Appunti Ingegneria del Software

Lezione M1_01

Durante l’evoluzione della produzione di software si è partiti da una fase in cui le applicazioni venivano sviluppate e usate da singole persone. Successivamente le applicazioni venivano sviluppate da piccoli gruppi di persone per singoli clienti. Oggi invece a causa della crescente diffusione di software e della loro complessità è necessario migliorare il loro sviluppo con approcci ingegneristici e con figure manageriali.

- Programma: Ha un unico autore e utente e non vi è un approccio formale

- Prodotto Software: Insieme di tutti gli "artefatti" che vengono rilasciati con il codice. È usato da persone diverse dagli sviluppatori e il suo costo è molto più elevato rispetto ad un programma e necessita formalità. Si suddividono in generici che hanno un utenza di massa e specifici ovvero sviluppati per speciali esigenze.

Problemi produzione software

- Costi elevati, principalmente manodopera, testing e manutenzione.

- Ritardi, molti dei progetti iniziati riscontrano nello sviluppo ritardi o mancanza di budget motivo per cui molti progetti vengono abbandonati.

- Affidabilità, i software risultano spesso inaffidabili e molti dei problemi vengono riscontrati solo durante l'operatività del sistema.

Metodo: Procedimento generale per risolvere classi di problemi.

Metodologia: Insieme di principi, metodi per garantire l'efficacia della produzione.

Processo: Metodologia operativa che definisce le operazioni fondamentali per ottenere un prodotto industriale.

L'ingegneria del software serve quindi per la costruzione di software di grandi dimensioni in team. Si occupa dei metodi e relative metodologie, dei processi e degli strumenti per la gestione professionale del software.

Lezione M1_02

Ciclo di vita del software: Periodo di vita di un software da quando è concepito fino a quando non è più disponibile per l'uso. Identifica che stati ha attraversato durante la sua vita e come ha modificato il suo stato.

Ciclo di sviluppo del software: Periodo che inizia quando si prende la decisione di sviluppare un software e che termina quando il prodotto è consegnato.

Un CSV o modello del ciclo di vita del software è una caratterizzazione descrittiva o prescrittiva di come un software viene o dovrebbe essere sviluppato.

Fasi sviluppo software:

- Definizione: Determinare requisiti, informazioni da elaborare prestazioni e altro, tutto ciò che è

indipendente dalla tecnologia utilizzata. (COSA FA)

- Sviluppo: Definizione del progetto, strutturare i dati scegliere il linguaggio.

- Manutenzione: Si occupa delle modifiche da fare, correzioni e migliorie.

Modello a cascata (Waterfall): Si tratta di un modello sequenziale lineare. Ogni fase raccoglie attività omogenee ed è caratterizzata dalle attività e prodotti di tale attività e dai relativi controlli. Una volta approvata una fase questa è congelata e non si torna indietro.

Studio fattibilità: Valutazione preliminare di costi e benefici, se io voglio che venga realizzato un sistema software devo capire quanto costa realizzarlo. Si valutano diverse alternative ad esempio se è già pronto. L'output di questa fase deve essere una definizione preliminare del problema con costi, tempi e modalità per ogni alternativa.

Analisi requisiti: Analisi completa dei bisogni dell'utente e dominio del problema. L'output è un manuale utente (come usare) e piano di test di accettazione ovvero i test che il cliente farà per convalidare il software.

Progettazione: Si definisce una struttura del sistema e si divide in componenti e moduli. Viene fatta una distinzione tra la struttura modulare complessiva e i dettagli interni a ciascuna competente. Si tratta della progettazione di basso livello perché spiega come fa. L'output è un documento di specifica del progetto.

Ogni modulo è codificato e testato nel linguaggio scelto. L'integrazione deve essere fatta in modo incrementale.

Pro modello a cascata: Ha definito concetti utili ed ha rappresentato un punto di partenza per lo studio dei processi ed è facilmente applicabile e comprensibile

Contro modello a cascata: Interazione con il committente avviene solo all'inizio o alla fine, il sistema software risulta installabile solo quando è completamente finito.

Nella realtà le specifiche del prodotto sono spesso incomplete e l'applicazione evolve in tutte le fasi facendo si che non ci sia una netta distinzione tra le varie fasi.

In una versione modificata del modello a cascata al termine di ogni fase c'è una fase di:

- Verifica: Ci dice se si sta facendo il prodotto bene
- Convalida: Ci dice se si sta facendo il prodotto giusto.

Modello a V: Ciclo di sviluppo software in modo sequenziale, con una forma a "V" dove ogni fase sulla sinistra è collegata a una fase di testing corrispondente sulla destra. Le fasi iniziali di pianificazione e progettazione (sinistra) trovano corrispondenza nelle fasi di verifica e validazione (destra). Ogni test eseguito nella fase destra è quindi pianificato nella fase sinistra, assicurando che i requisiti siano tracciabili fino alla fase di testing finale.

La trasformazione formale è simile alla trasformazione dal codice che il compilatore fa per trasformarlo in linguaggio macchina.

Un prototipo serve a valutare i requisiti e vederne la fattibilità, come se fosse una prova per accertarci della fattibilità del prodotto e validare i requisiti. Inoltre è anche un potente mezzo con cui ci si può interfacciare con l'utente.

Tipi di prototipo

- Mock-up: Produzione completa dell'interfaccia utente e permette di definire i requisiti.

- Breadboards: Produce feedback su come implementare la funzionalità, concentrandosi sui requisiti non funzionali.

- Throw-Away: Prototipo usa e getta, tecnologie che non uso e mi serviva solo per fare una cosa veloce e lo applico sulle cose che non conosco e su cui non sono certo.

- Esplorativa: Si applica quando si ha capito cosa si deve fare e serve a fare qualcosa che può confluire nello sviluppo evolutivo.

Vantaggio sviluppo incrementale:

- Cliente: Il cliente paga a stato d'avanzamento del lavoro e vede questo avanzamento, salvaguardandosi nel caso il progetto finale non fosse consono alle richieste. Permette di fare un testing anticipato lato "utente"

- Fornitore: Permette di salvaguardarsi ad esempio se il cliente fallisce non potendo più pagare il prodotto. Permette anche di fare testing più esaustivo visto che oltre a farlo ai nuovi aggiornamenti lo si fa anche sulle cose già rilasciate.

Nel modello iterativo vado a migliorare quello che ho già rilasciato, su quelli che sono i requisiti non funzionali a differenza dall'incrementale ho già tutte le funzionalità.

Metodi agili: Approccio recente allo sviluppo del software basato su iterazioni veloci che rilasciano piccoli incrementi delle funzionalità, si fanno i test concentrandosi su una singola funzionalità.

Il problema è che bisogna sempre trovare un architettura che soddisfi le cose già fatte e i nuovi requisiti cosa che non è scontata perché quando ho realizzato le prime funzionalità non avevo tenuto conto di quella nuova.

Refactoring dell'architettura: Andare a rimodulare il codice, l'architettura del sistema per nuove funzionalità.

Modello a spirale (meta-modello): Ogni giro della spirale corrisponde ad una iterazione. Le fasi non sono predefinite ma sono scelte in accordo al prodotto e ognuna di esse prevede la determinazione degli obiettivi della fase, identificazione e riduzione dei rischio, sviluppo e verifica della fase e pianificazione della fase successiva.

Valutazione dei rischi, ad esempio chiedono una modifica ad una funzionalità dopo che questa è stata implementata. Nel modello a cascata ci sono alti rischi per i sistemi che sono nuovi , mentre se non sono nuovi il modello a cascata va benissimo. Se non si conoscono le tecnologie il dominio meglio un modello iterativo incrementale

---
Lezione M1_03

Modellare significa costruire un astrazione della realtà. La modellazione ricopre uno scopo importante nella progettazione dei software perché ci permette di creare delle astrazioni dei sistemi, i modelli. Queste astrazioni ci permettono di concentrarci su quelli che sono i dettagli importanti del sistema software senza andare troppo nei dettagli. La modellazione serve inoltre da mezzo per esplorare e validare concetti complessi e individuare possibili difetti

Per creare una struttura modulare e modellabile utilizziamo i concetti di:

- Model: Un astrazione che descrive un sottoinsieme di un sistema.

- View: Raffigura aspetti selezionati di un modello.

- Notation: Insieme di regole grafiche o testuali per raffigurare una View.

Fenomeno: Un oggetto del dominio per come noi lo percepiamo.

Un concetto spiega le proprietà del fenomeno che sono comuni con:

- Nome per distinguerlo da altri concetti.
- Scopo, per le proprietà che determinano se un fenomeno è parte di un concetto.
- Membri, l'insieme di fenomeni che sono parte del concetto.

Una astrazione è una classificazione di fenomeni in concetti e svilupparla permette di rispondere a specifiche domande riguardo ad un insieme di fenomeni ignorando dettagli irrilevanti. Un tipo è ad esempio un astrazione in un linguaggio di programmazione, un istanza è un membro di uno specifico tipo.

Dominio applicazione: Ambiente dove opera il sistema.

Dominio delle soluzioni: Tecnologie disponibili per costruire il sistema.

UML (Unified Modeling Language): Standard per modellare software orientati ad oggetti. Lo scopo dei modelli UML è quello di semplificare l'astrazione di un progetto software e nascondere i dettagli non necessari alla comprensione della struttura generale.

Convenzioni di UML

- I rettangoli sono classi o istanze
- Ellissi (ovali) sono funzioni o casi d'uso
- Le istanze sono segnalate con nomi sottolineati
- I tipi sono dei nomi non sottolineati
- I diagrammi sono grafi dove i nodi rappresentano le entità e gli archi sono relazioni tra entità

Attore: Modella un entità esterna che comunica con il sistema. Un attore ha un nome unico e una descrizione opzionale. Gli attori sono quindi delle figure del mondo reale che utilizzano e comunicano con il sistema cosa che invece il sistema interno non può fare, un sistema esterno potrebbe invece essere un attore del sistema

Differenza tra un attore una classe e un oggetto: Un attore è un entità che deve essere modellata che interagisce con il sistema e non ha sempre ha un corrispettivo con una classe, ad esempio un utente non registrato. La classe rappresenta invece un astrazione modellata di un entità nel dominio del problema. Un oggetto è una specifica istanza di una classe.

Ci sono diversi diagrammi:

- Case use Diagrams: Descrivono il comportamento funzionale del sistema per come è visto dall'utente. Nel diagramma compaiono gli attori, un ovale che rappresenta una sequenza di interazioni per un tipo di funzionalità. Si crea anche un modello di caso d'uso che è l'insieme di tutti i casi d'uso ed è un descrizione completa della funzionalità di un sistema e del suo ambiente.

- Class Diagrams: Descrive la struttura del sistema come oggetti, attributi e relazioni. Viene usato per modellare i concetti del dominio del problema, modellare sottosistemi e interfacce e per modellare le classi. Una classe rappresenta un concetto ed è un rettangolo da cui partono le frecce in un UML, le frecce rappresentano l'associazione ovvero una relazione tra classi. Con i numeri vicino ai rettangoli si

stabiliscono inoltre anche le varie molteplicità. Il nome della classe è l'unica informazione obbligatoria. Una classe incapsula attributi e operazioni rappresentati con altri rettangoli, dove ogni attributo ha un tipo e ogni operazione ha una firma. In ordine dall'alto si mettono nome della classe, attributi e metodi.

- Sequence Diagrams: Descrive il comportamento dinamico tra attori e sistema e oggetti del sistema. Viene utilizzato durante l'analisi dei requisiti per raffinare la descrizione dei casi d'uso e trovare oggetti aggiuntivi. Usato anche nel system design per raffinare le interfacce dei sottosistemi. Le classi sono rappresentate da colonne mentre i messaggi da frecce. Le attivazioni sono rappresentate da stretti rettangoli mentre la vita da linee tratteggiate verticali . La fonte di una freccia descrive l'attivazione che invia il messaggio. Le linee tratteggiate orizzontali indicano il flusso di dati. Le iterazioni si indicano inserendo * davanti al nome del messaggio. Abbiamo poi anche le condizioni che si identificano con un espressione all'interno di [] prima del nome del messaggio. La creazione è rappresentata con una freccia di messaggio che punta l'oggetto. La distruzione è denotata da una X alla fine di un'attivazione di distruzione.

- Statechart Diagrams: Descrive il comportamento dinamico di singoli oggetti come il loro stato e le loro evoluzioni. C'è uno stato iniziale rappresentato da un pallino nero. Poi ci sono i vari stati anche qui rappresentati con delle ellissi. C'è un evento che genera una transizione verso un altro evento. Lo stato finale è rappresentato da un pallino nero.

- Activity Diagrams: Modella il comportamento dinamico del sistema concentrandosi sul flusso di lavoro. È un caso speciale di statechart diagram dove gli stati sono attività (funzioni). Modellano decisioni e modellano la concorrenza sincronizzando attività multiple.

Casi d'uso: Rappresentano una classe di funzionalità date dal sistema come un flusso di eventi. E sono formati da:

- Nome unico
- Attori partecipanti
- Condizione di ingresso
- Flusso di eventi
- Condizione di uscita
- Requisiti speciali

Le linee tratteggiate rappresentano una dipendenza, un caso d'uso che dipende da un altro. Procedere da un caso d'uso primario ad uno secondario.

Le parole tra << >> sono stereotipi. Ad esempio

- <<Extend>> rappresenta un caso che dipende da un altro come se fosse una possibile evoluzione di un caso primario, una delle tanta ma non l'unica. Rappresenta un caso eccezionale o che si verifica di rado.

- <<Include>> due casi d'uso, come pezzi di codici usati da due funzioni, in questo caso è il caso primario che dipende dal caso d'uso secondario. Relazione tra due entità che indica che una determinata funzionalità. Va messa nel flusso di eventi come chiamata a sottoprogramma. Spesso un caso d'uso con Include serve per concludere delle operazioni

Distinguiamo due tipi di relazioni di comunicazione tra attori e casi d’uso:

- La prima <<initiate>> viene usata per indicare che un attore può iniziare un caso d’uso.

- La seconda <<participate>> invece indica che l’attore (che non ha iniziato il caso d’uso) può solo comunicare.

Differenza tra collegamenti (link) e associazioni:

- Link: Connessione tra due istanze di oggetto ed è un istanza di un'associazione. Rappresenta una relazione (fisica o concettuale) tra oggetti la cui conoscenza deve essere preservata per un certo periodo di tempo (Filomena Ferrucci lavora per Facoltà di Scienze).

- Associazione: Mappatura bidirezionale che descrive un insieme di collegamenti proprio come una classe descrive una serie di oggetti. Le associazioni possono, se lo vogliamo, essere ad unica direzione dove la freccia punta a chi interpreta il ruolo. Un'associazione descrive un gruppo di legami aventi struttura e semantica comuni (Docente lavora per Facoltà).

Ruoli

Il nome di un ruolo è il nome che identifica univocamente un associazione. È scritto vicino alla linea di associazione e vicino alla classe che interpreta il ruolo. I nomi di un ruolo sono necessari per associazioni tra oggetti della stessa classe e sono anche utili per differenziare due associazioni tra la stessa coppia di classi. Non è necessario utilizzarli se c'è una singola associazione tra una coppia di classi distinte.

Vi sono tre tipi di ruoli:

- Cliente: Oggetto che può operare su altri oggetti ma non è mai manipolato da altri oggetti.

- Server: Oggetto che non manipola mai altri oggetti ma viene solo manipolato

- Agente: Oggetto che può manipolare e essere manipolato.

Classi di associazioni: Associazioni che possono avere attributi e operazioni collegate a loro. Possono essere trasformate in classi con semplici associazioni.

Aggregazione: Caso speciale di associazione che denota una sorta di gerarchia. L'aggregato è la classe genitore e i componenti sono le classi figlie. Una relazione di aggregazione può essere descritta in parole semplici come "un oggetto di una classe può possedere o accedere agli oggetti di un'altra classe". Con un diamante nero indichiamo una composizione ovvero una forte forma di aggregazione dove le componenti non possono esistere senza l'aggregazione.

Qualificatore: Migliora le informazioni di una molteplicità di un'associazione tra classi, usata per esempio per ridurre un riduzione 1 to * a 1 to 1. Ad esempio senza un qualificatore una cartella ha molti file, un file (con relativo nome) appartiene solo ad una cartella. Con un qualificatore un cartella ha molti file ognuna con un nome unico.

Ereditarietà: La classe figlio eredita gli attributi e le operazioni dalla classe genitore, utile per eliminare ridondanze.

Nella modellazione di oggetti i passi da seguire sono:

1) Trovare nuovi oggetti
2) Trovargli nome, attributi e metodi
3) Trovare associazioni tra oggetti
4) Nominare le associazioni
5) Determinare la molteplicità delle associazioni

Pacchetto: Meccanismo UML per organizzare elementi in gruppi. Sono alla base del costrutto di raggruppamento con il quale si possono organizzare i modelli UML per migliorare la loro leggibilità. Un sistema complesso può essere scomposto in sottosistemi dove ogni sottosistema è modellato come un pacchetto.

Tipi di stato:

- Azione: Non possono essere decomposti ulteriormente e accadono istantaneamente rispettando il livello di astrazione usato nel modello. Possono essere divise in corsie per denotare l'oggetto e il sottosistema che implementa le azioni. (Swimlanes)

- Attività: Possono essere ulteriormente decomposte e l'attività è modellata da un altro diagramma di attività.

Forward Engineering: Creazione di codice da un modello.

Reverse Engineering: Creazione di un modello da codice.

Roundtrip Engineering: Muoversi costantemente tra il modello e il codice.

Lezione M1_04

Componenti di un progetto:

- Prodotti di lavoro: Tangibile risultato di un compito. Qualsiasi tipo di oggetto fatto dal progetto come pezzi di codice modelli e documenti. Questi quando sono stati prodotti per il cliente sono chiamati Deliverables altrimenti sono lavori interni.

- Programma: Specifica quando il lavoro sul progetto dovrebbe essere completato.

- Partecipanti: Qualsiasi persona partecipante in un progetto, a volte possiamo anche chiamarli membri del progetto.

- Attività: Attività da far fare a un partecipante del progetto per creare un prodotto di lavoro

Tipi di comunicazione:

- Programmata: Aiuta a diffondere informazioni che i partecipanti dovrebbero conoscere. Cose come ispezione del problema, stato degli incontri , revisioni di persona , revisioni di cliente e progetto o rilascio versioni.

- Non programmata: Aiuta in caso di crisi e di inaspettati bisogni. Cose come richiesta di chiarimenti, di cambi o soluzioni di problemi.

Fasi di inizio di un progetto:

1) Kick-off meeting: I partecipanti del progetto si informano del problema da risolvere.
2) Join a team: I partecipanti sono assegnati ad un team in base alle loro abilità e interessi
3) Training session: I partecipanti che non hanno le abilità richieste ricevono addestramento
4) Join Communication Infrastructure: La comunicazione del progetto supporta sia le comunicazioni programmate e non programmate.
5) Extend communication Infrastructure: Altre cose sono aggiunte in modo specifico per il progetto
6) Attend first team status meeting: Viene insegnato a condurre riunioni su stato di avanzamento, registrare informazioni e a diffonderle
7) Understand the review schedule: La revisione del programma contiene un insieme di obiettivi da comunicare come risultati del progetto in forma di revisione al manager e al cliente.

Organizzazione base team:

- Team: Piccolo insieme di partecipanti che lavorano su una stessa attività o compito

- Gruppo: Insieme di persone a cui sono assegnati compiti comuni che lavorano in autonomia senza nessun bisogno di comunicazione.

- Committee: Composto di persone che si riuniscono insieme per revisionare e analizzare problemi e proposte di azioni.

Tipi di interazione:

- Report: Usata per segnalare le informazioni di stato.

- Decisione: Usato per propagare decisioni.

- Comunicazione: Questo tipo di interazione e usato per scambiarsi tutte le informazioni necessarie per decisioni o stati.

La struttura di report è gerarchica sia lo stato che le informazioni sono unidirezionali. Le decisioni sono fatte alla radice dell'organizzazione e scambiate con interazioni alle foglie. Lo stato è generato alle foglie dell'organizzazione e riportato alla radice con le interazioni.

Il problema di questa struttura gerarchica è che molte delle decisioni tecniche devono essere fatte localmente dagli sviluppatori ma queste decisioni dipendono da informazioni di altri team, questo può portare ad una comunicazione lenta. La soluzione a questo problema è quella di scambiare informazioni tramite un altra struttura di comunicazioni che permette ai partecipanti di comunicare direttamente tra di loro.

Tipi di comunicazione:

- Liaison based: Delegata ad uno sviluppatore chiamato liaison che è responsabile di portare le informazioni avanti e indietro

- Peer based: Gli sviluppatori parlano direttamente tra di loro.

Ruolo: Definisce l'insieme di compiti tecnici e manageriali che ci si aspetta da un partecipante o da un team. Ogni ruolo definisce ciò che ci si aspetta in termini di contributo tecnico o gestionale. Un partecipante, invece, è una persona coinvolta nel progetto, indipendentemente dal ruolo assegnato. Un ruolo può essere condiviso tra due o più partecipanti perché alla fine un ruolo indica semplicemente cosa ci si aspetta che una o più persone sappiano fare e facciano. Inoltre dividere un ruolo è un ottima tecnica per distribuire il carico di lavoro.

Ruoli:

- Manageriali: Interessati all'organizzazione e all'esecuzione del progetto entro i vincoli
- Sviluppo: Interessati allo specificare, disegnare e costituire sottosistemi.
- Cross-functional: Interessati alla coordinazione tra i team
- Consultant: Interessati a provvedere supporto temporaneo in aree dove i partecipanti del progetto mancano di esperienza.

Compito (Task): Assegno di lavoro ben assegnato per un ruolo. Gruppi relativi ad un compito sono chiamati attività.

Un compito è una singola unità di lavoro ben definita ed è assegnata per ruolo con obiettivi specifici da raggiungere. Un attività è invece un insieme di compiti correlati che contribuiscono al completamento di

una fase o di un obiettivo più ampio nel progetto.

Milestone e deliverables: Il primo rappresenta il punto finale di un'attività di processo, il secondo è un risultato fornito al cliente.

La specifica di un lavoro da completare in un compito o attività è descritta in pacchetto di lavoro.

Il pacchetto di lavoro include:

- Nome e descrizione del compito
- Risorse necessarie per svolgere un compito
- Dipendenze da altre attività e il prodotto dell'attività

Un prodotto di lavoro è un qualsiasi elemento tangibile che è stato creato da un task. Il work package rappresenta invece tutta la specifica di lavoro attorno ad una task per ottenere alla fine il prodotto di lavoro.

Quindi un work package descrive un unità di lavoro che è il risultato di un prodotto di lavoro. L'unità di lavoro appartiene ad un unica attività o task che viene assegnata ad un ruolo.

Programma (Schedule): Organizzazione delle attività nel tempo. Ogni compito inizia e termina in un tempo, questo permette di pianificare le scadenze per i deliverables. I più famosi schemi sono quello di Gantt e PERT.

Meccanismi di comunicazione: Strumenti o procedure che possono essere usati per trasmette informazioni, possono essere di tipo sincrono dove mittente e ricevente sono disponibile allo stesso tempo (meeting questionari e interviste) oppure asincroni dove i due non comunicano allo stesso tempo (email, gruppi di informazione e internet).

Definizione del problema: Ha come obiettivo presentare gli obiettivi requisiti e vincoli, attività da fare all'inizio di un progetto.

Revisione del progetto: Ha come obiettivo valutare stati e revisionare modelli di sistemi e decomporre interfacce di sottosistemi. Da fare al raggiungimento di obiettivi e deliverables.

Revisione del cliente: Ha come obiettivo aggiornare il cliente e accordasi sui cambiamenti di requisiti. Da fare dopo la fase di analisi.

Walkthorough: Informale e ha come obiettivo incrementare la qualità del sistema e deve essere programmato da ogni team.

Inspection: Conformità con i requisiti da programmare dal manager del progetto.

Revisione dello stato: Obiettivo di trovare deviazioni dal programma e correggerli o identificare nuovi errori, da fare ogni settimana.

Brainstorming: Ha come obiettivo generare e valutare un grande numero di soluzioni per un problema, da fare ogni settimana.

Ruoli nei meeting:

- Primary facilitator: Organizza il meeting e ne guida l'esecuzione. Scrive l'agende descrivendo l'obiettivo e l'ambito del meeting.

- Minute taker: Responsabile di registrare il meeting identificando azioni da fare e problemi.

- Time Keeper: Tiene traccia del tempo.

Lotus Notes: Ogni utente visualizza le informazioni come un insieme di database contenenti documenti, ciascuno composto da una serie di campi. Gli utenti collaborano creando, condividendo e modificando documenti.

Lezione M1_05

La configurazione di un software va gestita perché più persone possono lavorare su un software che sta cambiando.

Configuration Management: insieme di discipline gestionali nell'ingegneria del software che organizza e controlla l'evoluzione di un sistema, ottimizzando i costi delle modifiche. Consente di tracciare, documentare e gestire le configurazioni e i cambiamenti in modo sistematico, garantendo stabilità e affidabilità.

Configuration item: Insieme di hardware e/o software che è designato per configuration management ed è trattato come singola entità nel processo. Item di cui mettiamo sotto controllo il cambiamento.

Baseline: Una specifica o un prodotto che è stato formalmente revisionato e approvato che da quel momento in poi serve come base per ulteriori sviluppi e può essere modificato solo tramite procedure formali di controllo delle modifiche. Quando un sistema è sviluppato una serie di baseline sono sviluppate dopo una revisione.

Diversi tipi di baseline sono:

- Sviluppo (Development): Servono a coordinare attività
- Funzionali: Servono a far avere una prima esperienza con il sistema funzionale
- Prodotto: Serve a coordinare costi e assistenza.

Change Management: processo di gestione delle richieste di modifica a un sistema. Ogni richiesta di cambio accettata può portare alla creazione di una nuova versione del sistema o di un suo componente.

In una Change Request, è necessario documentare dettagli specifici come il CI (confiuration item) coinvolto, il richiedente, la data, l'urgenza, la motivazione e la descrizione della modifica. Questo aiuta a garantire che tutte le informazioni critiche siano disponibili per una gestione efficace della richiesta.

Esistono due tipi di controllo delle modifiche:

Promotion: Lo stato dello sviluppo interno di un software è cambiato, sono mantenute nella master directory

Release: Un sistema software cambiato e la distribuzione lo ha approvato ed è reso visibile al di fuori dell'organizzazione produttrice, sono mantenute nella repository.

Ci sono diversi tipi di directories:

- Programmatore: Libreria per mantenere nuove o modificate entità software

- Master: Directory centrale per tutte le promozioni, controlla tutte le baseline a controlla i cambiamenti da loro fatti, e i cambi devono essere autorizzati

- Software: Archivio per le varie baseline rilasciate per uso generale.

Version: Rilascio di una versione di un configuration item ottenuta tramite una compilazione completa o una ricompilazione dell'elemento in punto ben definito del tempo. Ogni versione presenta caratteristiche o funzionalità diverse rispetto alle versioni precedenti.

Revision: Cambio ad una versione che corregge solo errori nel design o codice ma non alle funzionalità.

Un configuration item può avere più versioni e può a sua volta contenere un configuration item.

Ruolo manageriali:

- Configuration manager: Responsabile di identificare configuration item. Inoltre può anche essere responsabile di definire procedure per creare promozioni e realese.

- Change control board member: Responsabile di approvare o rifiutare una richiesta di cambio

- Developer: Crea promozioni da una change request o da una normale attività di sviluppo.

- Auditor: Responsabile per la selezione e valutazione di promozioni per realese e per assicurare la consistenza e completezza della realese.

Audit: Determina per ogni configuration item se ha i requisiti fisici e funzionali.

Revisione (Review): Strumento manageriale per stabilire baseline.

Branch: Identifica un percosso di sviluppo concorrente che richiede un'indigente configurazione. Ogni sequenza di versioni è una branch che è indipendente dalle branch di altri. Le branch sono identificate con la versione da cui derivano seguite da un numero unico.

Variant: Sono versioni che coesistono con altre.

Metodi per gestire le varianti:

- Team ridondante: Un team è assegnato ad ogni variant e ognuno di essi ha stessi requisiti ed è responsabile di tutto. Alcuni configuration item possono essere condivisi.

- Progetto singolo: Progetto una decomposizione di un sottosistema che massimizza il codice condiviso.

Lezione M2_01

Per ridurre la complessità di un software le opzioni sono:

- Astrazione
- Decomposizione
- Uso della gerarchia.

Tipi di decomposizione:

- Decomposizione ad oggetti.
- Decomposizione funzionale, che potrebbe portare però a un codice non mantenibile.

Requirement Elicitation: Definisce un sistema in termini che l'acquirente possa comprendere.

Requirement Analysis: Specifica tecnica del sistema per farlo comprendere agli sviluppatori.

Specifica sistema e analisi del modello: Entrambi i modelli si concentrano sui requisiti dal punto di vista dell'utente sul sistema. La specifica del sistema avviene però usando il linguaggio naturale mentre l'analisi usa delle notazioni formali come UML.

Descrizione del problema: Sviluppato dal cliente ed è una descrizione del problema che contiene:

- La situazione corrente: Il problema che deve essere risolto.
- Nuove funzionalità supportate
- Macchina su cui verrà utilizzato
- Date di consegna
- Criteri di accettazione: Criteri per il test del sistema.

Tipi di requisiti:

- Requisiti funzionali: Descrivono l'interazione tra il sistema e il luogo d'uso indipendentemente dall'implementazione.

- Requisiti non funzionali: Aspetti osservabili dagli utenti che non hanno però relazione diretta a comportamenti funzionali.

- Costrizioni: "Regole imposte" dal cliente come linguaggio di programmazione o altro.

Criteri per la validazione dei requisiti

- Correttezza: Sono quello che il cliente vuole.
- Completezza: Tutti i possibili scenari in cui possa essere utilizzato il sistema sono descritti includendo possibili eccezioni.
- Consistenza: Non ci devono essere contrasti tra requisiti funzionali e non funzionali.
- Realismo: I requisiti possono essere sviluppati e consegnati.
- Tracciabilità: Ogni funzione del sistema può essere associata a dei requisiti funzionali. La tracciabilità aiuta gli sviluppatori a dimostrare che il sistema è completo, ai tester che il sistema è conferme ai requisiti e agli analisti di identificare tutte le componenti e le funzionalità del sistema su cui il cambiamento avrebbe conseguenze.

Specifica dei requisiti:

- Da zero: Si sviluppa da zero e i requisiti vengono ottenuti dall'utente finale e dal cliente. Ottenuti dai bisogni dell'utente.

- Reingegnerizzazione: Rifare il design e una nuova implementazione di un sistema esistente usando tecnologie più recenti.

- Di interfaccia: Fornisce i servizi di un sistema esistente in un nuovo ambiente.

Scenari: Da usare perché sono facilmente comprensibili dall'utente. Sono delle descrizioni sintetiche di un evento o una serie di eventi dal punto di vista delle persone e della loro esperienza mentre usano il sistema.

Diversi tipi di scenario sono:

- As-is: Usato per descrivere una situazione corrente, spesso usato nei progetti di reingegnerizzazione

- Visionary: Usato per descrivere un sistema futuro.

- Evaluation: Attività dell'utente con cui il sistema deve essere valutato.

- Training: Istruzioni che passo dopo passo guidano un novizio nell'utilizzo del sistema.

Dopo che uno scenario è stato trovato si procede a ricercare al suo interno tutti i casi d'uso.

Un caso d'uso è un flusso di eventi nel sistema includendo anche le interazioni tra gli attori. Viene innescato da un attore e ognuno di essi ha un nome e una condizione per concludersi. Trovato un caso d'uso gli si da un nome poi si evidenziano gli attori coinvolti e poi ci si concentra sul flusso di eventi. Importante anche definire entry condition (stato iniziale del problema), exit condition (come termina). In caso ce ne fossero è importante definire anche le possibili eccezioni. Infine anche i requisiti speciali ovvero una lista di requisiti non funzionali e vincoli.

Guida per formare casi d'uso:

1) Usare un verbo per nominarlo, il nome dovrebbe indicare cosa l'utente sta cercando di fare

2) Non fare un caso d'uso troppo lungo che superi uno o due pagine, nel caso fosse più lungo utilizzare include e farne uno separato

3) Per il flusso di eventi iniziare sempre ad esempio con l'attore o il sistema, rendere chiara la relazione tra i vari passi. Tutti i flussi d'eventi devono essere descritti. I limiti del sistema devono essere chiari e le componenti esterne al sistema devono essere descritte in modo funzionale.

4)Definire un glossario dei termini.

Requisiti non funzionali:

- Usabilità: Rappresenta la facilità con cui un utente può imparare ad usare il sistema

- Reliability: Abilità del sistema o di una componente a performare la funzione richiesta sotto determinati stati per un tempo specifico. Questa riguarda anche la robustezza ovvero la capacità del software di rispondere a dati imprevisti. Poi la correttezza ovvero se l'input rispetta la pre-condizione e di conseguenza l'output rispetta la post-condizione. Infine safety se il problema non porta a danni a persone e ambiente (esempio centrale nucleare o volo automatico di un aereo), la sicurezza che riguarda cose come la privacy ad esempio quindi il sistema.

- Performance: Tempi di risposta, disponibilità e accuratezza.

- Supportability: Facilita con cui il software può essere modificato, cose come l'adattabilità e la manutenibilità.

Pseudo requsiti (FURPS+):

- Requisiti di implementazione: Vincoli sull'implementazione del sistema incluso l'uso di strumenti specifici.

- Requisiti di interfaccia: Vincoli imposti da sistemi esterni. Interfaccia per comunicare con altri sistemi

- Requisiti di operazioni: Vincoli sull'amministrazione e gestione del sistema.

- Requisiti di pacchetto: Vincoli sull'attuale consegna del sistema. Come avviene il deploy dell'applicazione.

- Requisiti legali: Si concentrano su licenze e altro.

Lezione M2_02

Durante la modellazione di oggetti il nostro compito è quello di trovare le astrazioni importanti.
I passi da seguire durante la modellazione degli oggetti sono:

1) Identificazione della classi, ovvero identificare gli oggetti esterni del sistema e le entità importanti del sistema.
2) Trovare gli attributi
3) Trovare metodi
4) Trovare relazioni tra le classi

Le componenti di un modello oggetti sono:

- Classi
- Relazioni, come quelle generiche che sono le associazioni, o quelle canoniche come aggregazione e generalizzazione.
- Attributi, scoperta di attributi specifici al sistema , attributi in un sistema potrebbero essere classi in un altro
- Operazioni, scoperta di operazioni che sono generiche o del dominio.

Un oggetto o istanza è esattamente una cosa mentre una classe descrive un gruppo di oggetti con proprietà simili.

Diagramma a oggetti: Una notazione grafica per modellare oggetti classi e le loro relazioni.

Possono essere:

- Class diagram: Un template per descrivere molte istanze di dati. Sono importanti perché servono agli esperti del dominio applicativo per modellare il dominio dell'applicazione. Inoltre sono anche usati dagli sviluppatori.

- Instance diagram: Un particolare insieme di oggetti legati tra di loro, utile ad esempio per la discussione di scenari o esempi.

Trovare gli oggetti è il punto centrale nella modellazione di oggetti e possibili approcci sono:

- Dominio applicativo: Chiedere a esperti del dominio applicativo di identificare astrazioni rilevanti

- Sintattico: Inizia con i casi d'uso e si estraggono gli oggetti dal flusso di eventi. I nomi sono candidati per classi e i verbi per operazioni (Abbott)

- Design Pattern: Si utilizzano design pattern riutilizzabili.

- Basato a componenti: Si identificano classi soluzioni esistenti.

Modi per trovare oggetti:

- Investigazione sintattica, si cerca nel problem statement e nel flusso di eventi di un caso d'uso.
- Uso di varie fonti di conoscenza come dell'applicazione o della struttura.
- Formulazione di scenari, descrizione dell'uso concreto del sistema.
- Formulazione di casi d'uso descrizione di funzioni con attori e flusso di eventi.

Tipi di oggetti:

- Entity object: Rappresentano l'informazione persistente tracciata dal sistema
- Boundary (Interface) Objects: Rappresentano l'interazione tra l'utente e il sistema
- Control Objects: Rappresentano le attività di controllo fatte dal sistema

Dominio della soluzione: Domini che aiutano nella soluzione del problema. Le classi del dominio della soluzione sono un astrazione che è introdotta per motivi tecnici visto che aiuta nella soluzione di un problema.

Ruoli dello sviluppatore:

- Analista: Si interessa delle classi applicative dove le relazioni tra classi sono relazioni tra astrazioni nel dominio applicativo. Inoltre si interessa se l'uso dell'ereditarietà nel modello riflette la gerarchia di astrazione (tassonomia) del dominio applicativo. Non si interessa invece nella esatta firma delle operazioni o delle classi di soluzione.

- Designer: Si concentra sulla soluzione dei problema che è il dominio della soluzione. Un suo compito importante è quello della specifica delle interfacce, si occupa infatti di descrivere le interfacce delle classi e dei sottosistemi. Il suo obiettivo è l'usabilità (l'interfaccia è usabile da più classi possibili) e riusabilità (definizione di interfaccia che la rende utilizzabile anche in sistemi futuri).

- Implementatore: Abbiamo l'implementatore di classi, ovvero sceglie le strutture dati appropriate e gli algoritmi e realizza l'interfaccia della classe con un linguaggio di programmazione. Poi abbiamo l'estensore di classi che estende la classe da una sottoclasse. Infine abbiamo il class-user ovvero il programmatore che vuole usare una classe esistente da una libreria. Lui è solo interessato nella firma delle operazioni delle classi, non gli interessa l'implementazione.

Operazione: Funzione o trasformazione applicata ad oggetti in una classe. Tutti gli oggetti in una classe condividono le stesse operazioni.

Firma: Numero e tipi di argomenti e tipo del risultato.

Metodo: Implementazione di un'operazione per una classe

Lezione M2_03

Digrammi di interazione: Descrivono il comportamento dinamico tra oggetti, ad esempio il sequence diagram che fa vedere il comportamento dinamico di un insieme di oggetti in una sequenza di tempo. Un altro tipo sono i collaborative diagram che mostrano la relazione tra oggetti ma non il tempo.

Statechart diagram: Descrivono il comportamento dinamico di un singolo oggetto, ad esempio l'activity diagram che è un tipo speciale dove tutti gli stati sono stati d'azione.

Modello dinamico: Collezione di più state chart diagrams, uno di essi per ogni classe con un importante comportamento dinamico. Il suo scopo è quello di scovare e fornire metodi per il modello oggetti.

Evento: Qualcosa che accade in un punto del tempo, gli eventi tra di loro possono essere relazionati casualmente o non relazionati. Un evento invia informazioni da un oggetto ad un altro. Gli eventi possono essere inoltre raggruppati in classi con una struttura gerarchica.

Sequence diagram: Descrizione grafica di oggetti partecipanti in un caso d'uso o in uno scenario utilizzando un grafo aciclico. Nei sequence diagram la prima colonna corrisponde all'attore che inizia il caso d'uso mentre la seconda colonna è un oggetto di interfaccia e la terza dovrebbe essere il controller che gestisce il resto del caso d'uso. Gli oggetti di controllo sono creati all'iniziazione del caso d'uso mentre gli oggetti di interfaccia sono creati dagli oggetti di controllo. Si accede alle entità dagli oggetti di

controllo e interfaccia (boundary). Le entità non dovrebbero mai chiamare quegli oggetti, in questo modo la condivisione di entità tra casi d'uso è semplificata e permette all'entità di essere resilienti.

Dai sequence diagram distinguiamo due strutture:

- Fork diagram: Il comportamento dinamico è posto in un singolo oggetto che è solitamente il controller che conosce tutti gli altri oggetti e che spesso li usa per domande dirette e comandi. Uno che manda frecce verso altri. Meglio degli stair per la responsabilità spalmata ed è decentralizzata.

- Stair diagram: Il comportamento dinamico è distribuito e ogni oggetto delega delle responsabilità ad altri. Ogni oggetto conosce infatti solo alcuni degli altri oggetti e sa quale di quale di essi può aiutarlo. Può essere meglio per il controllo centralizzato.

Statechart diagrams: Grafi i cui nodi sono stati e cui archi diretti sono transizioni con un nome. Relaziona eventi per una e una sola classe.

Negli statechart abbiamo due tipi di operazioni:

- Activity: Operazioni che hanno bisogno di tempo per completarsi e sono associate con gli stati.

- Azioni: Operazioni istantanee associate con gli eventi associati con gli stati di ingresso, uscita e azioni interne.

Stato: astrazione degli attributi di una classe, è infatti l'aggregazione di più attributi di una classe. Lo stato ha una durata. Uno stato rappresenta una situazione in cui un oggetto ha un insieme di proprietà considerate stabili.

Diagrammi di stato annidati, le attività negli stati sono elementi composti che che denotano altri diagrammi di stato di basso livello. Un diagramma di uno stato di basso livello corrisponde ad una sequenza di stati di basso livello e eventi che son invisibili nel livello più alto del diagramma.

Super-stati: Insieme di sottostati in un diagramma di stato annidato e sono racchiusi in una scatola chiamata contorno. Servono a ridurre il numero di linee in un diagramma di stato. Le transizioni da uno stato ad un superstato iniziano con il primo sottostato del superstato, inoltre tutte le transizioni verso altri stati da un superato sono ereditate da tutti i sottastati.

Tipo di concorrenza:

- Di sistema: Stato del sistema complessivo come aggregazione dei diagrammi di stato, uno per ciascun oggetto. Ogni diagramma di stato viene eseguito in modo concorrente con gli altri.

- Di oggetto: Un oggetto può essere partizionato in sottostato di stati, cosi che ognuno di essi abbia il proprio sotto-diagramma. Lo stato dell'oggetto consiste in un insieme di stati, uno stato da ogni sotto diagramma. I diagrammi di stato sono divisi in sotto-diagrammi da linee tratteggiate.

Uno state chart diagram aiuta ad identificare i cambiamenti su un unico oggetto nel tempo, il sequence diagram aiuta ad identificare la relazione temporale tra oggetti nel tempo e la sequenza di operazioni come risposta ad uno o più eventi.

Transizioni di stato sono il risultato di un azione di uscita (click di un bottone, selezione di un menu...)

Dominanza dei modelli:

Modello oggetti: Il sistema ha oggetti con stati non banali
Modello dinamico: Il modello ha diversi tipi di eventi

Modello funzionale: Il modello fa delle trasformazioni complicate.

Verifica: Controllo di equivalenza tra la trasformazione di due modelli
Validazione: Confrontare il modello alla realtà, dovrebbe avvenire dai clienti e dagli utenti.

Consistenza: Identificazione di fili tra le classi.
Completezza: Identificare associazioni penzolanti, doppia definizioni di classi e classi mancanti.
Ambiguità: Nomi sbagliati e classi con lo stesso nome con significati diversi

Priorità sui requisiti:

Alta: Devono essere affrontati nell'analisi, design e implementazione e devono essere dimostrati durante l'accettazione del cliente.

Medi: Devono essere affrontati durante l'analisi e il design, implementati e dimostrati in un secondo momento.

Bassi: Devono essere affrontati durane l'analisi e illustrano come il sistema sarà in futuro.

FINE PRIMA PROVA INTERCORSO

Lezione M3_01

Partizionamento e accoppiamento.

La fase di design si concentra sul dominio delle soluzioni.

Per la fase di definizione degli obiettivi del design si utilizzano i requisiti non funzionali.

Il modello funzionale serve per la decomposizione dei sistemi.

Il modello a oggetti serve per il mapping hardware/software e per gestione della persistenza.

Il modello dinamico è utilizzato per la concorrenza e la gestione delle risorse globali e il controllo del software.

I sottosistemi sono quelli che diventano i package UML. Rappresentano un'insieme di classi, associazioni, operazioni eventi e vincoli che sono collegate.

I servizi sono un gruppo di operazioni date dal sottosistema e si ritrovano dai casi d'uso, che condividono un scopo comune. Un servizio è specificato dall'interfaccia di un sottosistema, che specifica il flusso di interazioni e le informazioni da e verso i bordi del sottosistema ma non al suo interno. Durante la fase di implementazione sono le API.

Coesione: Misura la dipendenza tra le classi, un'alta coesione indica che le classi nel sottosistema eseguono attività simili e sono collegate tra loro con associazioni. Una bassa coesione indica invece varie e ausiliarie classi senza associazioni. Indica che una classe implementa un unica responsabilità.

Accoppiamento: Misura la dipendenza tra sottosistemi, un alto accoppiamento indica che i cambiamenti fatti ad un sottosistema avranno un grande impatto sugli altri.

Per avere un basso accoppiamento utilizziamo tecniche di layering e partizionamento.

- Partizionamento: Divisione verticale di un sistema in più sottostanti sottosistemi indipendenti o debolmente accoppiati che forniscono servizi allo stesso livello di astrazione.

- Layer: Un sottosistema che fornisce servizi a livelli più alti di astrazione. Un layer può dipendere solo dai layer sottostanti e non ha conoscenza dei layer superiori.

Le relazioni tra sottosistemi si dividono in relazioni di layer (un layer A chiama un layer B e dipende da esso) e di partizione dove i sottosistemi hanno una conoscenza superficiale (una partizione A chiama una partizione B e B chiama A).

Bisogna avere almeno 3 layer, Application data e logic.

Per le partizioni usiamo i casi d'uso con cui dividiamo verticalmente il sistema e questi comunicano tra di loro con lo strato sottostante (debolmente accoppiati)

Architettura chiusa: Qualsiasi layer può solo invocare operazioni dal layer immediatamente sottostante. Questo permette un'alta munetenibilità e flessibilità.

Architettura aperta: Qualsiasi layer può invocare operazioni da qualsiasi layer sottostante, questo garantisce un alta efficienza in fase di esecuzione.

Decomposizione in sistemi: Identificazione di sottosistemi, servizi e la loro relazione reciproca.

Architettura Client/Server: Uno o più server forniscono servizi alle istanze di un sottosistema (client). I client effettuano chiamate al server che esegue alcuni servizi e restituisce il risultato. Un client conosce l'interfaccia del server (i suoi servizi). Gli obiettivi di questa architettura sono la portabilità dei servizi, un server può infatti essere installato su una varietà di macchine e sistemi operativi. Fornisce inoltre trasparenza di locazione visto che potrebbe essere distribuito. Garantisce le performance visto che client e server hanno dei ruoli ben definiti. Risulta scalabile , flessibile e affidabile.

Architettura Peer-to-Peer: Rappresenta una generalizzazione dell'architettura client server dove i client possono essere server e server possono essere client. Ci sono più possibilità deadlock. (Esempio ISO/OSI)

Architettura del repository: I sottosistemi accedono e modificano dati da una singola struttura dati. I sottosistemi sono bassamente accoppiati e il controllo del flusso è dettato da una repository centrale o dal sottosistema.

Model/View/Controller: I sottosistemi sono classificati in 3 tipi differenti. Il sottosistema model che è responsabile della conoscenza del dominio dell'applicazione. Il sottosistema View che è responsabile per mostrare gli oggetti del dominio applicativo all'utente. Infine il controller che è responsabile della sequenza di interazioni con l'utente e notifica le View dei cambiamenti del modello.

Il system design riduce il divario tra i requisiti e la macchina e decompone il sistema generale in parti più gestibili.

Lezione M3_02

Nella sezione dedicata alla concorrenza si devono identificare i thread concorrenti e i possibili problemi.

Sistemi concorrenti possono essere implementati su un qualsiasi sistema che fornisce concorrenza fisica o logica.

Nel mapping software/hardware affrontiamo come dovremmo realizzare il sistema e come il modello ad oggetti si collega ai software e all'hardware scelto. Durante il mapping si devono tenere conto dei problemi di computazione di memoria e di I/O. Si descrivono quindi le connessioni fisiche e quelle logiche come le associazione dei sottosistemi.

Component Diagram: Grafo delle componenti connesse da relazioni di dipendenza, mostra le dipendenza tra componenti software. Le dipendenze sono mostrate da frecce tratteggiate dal client al componente del fornitore. Struttura statica che mostra la struttura a tempo di progettazione o di compilazione.

Deployment Diagram: Sono utili per mostrare la progettazione di un sistema dopo la decomposizione in sistemi , la gestione della concorrenza e il mapping hardware/software. È un grafo i cui nodi sono connessi come associazioni di comunicazione. Mostra la struttura a tempo di esecuzione.

Nella parte della gestione dei dati ci occupiamo di come alcuni oggetti nel modello devono essere resi persistenti. Un oggetto può essere reso persistente in una struttura dati, all'interno di un file o all'interno di una base di dati.

Un file andrebbe usato nel caso in cui i dati siano voluminosi e abbiamo tante informazioni grezze, o per mantenere i dati per poco tempo. Un database se ai dati vi accedono più persone o applicazioni e se i dati devono essere portati tra più piattaforme.

I problemi di un database sono:

- Lo spazio richiesto visto che richiedono di solito il triplo dello spazio rispetto alla grandezza effettiva del dato.

- Tempi di risposta

- I modi di locking: Dove con il pessimistico blocchiamo prima di un accesso ad una risposta e la liberiamo quando abbiamo completato , oppure con l'ottimistico facciamo le operazioni e solo alla fine controlliamo se c'è stata concorrenza.

- Amministrazione: Grandi database richiedono personale addestrato.

Database orientati agli oggetti: Supportano i concetti di modellazione degli oggetti.

Database relazionali: Sono basati sull'algebra relazione e i dati sono mantenuti in una tabella a due dimensioni.

Nella gestione delle risorse globali e della sicurezza:

Si definiscono gli accessi: Nei sistemi con più utenti gli attori potrebbero avere diverse funzionalità e dati. In questa fase modelliamo le differenze di acceso esaminando il modello ad oggetti determinando quali oggetti sono condivisi tra i vari attori. In basi ai requisiti di sicurezza definiamo anche come gli attori sono autenticati al sistema e come i dati selezionati dovrebbero essere criptati.

Matrici di accesso: Si modella l'acceso alle classi con una matrice dove le righe rappresentano gli attori del sistema e le colonne rappresentano le classi di cui vogliamo controllare gli accessi. Un entry della tabella indica l'operazione disponibile su istanze della classe dall'attore. Questa può essere implementata con una tavola globale , oppure con una lista di attori e operazioni associate ad una classe. Oppure una coppia tra la classe e l'operazione legata all'attore.

Controllo del software:

- Controllo guidato dalle procedure: Un programma principale chiama procedure di un sottosistema.

- Controllo guidato da eventi: C'è un dispatcher che chiama le funzioni, questo lo rende molto flessibile e di facile estensione.

- Controllo decentralizzato: Il controllo risiede in più oggetti indipendenti.

Boundary condition: In fase di configurazione per ogni oggetto persistente dobbiamo identificare in che caso d'uso è stato creato o distrutto. Nella fase di inizializzazione descriviamo come il sistema o una componente passa da uno stato non inizializzato ad uno stato pronto. Nella fase di terminazione descriviamo quali risorse sono pulite e quali sistemi. Nella fase ci configurazione cerchiamo di gestire i possibili errori.

Admin: Utente che conosce un minimo di informatica che fa partire/spegne il server non una persona che utilizza il sistema

---

Lezione M3_03
/* Inutile */

Lezione M3_04

Nel system design l'architetto è il ruolo principale, si occupa di assicurare la consistenza nelle decisioni di design e negli stili delle interfacce. Assicura anche la consistenza delle policy e dell'integrazione del sistema.

Il team di architettura inizia a lavorare appena il modello dell'analisi è stabile. Il numero di persone in questo team dipende in base al numero di sottosistemi.

I membri del team d'architettura sono gli architecture liason, questi rappresentano i team dei sottosistemi e servono per la negoziazione dei cambiamenti delle interfacce. Durante il system design si concentrano sui servizi del sottosistema mentre durante la fase di implementazione si concentrano sulla consistenza delle API.

Alcune difficoltà di coordinamento durante questa fase possono essere il livello di astrazione visto che i problemi che sorgono a questo punto saranno più chiari in fase di testing e implementazione, questo porta spesso a rimandare a quella fase la discussione del problema.

Design window: Serve per raggiungere la progressiva stabilizzazione della decomposizione del sistema. Dare delle finestre temporali in cui un determinato problema deve essere risolto.

L'analisi dei requisiti la può fare un'esperto del dominio del problema, qui invece dovrebbero esserci sviluppatori che non hanno partecipato alla progettazione e devono controllare se queste cose hanno la controparte di analisi dei requisiti. Bisogna controllare se sono rispettati i design goal e vedere se i modelli sono consistenti e realistici. E se le scelte di progettazione sono riconducibili a requisiti funzionali e non (spiegando cosa mi ha portato a quella scelta).

I servizi sono operazioni delle classi che corrispondo ai casi d'uso.

La correttezza è se ho considerato tutti gli aspetti nella mia progettazione.

Consistente se non contiene contraddizioni. I nomi che abbiamo usato sono consistenti per esempio ai nomi delle classi devono corrispondere a quelli dell'analisi dei requisiti.

I requisiti non funzionali possono andare in conflitto e c'è bisogno che gestiamo le priorità e queste priorità si devono riscontrare negli obiettivi di design che ci siamo posti.

Realismo: Le soluzioni devono essere realizzabili.
Robustezza: Capacità di gestire condizioni limite.
Concorrenza: Supporto per operazioni parallele.
Leggibilità: I nomi e i modelli devono essere comprensibili, verificati anche da sviluppatori esterni.

Hardware/Software mapping: Classi che si occupano delle cose come per esempio i DAO classi che si occupano dell'accesso allo Storage. Data Layer classi che si occupano di gestire la persistenza de dati.

Lezione M3_05

Rational Management tutto quello si discute e porta alla scelta di progettazione e presenta:

- Problematiche che si discutono
- Alternative, ovvero cosa è meglio o peggio
- Decisioni che sono state fatte per risolvere i problemi
- Criteri usati per guidare le decisioni
- E argomentazioni tra le varie alternative

Questo migliora il supporto al design e migliora il supporto di documentazione rendendo facile a chi non ha fatto il sistema di rivedere delle cose. Inoltre migliora anche la manutenzione e l'apprendimento.

Issue: Problemi concreti che spesso non hanno una sola ed unica soluzione.
Poposal: Sono le possibili alternative agli issue e una proposal può essere condivisa tra più issue.
Consequent issue: Issue che sono sono creati dall'introduzione di una proposal
Criteri: Sono delle misure di bontà e sono spesso gli obiettivi di design e requisiti non funzionali.
Arguments: Rappresentano la discussione per giungere alla risoluzione di un problema.
Resolutios: Rappresentano le decisioni che sintetizzano l'alternativa scelta e le argomentazioni che la supportano.

M4_01

Nel system design abbiamo la decomposizione in sottosistemi mentre nell'object design abbiamo la scelta del linguaggio e gli algoritmi e strutture date che andremo ad utilizzare, scelte implementativi quindi.

Abbiamo due attività principali

- Preliminary Design: Si concentra sulla struttura generale del sistema e fa una prima scelta di strutture dati.

- Detailed Design: Si concentra sui dettagli implementativi come gli algoritmi, le strutture dati da raffinare e il linguaggio. Questa fase può avvenire in modo contemporaneo alla fase di Preliminary Design.

Nell'analisi dei requisiti ci si concentrava sul dominio dell'applicazione mentre nel fase di system design ci si concentrava anche in parte sul dominio dell'implementazione. Nell'object design adesso ci si concentra solo sul dominio dell'implementazione.

Application Objects (Domain objects): Concetti del dominio che sono rilevanti per il sistema, sono identificati da specialisti del domino dell'applicazione e dall'utente finale. Trovati durante l'analisi dei requisiti.

Solution Objects: Oggetti che rappresentano concetti che non hanno un contro parte nel dominio dell'applicazione , sono identificati dagli sviluppatori.

Custom Objects: Oggetti che faccio io e anche soluzioni esterne che cerco di adattare al mio sistema.

Off-the-shell: Componenti dell'architettura, dell'infrastruttura del sistema come ad esempio un particolare DBMS o un particolare server.

Con OCR facciamo la firma dei metodi e facciamo il contratto delle operazioni inoltre permette di definire

contratti che fanno da vincoli.

Nello sviluppo incrementale sviluppiamo un sistema alla volta e le operazioni che facciamo sono:

- Specifica, per esempio se vogliamo che gli attributi sono privati dobbiamo mettere i set e get, specifica dei vincoli e eccezioni, cosa fa un oggetto quando viene violata una precondizione.

- Riuso: Si vedono che componenti possiamo usare e si utilizzano dei design pattern eventualmente modificandoli.

- Controllo casi d'uso.

- Ristrutturazione e ottimizzazione (hanno a che fare con i requisiti di performance).

Design Patterns: Soluzione nota ad un problema noto e frequente.

Il Composite Pattern è un design pattern strutturale che consente di trattare oggetti individuali e gruppi di oggetti in modo uniforme.

In un composite pattern troviamo:

- Component: Classe astratta o interfaccia che definisce le operazioni comuni. Rappresenta l'elemento base della gerarchia.

- Leaf (Foglia): Classe concreta che implementa le operazioni definite in Component. Rappresenta un elemento foglia, cioè un oggetto senza ulteriori sotto-componenti.

- Composite: Classe concreta che implementa Component. Contiene una collezione di altri Component (foglie o altri composite) e consente la ricorsione nella struttura. Implementa metodi per aggiungere, rimuovere o accedere ai suoi sotto-componenti.

Il Facade Pattern fornisce un'interfaccia unificata ad un'insieme di oggetti in un sottosistema. Una facade definisce un alto livello di interfaccia che rende il sottosistema più facile a usare, questo design pattern ci permette di realizzare architetture chiuse.

Package: Insieme di classi che sono raggrupate. I packages sono spesso usati per modellare i sottosistemi. In UML è un rettangolo con una linguetta (tab).

Ereditarietà: Serve a definire tassonomie e verificare se un concetto può ereditare proprietà da un altro. Permette di specificare servizi che saranno ereditati da altre classi, consentendo specializzazione o generalizzazione. È usata sia in fase di analisi (per tassonomie) sia per favorire il riuso a livello di object design.

Tipi di riuso:

- Composizione (Black Box Reuse) ovvero una nuova funzionalità ottenuta tramite aggregazione di componenti esistenti.

- Ereditarietà (White Box Reuse) dove otteniamo una nuova funzionalità tramite ereditarietà.

Per ottenere una nuova implementazione possiamo:

- Ereditare un'implementazione: Si estende un'applicazione ereditando le funzionalità di una classe esistente.

- Ereditare da un'interfaccia: Si eredita la definizione delle operazioni (specifica), ma non la loro implementazione.

- Delegazione: Un oggetto delega ad un altro l'esecuzione di determinate operazioni.

La delegazione rende la composizione potente quanto l'ereditarietà per il riuso. Coinvolge due oggetti per gestire una richiesta ,l'oggetto ricevente delega le operazioni a un oggetto delegato. La delegazione ci dà flessibilità ma incapsula gli oggetti.

Adapter Pattern: L'Adapter Pattern viene utilizzato quando un client si aspetta un oggetto con una determinata interfaccia, ma l'oggetto disponibile ne implementa un'altra incompatibile. Si introduce quindi un adapter che converte l'interfaccia dell'oggetto disponibile (Adaptee) in quella attesa dal client (Target). I due adapter patterns sono l'object Adapter che usa la composizione per delegare le richieste e il class Adapter che usa l'ereditarietà per adattare le interfacce.

Bridge Patter: Serve a disaccoppiare un'astrazione da una sua implementazione, così posso avere più implementazione di un astrazione facendomi decidere dinamicamente quale usare.

Differenze tra adapter e bridge: Entrambi nascondono i dettagli dell'implementazione, l'adapter usato per componenti riutilizzabili e per far lavorare insieme componenti non legati. Il bridge serve per estendere e far variare implementazioni. Adapter più reingeneering, mentre bridge Green field engeenerring.

Proxy Pattern: Utilizza un proxy come sostituto o rappresentante dell'oggetto reale, incapsulandolo. Invece di interagire direttamente con l'oggetto reale (Subject), si utilizza un'implementazione intermedia (il Proxy) che gestisce l'istanza dell'oggetto reale e può aggiungere funzionalità. Si usa la delegazione per catturare e inoltrare qualsiasi accesso all'oggetto reale.

Tipologie di Proxy:

- Remote Proxy: Agisce come rappresentante locale di un oggetto remoto. Utile per caching quando le informazioni non cambiano frequentemente.

- Virtual Proxy: Funziona come un placeholder per un oggetto costoso da creare o scaricare, l'oggetto reale viene creato solo quando necessario.

- Protection Proxy: Controlla l'accesso all'oggetto reale. Permette a diversi attori di accedere solo alle informazioni a cui sono autorizzati, implementando metodi personalizzati per ogni attore.

I Vantaggi sono il maggiore controllo e gestione delle interazioni con l'oggetto reale.

FINE PATTERN STRUTTURALI

Command Pattern: Creare oggetti che rappresentano operazioni, consentendo flessibilità nel loro utilizzo, tracciamento e gestione.

Un Command pattern è composto da:

- Command: Interfaccia con il metodo execute().
- ConcreteCommand: Implementa Command, conosce il receiver e gestisce i comandi, ad esempio tramite uno stack.
- Receiver: L'oggetto su cui viene eseguita l'azione.
- Invoker: Chi richiama il comando, eseguendolo dal top dello stack.

Observer Pattern: Definisce una dipendenza uno-a-molti tra oggetti, seguendo il modello publish-subscribe, permette di sincronizzare automaticamente gli oggetti collegati, garantendo che ogni

modifica al subject si rifletta negli observer.

Struttura:
- Subject: L'oggetto osservato nel suo stato attuale, mantiene una lista di observer e fornisce metodi per aggiungerli, rimuoverli e notificarli.

- Observer: Interfaccia o classe base implementata dagli oggetti che vogliono essere notificati.

- Notify: Metodo del subject che informa tutti gli observer quando c'è un cambiamento di stato.

Il subject registra gli observer , quando il subject cambia stato, chiama notify per informare tutti gli observer registrati. Gli observer aggiornano il proprio stato in base al cambiamento del subject.

Strategy Pattern: Permette di definire una famiglia di algoritmi, incapsularli in classi separate e selezionarli dinamicamente a runtime. Diverso dal Bridge Pattern, che è progettato per separare astrazione e implementazione in modo statico.

Struttura:
- Strategy: Interfaccia comune per tutti gli algoritmi.
- ConcreteStrategy: Implementazioni concrete della strategia.
- Context: Classe che utilizza una Strategy e delega ad essa l'esecuzione dell'algoritmo.

Fine Pattern Comportamentali

Abstract Factory Pattern: Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. Permette di cambiare dinamicamente la famiglia di oggetti creata scegliendo una diversa factory. Assicura che gli oggetti creati siano compatibili tra loro, poiché appartengono alla stessa famiglia.

Struttura:
- AbstractFactory: Interfaccia che definisce metodi per creare oggetti correlati.
- ConcreteFactory: Implementa l'AbstractFactory e fornisce le istanze concrete degli oggetti.
- Product: Interfaccia o classe base per gli oggetti prodotti.
- ConcreteProduct: Implementazioni concrete dei prodotti.
- Client: Utilizza l'AbstractFactory per creare gli oggetti senza conoscere le classi concrete.

Il Builder Pattern: Separa la costruzione di un oggetto complesso dalla sua rappresentazione, permettendo di creare diverse rappresentazioni dello stesso oggetto. Utile per oggetti con molteplici configurazioni o step di costruzione.
Consente di riutilizzare lo stesso processo di costruzione per creare oggetti con rappresentazioni diverse. Isolare il processo di costruzione per mantenere il codice flessibile e leggibile, separando i dettagli costruttivi dal client.

Struttura:
- Builder: Interfaccia che definisce i metodi per costruire le parti di un oggetto complesso.
- ConcreteBuilder: Implementa il Builder, specificando i dettagli della costruzione.
- Director: Dirige il processo di costruzione usando il Builder.
- Product: L'oggetto finale costruito.

Abstract Factory non nasconde il processo di creazione e il prodotto è immediatamente disponibile , builder invece viene nasconde il processo di costruzione all'utente e restituisce il prodotto solo dopo che è completamente costruito.

Abstract Factory e Builder possono essere usati insieme per gestire famiglie di prodotti complessi, combinando la creazione di famiglie (Abstract Factory) con la gestione della complessità costruttiva

(Builder).

M4_02

Object design avvicina i modelli che abbiamo definito al codice.

Attività dell'object design:

1) Aggiungere informazioni sulla visibilità, fino ad ora non avevamo get() , set() ma solo le operazioni più complesse. Ora dobbiamo definire la visibilità. In UML se un attributo è private si mette (-), protected con (#) , public con (+).

2) Aggiungere l'informazione della firma del tipo.

3) Aggiungere contratti, permettono al chiamato e al chiamante di condividere le stesse assunzioni di una classe.

Invariate un predicato che è sempre vero per tutte le istanze della classe. Le invariati sono vincoli associati con classi o interfacce e sono usate per specificare vincoli di consistenza tra gli attributi della classe.

Precondizioni predicato che deve essere vero prima che che un operazione deve essere invocata, deve essere vera sui parametri, ma visto che l'operazione va a modificare lo stato di un oggetto deve essere definita sugli attributi anche.

La post-condizione deve essere definita sul valore di ritorno e sugli attributi dopo l'operazione , ma la post-condizione mette in relazione l'output in base ai parametri e le operazioni di input.

OCL permette ai vincoli di essere formalmente specificati su elementi di un singolo modello o gruppi di elementi del modello. Un vincolo è espresso come un espressione OCL che restituisce vero o falso.

Un metodo della sotto classe più indebolire le condizione. Mentre per la post-condizione posso solo rafforzare perché devo garantire. Un inaridente invece deve rispettare quelle le cose della superclasse.

M4_03

OCL è un linguaggio logico, e sono predicati che assumono valori, vero e falso. Usare le operazioni invece degli attributi perché gli attributi sono privati.
OCL set tra un singola associazione
OCL sequenze quando devo navigare in un certo ordine
OCL bags per avere un insieme di una stessa istanza , quando navigo li ottengo poi passo da bags a set.

M4_04

Mappare modelli su codice

Operazioni da fare quando mappiamo:

- Ottimizzazione, questa attività affronta i requisiti di performance del modello del sistema perché a volte potremmo avere dell'inefficienza nonostante una semantica corretta.

- Realizzare associazioni: Le associazioni sono mappate ai costrutti del codice sorgente.

- Mappare contratti su eccezioni: Descrivere il comportamento delle operazioni quando i contratti sono rotti.

- Fare modello dei dati su uno schema.

Le trasformazioni sono applicate al modello ad oggetti per ottenere un altro modello a oggetti. Gli obiettivi delle trasformazioni sono quelli di semplificare, ottimizzare il modello rendendolo più in linea alle specifiche. Effettuare delle trasformazioni per avvicinarmi al codice, miglioro l'eredità o converto un classe in un attributo.

Devo preparare l'eredità, bisogna avere la stessa signature , avere gli stessi nomi, devo uniformare per generalizzare il comportamento. Possono sorgere dei problemi a volte in questi procedimento come ad esempio che delle operazioni definite in una classe ma non in altre potremmo pensare quindi di usare delle funzioni virtuali da poi specializzare.

Tipi di trasformazioni:

Noi abbiamo lo spazio dei modello ma anche lo spazio del codice. Se ci rendiamo conto di dovere fare una modifica possiamo farla prima su modello e poi al codice (Re-engineering) oppure prima dal codice e poi dal modello (reverse-engineering).

Principi delle trasformazioni:

- Deve andare incontro ad un unico obiettivo di design.

- Deve essere locale, deve cambiare pochi metodi o classi per volta

- Deve essere fatta in isolamento ad altri cambiamenti (per il test).

- Devo fare la verifica ovvero il testing.

Possiamo ottimizzare l'accesso ai percorsi e migliorare l'efficienza del sistema adottando diverse strategie:

Accesso ai percorsi: Aggiungere associazioni ridondanti per accedere rapidamente ai dati frequentemente richiesti. Ridurre le associazioni da "many" a "one" utilizzando qualificatori, per limitare il numero di oggetti coinvolti.

Ottimizzazione degli attributi: Trasformare oggetti in attributi quando non è necessario gestirli come entità separate. Salvare attributi derivati direttamente, evitando calcoli ripetuti che possono essere costosi.

Esecuzione efficiente: Riorganizzare l'ordine delle operazioni per ottimizzare le dipendenze. Utilizzare metodi get e set per controllare l'accesso e la modifica degli attributi, migliorando l'incapsulamento e la coerenza.

Per mappare le eccezioni ai contratti in Java dobbiamo definire classi di eccezioni , controllare i contratti con i costrutti di programmazione e sollevare un eccezione.

Per ogni operazione nel contratto dobbiamo controllare la precondizione prima di iniziare , controllare la postcondizione alla fine del metodo e mentre facciamo questo controllare anche le invarianti. Questo approccia risulta essere molto dispendioso.

Quindi ci sono alcune euristiche da seguire.

- Non verificare postcondizioni e invarianti nel codice: Di solito sono ridondanti rispetto alla funzionalità della classe. Sono poco utili per trovare bug, a meno che non siano scritti da sviluppatori diversi. La verifica di queste condizioni può essere gestita attraverso i test.

- Concentrarsi sulle interfacce dei sottosistemi: Non aggiungere controlli per metodi privati o protetti.

- Dare priorità ai contratti per componenti longevi: Gli oggetti entità (entity objects) sono spesso i più rilevanti per la durata e stabilità del sistema.

- Riutilizzare il codice di verifica dei vincoli: Molte operazioni condividono precondizioni simili. Incapsulare i controlli dei vincoli in metodi riutilizzabili e far sì che condividano le stesse classi di eccezioni.

- Commentare sempre il codice di verifica: Documentare il motivo e il contesto dei controlli per facilitare la manutenzione e la comprensione del codice.

M5_01

Affidabilità: Misura di successo con il quale il comportamento osservato conferma alcune specifiche del comportamento.

Failure: Qualsiasi deviazione del comportamento osservato dal comportamento specificato.

Stato erroneo: Parliamo di failure quando il programma si trova in uno stato erroneo e l'utente se ne accorge. C'è perché ad un certo punto il sistema ha elaborato delle informazioni non corrette. Quindi si trova in uno stato dove ogni ulteriore operazione porta ad una failure.

Fault (Bug): Difetto del codice come algoritmico ovvero che abbiano commesso errori nello scrivere il codice, meccanico legato alla robustezza del sistema per esempio va via la connessione o si rompe qualcosa.

Gestione errori:

- Verification: Dimostrazioni di correttezza, ma potrebbe esserci un grado di astrazione che poi non andremo ad usare, perché i vincoli ipotetici sono corrispondono alla realtà.

- Ridondanza modulare ma è costoso, fare delle patch.

- Testing si occupa di trovare i malfunzionamenti, io pianifico il testing con l'obiettivo di far fallire un programma.

Possiamo fare individuazione degli errori, creando delle failure in modo programmato oppure facendo monitoraggio consegnando informazioni sullo stato e vedere bug di performance.

Error recovery quando ci si trova in uno stato di errore e dobbiamo recuperare.

Definizioni base del testing:

- Error: Errore che commette chi introduce un fault.

- Fault: Risultato di un errore, difetto che c'è nel codice

- Failure la visione del fault.

- Incident: Conseguenza di una failure.

- Testing esercitare il codice con dei casi di test per vedere failure e acquisire maggiore fiducia nel codice.

- Test Case: Insieme di risultati attesi in base a dati in input.

- Test Suite: Insieme di test case.

Component: Parte del sistema che può essere isolata per il testing. Può essere anche un gruppo di oggetti o uno o più sottosistemi.

Test stub: Implementazione parziale di un component sul quale la componente testata dipende. Stub che è un implementazione parziale di un componete , ovvero un implementazione fittizia visto che quel componente isolato dipende da altri.

Test driver: Implementazione parziale di un componente che dipende dal componente testato. Test driver è implementato parzialmente, un componente che usa un competente che devo testare che mi serve per testarlo.

Driver e stub possono essere:

- Interattivo: Mettiamo noi dati
- Automatico: Metto dati di test in un file e controllo i dati e scrivo se i risultati sono corretti o meno.

Correzione: Faccio debugging e correggo il codice, ma potrebbe capitare di aggiungere nuovo errore quindi devo fare regressione-test e non devo testare solo quello che ha trovato il fault ma tutto perché potrei aver rotto qualcos'altro.

Test deve essere fatto da persone che non sviluppano il sistema. Integration Test e test di sistema non è fatto dagli sviluppatori dei moduli.

Test funzionale: Test di sistema fatto per vedere la correttezza delle funzionalità del mio sistema. Vedere che il programma fa quello che dicono i casi d'uso.

Test di performance: Test dei requisiti non funzionali.

Component testing:

- Testing di unità: Singolo sottosistema, fatto dagli sviluppatori e ha come obiettivo confermare che il sottosistema è programmato correttamente e fa le funzioni richieste.

- Integration Testing: Gruppi di sottosistemi e eventualmente l'intero sistema. Fatto dagli sviluppatori ed ha come obiettivo testare l'interfaccia tra i sottosistemi.

- System testing: riguarda l'intero sistema, fatto dagli sviluppatori ed ha come obiettivo quello di determinare se il sistema soddisfa i requisiti.

- Test di accettazione: Valutare il sistema consegnato dagli sviluppatori , fatto dal cliente ed ha come obiettivo quello di dimostrare che il sistema incontra l'obiettivo dei clienti.

Test Plan: È un documento che si concentra sugli aspetti gestionali del processo di testing. Descrive lo scopo, l'approccio, le risorse necessarie e l'organizzazione delle attività di testing. Inoltre, identifica i requisiti e le componenti che devono essere sottoposte a test, fornendo una guida chiara per l'esecuzione del processo.

Test Case Specification: Documenta ogni test, ha gli input i driver e gli output attesi.

Test Execution Report: Documento che riporta i dettagli di ogni esecuzione del test.

Test Item Transmittal Report: Documento che accompagna un elemento software testato. Contiene

informazioni per identificare chiaramente l'elemento software sottoposto a test.

Test Log: Riporta i risultati effettivi dei test, evidenziando le differenze rispetto agli output attesi.

Test Incident Report: Descrive gli incidenti scoperti durante i test che richiedono ulteriori analisi. Utilizzato dagli sviluppatori per analizzare e prioritizzare i problemi e per pianificare modifiche al sistema e ai modelli e anche per generare nuovi casi di test e nuove esecuzioni.

Test Summary Report: Sintesi dell'attività di testing. Elenca gli incidenti risolti e quelli ancora aperti.Valuta le tecniche di testing adottate, descrivendone i limiti.

M5_02

Le tecniche di verifica possono essere:

- Informali e Statici, come leggere semplicemente il codice facendo un esecuzione mentale da soli o con qualcuno. Per una cosa più formale c'è la code Inspection dove c'è un team a cui presentiamo. Abbiamo anche tool statici simili al lavoro dei compilatori, riconoscono errori sintattici e semantici.

-Dinamica:

Black box (funzionale) dove il test è basato sulla specifica formale o informale , scalabile perché lavora al livello di funzionalità.
White-box (strutturale) si guarda la logica interna di un programma, si usa di solito al livello di una singola classe e non è scalabile.

Gli input per il testing possono essere scelti:

- Random: Considero tutti gli input con la stessa capacità di rilevare una failure. Non è ottimale perché la distribuzione dei fault non è uniforme.

- Systematic: Cerco di dividere lo spazio di input in partizioni , utilizzando la conoscenza del sistema, in modo che gli input in una singola partizione abbiano la stessa probabilità di rilevare un malfunzionamento. Prendo poi un rappresentante per ciascuna di queste partizioni.

Principio di partizionamento: Sfruttare la conoscenza per scegliere campioni in cui è più probabile trovare regioni speciali o problematiche dello spazio di input.

Quando usiamo il principio di partizionamento usiamo delle euristiche e abbiamo o un partizionamento perfetto o quasi(la loro intersezione non è vuota) questa è meno efficace ma riduco i casi di test. Coprendo una partizione abbiamo coperto una parte di input.

Testing black-box: Testing funzionale che sfrutta la specifica per partizionare lo spazio di input in classi di equivalenza. Ha come obiettivo quello di ridurre il numero di casi di test. Quando un input è valido tra a e b cerco di individuare almeno tre classi (i bordi e un valore centrale), se invece di un intervallo ho un enumerativo devo testare il mio programma su tutti gli input semantici e in più devo testare un valore non valido e vedere cosa succede quando l'input non rispetta la precondizione.

Nel partizionare il dominio in classi di equivalenza l'ideale sarebbe testare ogni classe di equivalenza con classi disgiunte.

Se abbiamo più parametri in una componente da testare selezioniamo un valore per ogni parametro. Per combinarli abbiamo due modi

- Weak: Scegliamo un valore di una variabile per ogni classe

- Strong: Creiamo tutte le possibili combinazioni dei valori delle classe di equivalenza.

Boundary value testing: Abbiamo partizionato l'input in classi supponendo che il comportamento con esse sia simile, ma spesso gli errori avvengono con i limiti di queste classi. Il boundary test si concentra su questo. Questa tecnica si aggiunge a quelle precedenti.

Una funzione con n variabili richiederà 4n + 1 casi di test e funziona bene con le variabili che rappresentano quantità fisiche limitate come intervalli numerici.

Passi della category partition:

1) Il sistema è diviso in singole funzioni che possono essere testate in modo indipendente

2) In ogni funzione il metodo identifica i parametri e per ognuno identifica diverse categorie. Le categorie sono proprietà o caratteristiche principali.

3) Le categorie vengono ulteriormente suddivise in scelte specifiche (possibili valori), applicando lo stesso principio del partizionamento in classi di equivalenza.

4) Si analizzano i vincoli tra le scelte, ad esempio come l'occorrenza di una scelta può influenzare l'esistenza di un'altra.

5) Sono generati i test frame, ovvero tutte le combinazioni di scelte possibili nelle categorie

6) Test frame sono convertiti i test dati.

Vincoli: Proprietà , selettori associati con le scelte. Annotazioni speciali sono [Error] e [Single]

[Error]: I test con questa annotazione sono test progettati per verificare una particolare funzionalità che causa un errore o un eccezione. Una scelta marcata con [Error] non è combinata con le scelte in altre categorie per creare test frame.

[Single]: Serve per descrivere condizioni speciali , insolite o ridondanti che non devono essere combinate con tutte le possibili scelte. Così come error il generatore non combina queste scelte con quelle di altre categorie.

White-box testing: Si concentra sulla minuziosità. Questo test non è scalabile ed è utilizzabile solo a livello di unità.

Tipi di white-box testing:

Statement Testing (Testing algebrico): Verifica l’esecuzione di singole istruzioni, come la scelta degli operatori nei polinomi.

Loop Testing: Testa il comportamento dei cicli in tre scenari: Il ciclo viene completamente saltato. Il ciclo viene eseguito esattamente una volta. Il ciclo viene eseguito più di una volta.

Path Testing: Garantisce che tutti i percorsi nel programma vengano eseguiti almeno una volta.

Branch Testing (Testing condizionale): Verifica che ciascun possibile esito di una condizione (vero o falso) venga testato almeno una volta.

White-box Testing: Richiede il testing di un numero potenzialmente infinito di percorsi. Si concentra su ciò che è implementato, piuttosto che su ciò che dovrebbe essere fatto. Non rileva casi d'uso mancanti.

Black-box Testing: Comporta una potenziale esplosione combinatoria di casi di test (dati validi e non validi). Non è sempre chiaro se i test individuino specifici errori. Non rileva funzionalità superflue o non richieste.

Passaggi per il Processo di Testing

Selezionare cosa misurare: Completeness, verificare la completezza dei requisiti. Reliability, testare il codice per verificarne l'affidabilità. Cohesion, valutare la coesione del design.

Sviluppare i casi di test: Un caso di test è un insieme di dati o situazioni utilizzate per verificare l'unità (codice, modulo, sistema) o l'attributo che si desidera misurare.

Creare l'oracolo di test: L'oracolo predice i risultati attesi per i casi di test. Deve essere documentato prima dell'esecuzione del testing.

M5_03

L'intero sistema è visto come un'insieme di sottosistemi determinati durante il system e l'object design. L'ordine in cui i sottosistemi sono scelti per il testing determina la strategia di testing

Bottom-up: I sottosistemi nel layer più basso sono testati singolarmente. Poi vengono testati i sottosistemi successivi che chiamano i sottosistemi testati in precedenza. Questo viene ripetuto fino a quando tutti i sottosistemi sono inclusi nel testing. Questo sistema non è ottimale per la decomposizione funzionale visto che prova i sottosistemi più importanti alla fine. Risulta utile per i sistemi OO e altro. Richiedono i driver

Top-down: Testa il layer più alto o il sottosistema di controllo per primo. Poi combina tutti i sottosistemi che sono chiamati dal sistema testato e test l'insieme risultante. Fa questo fino a quando tutti i sottosistemi sono incorporati nel test. Richiedono gli stub che potrebbero essere difficili da scrivere.

Sandwich: Combina top-down e bottom-up. Il sistema è visto con 3 layer un layer obiettivo nel centro , e due con uno sopra e un altro sotto l'obiettivo. Il testing converge verso il centro. Questa soluzione non testa singolarmente i sottosistemi del layer obiettivo accuratamente.

Modified Sandwich Strategy: Testa in parallelo il layer centrale, il top con gli stub e bottom con I driver. Entrambi top e bottom acceded al layer centrale.

Structure Testing = White-box testing.
Functional Testing = Black-boc testing

Performance testing: Spinge il sistema verso il suo limite, ha come scopo infatti quello di rompere il sottosistema e vede come si comporta quando è in sovraccarico. Prova cose in ordine insolito

Acceptance Testing: Hanno come scopo quello di dimostrare che il sistema è pronto per l'uso. Viene fatto dal cliente e non dallo sviluppatore. Alpha test provato dallo sviluppatore ed è usato con impostazione controllate. Beta test fatto senza sviluppatori ed ha un'esecuzione più realistica.
