Università degli Studi di di Verona

FACOLTÀ DI SCIENZE E INGEGNERIA
Corso di Laurea in Informatica

Ingegneria del Software

Riassunto dei principali argomenti

Autori:
Davide Bianchi
Matteo Danzi

Anno Accademico 2016/2017

Indice

1 Introduzione

L' ingegneria del software è lo *studio delle metodologie, strumenti e teorie che stanno alla base della creazione dei software a livello professionale.*
È lo studio a livello *professionale* perché a livello *artigianale* non sempre vi è la garanzia del prodotto, in quanto non è garantito che la soluzione sia **scalabile** (cioè permette di utilizzare il software per un ambito/problema più grande o differente rispetto a quello in cui è utilizzato).
I prodotti software si suddividono in:

- **prodotti software generici** (e.g.: S.O., Applicazioni ...)

  prodotti in cui il progettista decide le caratteristiche del prodotto e deve accontentare le richieste da parte di molti utenti cercando di creare un progetto generico diretto a una vasta categoria di persone.

- **prodotti software specifici/personalizzati**(e.g.: Applicativi per monitorare l'inquinamento)

  prodotti in cui l'utilizzatore richiede al progettista le caratteristiche che deve avere il prodotto, e il software è pesantemente influenzato da chi lo usa.

Ma che cos'è il software?
Il software non è altro che un insieme di programmi che svolgono un compito qualsiasi e che viene opportunamente documentato, la cui documentazione indica com'è strutturato.

1.1 Caratteristiche di un buon prodotto software

1. **Mantenibilità**:
   - deve poter essere gestito su un periodo vita medio/lungo.
   - deve saper gestire eventuali errori che non si erano trovati precedentemente.
   - deve fare in modo che il software giri su piattaforme diverse.
2. **Affidabilità**: un software deve eseguire quello che gli viene imposto senza compromettere altri aspetti come ad esempio la sicurezza degli utenti.
3. **Usabilità**: quanto è facile da utilizzare un software, saperlo utilizzare senza dover leggere le istruzioni. Ad esempio per utilizzare un browser web utilizzo sempre gli stessi criteri di usabilità e faccio in modo che sia utilizzabile in modo intuitivo.

1.2 Quattro attività principali dell' ingegneria del software

1. ***Specifica del software*** : si considerano i requisiti di chi mi richiede di creare il software.

   - l'ingegneria dei requisiti considera i requisti dati e li trasforma in requisiti ad alto livello, ma più rigorosi e meno ambigui.

2. ***Realizzazione del software*** : si appoggia sui requisiti e consiste di due fasi:

   - progettazione
   - implementazione

3. ***Validazione del software*** : è un fattore molto importante e consiste nella validazione da parte di un utente del software creato.

4. ***Evoluzione del software*** : consiste in modifiche e aggiornamenti apportati dopo la creazione finale del software; rappresenta il costo principale (più del 50%!!!) di una vita media di un sw.

Problemi e sfde attuali dell' ingegneria del software

- sicurezza: privacy, frodi, danni a cose o persone ecc.
- costi: si deve contenere i costi di produzione sotto controllo.
- etereogeneità: sfruttare il sw su più macchine, soddisfare più richieste in contemporanea ecc.
- tempistiche: sono sempre più ridotte.

1.3 Efficienza VS efficacia

L' *efficienza* indica quanto materiale uso per risolvere un problema, è un'aspetto più computazionale, mentre l' *efficacia* consiste nel chiedersi se un sw svolge il proprio compito per cui è stato creato, quanto un sw è in grado di risolvere il problema richiesto dall'utente e quindi se raggiunge l'obbiettivo delle specifiche.

Tipologie software:

(a) Sistemi Stand-Alone: sistemi sw isolati a livello di rete, funzionano su una macchina isolata.

(b) Applicazioni interattive basate su transizioni: scambio di dati concordato da entrambe le parti (vi è il concetto di "o tutto, o niente"). Sono per lo più basate sul web e quindi distribuite sulla rete. Ad esempio operazioni di scambio di dati tipo commercio elettronico(Amazon) oppure tutte le applicazioni bancarie.

(c) Sistemi embedded: sistemi sw che sono innestati all'interno di apparecchi di varia natura (controllo treni, bancomat). Il software è a bordo di un sistema dedicato e quindi ho vincoli architetturali.

(d) Applicazioni batch: applicazioni che elaborano dati in grossi gruppi. Ad esempio stampare e produrre dati bancari di tutti i clienti di una banca in un certo istante oppure inserimento pesante di dati.

(e) Sistemi d'intrattenimento: tipo videogiochi, film on demand, il daltonismo del Dalty ecc. Pesantemente basati sul web.

(f) Applicazioni di raccolta dati: tipo Arpav, centraline per rilevare dati di inquinamento, i quali vengono raccolti e compongono un archivio.Wearable Systems: cioè sistemi che raccolgono parametri vitali (tipo elettrocardiogramma, per monitorare ...).

(g) Sistemi di sistemi: sistemi che si basano su altri sistemi
esempio: Amazon che ha una parte basata su transazioni, che gestisce i pagamenti, e una parte di applicazione raccolta dati, che permette di avere informazioni sui clienti per dare consigli e pubblicità (cookies, banner...).

1.4 Fondamenti dell'ingegneria del Software

Ogni sviluppo del SW dovrebbe avere:

- Chiara metodologia/processo di sviluppo:
  - devo avere un percorso ben definito all'inizio
  - devo dotarmi di strumenti che mi supportano le fasi di sviluppo
  - devo sapere come avvengono i vari passi di sviluppo
- Affidabilità e prestazioni (Efficienza-Efficacia)
- Specifica del Software e requisiti
- Riuso del Software: esistono componenti SW che permettono di iniziare a costruire un nuovo progetto prendendo questi esempi e usando le *librerie di base*.

2 Etica dell'ingegneria del Software

Gli ingegneri del software devono devono comportarsi in modo onesto e in modo eticamente responsabile se vogliono essere rispettati dal punto di vista professionale. Il comportamento etico non è solamente il semplice rispetto della legge ma implica il rispetto di un insieme di principi che sono moralmente corretti.

- **Confidenzialità**: Gli ingegneri dovrebbero rispettare la confidenzialità dei loro datori di lavoro o clienti, indipendentemente dal fatto che si sia firmato un accordo di confidenzialità.
- **Competenza-Onestà**: Gli sviluppatori non dovrebbero fornire false competenze o accettare lavori che sono al di là del proprio livello di conoscenza. Dovrebbero invece costantemente documentarsi e aggiornare le proprie competenze se il lavoro che devono svolgere lo richiede.
- **Proprietà Intellettuale**: Gli ingegneri dovrebbero essere a conoscenza delle leggi che governano l'uso della proprietà intellettuale, come ad esempio brevetti, licenze, copyright, ecc. Dovrebbero fare in modo di assicurare che la proprietà intellettuale dei dipendenti o dei clienti sia protetta.
- **Uso Scorretto del SW**: Gli ingegneri del software non dovrebbero usare le loro conoscenze tecniche per danneggiare gli altri. Non possono usare il software in modo ricreativo (giocare a videogiochi sui pc dei dipendenti) oppure malevolo (generare virus), devono generare software senza arrecare danni.
- **Giudizio e Rapporto con colleghi**: Gli ingegneri del sw dovrebbero essere giusti e coadiuvare in modo corretto i propri colleghi, se necessario.

Le organizzazioni statunitensi *ACM/IEEE* hanno cooperato per costruire un codice etico, che viene firmato dai membri di queste quando vengono assunti. Il codice contiene 8 princìpi legati al comportamento e alle decisioni fatte dagli ingegneri del software professionisti, includendo anche altre professioni (manager, insegnanti, medici ...).

2.1 Dilemmi Etici

Esempi di dilemmi etici e relative ipotetiche soluzioni:

- Disaccordo con i principi e con le politiche aziendali.

  Soluzione: proposta di una soluzione alternativa corretta dal punto di vista etico
- I dipendenti non si comportano in modo responsabile ed eticamente scorretto pubblicando sistemi non stabili dal punto di vista della sicurezza, senza eseguire gli adeguati test sul software.

  Soluzione: Devo garantire il controllo sui dipendenti per migliorare la qualità e la professionalità.

3 Processi per lo sviluppo di Software

I processi per il Software sono l'insieme strutturato di attività richieste per sviluppare un sistema software. Ci sono molti modelli di processo diversi, cioè molti diversi schemi per realizzare sw utilizzabili per molte diverse tipologie. Tutti questi modelli comprendono:

1. Specifica: cosa il sistema dovrebbe fare.
2. Progettazione e Implementazione: definire l'organizzazione del sistema e l'implementazione dello stesso.
3. Validazione: controllare che faccia ciò che il cliente vuole.
4. Evoluzione: cambiare il sistema in risposta ai cambiamenti di ciò che il cliente vuole.

Un modello di processo per il software è una rappresentazione astratta di un processo. Presenta una descrizione di un processo da uno specifico punto di vista.
Quando si descrivono i processi solitamente si parla di attività come ad esempio la specifica di strutture dati, la costruzione di un'interfaccia utente, ecc. e l'ordinamento di tali attività.

3.1 Processi Plan-driven e Agile

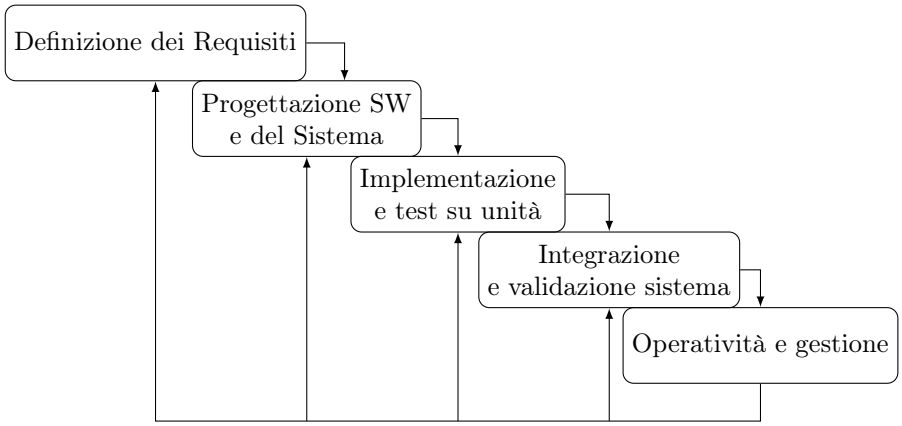
Nei processi **plan-driven** tutte le attività vengono pianificate all'inizio e mentre si esegue lo sviluppo viene verificata la progressione dell'esecuzione del piano iniziale.
Nei processi **agile** la pianificazione viene fuori man mano che le attività vengono eseguite e quindi risulta più facile modificare il processo a fronte del cambiamento delle richieste da parte del cliente.
Nella pratica la maggior parte dei processi pratici include sia elementi dell'uno che dell'altro approccio.

3.2 Modelli di processi Software

Ci sono 3 tipi di modelli di processi software:

3.2.1 Modello a cascata

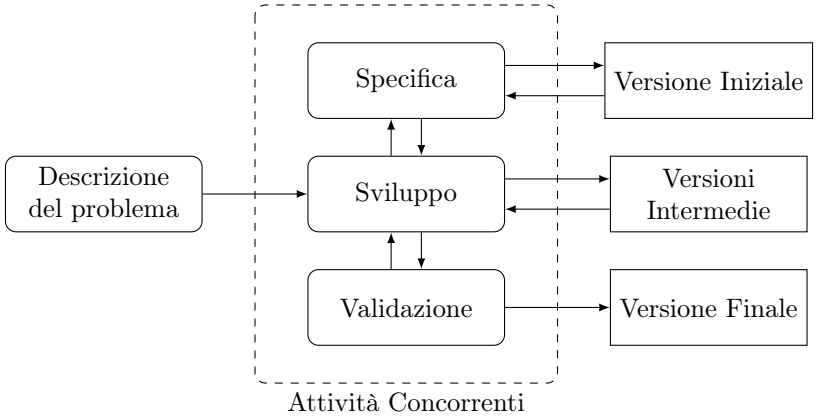
Modello *Plan-driven* in cui le fasi di specifica e di sviluppo sono distinte e separate.



L'output diventa input, ogni fase non inizia prima che la precedente sia terminata. Solo quando arrivo all'ultima fase ho *retroazione* e posso tornare indietro.
Lo **svantaggio** principale è la difficoltà di cambiamento dopo che il processo si è avviato. La sua partizione in fasi distinte rende difficile rispondere ai cambiamenti delle specifiche da parte del cliente. Questo modello è appropriato quando i requisiti sono stati ben compresi e i cambiamenti saranno limitati durante il processo di sviluppo.

Questo modello è principalmente usato per grossi progetti di sistemi di ingegneria dove un sistema è sviluppato in diverse zone.

3.2.2 Sviluppo incrementale



Le fasi di Specifica, Sviluppo e Valutazione sono concorrenti e non è detto che avvengano sempre nell'ordine se ho diverse parti di un progetto. Fornisce risultati intermedi che vengono raffinati sulla base del confronto con l'utente il quale rimane coinvolto in tutto il processo. Può essere *plan-driven* e quindi fisso il numero di prodotti intermedi, altrimenti è *agile*.
Vantaggi:

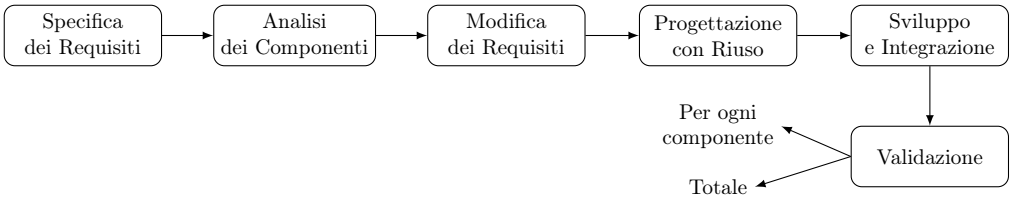
- Se i requisiti non sono chiari o vengono cambiati durante la costruzione, posso modificare a run-time gestendo dinamicamente gli aggiornamenti e riducendo quindi il costo. La quantità di documentazione che deve essere rifatta è molto inferiore rispetto al modello a cascata.
- Ho un maggior coinvolgimento del committente il quale riesce a vedere in modo periodico gli sviluppi del progetto e riesce a modificarlo eventualmente.
- Ho una consegna di software utile più rapida al committente.

Svantaggi:

- Il processo non è così visibile come nel modello a cascata. Documento le parti più importanti senza mostrare i cambiamenti. Si hanno quindi fasi intermedie prive di documentazione.
- L'idea di aggiungere parti ad un software produce un degrado del software. Non è più così efficiente e organico come si era pensato inizialmente. Bisogna quindi riorganizzare il software raffinandolo per evitare il peggioramento dell'efficienza; questo però aggiunge costi ulteriori.

3.2.3 Processi basati sul riuso

I sistemi non vengono ricreati da zero, bensì vengono riciclati da componenti già esistenti o da applicazioni COTS (commercial-off-the-shelf systems). Gli elementi riusati possono essere riconfigurati per adattare le loro funzionalità alle richieste dei committenti. È l'approccio standard per costruire la maggior parte dei sistemi commerciali.



Vantaggi e svantaggi:

- Si riducono i costi e i rischi poiché meno software è sviluppato da zero.
- Consegna e pubblicazione dei sistemi avviene in modo più veloce.

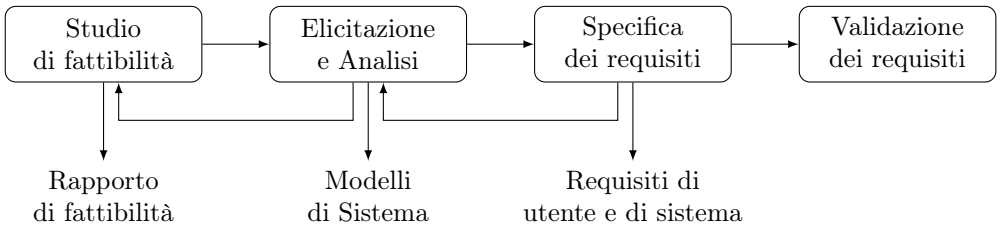
- Compromessi sulle specifiche sono inevitabili, quindi i sistemi potrebbero non soddisfare pienamente i reali bisogni dei committenti.
- Perdita di controllo sull'evoluzione degli elementi del sistema riusati.

3.3 Attività dei processi

3.3.1 Specifica del software

La specifica comprende 4 fasi principali:

1. **Studio di fattibilità**: quali costi e quali risorse/competenze bisogna avere per specificare il software. Si sviluppa un modello che simula i costi e si ricava una stima del costo.
2. **Elicitazione(estrazione) e analisi dei requisiti**: si analizza ciò che l'utente utilizza normalmente per estrarre informazione e per fondare le parti del software. Questa fase comprende anche l'intervista di utenti, i quali possono avere attese e esigenze diverse.
3. **Specifica dei requisiti**: si utilizzano formalismi diagrammatici per definire e rappresentare i requisiti.
4. **Validazione dei requisiti**



4 Sviluppo agile

4.1 Introduzione

Nel mondo moderno, lo sviluppo rapido e la consegna in breve tempo di un prodotto finito sono i requisiti più importanti per i sistemi software.

Il metodo di sviluppo agile si è sviluppato nei tardi anni 90 e il suo scopo principale era quello di ridurre il tempo di consegna di un prodotto software finito.

In un sistema agile di norma la progettazione e l'implementazione di un software sono sempre a stretto contatto, e le parti interessate al software sono spesso coinvolte nel processo di sviluppo, per scopi di specifica e validazione del prodotto stesso, che viene consegnato in più versioni (spesso con una documentazione minima, per focalizzare gli sforzi dello sviluppo nella scrittura di codice funzionante).

Lo sviluppo agile del software ruota in generale attorno a 5 principi cardine:

- Coinvolgimento del cliente: sfruttato soprattutto per le sezioni di verifica del prodotto;
- Consegna incrementale: il cliente specifica i requisiti una porzione alle volta, e la loro realizzazione è inclusa nella consegna del prodotto più vicina;
- Sfruttamento delle abilità del team;
- Progettare basandosi sui cambiamenti dei requisiti, quindi sviluppare un prodotto che sia semplice da cambiare;
- Mantenere la semplicità del prodotto: se possibile lavorare anche attivamente per ridurre la complessità del software sviluppato finora.

4.2 Extreme programming

La tecnica di extreme programming estremizza le condizioni di lavoro per lo sviluppo agile del software, a volte anche facendo build del prodotto diverse volte al giorno, pubblicando release dopo brevissimi periodi di sviluppo e test (circa 2 settimane), e assicurandosi che la release venga pubblicata solo quando tutti i test siano passati.

Le caratteristiche principali dell'extreme programming sono le seguenti:

- Planning dei task incrementale: le caratteristiche da includere nel software sono decise sulla base del tempo che richiedono per essere sviluppate, e sono divise in tasks;
- Piccole release: una versione con delle funzionalità di base è la prima ad essere rilasciata, successivamente vengono fatte release frequenti e con aggiunte di feature extra in maniera incrementale;
- Design semplice: il design è spesso progettato in maniera minimale, in modo da concentrare lo sviluppo per soddisfare al massimo i requisiti richiesti;
- Sviluppo del test necessario per testare una funzionalità prima di implementare la funzionalità stessa;
- Refactoring: tutti gli sviluppatori eseguono un refactoring del codice ogni volta che sono possibili delle ottimizzazioni, per mantenere il codice più semplice e mantenibile possibile;
- Pair programming: programmazione di coppia, per verificare ognuno il lavoro dell'altro;
- Proprietà collettiva: le coppie di sviluppatori lavorano su tutto il software, in modo da non creare zone di codice che solo una coppia sa interpretare e tutti si prendono la responsabilità di tutto il codice;
- Integrazione continua: ogni volta che una nuova funzionalità è implementata viene immediatamente integrata nel prodotto finito;
- Ritmo sostenibile: non sono permessi grandi overtime, per non abbassare la qualità del codice;
- Cliente sempre in loco: un rappresentante del cliente è sempre disponibile al team di sviluppo ed è responsabile per la consegna delle specifiche.

Refactoring Il refactoring è una procedura (in alcuni punti anche automatizzata) per migliorare la qualità del codice e mantenerlo leggibile e facilmente modificabile, ad esempio riorganizzando la gerarchia delle classi per evitare codice duplicato, tenere un codice pulito per renderlo semplice da capire, usare delle librerie per evitare di sovraccaricare il codice che si scrivere con molte chiamate consecutive.

I motivi principali per cui il refactoring può dimostrarsi necessario sono:

- Codice duplicato;
- Metodi lunghi;
- Blocchi switch/case;
- Data clumping;
- Codice derivato da speculazioni.

Test-first Il metodo test-first è basato sull'idea di sviluppare prima il test della feature che la feature stessa, in questo modo vengono a galla molti chiarimenti sulla funzionalità da implementare. Spesso i test di questo tipo sono vengono eseguiti facendo affidamento su un testing framework, ad esempio JUnit, che aiuta lo sviluppatore nella stesura e nell'esecuzione dei test stessi.

4.3 Sistema Scrum

Il sistema *Scrum* è un metodo di sviluppo basato più sul controllare lo sviluppo iterativo. È contruito su 3 fasi:

- Fase iniziale di pianificazione dove vengono specificati gli obiettivi del software e disegnata l'architettura del sistema;
- Fase di sviluppo, costituita da una serie di sprint iterativi che aggiungono sempre nuove feature al sistema;
- Al termine dello sviluppo, vengono scritti la documentazione e i manuali utente.

I vantaggi di un sistema Scrum sono dati dal fatto che il prodotto completo per essere sviluppato viene diviso in una serie di sotto-chunk, che vengono risolti man mano dagli sviluppatori durante ogni ciclo di lavoro.

4.4 Problematiche dello sviluppo agile

Le più grandi problematiche dello sviluppo agile sono le seguenti:

- I metodi di lavoro agili sono più adatti alla scrittura di nuovo software piuttosto che alla manutenzione di software precedente, e la maggior parte delle grandi compagnie si affida di più a software datato che viene continuamente aggiornato;
- I metodi sono concepiti più per lo sviluppo con pochi sviluppatori concentrati in un solo luogo, non per grandi compagnie, per le quali potrebbero risultare dispersivi approcci di questo tipo;
- Il sistema agile spesso è in conflitto con quello che è l'approccio dell'azienda al sistema di sviluppo del software;
- Il sistema agile è concentrato maggiormente sulla frequenza delle release che non sulla documentazione, cosa che può rendere problematica la manutenzione di software scritto con questo sistema o software di grandi dimensioni. Per questo motivo si preferisce più che altro usare un approccio plan-driven e agile (una sorta di metodo ibrido), anche per adattarsi a progetti di software con un ciclo di vita lungo (che hanno quindi bisogno di documentazione) o di grandi dimesioni (in quanto richiedono un numero di sviluppatori che può essere sparso per il mondo).

5 Ingegneria dei requisiti

Un requisito è una specifica che può andare dalla descrizione ad alto livello di un servizio fino ad un vincolo che il prodotto deve soddisfare. I requisiti si dividono in due macro-categorie:

- Requisiti utente: definiti in linguaggio naturale, specificano i servizi che il sistema deve fornire una volta sviluppato;
- Requisiti di sistema: sono indicati in un documento ben strutturato, che da una specifica dettagliata delle funzioni del sistema, dei servizi che offre e dei vincoli di funzionamento (di performance, ecc.).

5.1 Requisiti funzionali e non funzionali

I requisiti si possono suddividere inoltre in altre due grandi categorie:

- Requisiti funzionali: specificano quali servizi il sistema deve offrire, come il sistema dovrebbe reagire di fronte a determinati input, come dovrebbe comportarsi in determinate situazioni
- Requisiti non funzionali: costituiscono dei vincoli di sistema come ad esempio vincoli di tempo, vincoli di sviluppo ecc.

È importante notare che i requisiti funzionali potrebbero anche essere ambigui o venire male interpretati, in tal caso è bene, prima di iniziare lo sviluppo, fare un controllo di completezza e consistenza dei requisiti forniti dall'utente, per evitare errori di progettazione del software.

I requisiti funzionali invece non hanno bisogno di particolari tipi di analisi e sono classificati come segue:

- Requisiti del prodotto: specificano come il prodotto deve comportarsi;
- Requisiti organizzativi: requisiti derivati da politiche di organizzazione;
- Requisiti esterni: derivano da fattori esterni (legali, di interoperabilità...)

5.2 Processi di ingegneria dei requisiti

Il processo di ingegneria dei requisiti è concentrato su 4 punti, svolti iterativamente:

1. Estrazione dei requisiti;
2. Analisi dei requisiti;
3. Validazione dei requisiti;
4. Gestione dei requisiti.

5.2.1 Estrazione dei requisiti

È il processo con il quale si determinano i requisiti da implementare:

1. Scoperta dei requisiti: richiede interazione diretta col cliente;
2. Specifica dei requisiti: stesura dei documenti che specificano i requisiti di sistema e i requisiti utente, in modo che possano essere compresi anche dai clienti;

5.2.2 Validazione dei requisiti

È la "dimostrazione" che i requisiti analizzati sono esattamente quello che il cliente vuole dal team di sviluppo. La sezione della validazione è importante, in quanto permette di rilevare gli errori nell'analisi dei requisiti che costerebbero molto tempo per essere risolti in fase di sviluppo.

I punti da tenere sotto controllo sono i seguenti:

1. Validità: verifica che il sistema fornisca le funzioni che il cliente desidera;
2. Consistenza: controllo di eventuali conflitti di requisiti;
3. Completezza: verifica che tutte le funzioni desiderate dal cliente siano coperte;
4. Realismo: verifica che i requisiti specificati siano realizzabili con il budget e le tecnologie disponibili;
5. Verificabilità: descrizione di metodi per verificare i requisiti.

5.2.3 Gestione dei requisiti

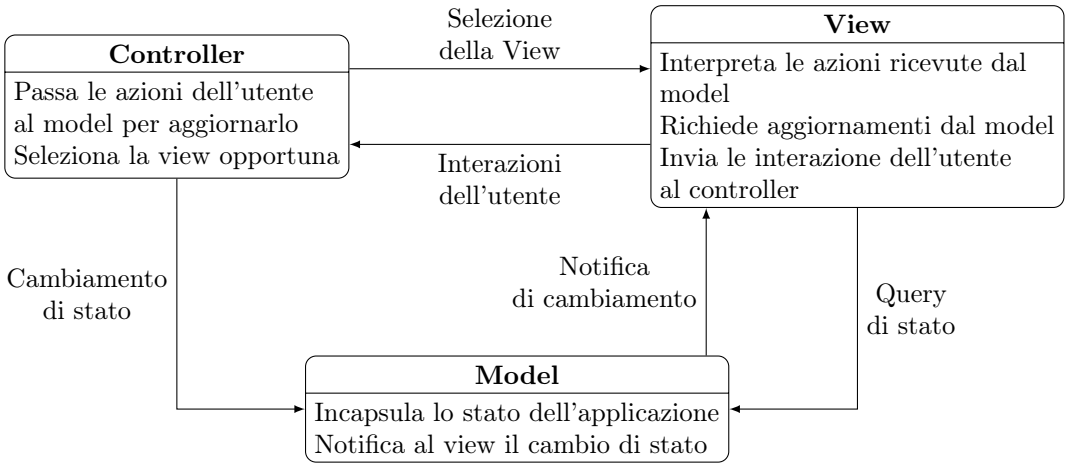
Questa parte del processo di ingegnerizzazione dei requisiti consiste nel definire delle politiche di gestione in caso di cambio di requisiti del sistema. Il processo di modifica dei requisiti passa attraverso un analisi dei costi di modifica del programma, un'analisi dei delle modifiche da apportare e poi viene avviata la modifica dell'implementazione del prodotto finito.

6 Pattern Architetturali

I pattern sono un mezzo per rappresentare condividere e riutilizzare le conoscenze software. Un pattern architetturale è una descrizione stilizzata di una buona pratica di progettazione che è stata provata e testata in diversi ambienti. I pattern dovrebbero includere informazioni su dove sono utili e dove quel dato pattern è da evitare. Possono essere rappresentati usando una descrizione a tabella oppure grafica.

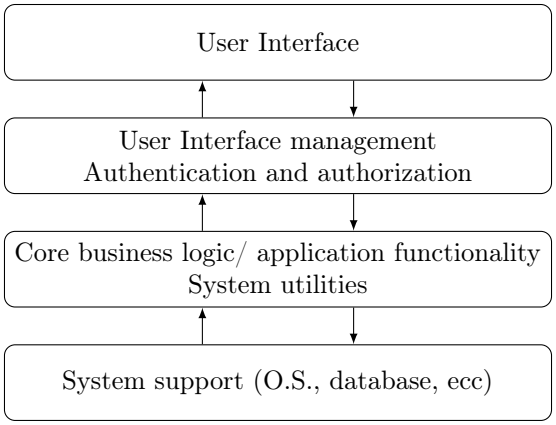
6.1 Model View Controller (MVC)

Separa i moduli di presentazione e interazione dai moduli che contengono i dati. Il sistema è strutturato in tre componenti logiche che interagiscono tra loro. Il **Model** si occupa della gestione dei dati e delle operazioni sui dati. La componente **View** definisce l'interfaccia e come i dati vengono presentati all'utente. Il **Controller** gestisce l'interazione dell'utente con questa interfaccia (key presses, mouse clicks, ecc) e passa queste interazioni al View e al Model.



6.2 Architettura a Strati (Layered)

L'architettura a strati è usata per gestire il modo in cui ci si interfaccia ai sottosistemi. Il sistema viene organizzato in un insieme di strati (layers o macchine astratte) ciascuno dei quali provvede ad un insieme di servizi. Le funzionalità di sono correlate di strato in strato. Ciascuno fornisce servizi allo strato sottostante, fino all'ultimo livello che rappresenta il nucleo delle funzionalità che facilmente verranno usate da tutto il sistema.

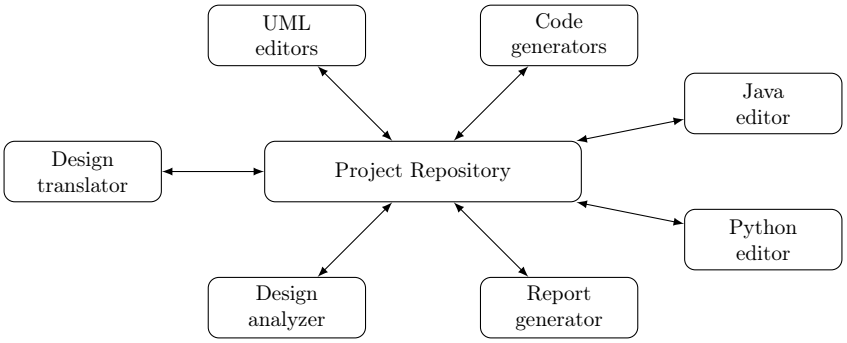


6.3 Architettura a repository

Repository: archivio in cui sono raccolti dati e informazioni, valorizzati earchiviati sulla base di metadati che ne permettono la rapida individuazione, anche grazie alla creazione di tabelle relazionali. Grazie alla sua peculiare architettura, un repository consente di gestire in modo ottimale anche grandi volumi di dati.
Nel caso in cui due sotto-sistemi debbano scambiarsi dati, solitamente ci sono due sistemi:

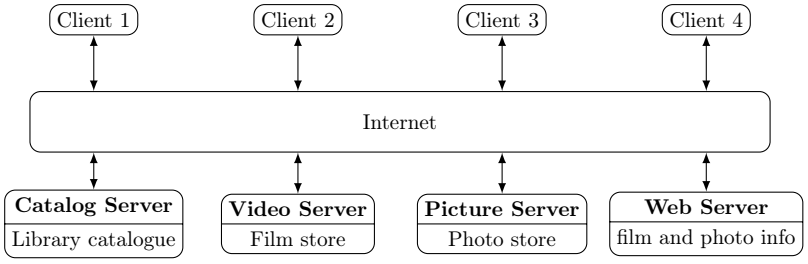
- I dati scambiati vengono mantenuti in un database centrale o in una repository e possono essere acceduti da tutti i sotto-sistemi
- Ogni sotto-sistema mantiene un database e passa i dati esplicitamente agli altri sotto-sistemi.

Quando c'è una grande quantità di dati da scambiare, il modello di scambio a repository è il più utilizzato e rappresenta un meccanismo molto efficiente di data sharing.



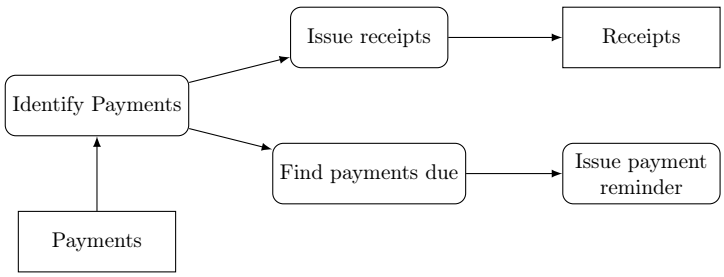
6.4 Architettura Client Server

Modello di sistema distribuito che presenta come i dati e l'elaborazione sono distribuiti attraverso un range di componenti. In questa architettura le funzionalità del sistema sono organizzate in servizi, con ogni servizio fornito da un server diverso. I Client sono gli utenti che possono accedere ai servizi attraverso i server.



6.5 Architettura Pipe and Filter

L'elaborazione dei dati nel sistema è organizzata in modo tale che ogni componente (**Filter**) del processo è discreto e genera in output un tipo di trasformazione dei dati. L'output di un componente è l'input del componente successivo (come in una **Pipe**).

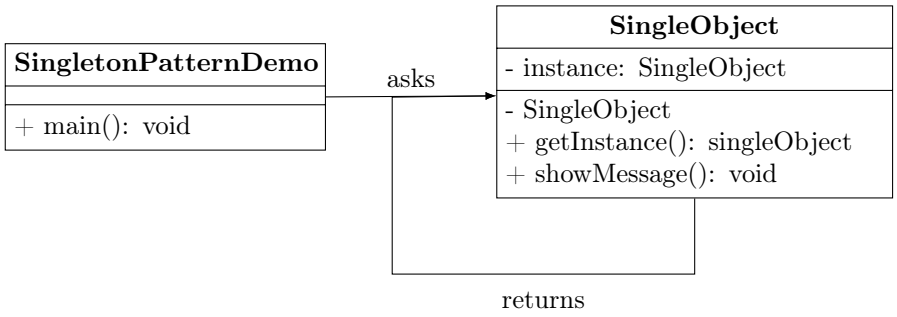


7 Pattern di progettazione

I pattern di progettazione costituiscono degli schemi di base per progettare alcune porzioni di software con specifiche caratteristiche. I principali sono:

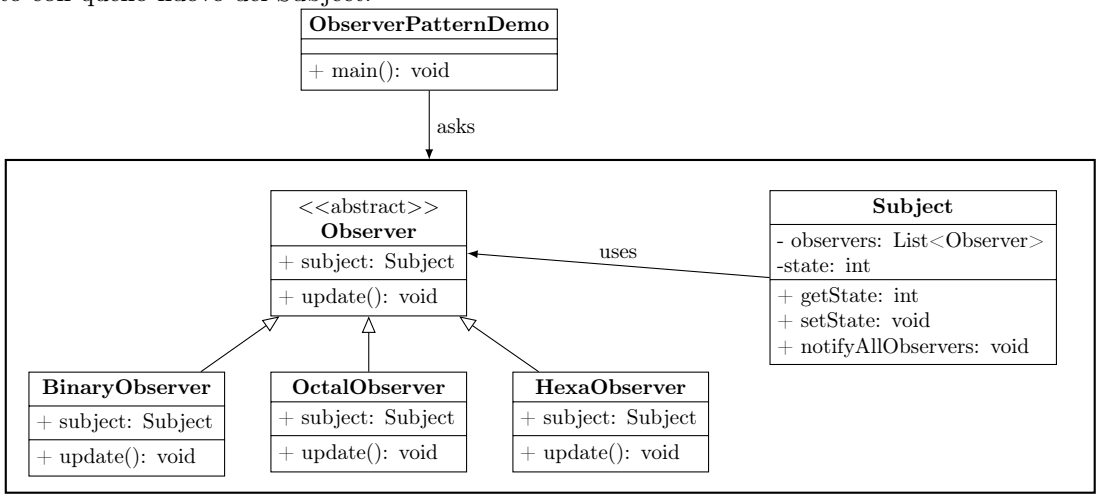
7.1 Singleton

In molte situazioni è necessario garantire l'esistenza di un solo oggetto di una classe.
Il Singleton è usato per assicurare che una classe abbia una sola istanza per tutto il ciclo di vita del software e un unico punto di accesso globale. Le classi Singleton hanno costruttori privati per evitare la possibilità di istanziare più oggetti della stessa classe. Esiste un metodo statico che fa in modo che nessuna istanza venga creata oltre alla prima restituendo un riferimento a quella esistente.



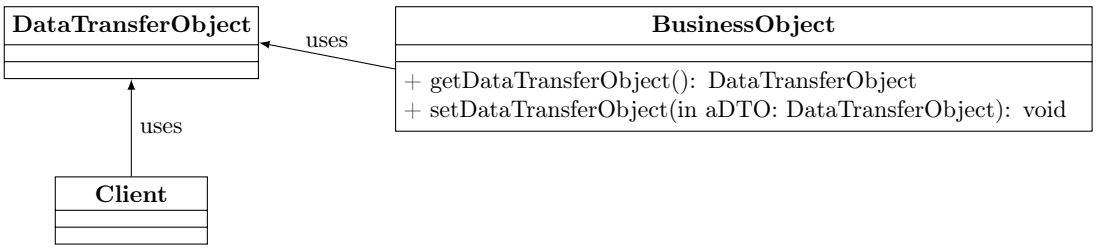
7.2 Observer

L'intento dell'Observer (o Publish/Subscribe) è quello di definire una dipendenza uno-a-molti tale che quando un oggetto (**Subject**) cambia il suo stato, tutti gli altri oggetti (**Observers**) che ne dipendono vengono notificati automaticamente e aggiornati di conseguenza. In risposta alla notifica gli Observers chiedono al Subject le informazioni necessarie per sincronizzare il proprio stato con quello nuovo del Subject.



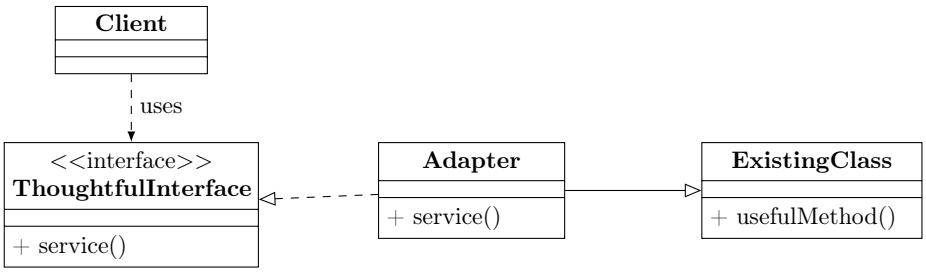
7.3 Data Transfer Object

Quando l'invocazione tra oggetti è **remota** è più conveniente permettere al client di gestire molti valori in un unica operazione piuttosto che avere invocazioni distinte.
Il pattern DTO è usato quindi per trasferire dati tra un client e un server e permette di gestire più oggetti con una singola invocazione piuttosto che singolarmente. L'oggetto DTO contiene un sottoinsieme dei dati del BusinessObject remoto, non ha metodi comportamentali e permette l'accesso diretto ai dati senza fare get/set.



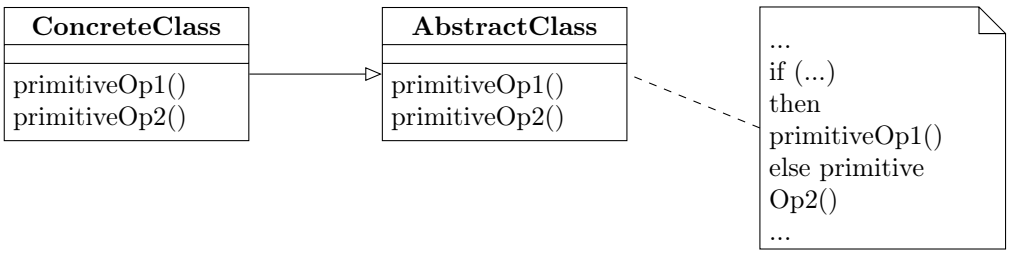
7.4 Adapter

L'adapter permette di fare da tramite tra due interfacce incompatibili. Converte un interfaccia di una classe in un'altra richiesta dal client, consentendo a diverse classi di operare insieme quando altrimenti ciò non sarebbe possibile. La classe Adapter implementa l'interfaccia, estende la classe esistente e ridefinisce il suo metodo chiamando quello richiesto.



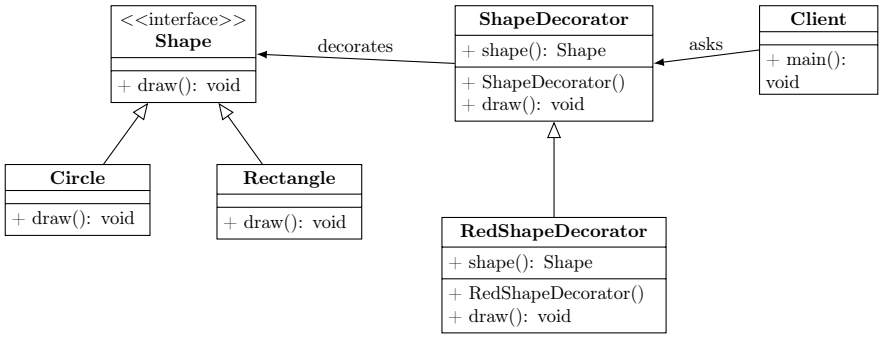
7.5 Template

Il pattern Template consiste nel definire la struttura di un algoritmo in una superclasse delegando alcuni passi dell'algoritmo a una o più sottoclassi. Le sottoclassi ridefiniscono alcuni passi dell'algoritmo senza dover implementare la struttura dell'algoritmo stesso (*hook*).



7.6 Decorator

Il pattern Decorator permette di aggiungere funzionalità ad un oggetto già esistente senza modificarne le proprietà o la sua struttura. Si crea una classe Decorator che fa da wrapper alla classe originale e aggiunge funzionalità mantenendo i metodi della classe intatti.

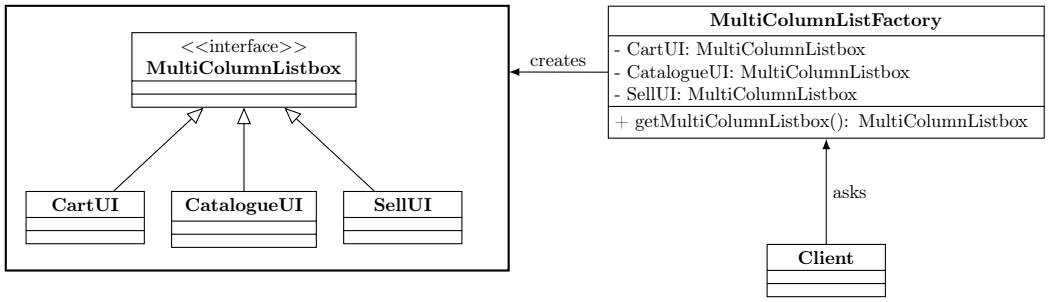


7.7 Facade

Il pattern Facade propone dei metodi che nascondono al client la complessità del sistema, poi la classe maschera andrà a chiamare i metodi più complessi all'interno del sistema.

7.8 Factory

Consiste nel creare un oggetto senza esporre la logica della creazione al client e si riferisce all'oggetto nuovo creato utilizzando un interfaccia comune.



7.9 Iterator

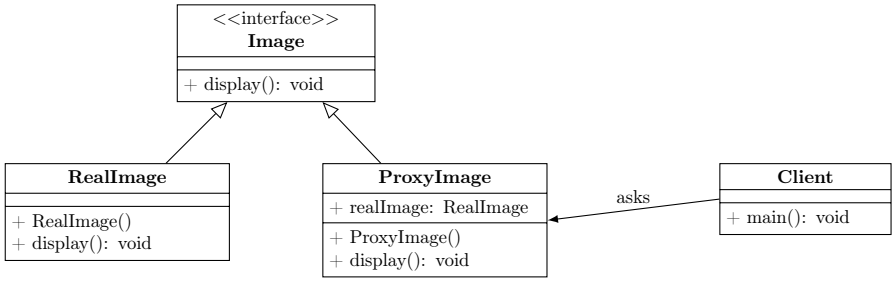
Questo pattern è usato per scorrere sequenzialmente gli elementi di una collezione senza dover conoscere la sua rappresentazione sottostante.

7.10 Abstract Factory

Il pattern abstract factory lavora attorno ad una super-factory che crea altre factory. È chiamato anche factory di factory. L'interfaccia crea una factory di oggetti senza specificare esplicitamente le loro classi.

7.11 Proxy

Una classe proxy rappresenta le funzionalità di un'altra classe. Creo l'oggetto originale che si interfaccia al mondo esterno.



8 Test del software

La fase di testing del software è pensata per dimostrare che il programma fa quanto richiesto e verificare che non ci siano difetti di sviluppo prima che il software venga rilasciato.

8.1 Test di sviluppo

Il test di sviluppo include tutte le attività di test che sono svolte dal team di sviluppo:

- Unit testing;
- Component testing;
- System testing.

8.1.1 Unit testing

Il test di unità è un processo che permette di sviluppare test individuali per ogni componente del sistema. È largamente usato per trovare difetti nei singoli componenti delle classi, che sarebbe più difficile individuare durante i test finali, specialmente se il software sviluppato è di grandi dimensioni.

In caso sia possibile, è bene automatizzare lo unit testing per affidare alla macchina il controllo dei risultati e per evitare di eseguire moltissime istanze di test manualmente. Per automatizzare le fasi di unit testing vengono usati dei test framework (es. JUnit per Java), che implementano classi di test specifiche in cui assemblare i singoli test.

Ogni metodo di test è suddiviso in 3 fasi:

- Setup del test, in cui vengono creati gli oggetti necessari allo svolgimento del test;
- Chiamata del metodo da testare;
- Generazione dell'asserzione, ovvero nel creare un'ipotesi di risultato che poi viene verificata in fase di esecuzione del test.

8.1.2 Test dei componenti

Consiste nel testare singolarmente funzionalità del programma, in modo da assicurarsi di avere una feature completamente funzionante prima di procedere allo sviluppo di altre parti del prodotto.

8.1.3 Test del sistema

Il test di sistema comprende il test di integrazione delle componenti, in modo da verificare eventuali incompatibilità o conflitti tra blocchi di codice.

8.2 Sviluppo test-driven

Lo sviluppo test driven è un metodo di testing che interlaccia test del sistema e sviluppo del codice, e consiste nello sviluppare una feature e poi testarla immediatamente, bloccando l'incremento dello sviluppo fino a quando i test non siano passati.

I benefici dello sviluppo test-driven sono:

- Ogni segmento di codice scritto ha almeno un test correlato;
- La test-suite è sviluppata incrementalmente;
- Il debug è semplificato, perchè quando un test fallisce è scontato verificare dove sta il problema;
- I test costituiscono già di per sè una forma di documentazione.

8.3 Release test

Il test di release è un particolare tipo di test pensato per un utilizzo esterno rispetto a quello del team di sviluppo. L'approccio usato solitamente è quello a black-box, in quanto il test è portato avanti guardando unicamente se vengono sviluppati correttamente i requisiti.

8.4 User test

Il test per l'utente è portato avanti soprattutto per verificare che il sistema sia usabile da utenti senza particolari problemi.

- Alpha test: gli utenti lavorano con il team per verificare l'utilizzo del software;
- Beta test: una verione è rilasciata agli utenti per verificare il software e segnalare problemi;
- Acceptance test: il software è usato dall'utente per verificare se può definitivamente essere rilasciato.

9 Evoluzione del software

Far evolvere il software è necessario, in risposta a cambiamenti di requisiti, cambio di tecnologie, ecc.

Bisogna innanzitutto separare l'aspetto di evoluzione da quello di servicing: l'evoluzione è effettuata quando sono necessarie modifiche all'implementazione del software in risposta a modifiche nelle specifiche dei requisiti, il servicing è la fase in cui il software viene modificato per aspetti puramente operazionali, quali bugfix, patch di vulnerabilità, ecc. Quando il software viene ritirato può rimanere usabile ma non essere più mantenuto.

Quando un software deve essere modificato deve essere svolta nuovamente l'analisi dei nuovi requisiti, deve essere fatto un nuovo piano e devono essere reimplementate o modificate le funzionalità coinvolte. L'unico problema che può nascere durante una fase evolutiva di un software può essere quello dell'approccio, ovvero quando due team (quello di implementazione e quello di evoluzione) hanno due sistemi di lavoro diversi (agile vs plan-driven).

9.1 Sistemi legacy

Una trattazione particolare va riservata ai sistemi *legacy*, ovvero quei sistemi che si appoggiano su un software o tecnologia vecchia, ma il cui strato superficiale viene continuamente adattato alle nuove tecnologie (ad esempio, molte banche usano ancora sistemi Cobol abbastanza vecchi, perchè la migrazione a sistemi più recenti è molto costosa e quindi poco conveniente.)

I componenti principali di un sistema legacy sono:

- Hardware del sistema: probabilmente obsoleto;
- Software di supporto: probabilmente obsoleto, vecchio e non più mantenuto;
- Application software/data: l'insieme di applicazioni fornite all'azienda titolare e i dati sui quali le applicazioni lavorano;
- Livello business: insieme di politiche, regole e processi che sono portati avanti dall'azienda titolare per ottenere i suoi obiettivi.

I sistemi legacy possono essere costosi da cambiare per motivi di documentazione, difficoltà di comprensione causata dalla complessità delle ottimizzazioni oppure a causa di errori o ridondanze nei dati. Le aziende che sono coinvolte in un utilizzo di sistemi legacy possono:

- Cambiare il sistema e riadattare i loro processi business;
- Continuare a mantenere il sistema;
- Migliorare la mantenibilità del sistema reingegnerizzandolo;
- Rimpiazzare il sistema.

9.2 Mantenimento del software

Ci sono 3 tipi di mantenimento del software:

- Bug repair: il software viene patchato a causa di bug trovati durante l'utilizzo;
- Adattamento dell'ambiente: il software viene adattato ai nuovi sistemi operativi o ai nuovi hardware;
- Modifica delle funzionalità: il software viene modificato per soddisfare nuove esigenze.

Il costo di mantenimento del software è solitamente maggiore del costo di sviluppo, e aumenta con l'aumentare della complessità del software.

9.2.1 Reingegnerizzazione

Consiste nel ristrutturare parte o tutto il codice di un sistema legacy senza modificare le sue funzionalità. Potrebbe inoltre essere necessario ristrutturare e ridocumentare il software. I principali vantaggi di reingegnerizzare il software sono:

- Riduzione del rischio, in quanto viene toccata una parte sola del codice;
- Riduzione del costo, dato dal fatto che il costo di reingegnerizzazione è sempre minore del costo di sviluppo.

Il processo di reingegnerizzazione solitamente è composto da:

- Analisi del codice da modificare;
- Miglioramento della struttura del programma;
- Modularizzazione del programma;
- Reingegnerizzazione del sistema.

10 Credits

Codice tex disponibile su https://github.com/alx79/dispense-info-univr. Questa dispensa è stata scritta da:

- Davide Bianchi (davideb1912@gmail.com)
- Matteo Danzi (matteodanziguitarman@hotmail.it)