INGEGNERIA DEL SOFTWARE

Alessandro Pioggia

22 gennaio 2022

Indice

Capitolo 1

Analisi dei requisiti

L'analisi dei requisiti è la fase che permette, attraverso la modellazione della realtà, di redigere la specifiche dei requisiti, documento che rappresenterà l'input per le successive fasi di progettazione. Per rendere possibile la creazione un coadiuvato è necessario, da parte dell'analista, intervistare il produttore, cercando di assimilare al meglio tutte le informazioni riguardanti il dominio applicativo.
In questa fase è necessario fare in modo che il documento sia chiaro, non ambiguo, accessibile e privo di contraddizioni.
L'analisi è incrementale e deve comunicare gli aspetti statici, dinamici e funzionali del progetto software.

1.1 Analisi orientata agli oggetti

In questa tipologia di analisi vengono curati principalmente gli aspetti statici, ovvero vengono definiti gli oggetti e le relazioni presenti fra essi. Definire gli oggetti significa individuare tutte le informazioni e proprietà ad essi connesse, esse tendono a rimanere invariate nel tempo, per questo viene definito un approccio **statico**.

1.2 Analisi funzionale

L'analisi funzionale si pone il problema di definire le specifiche prendendo come riferimento le funzioni, ovvero viene studiata solo ed esclusivamente la relazione presente fra dati in ingresso e dati in uscita. Vengono presi in considerazione i flussi informativi, che verranno poi modificati da processi.

1.3 Analisi orientata agli stati

L'analisi orientata agli stati analizza i vari stati evolutivi del prodotto, in funzione di ciò, vengono analizzati i comportamenti e successivamente redatta la documentazione delle specifiche dei requisiti.

1.4 Astrazione

I meccanismi di astrazione utilizzati sono:

- classificazione

- ereditarietà (is-a)
- aggregazione (part-of)
- associazione

1.5 Linguaggi utilizzati per la specifica dei requisiti

- informale (linguaggio naturale, molto contradditorio e poco chiaro, sconsigliato)
- semiformale (utilizzo di diagrammi secondo un preciso standard, che però è semplice ed intuitivo (e/r, dfd))
- formale (linguaggio tecnico, difficile da comprendere e inutilmente complicato, indipendentemente dal contesto)

Una volta definito il tipo di linguaggio da utilizzare è necessario effettuare una distinzione fra formalismi dichiarativi e operazionali. I primi definiscono il problema indicando le proprietà che esso deve avere, i secondi ne descrivono il comportamento, nella maggior parte dei casi attraverso un modello. I formalismi operazionali sono i più indicati in questo contesto, in quanto più comprensibili e modellabili.

Capitolo 2

Progettazione

Se la fase di analisi vuole sincerarsi sul "che cosa" sviluppare, nella parte di progettazione ci si chiede "come" svilupparlo, dunque si ha una sorta di ponte fra analisi e codifica.

2.1 Modalità

Nella progettazione si suddivide il problema iniziale in più sottoproblemi, il più possibile indipendenti fra loro, questo dà la possibilità di fare gestire il lavoro a più team, anche in parallelo. Una fase di progettazione ben realizzata consente un notevole risparmio di risorse, dal momento che un eventuale errore in fase di produzione o peggio, durante la manutenzione comporta un costo molto più elevato per l'azienda.

2.2 Approccio

L'approccio varia in base alla strategia adottata, ne conosciamo due:

- generale
- specifico

Il primo permette di mantenere una buona elasticità, può essere modellato anche nelle fasi successive. Il secondo invece è rigido, non contempla modifiche nelle fasi successive, questo semplifica il passaggio dalla progettazione alla codifica.

2.3 Il buon progettista

Il buon progettista è colui che ha una buona conoscenza di tutto ciò che riguarda lo sviluppo software, sa anticipare i cambiamenti ed ha un buon grado di esperienza. Inoltre, è colui che si pone come obiettivi della progettazione l'affidabilità, la modificabilità, la comprensibilità e la riusabilità del software.

Capitolo 3

Paradigma ad oggetti

La programmazione OO offre un nuovo e potente modello per scrivere programmi, gli oggetti sono scatole nere che mandano e ricevono messaggi. Questo sistema velocizza l'approccio software e migliora la riusabilità, la modificabilità e il mantenimento, richiedendo però un maggiore sforzo in fase di progettazione. I concetti fondamentali sono : oggetto, astrazione, classe, incapsulamento, ereditarietà, polimorfismo, late-binding e delegazione.

3.1 Oggetto

Un oggetto è una entità del dominio applicativo caratterizzata da un identificatore (unico), delle proprietà (attributi) e da un comportamento (definito da un insieme di operazioni). Gli oggetti possono essere aggregati fra loro per poter poi formare oggetti complessi.

3.2 Operazione

Ogni operazione è caratterizzata da un nome, dai parametri che prende come argomento e dal tipo di ritorno (signature). L'insieme di tutte le signature delle operazioni di un oggetto sono dette interfaccia dell'oggetto.

3.3 Classe

Una classe è la realizzazione di un tipo di dati astratto (tipo di dati astratto : insieme di oggetti simili, caratterizzati da dati e da un insieme di operazioni associate agli oggetti), cioè una implementazione per i metodi ad esso associati.

3.4 Incapsulamento

L'incapsulamento permette di **proteggere** dati e implementazione delle operazioni di un oggetto, attraverso una interfaccia pubblica. L'unico modo per modificare lo stato dell'oggetto (o ottenere informazioni riguardanti gli attributi) è attraverso l'utilizzo dei metodi presenti nell'interfaccia. Questo meccanismo favorisce la diminuzione della quantità di errori commessi (es:modifica di uno stato manualmente), una maggiore sicurezza, l'utilizzo (per utilizzare una classe è sufficiente conoscerne l'interfaccia pubblica) ed infine la modifica dell'implementazione di un metodo di una classe (non si ripercuote sull'applicazione).

3.5 Metodi

Un metodo cattura l'implementazione di una operazione, possono essere classificati in: costruttori, distruttori, accessori e trasformatori. Un metodo può essere o privato, o pubblico o protetto.

3.6 Ereditarietà

Il meccanismo di ereditarietà permette di basare la definizione dell'implementazione di una classe su quella di altre classi. Una classe può **ereditare** da una superclasse ed essere **estesa** da una sottoclasse. Le sottoclassi ereditano attributi e metodi della superclasse.
In certi contesti è ammessa l'ereditarietà multipla, fenomeno per il quale una sottoclasse eredita **contemporaneamente** da due superclassi.

3.7 Polimorfismo

Per polimorfismo si intende la capacità di assumere forme molteplici, si verifica quando:

- in una classe posso definire due metodi con stessa intestazione, a patto che abbiano una diversa signature (parametri in ingresso)
- posso ridefinire un metodo attraverso il meccanismo di override

3.8 Late binding

Il late binding, o istanziamento dinamico, permette a ciascun oggetto di rispondere a uno stesso messaggio in modo appropriato a seconda della classe da cui deriva.

3.9 Delegazione

Si parla di delegazione quando un oggetto A contiene un riferimento ad un oggetto B, in questo modo A può delegare delle funzioni alla classe a cui appartiene B. Questo è il meccanismo fondamentale per quanto riguarda l'implementazione dell'associazione fra classi.

Capitolo 4

Ingegneria del software

L'ingegneria del software è la disciplina che, attraverso un approccio sistematico (tecnico e preciso), si pone come obiettivo quello di gestire l'operabilità, la manutenzione e l'eventuale ritiro dal commercio del software. Questa disciplina, oltre all'aspetto tecnico, comprende un aspetto manageriale, il quale si occupa della gestione dei flussi, dei tempi e delle risorse dell'azienda.

4.1 Qualità del software

Un software che funzioni non è sufficiente, è necessario che sia di qualità, esse possono essere:

- interne (visibili solamente dagli sviluppatori)
- esterne (visibili dall'utente finale)
- di prodotto (guardano la qualità del prodotto)
- di processo (guardano la qualità del processo produttivo)

4.1.1 Qualità esterne, di prodotto e di processo

Robustezza (si comporta bene anche su aspetti non menzionati nelle specifiche di progetto)

4.1.2 Qualità esterne e di prodotto

Correttezza (funziona?), affidabilità (un software è affidabile se posso dipendere da esso), Efficienza (tempi e prestazioni), Facilità d'uso, portabilità e interoperabilità (utilizzo di software unito ad altri, ad esempio word e excel).

4.1.3 Qualità interne, di prodotto e di processo

Verificabilità (il sw è certificabile, si osserva la correttezza, performance, ecc.)

4.1.4 Qualità interne e di prodotto

Riusabilità e facilità di manutenzione (aperto a modifiche correttive, perfettive ed adattive).

4.1.5 Qualità esterne e di processo

Produttività, tempestività e trasparenza.

4.2 Software design

Il sw design rappresenta il processo che trasforma le specifiche utente in un insieme di specifiche utilizzabili dai programmatori, il risultato del processo di design è l'architettura software.

4.2.1 Principi di progettazione (good practices)

- **Formalità** (suggerisce l'uso di formalismi, ad esempio uml)
- **Anticipazione dei cambiamenti** (un sw robusto deve essere aperto ai cambiamenti, che siano noti o meno a priori. Le modifiche possono riguardare : hw, sw, algoritmi e dominio di applicazione)
- **Separazione degli argomenti** (suggerisce di spezzare il problema in sottoproblemi, in funzione del: tempo, livello di qualità (approccio incrementale), vista (in fase di analisi curare aspetti statici, dinamici e funzionali), livello di astrazione e dimensione)
- **Modularità** (divisione del sistema in moduli, un modulo è un componente di base di un sistema software, che permette di scomporre un sistema complesso in più componenti semplici. Tutti i servizi strettamente connessi devono appartenere allo stesso modulo e inoltre devono essere indipendenti fra loro, i programmatori devono poter operare su un modulo con una conoscenza minima del contenuto degli altri. Un modulo inoltre deve definire una interfaccia, la quale mostra le informazioni relative ad esso ma nasconde l'implementazione, si tratta di information hiding, dunque è suff. indicare una linea guida su come utilizzare i servizi. Essi interagiscono fra loro, attraverso le dipendenze [uses], composizione [part-of] e in base al tempo.)
- **Astrazione**
- **Generalità** (Ogni volta che occorre risolvere un problema, si cerca di capire quale è il problema più generale che vi si nasconde dietro.)

4.3 Misurazione

La misurazione si occupa di prevedere e stimare tempi di consegna e qualità di lavorazione del software. Non è una misurazione esatta, in quanto è molto difficile quantificare, si tratta di una stima, eseguibile attraverso strumenti di diverso genere.

4.3.1 Fasi in cui è opportuno eseguire la stima

Scopi:
La stima può avvenire in diversi stadi di lavorazione, può essere utile per prevedere le caratteristiche che avrà il software in una fase successiva alla valutazione (utile per definire azioni correttive) oppure per stimare le caratteristiche nello stadio attuale, utile per capire il costo adeguato di un software.
Fasi
Bisogna inoltre definire in quale momento effettuare la misurazione, essa può avvenire in:

- fase di progettazione (serve a prevedere la manutenibilità e prevenire problemi)
- fase di collaudo/test (eseguire un confronto con le specifiche)
- fase dopo il rilascio in esercizio (serve a misurare l'impatto del software, in modo da assegnargli eventualmente un costo adeguato)

4.3.2 Costi

Prima di stimare i costi occorre effettuare una classificazione, dividendo le fonti di costo (costi per le attività generatrici) dai fattori di costo (hanno incidenza sul costo). **Fonti di costo**

- Costo delle risorse per lo sviluppo del sw (costo diretto)
  - personale tecnico
  - personale di supporto (bidelli, donne delle pulizie, manutentori)
  - risorse informatiche
  - materiali di consumo
  - costi generali della struttura
- Costo per l'indisponibilità di un'applicazione (costo indiretto)

Fattori di costo

- Lines of code (LOC)
- bravura del team (un team bravo, anche se lo pago molto, mi fa risparmiare sul processo produttivo)
- complessità del programma
- stabilità dei requisiti (se cambiano idea spesso è difficile costruire un sw solido e in poco tempo)
- caratteristiche dell'ambiente di sviluppo

4.3.3 Dimensioni del software

Le dimensioni del software si possono stimare in funzione di due tipi di metriche:

- dimensionali (si basano sul numero di istruzioni del programma)
- funzionali (si basano sulle caratteristiche funzionali del programma)

4.3.4 Il metodo function points

Il metodo FP è empirico (basato sull'esperienza) e misura la dimensione di un sw in termini delle funzionalità offerte all'utente. Può essere utilizzato a partire dalla prima fase dello sviluppo per poi ripetere la misura nel caso le specifiche siano cambiate, è inoltre indipendente dall'ambiente tecnologico in cui si sviluppa il progetto.

Può essere utilizzato per:

- capire che beneficio un sw porterà alla mia organizzazione
- controllare che non si ladri alla vendita di un sw, c'è un fattore di proporzionalità (non posso vendere un sw di hello kitty a 3 milioni di euro)

4.3.5 Conteggio dei function points

In ordine:

- individuazione del tipo di conteggio
  - sviluppo software o manutenzione?
- individuazione dei confini
  - tutto il sistema o una sola componente?
- conteggio dei FP non pesati
  - funzioni di tipo dati
    - file interni logici (informazioni tenute all'interno dei confini dell'applicazione)
    - file esterni di interfaccia (dati a disposizione dell'applicazione ma mantenuti dentro i confini di un'altra applicazione)
  - funzioni di tipo transizione
    - input esterno (dati provenienti dall'esterno elaborati all'interno dei confini dell'applicazione, es: un form)
    - output esterno (manda dati di controllo fuori dai confini dell'applicazione, richiede almeno una formula matematica, es:una stampa)
    - interrogazioni esterne (manda dati o informazioni fuori dai confini dell'applicazione, non richiede una formula matematica)
- calcolo dei FP pesati, in funzione del fattore di aggiustamento(varia da 0.65 a 1.35, con incidenza max sul totale del 35%)

fattore di influenza $= 0.65 + (TDI \cdot 0.01)$
Il TDI è il total degree of influence e somma i gradi per diverse caratteristiche (es : riusabilità, comunicazione dati, prestazioni, facilità di installazione, complessità elaborativa, ecc.) ed assume valori fra 0 e 5.

4.4 Il numero ciclomatico

Mentre il calcolo dei FP è empirico, in questo contesto siamo teorici e molto specifici, in particolare si quantifica la complessità del flusso di controllo (cicli, if, switch, ecc.). Il numero ciclomatico cattura solo in parte ciò che è la complessità del flusso di controllo. Il numero ciclomatico viene calcolato su un grafo fortemente connesso (ogni nodo può raggiungere indirettamente qualsiasi altro nodo). In termini di conti, il suo valore è determinabile attraverso:

- $e - n + 1$ (se il grafo è già fortemente connesso)
- $e - n + 2$ (se ho un grafo normale, lo posso trasformare in fortemente connesso collegando il nodo iniziale al nodo terminale)
- $e - n + 2p$ (se il programma ha procedure al suo interno, il numero ciclomatico dell'intero grafo è dato dalla somma dei numeri ciclomatici dei singoli grafi indipendenti. p è il numero di grafi (procedure) indipendenti)

4.5 Cocomo (COnstructive COst MOdel)

Si calcola una stima iniziale dei costi di sviluppo in base alla dimensione del software da produrre, poi la si migliora sulla base di un insieme di parametri.

4.5.1 Stima della dimensione del software

Questa stima la si opera contando il numero di linee di codice scritte (KDSI), il conto lo si può fare sfruttando i FP, assegnati diversamente in funzione del linguaggio.

4.5.2 Determinazione della classe del software

Ci sono differenti classi software, determinate in funzione del tempo e personale richiesto, ogni classe presenta un calcolo diverso per la determinazione del costo.

- Organic $M_{nom} = 3.2 \cdot KDSI$
- Semi-detatched $M_{nom} = 3 \cdot KDSI$
- Embedded $M_{nom} = 2.8 \cdot KDSI$

4.5.3 Applicazione degli estimatori di costo

Gli estimatori di costo c_i sono dei coefficienti che sono determinati in funzione di :

- Proprietà del prodotto (affidabilità, complessità, ecc.)
- Caratteristiche hardware (efficienza, memoria, ecc.)
- Caratteristiche del team (esperienza e capacità del team)
- Caratteristiche del progetto (Modernità del processo di sviluppo, ecc.)

La formula esatta è la seguente:

$M = M_{nom} \cdot \prod_{i=1}^{15} c_i$

4.6 Produzione

Il processo di produzione è la sequenza di operazioni che viene seguita per costruire, consegnare e modificare un prodotto.

4.6.1 Modelli di processo prescrittivi

Si pongono come obiettivo quello di organizzare e dare una struttura a ciò che altrimenti risulterebbe caotico. Tutti i modelli prescrittivi prevedono una serie di attività da seguire, ovvero:

- comunicazione (raccolta requisiti)
- pianificazione (stima)
- modellazione (analisi e progettazione)
- costruzione (programmazione e testing)
- deployment (consegna, supporto e feedback)

4.6.2 Modello a cascata

Il modello a cascata è **rigido**, poco flessibile, non prevede il concetto di "torno indietro" (impossibile modificare i risultati delle fasi precedenti). Il vantaggio è che alla fine del processo porta ad un prodotto funzionante. Questo modello è consigliato nel caso in cui le specifiche siano note a priori, per ovvi motivi.

4.6.3 Modello incrementale

Il modello incrementale consiste nella creazione di un software funzionalità per funzionalità, fino a terminarlo. Si suddivide in più stadi operativi, al primo stadio viene generato un prodotto base, dopodichè, in seguito ad una valutazione utente si imposta uno stadio successivo, il quale aggiunge ulteriori funzionalità, rendendolo un prodotto più complesso. Questo modello è consigliato se i requisiti sono noti a priori ma le dimensioni finali del software non sono chiare.

4.6.4 Modello RAD

Il modello RAD è incrementale e punta ad uno sviluppo in breve termine, permesso grazie alla modularizzazione in componenti (ogni team si occupa di una componente). Una applicazione è adatta al RAD se è possibile sviluppare in 3 mesi max tutte le funzionalità base, rendendola già funzionante. Non è adatto se gli utenti non sono attivi e disponibili alla comunicazione, nemmeno se sono richieste alte prestazioni da ottenere tramite l'ottimizzazione delle interfacce tra i componenti.

4.6.5 Modello evolutivo

Il modello evolutivo prevede la creazione di una prima versione sw, sulla base delle specifiche note a priori, dopodichè si realizzano delle estensioni. Le estensioni possono essere create attraverso un modello a prototipi (incremento progressivo di un prototipo) o a spirale (fa crescere incrementalmente il grado di definizione del sistema). Questo modello è consigliato nel caso in cui i requisiti non sono chiarissimi e cambiano col tempo, sono modelli iterativi.

4.6.6 Model driven development

MDD ha come idea quella di automatizzare il processo di sviluppo, infatti prevede, in seguito alla fase di analisi e progettazione, l'utilizzo di tool per la generazione automatica di codice.

4.6.7 Modelli agile

I modelli agile, a differenza di quelli visti fino ad ora non sono prescrittivi, la loro filosofia si racchiude nei seguenti punti:

- cliente al 1° posto
- team di sviluppo molto motivato
- abbattimento formalismi (poca ingegneria del software)
- semplicità di sviluppo

Il prerequisito è la continua comunicazione con l'utente, se manca, non ha senso adottare un modello agile.

4.6.8 Extreme programming

Modello agile più diffuso, include 4 attività procedurali:

- Pianificazione
  - Definizione di user story, ovvero micro-funzionalità del sistema viste dal punto di vista dell'utente, devono essere sviluppabili in non più di 3 settimane. Se la funzionalità proposta dall'utente è troppo grande, si richiede di frammentarla, per rientrare nelle 3 settimane
- Design
  - Persegue la massima semplicità, incoraggia l'uso di CRC e il refactoring, in caso di problema di design si creano spike solution, prototipi usa e getta con 1 funzionalità.
- Programmazione
  - pair programming, due sviluppatori per ogni postazione, 4 occhi sono meglio di 2.
- Testing
  - Unit test e test di regressione (ogni volta che aggiungo una nuova funzionalità testo anche le precedenti, in modo da evitare il butterfly effect).

4.6.9 Test Driven Developement

Tecnica di sviluppo che consiste nel definire i test automatici ancora prima di programmare. Questo approccio diminuisce i tempi totali di sviluppo, grazie alla possibilità di trovare errori in maniera immediata, inoltre il TDD comporta una struttura del codice modulare, flessibile e modificabile. Concetto cardine : mock objects, pseudo-oggetti che si comportano come oggetti reali ma che ne riproducono solo i risultati, per facilitare la fase di test.

4.7 Prototipazione

Un prototipo software permette di animare e dimostrare i requisiti, l'uso principale consiste nell'aiutare i clienti e gli sviluppatori a capire i requisiti del sistema.

4.7.1 Tecniche di prototipazione

- prototipazione evolutiva
  - L'obiettivo è di fornire un sistema funzionante all'utente, parte come uno sketch, che ad ogni iterazione migliora, fino ad arrivare al prodotto finito.
  - Problemi: è presupposto un modello a cascata, inoltre cambi continui corromperanno il sistema sul lungo termine e sono richieste capacità di progettazione e programmazione non indifferenti.
- prototipazione usa e getta
  - Serve per validare i requisiti del sistema, non è pensato per essere un punto di partenza. Si fa sui requisiti non capiti, sapendo che poi, alla fine, il prototipo verrà buttato. Deve essere usato per sistemi in cui le specifiche non possono essere sviluppate in anticipo, es: interfacce grafiche.

4.8 Verifica del software

La fase di verifica del software ha lo scopo di controllare se il sistema realizzato risponde alle specifiche di progetto. Nel caso delle tecniche dinamiche il sw viene messo in esecuzione, mentre in quelle statiche no.

4.8.1 Testing

Non c'è modo di dimostrare formalmente la correttezza del software, dal momento che, dato un sw ho un numero infinito di possibili combinazioni di output. Per testare occorre trovare un numero di operazioni(non troppe, per non far sparare in alto il prezzo) da eseguire, vanno scelte correttamente, per gestire il maggior numero di casi possibili.

Testing in the small

Si tratta di tecniche di testing white-box, dunque per fare il test devo sapere come è fatto dentro l'algoritmo. Torna utile schematizzare il processo con un grafo di controllo. L'idea è di dare dei criteri di copertura che sono progressivamente più ampi, lo scopo è di far eseguire ogni operazione almeno una volta.

Statement-test(Criterio di copertura dei programmi) il senso è : "se ho una istruzione in un codice che non viene mai raggiunta, non posso capire se funziona o meno, quindi un caso prova può essere quello che mette in esecuzione qualsiasi istruzione almeno una volta, indipendentemente dai controlli che faccio". Il criterio degli statement non permette di trovare tutti i bug.

Branch-test (Criterio di copertura delle decisioni) il branch test fa in modo che ogni arco del grafo di controllo venga attraversato almeno una volta (di conseguenza viene rispettato anche il criterio dello statement-test). Anche il branch-test non ci aiuta più di tanto, il problema nasce da una condizione composta, prendiamo un ramo o l'altro del grafo di controllo, non ci siamo posti il problema di porre una volta vera una condizione dell'if e una volta l'altra(else).

Criterio di copertura delle decisioni e delle condizioni considera sia il true che il false di ogni predicato, risolvendo il problema nato nel secondo criterio. Anche in questo caso non ci dà garanzia di trovare l'errore. Quando dico quali criteri ho usato posso dare una garanzia riguardo quanto sono andato profondo con il testing del programma.

Testing in the large

Il numero elevato di possibili combinazioni delle porzioni di codice di un sistema di grandi dimensioni rende impossibile l'utilizzo di tecniche white-box. Si rende necessario l'utilizzo di tecniche black-box, in cui il sistema è considerato una scatola nera, nel quale si valuta il funzionamento sulla base della corrispondenza input-output. Devo selezionare i casi prova significativi nei casi di funzionamento normale. Lo schema uml che si può usare nel testing in the large è il diagramma dei casi d'uso, se un caso d'uso non lo testo non posso avere la garanzia del corretto funzionamento. I test possono essere fatti su livelli diversi :

- modulo (unit-test, verifica se un modulo si comporta correttamente in base al suo comportamento esterno)
- integrazione (verifica il comportamento di sottoparti del sistema sulla base del comportamento esterno)

- sistema (verifico il comportamento dell'interno sistema sulla base del suo comportamento esterno)

Il beta-test è un test di sistema, dal momento che è fatto dall'utente, il quale non conosce la modularità del sistema.

4.8.2 Analisi del software

L'idea è di ispezionare il sistema e metterlo mentalmente in esecuzione per capire se ci sono errori o no. Ci sono due sottofamiglie.

Code walk-through Senza avere in mente nessuna tipologia di errore, viene messo in funzionamento (nella mente) il programma, cercando l'errore.

Code inspection Si cerca invece un errore specifico, sono dunque alla ricerca di una particolare classe d'errore(Uso di variabili non inizializzate, loop infiniti). Fra le tecniche di code inspection c'è l'analisi di flusso di dati. Lo scopo è di analizzare staticamente qualcosa che dovrebbe essere dinamico, ovvero l'insieme di istruzioni di un programma, ad ogni comando associo il tipo di operazione eseguito sulle variabili (definizione = d, uso = u, annullamento = a). L'analisi è focalizzata sul controllo di una variabile alla volta. Le regole dell'analisi di flusso di dati sono 2:

- L'uso di una variabile x deve essere sempre essere preceduto in ogni sequenza da una definizione della stessa variabile x, senza annullamenti intermedi
- Una definizione di una variabile x deve sempre essere seguita da un uso della variabile x, prima di un'altra definizione o di un annullamento della stessa variabile x

4.9 Certificazione

La certificazione consiste nel dare al cliente una assicurazione della correttezza del prodotto consegnato, viene rilasciata da terzi. L'ue considera regole tecniche e norme tecniche consensuali, le prime sono obbligatorie mentre le seconde sono solo consigliate.

4.9.1 Normativa ISO 9000

La normativa ISO 9000 è una norma tecnica consensuale, essa incoraggia le sw house a seguire delle direttive che, a detta loro, siano in grado di garantire una maggiore qualità del software. Oltretutto dà delle garanzie all'utente, che sa che un software certificato ha un livello di qualità non banale. Il processo che porta alla assegnazione della certificazione al sw prende il nome di accreditamento. La certificazione può essere rilasciata anche su sotto-porzioni dell'azienda oggetto di valutazione, sarà compito degli ispettori definire i confini.

Manuale di qualità comprende di tutti i processi su cui è applicato il sistema qualità.

Visita ispettiva la visita ispettiva è la fase cardine della certificazione, gli ispettori visitano i reparti e intervistano i diretti interessati, in seguito viene deciso se rilasciare o meno il certificato. In seguito al rilascio, gli ispettori eseguono sporadicamente delle visite di sorveglianza (4 volte all'anno max).

Non conformità Le non conformità del progetto sono di diverso tipo, ovvero : non conformità rispetto ai requisiti della norma, non conformità sulla documentazione, non conformità sulla attuazione delle procedure (la politica per la qualità non risulta compresa e attuata appieno)

4.9.2 Documenti del progetto

Un prodotto software per poter essere mantenuto e per poter evolvere deve essere descritto mediante documenti tecnici : specifiche, manuali, programmi software, registrazione dei risultati. Nel manuale mostro come gestisco tutta la documentazione, ovvero:

- piano qualità
  - Gestisce la qualità del progetto, dunque deve descrivere : il livello di qualità desiderato, le metriche utilizzate per le verifiche e le tecniche e analisi di verifica. Il piano di qualità dovrebbe essere concordato, verificato e approvato da tutte le parti interessate e coinvolte nel progetto.
- piano del progetto
  - serve per definire il processo e la metodologia per trasformare la specifica dei requisiti del committente in un prodotto software. Va verificato prima dell'inizio del progetto (es: gantt)
- piano gestione configurazione
  - serve per determinare la gestione delle release del progetto, deve permettere di determinare le modifiche intervenute e gli effetti sui restanti documenti del progetto.
- documenti tecnici

4.9.3 Manutenzione

Si entra nella fase di manutenzione solo una volta che il sw è stato messo in produzione, può essere di 3/4 tipi (2 di questi tipi spesso vengono considerati insieme).

Manutenzione correttiva

Errori(fatti dall'uomo) $\rightarrow$ difetto $\rightarrow$ malfunzionamento globale.

Ha costi altissimi, soprattutto quando non si capisce il problema alla radice e si sistema con una patch ed interessa tutte le fasi del ciclo di sviluppo. La manutenzione sporca la struttura del programma e comporta un calo della qualità del sw, anche dal punto di vista della pulizia del codice. Il peso della manutenzione correttiva varia in funzione del sw.

Manutenzione adattiva

La manutenzione adattiva consiste nelle modifiche spesso dovute a delle variazioni dell'ambiente circostante, tutto ciò che non è errore va nella manutenzione adattativa (es: ricalcolo tasse e imposta, aggiornamenti listini e tariffari). Porta ad un ripristino della qualità.

Manutenzione perfettiva

Ho già una funzionalità e la estendo, introduco dei nuovi design patterns (modificando l'architettura). Il refactoring non fa parte di questa categoria perché fa parte del ciclo di sviluppo, è trasparente all'utente. Porta ad un aumento del valore informativo del sistema, oltre all'utilizzabilità ed ahimè complessità.

Manutenzione evolutiva

Migliora qualitativamente e quantitativamente le caratteristiche del sistema, porta a dei costi molto alti. L'architettura originale di un sistema lo rende o meno manutenibile, i programmi spesso sono costruiti come case, però si cerca di mantenerli come se fossero automobili(smontabili).

Capitolo 5

Interfacce grafiche

Una interfaccia è un qualcosa che permette il dialogo fra 2 entità. Quando si decide di progettarne una è necessario tenere in considerazione più aspetti, quali:

- obiettivi
- tecnologia usata
- persone

Gli obiettivi devono essere sviluppati in funzione delle persone che utilizzeranno il servizio e della tecnologia scelta.

5.1 Tipologie di interfacce

Questa è una classificazione delle varie tipologie interfacce che abbiamo a disposizione, variano in funzione dell'uso che se ne vuole fare.

5.1.1 Interfacce code-based

Consiste nell'interazione attraverso la linea di comando, attualmente non molto diffuse per via della bassa flessibilità.

- vantaggi → mole di lavoro accettabile, discreta qualità.
- svantaggi → pessima facilità di apprendimento, pessima riusabilità e pessima soddisfazione.

5.1.2 Interfacce 3270

Si tratta di una interfaccia a caratteri, adatta per data entry.

- vantaggi → mole di lavoro accettabile, buona qualità.
- svantaggi → subottima facilità di apprendimento (per via della navigazione attraverso tasti funzionali), pessima soddisfazione.

5.1.3 Pseudo-gui

chiamate pseudo perché strutturata come una interfaccia a caratteri, vanno bene per riempire dei form (gestione di dati fortemente strutturati).

- vantaggi → buon grado di flessibilità (perché posso utilizzare il mouse) riusabilità delle conoscenze acquisite (se standardizzazione), buona qualità .
- svantaggi → decente facilità di apprendimento, qualità decente.

5.1.4 Standard gui

Pensate per ottimizzare le modalità di interazione che al momento vengono date per scontato (drag-and-drop, cut-and-paste), progettata e sviluppata per un ambiente grafico.

- vantaggi → buona mole di lavoro, buona qualità e riusabilità, come il riutilizzo della conoscenza. Tutto sommato ottima soddisfazione.
- svantaggi → facilità di apprendimento non ottimale.

5.1.5 Special gui

Massimo risalto alla componente grafica, si vuole rendere il tutto più semplice per l'utente. Il pensiero è : voglio rendere il mio sw utilizzabile senza la necessità di un manuale → questo perché, spesso, mi interfaccio con utenti poco esperti.

- vantaggi → massima facilità di apprendimento, buona qualità, soddisfazione ottima.
- svantaggi → mole di lavoro da svolgere non indifferente, pessimo riutilizzo della conoscenza.

5.2 Strutturazione della interfaccia

Le strutture più verticali tendono a dare poche scelte ad ogni step (più intuitive), però creo dei percorsi più lunghi(quindi più click). Le strutture più basse hanno un numero elevato di scelte ma richiedono meno step (più semplici). Ci sono 3 tipi di strutture a finestre:

- modello multi-window
  - ha molte finestre principali ognuna con un proprio menù, ogni finistra principale ha anche delle children windows, che sono senza menù. C'è una estrema flessibilità ma la navigazione risulta complessa.
- modello multi-document
  - Composta da una sola top window (sempre aperta) che contiene una serie di document windows. Ha poca flessibilità, però è semplice da capire, ottima per utenti inesperti
- modello multi-paned
  - Ho una unica finestra, essa è divisa in pannelli (pane) ridimensionabili ma con posizione fissa. Flessibilità → poca, comprensibilità → alta.

5.3 Project standards

Consiste nella definizione di standard per la scelta di : terminologia, metafore/icone e caratteristiche delle finestre. L'obiettivo prioritario è di facilitare l'utilizzo all'utente finale.

5.4 Comunicazione visiva nelle GUI

- Affordance
  - Consiste nel mostrare che un oggetto è manipolabile, questo effetto lo si ottiene attraverso tridimensionalità, ombreggiatura e puntamento.
- Metafore
  - Parole, frasi o figure che hanno una analogia con il mondo reale.
- Layout
  - Consiste nel posizionamento degli elementi all'interno della pagina, è uno strumento di comunicazione. La distanza fra elementi deve essere collegata al loro grado di associazione.
- Colori
  - Possono essere sfruttati per focalizzare l'utente o come associazione (ogni stato ha un suo standard, es: in italia rosso : pericolo, verde : via libera). Se il colore è usato come associazione bisogna evitare di utilizzare più di 5 varianti, altrimenti si crea confusione.
- Icone (disegni piccoli e metaforici)
  - Desktop icon : per applicazioni collegate per l'utente, icone simili graficamente.
  - Menu icon : sempre visibili accanto ai menù, è un invito alla sperimentazione.
  - Button icon : In aggiunta all'icona c'è il testo, che rafforza la funzione del bottone.
- Font
  - La leggibilità è determinata dal tipo di carattere (es: per singola linea usare Sans serif, mentre per testi articolati su molte righe meglio il Serif).

5.5 I 6 criteri di usabilità

Rappresenta l'efficacia (in che misura i compiti previsti dal funzionamento vengono eseguiti), l'efficienza (risorse da impiegare) e soddisfazione dell'utente.

- apprendibilità (quanto in profondo gli utenti imparano?)
- velocità (in quanto tempo un utente porta a termine n task?)
- soddisfazione (in quanti mi hanno lasciato una recensione positiva?)
- facilità di navigazione (quanta libertà ha l'utente in merito alla presa di decisioni riguardanti attività da svolgere?)
- memorabilità (quanto bene riesce ad utilizzare l'interfaccia un utente inattivo da 6 mesi?)
- prevenzione degli errori (quanti errori catastrofici si verificano in date condizioni? è possibile prevederli?)

5.6 Metodologia di progetto

- Prima del termine dello studio di fattibilità
  - definire le attività legate alla realizzazione dell'interfaccia
  - definire i parametri di riferimento ed i criteri di usabilità
  - pianificare le attività di valutazione dell'usabilità
  - realizzare il modello concettuale dell'interfaccia
- Precocemente nella fase di analisi e progettazione
  - Definire e realizzare le strutture base (dialogo, look e feel)
  - Stabilire gli standard di progetto per l'interfaccia
  - Prototipare le parti ritenute critiche
  - Verificare l'allineamento con modello concettuale e standard
- Nella fase di sviluppo
  - Ultimare l'interfaccia in dettaglio legandola alla logica applicativa

5.7 Test con l'utente

- Simulatore (l'utente è passivo)
- Dimostratore (l'utente agisce sulle parti critiche)
- Prototipo (l'utente agisce sull'intero sistema in beta-release)