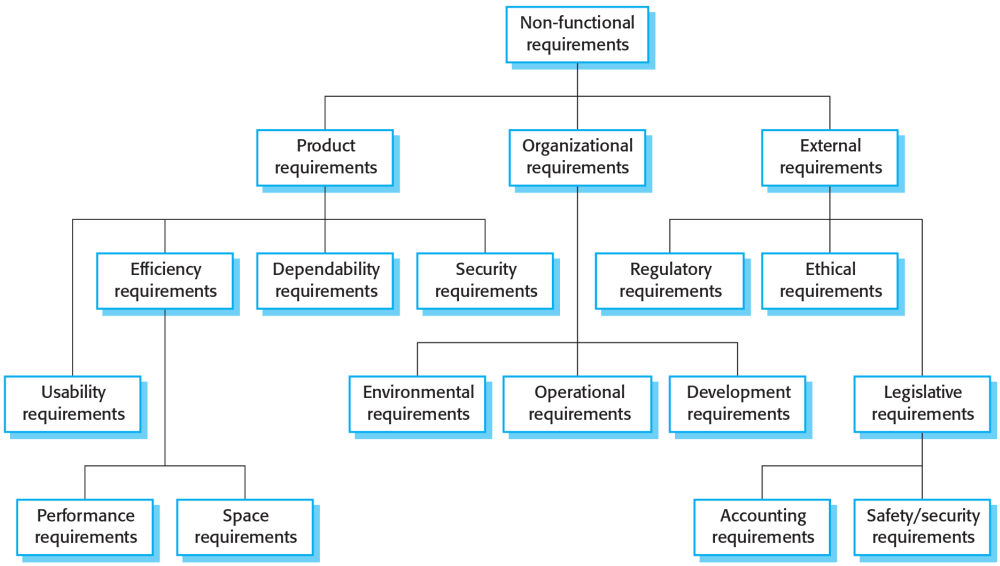
Alcune inconsistenze potrebbero sfuggire durante la fase di analisi dei requisiti, e potrebbe emergere quando tale requisito deve essere implementato correttamente.

Requisiti non funzionali

I requisiti non funzionali sono tutti quei requisiti che non sono legati alle funzionalità del sistema, ma che riguardano le proprietà del sistema.

Tipicamente sono vincoli legati al sistema e non sono legati al singolo componente del sistema.

Il problema è che tali requisiti non emergono in maniera spontanea da una frase pronunciata dal nostro cliente, perché tipicamente gli utenti hanno in mente le funzionalità del sistema e non le proprietà del sistema. Potrebbe quindi aiutare l'avere una tassonomia legata a quali sono i requisiti non funzionali, in modo da poterli riconoscere e quindi poterli catturare.



3.1.3 Verifica dei requisiti non funzionali

Siccome i vincoli non funzionali sono di alto livello, potrebbe risultare abbastanza difficile capire se sono soddisfatti o meno. Sarebbe opportuno quindi scrivere tali requisiti in maniera più quantitativa possibile invece che qualitativa. Supponiamo di disporre di tale vincolo:

> Il sistema deve essere facile da usare da parte dello staff medico in modo da minimizzare gli errori.

Ma come facciamo a capire se il sistema è facile da usare? È opportuno quindi trasformare da qualitativo a quantitativo, in modo da poterlo misurare.

> Il sistema deve essere facile da usare da parte dello staff medico in modo da minimizzare gli errori. Il sistema deve essere usabile da parte di un medico con meno di 4 ore di formazione. Il numero medio di errori per ora di utilizzo non deve superare 2 per ora.

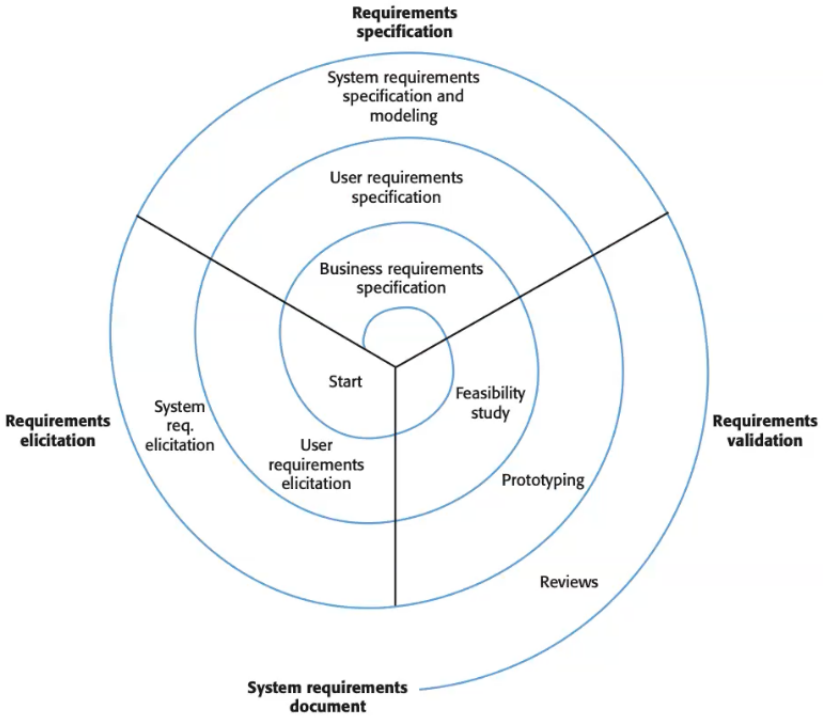
Fondamentale quindi è la capacità di tradurre i requisiti non funzionali qualitativi in requisiti non funzionali quantitativi, in modo da poterli misurare e quindi verificare, anche mediante metriche.

3.2 Processi di ingegneria dei requisiti

Di fatto ci sono tre fasi nella gestione dei requisiti:

- **Elicitazione dei requisiti**: si cerca di capire quali sono i requisiti, in modo da poterli catturare e quindi poterli documentare. Dove l'output è la descrizione del sistema.

- **Specifica dei requisiti**: ovvero come vengono scritti tali requisiti. Dove l'output sono i requisiti dell'utente e del sistema.
- **Validazione dei requisiti**: si verifica che i requisiti siano completi e non contradditori. Dove l'output è il documento dei requisiti.



In generale, il processo di sviluppo può essere descritto come un ciclo a spirale. Inizia con la specifica dei requisiti, in cui si definiscono i business requirements iniziali, ovvero i requisiti legati al dominio applicativo. Dopo, si passa alla validazione dei requisiti, verificando la loro completezza e coerenza.

Si riparte con la elicitazione dei requisiti, dove con la ripetizione di questo processo, emergono nuovi requisiti legati agli utenti. Questo permette di aggiornare la specifica dei requisiti utente e procedere con la loro validazione. Il ciclo si ripete ancora, focalizzandosi sui requisiti di sistema.

3.2.1 Elicitazione dei requisiti

Gli sviluppatori devono capire il dominio applicativo, le funzionalità che il sistema dovrà fornire, e i vincoli non funzionali come le performance e l'interazione con l'hardware. Possono presentarsi ostacoli: per esempio, i clienti potrebbero esprimere i requisiti usando termini non noti agli ingegneri del software o dare per scontati alcuni aspetti che sono ovvi nel loro dominio applicativo, oppure i requisiti potrebbero essere irrealizzabili.

La presenza di diversi stakeholder può portare a differenti obiettivi, risultando in requisiti potenzialmente in conflitto. La collaborazione può essere compromessa se gli stakeholder non si sentono rappresentati e possono emergere fattori politici. I requisiti potrebbero cambiare o emergere nel corso dello sviluppo.

Intervista

Un metodo primario per raccogliere i requisiti è l'intervista, che può includere domande sia chiuse che aperte, generalmente in una forma mista. Tuttavia, gli stakeholder non sempre forniscono requisiti dettagliati.

È importante condurre le interviste in modo efficace, cercando di evitare pregiudizi e verificando le proprie assunzioni. È consigliabile iniziare con domande semplici, evitando di partire subito con concetti troppo astratti.

I problemi che possono verificarsi sono legati alla difficoltà di capire il dominio applicativo, e quindi di catturare i requisiti. Una determinata classe di medicinali per un medico potrebbe essere ovvia, ma non per un ingegnere del software, non sentendo quindi il bisogno di descriverlo nei requisiti. Inoltre, potrebbero esserci dei problemi aziendali che portano gli stakeholder a non voler condividere informazioni, o a non voler cambiare i processi aziendali.

I suggerimenti sono quindi di lavorare con mentalità aperta, cercando di capire il dominio applicativo, e di non dare per scontato nulla ed evitare domande preconfezionate.

Etnografia

Non intervistare le persone, ma lavorare insieme a loro, per capire il loro necessità. Questo aiuta molto la conoscenza implicita che potrebbe non emergere durante l'intervista. Questo tipo di studio può essere attuato con un sistema software esistente, oppure se l'azienda ha già un'operatività anche cartacea.

Storie e Scenari

Storie Le storie sono narrazioni ad alto livello che forniscono esempi concreti del funzionamento del sistema software in un contesto specifico. Sono formulate in modo narrativo e facile da comprendere, e permettono agli stakeholder di visualizzare l'interazione con il sistema e di esprimere feedback su ciò che ritengono appropriato o meno.

Scenari Gli scenari, invece, rappresentano esempi di utilizzo più dettagliati del sistema. Descrivono un'interazione specifica tra l'utente e il sistema, includendo dettagli come l'input specifico. Questo li rende strumenti più precisi per illustrare le funzionalità del sistema e per validare i requisiti con gli stakeholder.

Una storia raccoglie può quindi raccogliere più scenari.

3.2.2 Specifica dei requisiti

La specifica dei requisiti è il processo di scrittura dei requisiti in un documento. I requisiti devono essere scritti in modo chiaro e preciso, in modo da essere comprensibili a tutti gli stakeholder. Gli user requirements devono essere comprensibili per gli utenti, mentre i system requirements possono includere dettagli tecnici. Tale documento può essere parte del contratto tra cliente e fornitore, e può essere usato come base per la verifica e la validazione dei requisiti, devono quindi essere più completi possibili.

In linea di principio viene descritto **cosa** il sistema dovrà fare non contenendo dettagli implementativi. In pratica però, è difficile separare i requisiti e potrebbero quindi emergere dettagli implementativi, legati anche all'architettura.

Linguaggio naturale

Il primo modo per scrivere i requisiti è il linguaggio naturale, che però è intrinsecamente ambiguo. Per ridurre l'ambiguità, si possono usare i verbi modali in modo corretto, e usare il grassetto per evidenziare parti chiave.

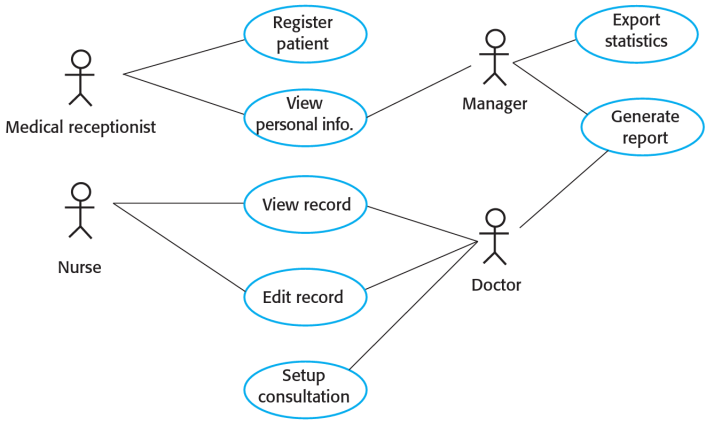
Si evita di norma l'utilizzo di gergo tecnico, e si usano invece termini che gli stakeholder possono comprendere. Di norma quando un requisito è inserito all'interno è accompagnato dalla motivazione per cui è necessario.

Struttura vincolata

Un altro modo per scrivere i requisiti è la struttura vincolata, che permette di avere una struttura standard per i requisiti. Questo permette di avere una struttura standard, ma non è sempre facile da usare e può essere limitante.

Use cases

L'use case è una descrizione di un insieme di sequenze di azioni che un sistema e serve per identificare gli attori e le funzionalità del sistema che fanno riferimento ai vari utenti.



Documento dei requisiti

Il **Documento dei Requisiti Software** è fondamentale per la specificazione dei requisiti. Questo documento può variare a seconda delle organizzazioni e dei contesti; tuttavia, esistono standard definiti da organizzazioni internazionali che possono fornire linee guida e struttura. Un esempio di tale standard è quello proposto dall'`IEEE` (*Institute of Electrical and Electronics Engineers*).

3.2.3 Validazione dei requisiti

L'obiettivo principale della validazione dei requisiti software è assicurare che i requisiti definiti corrispondano effettivamente al sistema desiderato dal cliente. È particolarmente costoso rilevare difetti nei requisiti nelle fasi avanzate dello sviluppo, arrivando a costi fino a 100 volte superiori.

Una checklist efficace per la validazione dei requisiti include i seguenti punti:

- **Validità:** Il sistema fornisce le funzionalità che meglio supportano le necessità del cliente?
- **Consistenza:** Ci sono conflitti tra i requisiti (*contraddizioni, differenze*)?
- **Completezza:** Tutte le funzioni richieste dal cliente sono incluse?
- **Realismo:** È possibile implementare i requisiti con il budget e la tecnologia disponibili?
- **Verificabilità:** È possibile controllare i requisiti?

Le tecniche di validazione includono:

1. **Revisioni dei Requisiti:** Analisi manuale sistematica dei requisiti da parte di un team di auditor.
2. **Prototipazione:** Modello eseguibile del sistema per verificare i requisiti con gli utenti finali.
3. **Casi di Test:** Sviluppo di test per i requisiti al fine di controllarne la testabilità. Se è difficile scrivere un test, probabilmente sarà difficile implementare il requisito.

3.2.4 Cambiamento dei requisiti

Nei grandi sistemi software, i requisiti sono soggetti a continui cambiamenti a causa della natura intrinsecamente complessa dei problemi, dell'evoluzione delle tecnologie e delle mutevoli priorità. Una gestione efficace di questi requisiti include l'identificazione univoca di ciascun requisito per permetterne il cross-referenziamento, un processo di gestione del cambiamento per stimare impatto e costi dei cambiamenti, policies di tracciabilità per mappare le relazioni tra i requisiti e il design del sistema, e il supporto di strumenti vari come database e fogli di calcolo.

Il processo di cambiamento dei requisiti implica l'analisi del problema e la specificazione dei cambiamenti, l'analisi degli effetti e dei costi dei cambiamenti proposti, e l'implementazione

del cambiamento attraverso modifiche al documento dei requisiti e, se necessario, al design del sistema.

Capitolo 4

Design architetturale

4.1 Design architetturale

L'obiettivo del design architetturale nel campo del software è di comprendere l'organizzazione del software, progettando la sua struttura generale e identificando i principali componenti strutturali e le loro relazioni all'interno del sistema. Un quesito rilevante in questo ambito è: perché è importante esplicitare un'architettura software?

Ci sono molteplici motivi per questo. Primo, l'architettura agisce come una rappresentazione ad alto livello che facilita la comunicazione con gli stakeholder. Inoltre, aiuta nell'analisi del sistema, poiché alcune decisioni architetturali possono influenzare aspetti non funzionali del sistema come le performance e l'affidabilità. Infine, un'architettura ben definita può favorire il riutilizzo dei componenti in diversi sistemi.

La **rappresentazione** dell'architettura è spesso realizzata tramite diagrammi a blocchi, che mostrano le relazioni tra le varie entità. Questi diagrammi, sebbene utili, sono talvolta criticati per la loro marcata astrazione e mancanza di significato dettagliato.

Gli **utilizzi** di tali rappresentazioni includono la semplificazione delle discussioni sulla progettazione del sistema e la documentazione dell'architettura, evidenziando componenti, connessioni e interfacce. Questo approccio fornisce una visione chiara e organizzata dell'intero sistema, essenziale per una progettazione e implementazione efficace.

Le **viste** dell'architettura sono le rappresentazioni dell'architettura che mostrano diversi aspetti del sistema. Queste viste sono utili per gli stakeholder che hanno interessi diversi nel sistema. Ad esempio, un programmatore può essere interessato alla vista dell'architettura che mostra i componenti del sistema e le loro interazioni, mentre un amministratore di sistema può essere interessato alla vista che mostra le risorse del sistema e le loro interazioni. Si distinguono quindi le seguenti viste:

- **Vista logica**: mostra le entità chiave del sistema e le loro interazioni.
- **Vista di processo**: mostra i processi del sistema e le loro interazioni.

- **Vista di sviluppo**: mostra come il sistema è organizzato per il processo di sviluppo.
- **Vista fisica**: mostra l'hardware e il software che compongono il sistema.

4.2 Pattern architetturale

4.2.1 Model View Controller

Il **Model View Controller** è un pattern architetturale che separa i dati e la logica dell'applicazione dalla loro rappresentazione e dal modulo che si occupa di gestire l'interazione con l'utente. Il pattern MVC prevede la suddivisione dell'applicazione in tre componenti:

- **Model**: rappresenta i dati dell'applicazione e le operazioni che possono essere eseguite su di essi. In questo componente non è presente alcuna logica di presentazione.
- **View**: rappresenta la visualizzazione dei dati contenuti nel model. In questo componente non è presente alcuna logica di business.
- **Controller**: riceve gli input dell'utente e li converte in comandi per il model o per la view.

L'utilizzo di MVC è consigliato quando si vuole separare la logica di business dalla logica di presentazione. In questo modo è possibile modificare la rappresentazione dei dati senza dover modificare la logica di business e viceversa. Quindi quando ci sono molti modi di vedere o interagire con gli stessi dati o quando il modo in cui le viste interagiscono con i dati è soggetto a evoluzione.

Vantaggi e svantaggi

Tra i vantaggi di questo pattern si possono citare:

- **Separazione delle responsabilità**: il pattern MVC separa le responsabilità dell'applicazione in tre componenti. Questo rende più facile la manutenzione e l'evoluzione dell'applicazione.
- **Supporta la presentazione dei dati in maniere diverse**: il pattern MVC permette di presentare i dati in maniere diverse.
- **Cambiamenti nella logica**: il pattern MVC permette di cambiare la logica dell'applicazione senza dover modificare la presentazione dei dati.

Tra gli svantaggi principali abbiamo che la separazione dei componenti può portare a un aumento della quantità di codice necessario per implementare l'applicazione.

4.2.2 Architettura stratificata

L'architettura stratificata è un pattern architetturale che prevede la suddivisione dell'applicazione in un insieme di livelli. Ogni livello è responsabile di un aspetto dell'applicazione. I livelli comunicano tra loro attraverso interfacce ben definite.

Viene utilizzata quando si progetta su sistemi esistenti, quando lo sviluppo è diviso tra vari team o se sono requisiti legati alla sicurezza multi-livello.

Vantaggi e svantaggi

Tra i vantaggi di questo pattern si possono citare:

- Permette di rimpiazzare un livello fintanto che l'interfaccia di comunicazione tra livelli adiacenti è rispettata.
- Si possono introdurre ridondanze tra i vari layer in maniera da incrementare l'affidabilità del sistema.

Tra gli svantaggi principali abbiamo che:

- Nella pratica una separazione dei livelli così netta e pulita non è banale da raggiungere.
- L'aggiunta di un nuovo livello può portare a un aumento della latenza.

4.2.3 Architettura Repository

L'architettura Repository è un pattern architetturale che prevede la presenza di un componente che si occupa di gestire la persistenza dei dati. Questo componente fornisce un'interfaccia per accedere ai dati e per modificarli.

Viene solitamente utilizzata quando ci sono grandi quantità di dati da gestire o quando devono essere gestiti dati che cambiano frequentemente, o in sistemi dove l'inclusione di dati nella repository fa si che venga azionata una serie di eventi.

Vantaggi e svantaggi

Il vantaggio è che i componenti sono indipendenti, inoltre i cambiamenti fatti su una risorsa da un componente sono immediatamente disponibili per gli altri componenti perché vengono propagati tramite la repository. I dati possono essere gestiti consistentemente dato che sono accessibili solo tramite la repository.

Gli svantaggi sono che:

- C'è un single point of failure. Se la repository fallisce, l'intero sistema fallisce.
- La repository può diventare un collo di bottiglia.
- Distribuire la repository su più server può essere difficile.

4.2.4 Architettura Client-Server

L'architettura Client-Server è un pattern architetturale che prevede la presenza di due componenti: un server che fornisce un servizio e un client che richiede il servizio. Il client e il server comunicano attraverso una rete utilizzando un protocollo ben definito.

Viene utilizzata quando i dati condivisi devono essere acceduti da più posizioni, e dato che i server sono replicabili per fare load-balancing e per garantire l'affidabilità del sistema.

Vantaggi e svantaggi

Tra i vantaggi di questo pattern si possono citare:

- I server possono essere distribuiti sulla rete.
- Funzionalità generali, sono disponibili a tutti i client senza che siano implementate da tutti i server.

Tra gli svantaggi principali abbiamo che:

- Ogni servizio è un single point of failure.
- La performance dipende dalla latenza della rete.
- Problemi di management quando i server sono posseduti da differenti organizzazioni.

4.2.5 Architettura Peer-to-Peer

Nelle architetture **Peer-to-Peer** (P2P), ogni componente agisce sia come client che come server. Ogni peer fornisce un'interfaccia che specifica sia i servizi offerti sia quelli richiesti. Quando un peer necessita di dati, esso comunica con gli altri peer della rete per identificare chi può fornirgli le informazioni richieste.

Le architetture P2P sono note per la loro capacità di scalare efficacemente e per la loro tolleranza ai guasti. La replicazione dei dati tra vari peer è un elemento chiave, aumentando la disponibilità dei dati e riducendo il rischio di perdita di informazioni a causa di guasti di singoli nodi. Questa caratteristica rende le reti P2P particolarmente robuste e affidabili, adatte per sistemi che richiedono una distribuzione ampia e decentralizzata delle risorse.

4.2.6 Architettura Pipe and Filter

L'architettura Pipe and Filter è un pattern architetturale che i dati vengano elaborati per step successivi, ad esempio prima generiamo i dati, poi li filtriamo, poi li analizziamo e infine li visualizziamo.

Si tratta di un'architettura molto appropriata per sistemi in cui i dati devono essere elaborati in maniera sequenziale, in cui i dati sono disponibili in quantità illimitate e in cui i dati possono essere elaborati in maniera indipendente. Non sono adatte per sistemi interattivi. La utilizziamo quindi in applicazioni che processano dei dati.

Vantaggi e svantaggi

Tra i vantaggi di questo pattern si possono citare:

- Facile da capire e supporta il riuso.
- Il workflow è simile alla struttura di danti processi di business.
- L'evoluzione aggiungendo trasformazioni è semplice.
- Può essere implementato come un sistema sequenziale o concorrente.

Tra gli svantaggi principali abbiamo che:

- Il formato dei dati deve essere comune a tutti gli step di trasformazione.
- Ogni trasformazione deve fare parsing dei dati come concordato, ciò incrementa l'overhead.

4.3 Architetture eterogenee

Ovviamente si possono combinare tra di loro i vari pattern architetturali se vantaggioso. Li si può organizzare in una composizione gerarchica, ossia ad esempio ho un sistema pipe and filter che ha un sottosistema organizzato a livelli. Nelle architetture eterogenee inoltre un componente può avere due ruoli diversi in due pattern distinti, ad esempio un sottosistema accede ai dati tramite una repository ma comunica con gli altri componenti tramite una pipe.

Capitolo 5

Testing

5.1 Introduzione

Una volta che il sistema software è stato implementato, l'obiettivo è verificare che sia corretto e allineato con quello che avremmo dovuto implementare in linea con i desideri sia in fase di specifica che di design e implementazione. Sostanzialmente si tratta di verificare che il software faccia quello che deve fare e verificare che non ci siano deformità, ovvero che non ci siano disallineamenti tra il comportamento che dovrebbe avere il software e quello che effettivamente ha.

Comportamento atteso

Il comportamento atteso è quello che è stato definito in fase di specifica e design.

Comportamento osservato

Il comportamento effettivo che il sistema espone nel funzionamento

Parliamo quindi di **matching** tra comportamento atteso e comportamento osservato, eseguendo quindi almeno un test per ogni requisito raccolto, più la combinazione di tali funzionalità.

Quando c'è una discrepanza tra comportamento atteso e comportamento osservato, si deve valutare se tale differenza possa essere un errore di raccolta dei requisiti, errore di design o errore di implementazione. Tipicamente si utilizza il termine **bug**.

5.1.1 Program testing

Di fatto, quando parliamo di Testing, parliamo di `SUT` (*System Under Test*), ovvero sistema che stiamo testando. Per testare il sistema dobbiamo mandarlo in esecuzione e ciò richiede che vangano inseriti dei dati di input e che vengano consumati dal sistema e che calcolano quindi un output. La decisione che viene presa è in relazione all'allineamento tra output atteso e output osservato.

In caso di allineamento, il test è **passato**, altrimenti è **fallito**.

La terminologia però non aiuta, perché un caso di test è tanto più utile quanto più rileva la presenza di un errore. Dicendoci quindi che su uno scenario di esecuzione il sistema rileva un errore.

Nota 5.1.1 *Il fatto che un test non rilevi un errore non significa che il sistema sia corretto, perché potrebbe esserci sempre scenari di esecuzione che non sono stati considerati. Quindi il test può solo mostrarci la presenza di errori ma non la loro assenza.*

5.1.2 Confidenza

Vogliamo avere una certa confidenza che il sistema non contenga bug, ovviamente per avere una confidenza del 100% dovremmo testare tutti gli scenari di esecuzione su tutti i possibili input, quindi si scarta la questione legata alla **completezza** dei test. A seconda del contesto dipendono dalle proprietà del software, legati anche alla criticità del software stesso, alle aspettazioni degli utenti, all'ambiente di marketing e di business, ecc...

Si tratta quindi di considerazioni ambientali più che tecniche, che ci portano a considerare la confidenza che vogliamo avere nel sistema.

Testing

Per testing intendiamo l'esecuzione di un sistema software su determinati input e la verifica che il comportamento osservato sia allineato con quello atteso.

5.1.3 Code review

Un'alternativa al software testing è il **code review**, ovvero la revisione del codice da parte di un altro sviluppatore, senza eseguire il codice, cercando di ottenere lo stesso risultato. Si tratta di un'attività legata all'ispezione del codice.

Ci sono diversi vantaggi nel code review, prima di tutto ciò non è limitato al codice sorgente, ma può essere esteso anche alla documentazione, alla grafica, ai test. Inoltre gli errori non sono mascherati, ciò significa che mandando in esecuzione il software con il test, al primo errore tipicamente l'esecuzione viene bloccata, nascondendo un possibile errore successivo. Di fatto è possibile verificare anche altri aspetti legati alla qualità del codice, come la leggibilità, la manutenibilità, ecc...

5.1.4 Test driven development

Parlando di testing, si può parlare anche di **test driven development**. Il test driven development è una metodologia di sviluppo software che prevede che i test vengano scritti prima del codice. Questo perché i test sono un ottimo strumento per capire cosa deve fare il software, inoltre i test sono un ottimo strumento per capire se il software funziona o meno.

I vantaggi del test driven development sono:

- Alta copertura delle funzionalità dovuta al fatto che i test vengono scritti prima del codice.
- Si utilizzano i test come test di non regressione, ovvero per che la qualità del software non regredisca nel tempo.
- Si semplifica la fase di debugging, in quanto si sa già dove cercare l'errore.
- Si semplifica la fase di documentazione, in quanto i test sono un ottimo strumento per capire cosa fa il software. Di fatto chiarisce le aspettative legate alla funzionalità prima ancora di implementarla.

5.2 Testing

Il testing si divide in tre fasi:

- **Development testing**: è il testing che viene condotto durante il processo di sviluppo, ovvero durante la fase di implementazione.
- **Release testing**: è il testing che viene fatto prima di rilasciare il software in produzione.
- **User testing**: è il testing che viene fatto dagli utenti, ovvero il testing che viene fatto in produzione.

5.2.1 Development testing

Tipicamente il development testing è fatto dagli sviluppatori, quindi è un testing svolto in parallelo con l'implementazione. L'obiettivo è quello di rilevare errori commessi dai programmatori in fase di sviluppo, tipicamente si esegue il software anche con attività di debugging, quindi si esegue il software in modalità `debug` per poter rilevare errori in particolari punti di esecuzione.

Quando parliamo di development testing abbiamo tre fasi distinte:

- **Unit testing**: è il testing che viene fatto a livello di unità, ovvero a livello di singola classe o di singola funzione. L'obiettivo è quello di verificare che la singola unità funzioni correttamente.
- **Component testing**: è il testing che viene fatto a livello di integrazione, ovvero a livello di integrazione tra più unità. L'obiettivo è quello di verificare che le unità funzionino correttamente anche quando sono integrate.
- **System testing**: è il testing che viene fatto a livello di sistema, ovvero a livello di sistema software. L'obiettivo è quello di verificare che il sistema software funzioni correttamente.

Unit testing

Quando abbiamo a che fare con il unit testing, abbiamo a che fare con il testing delle singole unità, ovvero delle singole classi o delle singole funzioni. Testare una classe significa testare

tutte le funzionalità che la classe mette a disposizione, quindi testare tutti i metodi della classe e testare gli oggetti istanziati dalla classe in tutti i modi possibili.

Si tratta di un problema difficile ottenere una copertura completa, ovvero eseguire tutti gli scenari di esecuzione possibili. Tipicamente si cerca di ottenere una copertura legata alla criticità del software, ovvero si cerca di ottenere una copertura che sia proporzionale alla criticità del software.

Tipicamente lo unit testing è di questa forma:

- Setup: il sistema è inizializzato in uno stato stabile.
- Chiamata al metodo da testare.
- Assert: verifica che il risultato sia allineato con quello atteso.

> È importante che i test di unità siano automatizzati, ovvero che possano essere eseguiti in modo automatico, senza intervento umano.

Spesso non è possibile testare una singola unità in modo isolato, ma è necessario testare una singola unità in modo integrato con altre unità o con altre componenti che non sono ancora state sviluppate. In questo caso si utilizzano dei **mock**, ovvero delle componenti che simulano il comportamento di altre componenti.

Ma quali scenari devono essere eseguiti? Una prima regola è quella di eseguire gli scenari che più si adattano agli scenari realistici, ovvero agli scenari che più si avvicinano a quelli che verranno eseguiti in produzione. Si dovrebbero testare anche valori anormali e valori limite, ovvero valori che potrebbero causare problemi al sistema. Quello che si può fare è utilizzare l'**input partitioning**, ovvero raggruppare gli input in classi di equivalenza, ovvero raggruppare gli input in classi che hanno lo stesso comportamento, quindi si esegue il test su un rappresentante per ogni classe di equivalenza.

Le linea guida per il unit testing sono:

- Con valori multipli si cerca di testarlo con liste di lunghezza diversa, cercando di scrivere test sugli elementi iniziali, sugli elementi finali e sugli elementi interni.
- Con valori booleani si cerca di testarlo con entrambi i valori.
- Si utilizza l'esperienza pregressa per capire quali sono i valori che potrebbero causare problemi.
- Si cerca di testare i valori che potrebbero causare problemi.
- ecc...

Component testing

Il component testing è il testing che viene fatto a livello di integrazione, ovvero a livello di integrazione tra più unità. L'obiettivo è quello di verificare che le unità funzionino correttamente anche quando sono integrate. L'obiettivo è quello di verificare che le assunzioni fatte

da una classe sul comportamento dell'altra siano rispettate. Tipicamente si verifica che il passaggio di dati avvenga in maniera corretta, se le due classi contengono memorie condivise, si verifica che il passaggio di informazioni avvenga in maniera corretta, o la consistenza tra i protocolli di comunicazione, ecc...

Questo avviene perché mettendo insieme due classi, è presente un contratto da rispettare. I tipici errori che si possono verificare sono:

- Errori di interfaccia: utilizzo scorretto di come vengono fatte le chiamate ad esempio mediante l'utilizzo del tipo di dato sbagliato.
- Errori di incomprensione: quando una classe non rispetta il contratto che è stato definito (*ad esempio liste ordinate che non sono ordinate*).
- Errori di tempi: quando una classe non rispetta i tempi che sono stati definiti (*ad esempio quando una classe non rispetta i tempi di risposta*).

Le linee guida sono quelle di testare i parametri che sono agli estremi del range, ovvero testare i parametri che sono al limite inferiore e al limite superiore. Si cerca di testare il passaggio di puntatori nulli, si testa con chiamate che si presume mandino in errore dei componenti, si eseguono stress test, ecc...

System testing

Il system testing è il testing che viene fatto a livello di sistema, ovvero a livello di sistema software. L'obiettivo è quello di verificare che il sistema software funzioni correttamente. Il focus è l'interazione tra le componenti, per rilevare ipotesi errate sul comportamento delle componenti e che i dati vengano passati in maniera corretta tra le componenti.

In questo caso è opportuno utilizzare gli **use case** per capire quali sono gli scenari che devono essere eseguiti e che solo in questo momento diventano testabili in quanto sono stati sviluppati tutti i componenti. Il diagramma di sequenza associato ai casi d'uso documenta i componenti e le interazioni che stanno venendo testati, i test dovrebbero anche considerare le eccezioni e verificare che siano correttamente gestite.

Testare tutte le possibili esecuzioni di un sistema è impossibile, si possono però utilizzare delle policies per considerare il testing adeguato, esempi di policies sono:

- tutte le istruzioni del programma devono essere eseguite da almeno un test.
- tutte le funzioni accessibili tramite un menu dovrebbero essere testate così come dovrebbero esserlo tutte le combinazioni di funzioni accessibili dal menu.
- se sono forniti input dall'utente, tutte le funzioni devono essere testate con input corretti o errati.

Release testing

Il release testing è il testing che viene fatto a livello di sistema software prima che il sistema venga rilasciato. L'obiettivo è quello di verificare che il sistema sia pronto per il rilascio, ovvero

che sia pronto per essere utilizzato dagli utenti finali. Tipicamente tale fase viene eseguita da un team diverso da quello che ha sviluppato il software, in modo da avere un team che non ha preconcetti sul software. Tipicamente si utilizza il **black box testing**, ovvero si testa il software senza conoscere la sua struttura interna, ma solo conoscendo le sue funzionalità.

Scenario testing Lo scenario è una storia che descrive un modo in cui il sistema potrebbe essere utilizzato, un testing degli scenari quindi può includere vari requisiti e in caso appunto scenari e user stories siano disponibili dalla fase di ingegneria dei requisiti possono essere direttamente utilizzati come scenari di test.

Performance testing Il performance testing è il testing che viene fatto a livello di sistema software per verificare che il sistema soddisfi i requisiti di performance. L'obiettivo è quello di verificare che il sistema soddisfi i requisiti di performance che sono stati definiti, ovvero che il sistema soddisfi i requisiti di tempo di risposta, di throughput, di utilizzo di memoria, ecc...

Se invece parliamo di stress testing, siamo in un contesto in cui il sistema è volutamente sovraccaricato per capire il suo comportamento quando fallisce.

User testing

L'obiettivo del user testing è quello di verificare che il sistema soddisfi le esigenze degli utenti finali. Tipicamente viene fatto in un ambiente di test controllato, in cui gli utenti finali utilizzano il sistema e vengono osservati dagli sviluppatori.

Ci sono delle tipologie di user testing:

- **Alpha testing**: è il testing che viene fatto con pochi utenti finali, in un ambiente controllato, in cui gli sviluppatori sono presenti e possono osservare gli utenti finali.
- **Beta testing**: è il testing che viene fatto con un numero maggiore di utenti finali, in un ambiente controllato, in cui gli sviluppatori sono presenti e possono osservare gli utenti finali.
- **Acceptance testing**: è il testing che viene fatto dagli utenti finali in un ambiente non controllato, in cui gli sviluppatori non sono presenti. In questa fase gli utenti finali utilizzano il sistema e verificano che soddisfi i requisiti che sono stati definiti. Decidendo quindi se accettare o meno il sistema.

Accettare il sistema implica l'avvenire del pagamento finale per il software, il processo di accettazione è cosi strutturato: Si definiscono i criteri, che se soddisfatti, comportano l'accettazione del sistema, si definisce poi il piano di accettazione quindi risorse,tempi,budget. Si prosegue quindi derivando i casi di test che verranno eseguiti e si eseguono questi test, i risultati poi vengono negoziati, se ci sono test falliti si capisce come sistemare questa condizione di incorrettezza, per finire il sistema viene o meno accettato e di conseguenza pagato o non pagato.

Metodi agili nell'acceptance testing

Nei metodi agili il customer è parte del team di sviluppo ed è responsabile di prendere decisioni per l'accettabilità del sistema, i test sono definiti dal customer e integrati con altri test e vengono eseguiti automaticamente quando sono fatti cambiamenti. Non c'è una fase di acceptance testing separata, il problema principale è **finire**.

Capitolo 6

Refactoring

6.1 Introduzione

Refactoring

Il codice sorgente, soprattutto con approcci incrementali, tende a subire cambiamento nel tempo. Questi cambiamenti possono degradare la struttura interna del software, rendendolo difficile da comprendere e mantenere. Il refactoring è un processo di trasformazione del codice sorgente che non altera il comportamento esterno del software ma migliora la sua struttura interna.

Si tratta di una manutenzione preventiva, la qualità del codice è normale che degradi. Limitando la normale decadenza del codice, migliorare la leggibilità (*non muore una volta scritto, bisogna continuare a modificarlo*), non migliorando è più facile inserire bug.

Il refactoring è diverso dal reingegnering. Il reingegnering è un processo che mira a riprogettare il software che è stato scritto per architetture datate o tecnologie obsolete (*soprattutto cambio di architetture, come per esempio da monolitiche e cloud*). Il refactoring invece è un processo che mira a migliorare la struttura interna e va in parallelo con lo sviluppo del software.

Debito tecnico

I debiti tecnici sono tutti i problemi di comprensione del codice che sono disposto ad avere per qualcosa che serve attualmente, ma probabilmente causerà in futuro.

Per ridurre il debito tecnico si può fare refactoring. Ci sono solamente in cui è opportuno fare refactoring, ad esempio quando si deve iniziare a sviluppare nuove funzionalità, in modo che sia facile nel codice recepire questa nuova funzionalità. Oppure quando si deve correggere un bug, o in caso di identificazione di *code smell*.

6.1.1 Code smell

Sensazione che qualcosa nel codice non va, e che gli sviluppatori esperti sanno che è un problema. Gli sviluppatori esperti hanno una sensazione di disagio quando vedono un code smell, lo sviluppatore non avrebbe fatto così.

Code smell comuni

Duplicazione di codice, quando avviene si utilizza una funzione per rimuovere la duplicazione. Preventivamente vengono evitati problemi, riducendo il costo di manutenzione futuro.

I metodi e le funzioni troppo lunghi, sono difficili da capire, e ci si accorge quando si tende a commentare il codice ad ogni step. In questo caso è utile spezzare il codice in più segmenti, già individuati dai commenti.

Spesso lo switch case rappresenta un code smell, soprattutto quando si utilizza in un linguaggi ad oggetti. In questo caso è meglio sfruttare il polimorfismo. In termine di oggetti è più naturale.

Data clumping quando delle strutture dati sono sempre usate insieme, è un code smell. Ha senso creare una classe che racchiude queste strutture dati.

Generalità speculativa, quando si crea un'interfaccia per qualcosa che non è ancora necessario, ma questo crea difficoltà nella complessità del codice.

6.1.2 Classi di smell

- Troppo codice:
  - **Large class**: una classe che ha troppe responsabilità
  - **Long method**: un metodo che è troppo lungo.
  - **Long parameter list**: una lista di parametri troppo lunga.
  - **Duplicated code**: codice duplicato.
  - **Dead code**: codice che non viene mai eseguito.
- Troppo poco codice:
  - **Classi con poco codice**: una classe che non ha abbastanza codice.
  - **Empty catch clause**: un blocco catch che non fa nulla.
  - **Classi di dati**: classi con solo getter e setter.
- Codice con troppi commenti: il codice non è autoesplicativo.

6.2 Software clone

Clone

Un clone è un frammento di codice che è simile ad un altro frammento di codice.

Spesso in copia incolla e vengono fatte modifiche, ma non sempre. Tuttavia abbiamo problemi di manutenzione, perché se si modifica un frammento di codice, bisogna ricordarsi di modificare anche il clone. Attorno al 5%. Esistono tool per identificare i cloni.

Tipi di cloni

- **Type 1**: cloni identici, copia incolla.
- **Type 2**: cloni simili, ma con modifiche, a meno di ridenominazioni di identificatori.
- **Type 3**: cloni simili, ma con modifiche e aggiunte, ad esempio quando i parametri sono di tipi diversi e anche il valore di ritorno.

6.3 Refactoring

Il fatto di avere i test automatizzati è importante, perché si può fare refactoring in modo sicuro. Test di non regressione, per verificare che il comportamento non cambi. Quando tutti i test passano, si può fare refactoring, si identifica il code smell e si determina il refactoring appropriato. Una volta completata questa modifica si rieseguono i test, se passano si può continuare.

Il ritmo da seguire è quello su scala piccola, si fa refactoring su una piccola parte di codice, si eseguono i test, si fa refactoring su un'altra piccola parte di codice, si eseguono i test, e così via. In questo modo si riduce il rischio di introdurre bug e ci si accorge subito se si è introdotto un bug.

6.3.1 Applicazione del refactoring

La ridenominazione è il refactoring più semplice, si cambia il nome di una variabile, di una funzione, di una classe, di un metodo. Per farlo possiamo utilizzare tool automatici, che ci permettono di fare refactoring in modo sicuro, e vengono gestite in automatico dipendenze.

I tool riescono anche a rilevare dipendenze anche in altri file, che possono sfuggire all'occhio umano. Lo svantaggio è che automatizzano solamente i refactoring più semplici, quelli più complessi vanno fatti manualmente. Ovviamente i casi di test devono esserci e devono coprire la funzionalità.

Di solito capita di dare nomi sbagliati alle variabili, soprattutto quando non è chiaro il dominio del problema. In questo caso è utile fare refactoring per rendere più chiaro il codice. Poiché le nuove funzionalità tenderanno a seguire la vita del progetto.

Estrazione di interfaccia

Una classe dipendente non dalla struttura, ma gli si da un'interfaccia che deve essere implementata. In questo modo si può cambiare la struttura della classe, ma la classe dipendente non cambia, riducendo il numero di cambiamenti da fare sulla classe. Si ragiona meglio in termini di funzionalità e non di implementazione in termini di cambiamento.

Pull up method

Si spostano dei metodi da una classe a una superclasse, in modo che le sottoclassi ereditino il metodo. In questo modo si evita di avere codice duplicato.

Extract method

Si estrae un metodo da una classe, in modo da avere un metodo che fa una sola cosa. In questo modo si evita di avere metodi troppo lunghi.

Move method

Si sposta un metodo da una classe a un'altra, in modo che il metodo sia più vicino ai dati che utilizza. In questo modo si evita di avere metodi che utilizzano dati di altre classi. Come ad esempio:

La persona partecipa al progetto `p.partecipate(x)`

Il progetto contiene la persona `x.partecipate(p)`

In questo modo il progetto viene controllato la classe progetto, poiché si tratta di una proprietà del progetto.

Replace temp

Si sostituisce una variabile temporanea con un metodo, in modo da rendere più chiaro il codice.

```
double basePrice = quantity * itemPrice;
if(basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

Si crea il metodo `basePrice()` che restituisce il prezzo base, in questo modo il codice diventa più chiaro.

```
if(basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
```

Replace method with method object

Si sostituisce un metodo con un oggetto, in modo da avere un metodo che fa una sola cosa. In questo modo si evita di avere metodi troppo lunghi.

```
int basePrice = quantity * itemPrice;
double discountLvel = getDiscountLevel();
double finalPrice = computeFinalPrice(basePrice,
   discountLevel);
```

Si elimina il parametro discountLevel, chiamando il metodo al suo interno per ridurre la complessità del metodo.

6.3.2 Extract class

Si estrae una classe da un'altra, in modo da avere una classe che fa una sola cosa. In questo modo si evita di avere classi troppo grandi.

6.3.3 Replace inheritance with delegation

Si sostituisce l'ereditarietà con la delega, in modo da avere una classe che fa una sola cosa. In questo modo si evita di avere classi troppo grandi.

Pensando all'esempio dello stack, potremmo utilizzare un design dove lo stack che al posto di ereditare informazioni del vettore come, utilizzerà funzionalità di delega. incapsulando il vettore all'interno della classe. Una eredità selettiva e parziale.

```
public void push(Object elemento)
{
    _vector.insertElementAt(elemento, 0);
}
```

6.3.4 Replace conditional with polymorphism

Si sostituisce un costrutto condizionale con il polimorfismo, in modo da avere una classe che fa una sola cosa. In questo modo si evita di avere classi troppo grandi.

Prendiamo il calcolo dell'area:

```
private double a, b, r;
...
public double area()
{
    switch(type)
    {
        case SQUARE:
            return a * a;
        case RECTANGLE:
            return a * b;
```

```
            case CIRCLE:
                return Math.PI * r * r;
        }
    }
```

A seconda della forma geometrica, si calcola l'area in modo diverso. Ha più senso creare una classe astratta `Shape` che ha un metodo `area()` che viene implementato dalle classi `Square`, `Rectangle`, `Circle`.

```
interface class Shape
{
    public double area();
}

class Square implements Shape
{
    public double area()
    {
        return side * side;
    }
}
...
```

6.3.5 Separate domain from presentation

Separazione dal dominio della presentazione, in modo da avere una classe che fa una sola cosa. In questo modo si evita di avere classi troppo grandi. Dividendo quindi le responsabilità.