PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

Per altri appunti consultare il sito: *https://luigi-v.github.io/Appunti_Universita/*

By *Vollono Luigi*

Ciclo di Vita del Software

Processo software: insieme organizzato di attività che guidano la costruzione di un prodotto software da parte di un team di sviluppo tramite tecniche, metodi, strumenti e metodologie. E' suddivisibile in varie fasi secondo uno schema di riferimento, ovvero il ciclo di vita del software. E' descritto da un modello formale, semiformale o informale.

Modello di ciclo di vita del software: caratterizzazione descrittiva o prescrittiva di come un sistema software dovrebbe essere sviluppato o è sviluppato.

Modello di processo software: precisa, formalizzata e dettagliata descrizione delle attività degli oggetti, delle trasformazioni e delle strategie che includono per avere k'evoluzione del software.

Le varie fasi del ciclo di vita del software sono:

Definizione: Determinazione dei requisiti, informazioni da elaborare, funzioni e prestazioni attese, comportamento del sistema, interfacce, vincoli progettuali, criteri di validazione.

Sviluppo: Definizione del progetto, dell'architettura software, della strutturazione dei dati e delle interfacce e dei dettagli procedurali; traduzione del progetto nel linguaggio di programmazione; collaudi.

Manutenzione: correzioni, adattamenti, miglioramenti, prevenzione.

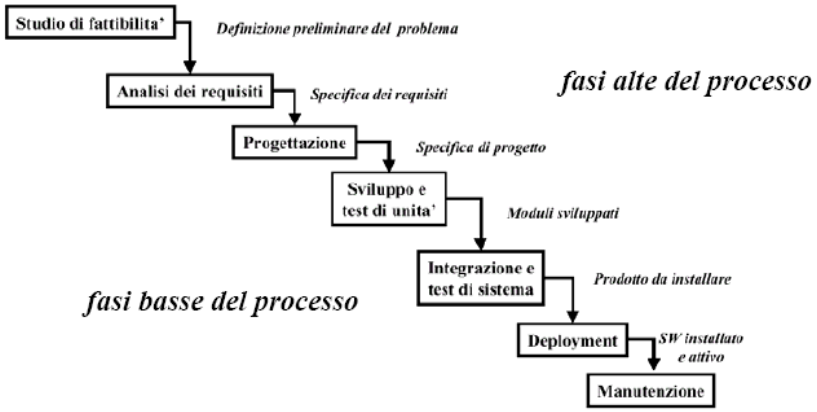
Deliverable: documento rilasciato al committente.

La definizione del modello di ciclo di vita del software è influenzato da molti fattori:
-specificità dell'organizzazione produttrice
-know-how
-area applicativa e particolare progetto
-strumenti di supporto
-diversi ruoli produttore/ committente

I tipi di modello di ciclo di vita del software sono:
Waterfall
Prototyping
Trasformazionale
Basato sul riuso
Spiral model

Waterfall

Modello sequenziale lineare, chiamato "a cascata" perché caratterizzato da una progressione sequenziale di fasi, senza ricicli, per controllare meglio sia tempi che costi. Le varie fasi del processo, in tal caso, sono, quindi, separate: tra di esse c'è overlap nullo o, in alcuni casi, minimo. Ha delle uscite intermedie: semilavorati del processo, cioè documentazione cartacea e programmi.



-ogni fase raccoglie un insieme di attività omogenee per metodi, tecnologie, skill del personale, etc.
-ogni fase è caratterizzata dalle attività (tasks), dai prodotti di tali attività (deliverables), dai controlli relativi (quality control measures)
-la fine di ogni fase è un punto rilevante del processo (milestone)
-i semilavorati output di una fase sono input alla fase successiva
-i prodotti di una fase vengono "congelati", ovvero non sono più modificabili se non innescando un processo formale e sistematico di modifica
-il limite del modello a cascata è la difficoltà ad effettuare cambiamenti nel corso del processo

Studio di fattibilità
Si valutano i costi ed i benefici preliminarmente. E' uno studio che varia a seconda della relazione tra consumatore e produttore.
Ha come obiettivi: stabilire se avviare il progetto, definire le scelte più adatte e le possibili opzioni, valutare le risorse umane e finanziarie necessarie.
Ha come output un documento di fattibilità, ovvero un documento che definisce il problema inizialmente, gli scenari, le strategie alternative di soluzioni, i costi, i tempi, le modalità di sviluppo per ogni alternativa.
Analisi dei requisiti
Si analizzano i bisogni dell'utente e del dominio del problema completamente; vengono coinvolti ingegneri e committenti del software.
Ha come obiettivo il descrivere le caratteristiche di qualità che il sistema deve soddisfare.
Ha come output: documento di specifica dei requisiti, manuale d'utente, piano di acceptance test del sistema.
Progettazione
Si definisce una struttura opportuna per il software, si scompone il sistema in componenti e moduli (vengono allocate le funzionalità ai moduli e vengono definite le relazioni tra i moduli), si definisce l'architettura del sistema.
Ha come obiettivo quello di scegliere come implementare il sistema.
Ha come output il documento di specifica del progetto, in cui a volte sono usati linguaggi o formalismi di progettazione.
Fasi basse del processo
Programmazione test di unità
Ogni modulo viene implementato nel linguaggio scelto e poi testato individualmente.
Integrazione e test di sistema
I moduli vengono composti nel sistema globale, il corretto funzionamento del sistema viene verificato e viene svolto un α test (il sistema è rilasciato al produttore) o un β test (il sistema è rilasciato a pochi utenti scelti).
Deployment
Il software viene distribuito all'utenza e gestito.
Manutenzione
C'è l'evoluzione del software, seguendo l'esigenza dell'utenza. C'è una sorta di iterazione delle fasi precedenti.

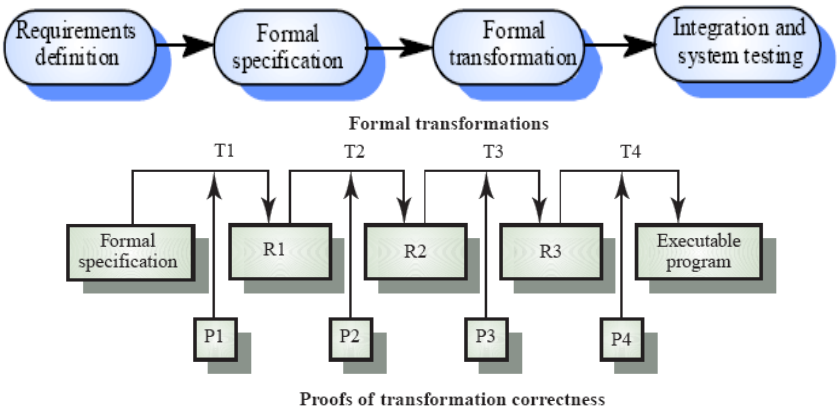
Il modello a cascata è stato molto importante nella storia dell'ingegneria del software, dato che ha introdotto concetti come quello del semilavorato, della fase del ciclo di vita del software ed altri, ha rappresentato un importante punto di partenza per lo studio dei processi software. Inoltre è un modello semplice sia da comprendere che da applicare.
L'interazione tra sviluppatori e committente ci sono solo all'inizio e alla fine delle attività: requisiti troppo tempo congelati e spesso imprecisi (dato anche il fatto che gli utenti sanno quello che vogliono solo quando lo vedono), gli errori nei requisiti si scoprono solo alla fine delle attività. Solo alla fine delle attività il software è installabile e giudicabile da parte degli utenti e del management. La manutenzione è considerata marginale, non c'è overlap o cicli.

V & V e Retroazione

Verifica: Stabilire la verità della corrispondenza tra un prodotto software e la sua specifica.
Convalida: Stabilire l'appropriatezza di un prodotto software rispetto alla sua missione operativa.

Modello Trasformazionale

Si basa sulla trasformazione di una specifica matematica in un applicazione eseguibile: ci sono varie trasformazioni che fanno passare da una rappresentazione ad un'altra e che devono rispettare la correttezza, in modo che il programma soddisfi il committente.

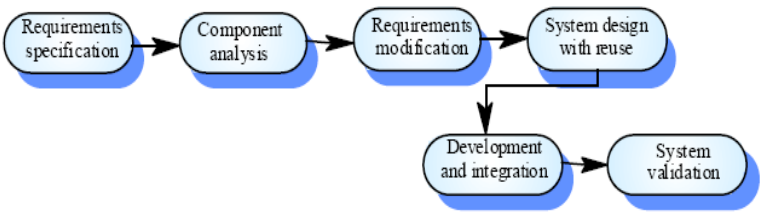


Tale modello può essere utilizzato con altri modelli per migliorare il processo di sviluppo di sistemi critici, soprattutto quelli in cui la sicurezza è molto importante.
Richiede una conoscenza molto approfondita delle proprie tecniche e molta esperienza nell'applicazione della tecnica. Rende difficile specificare formalmente aspetti come le interfacce. Si hanno pochi esempi di applicazione per sistemi complessi.

Modello basato sul riuso

Modello basato su un sistematico riuso. E' adatto a sistema ottenuti integrando componenti esistenti. Consente un rapido sviluppo delle applicazioni. Molto adatto a sitemi software objet-oriented. Componente Based, Design Pattern.

Full Reuse Model
Prevede repository di componenti riusabili a diversi livelli di astrazione, prodotti durante le diverse fasi del ciclo di vita (specifiche, progetti, codice, test case, ecc.).
Durante lo sviluppo di un nuovo sistema avvengono:
-il riuso di componenti esistenti.
-il popolamento delle repository con nuove componenti.



Modello evolutivo

Modello basato sulla prototipazione.

prototipo: mezzo tramite cui si interagisce con il committente per assicurarsi di aver capito bene le richieste, per specificare meglio queste ultime e valutare la fattibilità del prodotto.

Ci sono due tipi di prototipazione in questo caso:

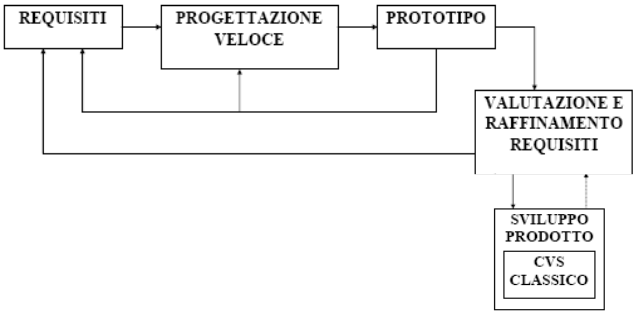
-**evolutivo (sviluppo esplorativo)**: lavorare con il cliente ed evolvere verso il sistema finale a partire da una specifica di massima. Lo sviluppo inizia con le parti del sistema che sono già ben specificate, aggiungendo via via nuove caratteristiche

-**usa e getta (throw away)**: capire i requisiti del sistema e quindi sviluppare una definizione migliore dei requisiti. Il prototipo sperimenta le parti del sistema che non sono ancora ben comprese.

Modello su prototipo throw away

In tal caso il prototipo è uno strumento di identificazione dei requisiti di utene ed è incompleto, approssimativo, realizzato utilizzando parti già possedute, ma soprattutto deve essere gettato.

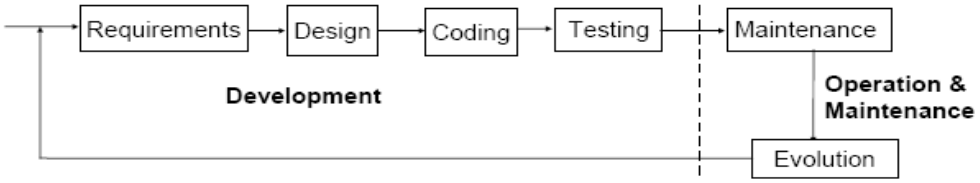
Realizzazione di una prima implementazione (prototipo), più o meno incompleta da considerare come una prova, con lo scopo di accertare la fattibilità del prodotto e validare i requisiti.
dopo la fase di utilizzo del prototipo si passa alla produzione della versione definitiva del Sistema SW mediante un modello che, in generale, è di tipo waterfall.



Nel caso di modello basato su prototipazione throw away, si possono avere due tipi di prototipazione:
mock up: produzione completa dell'interfaccia utente. Consente di definire con completezza e senza ambiguità i requisiti (si può, già in questa fase, definire il manuale di utente).
breadboard: implementazione di sottoinsiemi di funzionalità critiche del SS, non nel senso della fattibilità ma in quello dei vincoli pesanti che sono posti nel funzionamento del SS (carichi elevati, tempo di risposta, ecc.), senza le interfacce utente. Produce feedbacks su come implementare la funzionalità (in pratica si cerca di conoscere prima di garantire).

Modello su prototipo a sviluppo esplorativo

Il prodotto SW evolve continuamente (aggiunta di funzionalità, cambio di piattaforma, cambiamenti nell'organizzazione dell'azienda che lo utilizza e quindi adeguamento del SW, ecc.).



Nella fase di evoluzione:
-si analizza l'esperienza di uso del SW sul campo e si utilizza la maggiore conoscenza per definire nuovi obiettivi;
-si determinano esigenze e nuove funzionalità emerse o non coperti in precedenza;
-si riprende il ciclo dalla definizione dei requisiti alla messa in esercizio e manutenzione del nuovo SW nato dall'arricchimento ed evoluzione del precedente.
Problemi: mancanza di visibilità del processo, sistemi spesso poco strutturati, possono essere richieste particolari capacità (ad esempio in linguaggi per prototyping rapido).
Applicabilità: sistemi interattivi di piccola o media dimensione, per parti di sistemi più grandi (es. interfaccia utente), per sistemi a vita breve.

Modello Incrementale

Risolvono la difficoltà a produrre l'intero Sistema in una sola volta nel caso di grandi progetti SW (sia per problemi del produttore che del committente (quest'ultimo potrebbe non avere l'immediata disponibilità finanziaria necessaria per l'intero progetto)

Modelli incrementali:
-**modello ad implementazione incrementale**: Le fasi alte del Waterfall Modell sono completamente realizzate. Il SW viene totalmente definito nei requisiti, specificato e progettato. I sottosistemi vengono implementati, testati, rilasciati, installati e messi in manutenzione secondo un piano di priorità in tempi diversi.
-**modello a sviluppo e consegna incrementale**: L'approccio incrementale viene esteso a tutte le fasi del ciclo di sviluppo waterfall-like. È un partizionamento del processo di produzione waterfall: ogni partizione definisce un insieme di ben definiti comportamenti del sistema e procede dalla definizione dei requisiti all'implementazione, etc. del software che riproduce tali comportamenti; i prodotti di ogni nuova slice vengono aggiunti ed integrati con l'esistente e costituiscono un incremento del sistema.

Modello a spirale

Formalizzazione del concetto di modello evolutivo e ripetizione ciclica di task regions:
-Determinazione obiettivi, vincoli, alternative
-Valutazione alternative, analisi dei rischi
-Sviluppo, verifica e convalida
-Pianificazione prossimo ciclo

Il ciclo a cascata si può vedere come un caso particolare, cioè una sola iterazione di tale modello.

Vantaggi
• Rende esplicita la gestione dei rischi
• Focalizza l'attenzione sul riuso
• Aiuta a determinare errori nelle fasi iniziali
• Obbliga a considerare gli aspetti di qualità
• Integra sviluppo e manutenzione
• Costituisce un framework di sviluppo hardware/software

Svantaggi
• Per contratto di solito si specifica a priori il modello di processo e i "deliverables". Lo sviluppo richiede un contratto nel contratto: vanno specificati vincoli, modello di processo, tempi di consegna e artefatti da consegnare
• Richiede persone in grado di valutare i rischi
• Per poter essere usato deve essere adattato alla realtà aziendale e/o al team

Il manager ha il compito di minimizzare i rischi, che possono essere di varie tipologie: personale adeguato, scheduling, budget non realistico, sviluppo del sistema sbagliato e altro.
Meno informazione si ha più alti sono i rischi e alcuni alti rischi provocano ritardi e costi imprevisti.
Cascata
-alti rischi per sistemi nuovi, non familiari per problemi di specifica e progetto
-bassi rischi nello sviluppo di applicazioni familiari con tecnologie note
Prototipazione
-bassi rischi per le nuove applicazioni, specifica e sviluppo vanno di pari passo
-alti rischi per la mancanza di un processo definito e visibile
Trasformazionale
-alti rischi per le tecnologie coinvolte e le professionalità richieste

Raccolta dei Requisiti

La raccolta dei requisiti si incentra sulla descrizione dello scopo del sistema. I clienti, gli sviluppatori e gli utenti identificano un problema e definiscono un sistema che possa risolverlo. La raccolta dei requisiti fa da contratto tra clienti e sviluppatori sul sistema.

Requisito: caratteristica che un sistema deve possedere o vincolo che il sistema deve soddisfare.
Documento di Specifica dei Requisiti: documento che rappresenta un contratto tra clienti e sviluppatori del sistema a cui fa riferimento, dato che contiene la specifica dei requisiti richiesti per il sistema. E' scritto in linguaggio naturale.
Modello di Analisi: modello che contiene la specifica dei requisiti richiesti per un sistema. E' scritto in un linguaggio di modellazione (come ad esempio l'UML), è caratterizzato di notazioni formali o semiformali e supporta la comunicazione tra gli sviluppatori del sistema.

Scenario: esempio concreto di utilizzo del sistema in termini di interazioni tra sistema ed utente, che serve a comprendere il dominio di applicazione.
Use case: astrazione che descrive una classe di scenari.

Attività della Raccolta dei Requisiti:
Identificazione degli attori: individuazione degli utenti che il sistema dovrà supportare.
Identificazione degli scenari: descrizione degli scenari che rappresentano le funzionalità del sistema.
Identificazione degli use case: derivazione di use case dagli scenari, in modo da rappresentare completamente e abbastanza dettagliatamente il sistema.
Raffinamento degli use case: revisione degli use case identificati precedentemente per far sì che tutte le funzionalità del sistema siano rappresentate con essi ed aggiunta di casi d'uso che descrivano comportamenti eccezionali in condizioni limite o di errore, ad esempio.
Identificazione delle relazioni tra gli use case: individuazione delle dipendenze e delle associazioni tra gli use case una volta che questi sono stati identificati e raffinati.
Identificazione dei requisiti non funzionali: descrizione degli aspetti che non hanno legami con la funzionalità del sistema ma che sono visibili all'utente finale, ovvero i requisiti non funzionali.

Requisito Funzionale: requisito che descrive le interazioni tra sistema e suo ambiente (utente e ogni alktro sistema esterno che interagisce con esso) indipendentemente dalla implementazione del sistema stesso.
Requisito non Funzionale: requisito che descrive aspetti del sistema che non sono direttamente realtivi al comportamento del sistema stesso e quindi alle sue funzionalità.

Il modello FURPS+ definì vari gruppi di requisiti non funzionali, chiamati anche requisiti di qualità:
Usabilità: facilità di imparare ad usare il sistema per l'utente (convenzioni per interfacce utenti, portata dell'Help in linea, livello della documentazione utente, ecc.).
Affidabilità: capacità di un sistema o di una sua componente di offrire in maniera corretta la funzione ad esso richiesta sotto certe condizioni o per periodi di tempo (tempo di fallimento accettabile, abilità di scoprire difetti, abilità di resistere ad attacchi alla sicurezza, ecc.)
Performance: valore di attributi quantificabili, in particolare di tre tipi:
- tempo di risposta: quanta velocità di risposta del sistema ad input utenti.
- throughput: quanto lavoro il sistema riesce a fornire in un determinato lasso di tempo.
- disponibilità: quanta accessibilità il sistema offre per una sua componente o per l'intero sistema stesso quando è richiesta.

Supportabilità: semplicità di modifica del sistema dopo il suo sviluppo, in particolare riguarda:
- adattabilità: abilità di modifica del sistema al fine di trattare ulteriori concetti del dominio di apllicazione.
- mantenibilità: abilità di modifica del sistema al fine di trattare nuove tecnologie e per correggere difetti.

Il modello FURPS+ definì anche vari gruppi di requisiti non funzionali particolari, chiamati anche pseudorequisiti, dato che non sono dei requisiti a tutti gli effetti, o chiamati vincoli.
Implementazione: sviluppo del software del sistema progettato (vincoli sull'uso di tool specifici, linguaggi di programmazione, piattaforme hardware, ecc.).
Interfacce: strumenti di comunicazione con altri sistemi (vincoli imposti da sistemi esterni, come i legacy, vincoli sui formati di scambio, ecc.).
Operazioni: amministrazione e gestione del sistema.
Packaging: consegna reale del sistema.
Legali: certificazioni, licenze e regolamentazioni.

Validazione dei Requisiti: controllo fatto sulla specifica delle informazioni (i requisiti) che servono per dare forma al sistema. Tale controllo è fatto continuamente dagli sviluppatori con la collaborazione di utenti e clienti ed è fatto principalmente per rispettare la correttezza, la completezza, la consistenza e la chiarezza della specifica.
Correttezza: accurata rappresentazione da parte della specifica del sistema richiesto dai clienti e inteso dagli sviluppatori.
Completezza: inclusione nella specifica di tutti gli scenari, compresi i comportamenti eccezionali.
Consistenza: mancanza di contraddizione dei requisiti della specifica.
Chiarezza: mancanza di ambiguità nella specifica e, quindi, mancanza di interpretazioni ulteriori a quella voluta dagli sviluppatori nella specifica.

Ci sono tre importanti criteri per la validazione dei requisiti:
Realismo: tenendo conto dei vincoli si può implementare bene la specifica.
Verificabilità: test ripetuti possono dimostrare che il sistema costruito a partire dalla specifica soddisfa i requisiti vari.
Tracciabilità: ogni funzione del sistema corrisponde ad un requisito funzionale e si possono tracciare le dipendenze tra i requisiti, le funzioni di sistema, gli artefatti (anche componenti, classi, metodi, attributi degli oggetti, ecc.).

Ci sono tre tipi di raccolta dei requisiti diversificati sulla base della sorgente dei requisiti:
Greenfield Engineering: lo sviluppo della raccolta parte da zero e non da altri sistemi, quindi i requisiti sono estratti da utenti e clienti, da cui dipende strettamente.
Reengineering: lo sviluppo della raccolta parte da uno già esistente al fine di costruire un sistema con nuove tecnologie, con nuove funzionalità o con un qualcos'altro in più o di diverso rispetto al sistema già progettato.
Interface Engineering: lo sviluppo della raccolta parte da un sistema già esistente al fine di adattarlo ad un nuovo ambiente e, quindi, di riproggettargli le interfacce, mentre tutto il resto del vecchio sistema rimane inalterato.

Identificazione degli attori
Un attore è un sistema esterno, un ambiente fisico o un utente che interagisce con i sistema; ha un unico nome e una descrizione opzionale.

Identificazione degli scenari
Uno scenario è una descrizione concreta, focalizzata ed informale di una caratteristica del sistema sfruttata da un solo attore.
Ci sono vari tipi di scenario:
As-is scenario: usati per descrivere un situazione corrente, in essi l'utente descrive il sistema. Sono spesso usati nel caso di reengineering.
Visionary scenario: usato per descrivere prospettive future del sistema, a volte non definiti solo da utenti del sistema e sviluppatori del sistema. Spesso usati nel caso di greenfield engineering e reengineering.
Evaluation scenario: usato per descrivere task rispetto ai quali poi viene valutato il sistema, i task sono assegnati agli utenti.
Training scenario: usato come tutorial per aggiungere al sistema il supporto a nuovi utenti.

Identificazione dei casi d'uso
Un caso d'uso è una descrizione di interazioni tra attore e sistema che avvengono dopo la sua inizializzazione; è inizializzato da un attore e specifica sostanzialmente una funzionalità del sistema.
Per costruire un caso d'uso bisogna descrivere: il nome, gli attori, le condizioni di entrata, le condizioni di uscita, le eccezioni, il flusso degli eventi e i requisiti speciali, detti anche vincoli, del caso d'uso stesso.

Raffinamento dei casi d'uso
Si dettagliano gli elementi manipolati dal sistema, si specificano le sequenze d'interazione di basso livello tra sistema e attore, si specificano gli accessi corretti alle funzionalità del sistema, si identificano e specificano le eccezioni mancanti, si separano le funzionalità comuni a più casi d'uso.

Stereotipo: mezzo tramite cui si può classificare gli elementi di un modello (se S è uno stereotipo di M, allora S conserva le caratteristiche di M, S può essere usato per definire una differenza nella semantica o nel modo d'uso di M e può soddisfare nuove constraint ulteriori a quelle che soddisfa M e avere nuovi tagged values).

Identificazione delle relazioni tra gli use case
Le relazioni tra use case sono di vari tipi:
Extend: un caso d'uso estende un altro caso d'uso (per estendere una funzionalità facendo in modo tale che si possa usufruire della funzionalità originaria anche senza considerare la sua estensione).
Include: un caso dìuso include un altro caso d'uso molto comune tra vari casi d'uso, il primo compreso (per scomporre una funzione molto complessa in funzioni più semplici o per riusare funzioni già esistenti e, quindi, comuni a più casi d'uso).
Generalization: un caso d'uso astratto ha varie specializzazioni (per ricondurre vari casi d'uso ad un caso d'uso più generale o fattorizzare).
Inheritance: un caso d'uso eredita delle caratteristiche da uno o più altri casi d'uso (per considerare in un caso d'uso funzionalità di uno o più altri casi d'uso).

Identificazione dei requisiti non funzionali
Per ogni tipo di requisito non funzionale ci si pone delle domande a cui si risponde grazie al supporto dei clienti e degli utenti del sistema.

Gestendo la raccolta dei requisiti si svolgono varie attività:
-Negoziazione delle specifiche con i clienti
-Mantenimento della tracciabilità, detto anche evoluzione dei requisiti
-Documentazione della raccolta dei requisiti

Negoziazione delle specifiche con i clienti

Metodo Joint Application Design

Utenti, cliente, sviluppatori, e il responsabile della sessione si siedono insieme intorno ad un tavolo e presentano i loro punti di vista, ascoltano i punti di vista degli altri, negoziano, e arrivano ad una soluzione accettabile per tutti.
Il risultato è un documento finale, che è una completa specifica dei requisiti e include definizioni dei dati, flussi di lavoro, e descrizione delle interfacce
JAD è composto da 5 attività: Definizione del progetto, Ricerca, Preparazione, Sessione, Documento finale.
Definizione del progetto: l'agevolatore intervista il project manager e il cliente per determinare gli obiettivi del progetto (output: Management Definition Guide).
Ricerca: l'agevolatore intervista gli utenti attuali e quelli futuri, raccoglie informazioni sul dominio di applicazione, e descrive ad alto livello gli use case (output: Session Agenda e Preliminary Specification).
Preparazione: l'agevolatore prepara la sessione, e crea un documento di lavoro, che è un prima bozza del documento finale, un'agenda per la sessione, e ogni altro documento cartaceo che rappresenta le informazioni raccolte durante la Ricerca.
Sessione: l'agevolatore guida il team nella creazione della specifica dei requisiti. Il team definisce e si accorda sugli scenari, sugli use case, sui mockup delle interfacce.
Documento finale: l'agevolatore prepara il documento finale, rivedendo i documenti di lavoro per includere tutte le decisioni prese durante la sessione.

Evoluzione dei requisiti

Si seguono i cambiamenti dei requisiti, tenendo traccia della provenienza delle richieste e quali aspetti del sistema i requisiti interessano (quale componente soddisfa una richiesta, quale test controlla tale soddisfazione, ecc.).
Per garantire tale tracciabilità si possono usare riferimenti incrociati ai vari modelli, artefatti e documenti costruiti in precedenza per il sistema (identificando, ad esempio, ogni componente individuale come requisito, operazione, componente, classe con un numero unico).
Per progetti molto grandi e complessi si possono utilizzare tool per gestire i requisiti catturando, editando e tracciando le dipendenze dei requisiti con un buon livello di dettaglio.

Documentazione della raccolta dei requisiti

I risultati della raccolta dei requisiti sono inseriti nel RAD (Requiremente Elicitation Document), un documento che descrive completamente l'aspetto dei requisiti funzionali e l'aspetto dei requisiti non funzionali del sistema e che fa da base per il contratto tra clienti e sviluppatori. Parte di tale documento è scritta durante l'analisi dei requisiti: la formalizzazione della specifica del sistema in termini di modelli degli oggetti è fatta durante l'analisi dei requisiti.

Analisi dei Requisiti

L'analisi dei requisiti ha come obiettivi: la formalizzazione della specifica delle richieste prodotta durante la raccolta dei requisiti; validare, correggere e chiarire errori che eventualmente sono presenti nella specifica delle richieste; esaminare molto dettagliatamente le condizioni limiti e i casi eccezionali.
Durante tale fase della progettazione del sistema, viene costruito un modello che descrive il dominio di applicazione. Il modello di analisi viene esteso per descrivere come attori e sistema interagiscono per manipolare il modello del dominio di applicazione.

Le attività dell'analisi dei requisiti sono:
Identificazione degli oggetti
Definizione del comportamento degli oggetti
Definizione delle relazioni tra gli oggetti
Classificazione degli oggetti
Organizzazione degli oggetti

La formalizzazione traduce la specifica dei requisiti in un modello del sistema formale o semiformale, quindi permette di conoscere meglio tale specifica e, di conseguenza, aiuta ad individuare aree di ambiguità, inconsistenze ed omissioni. La specifica potrebbe avere, dunque, uno di questi tre problemi, risolvibili grazie all'acquisizione di maggiori informazioni da parte dei clienti. Dato ciò, la raccolta dei requisiti e l'analisi dei requisiti sono attività svolte in concorrenza, oltre che incrementali ed iterative.

Il modello di analisi è composto da tre modelli:
Modello funzionale: modello che è rappresentato da scenari e use case.
Modello ad oggetti di analisi: modello che è rappresentato dal diagramma delle classi e dal diagramma degli oggetti, rappresenta il sistema dal punto di vista dell'utente, è incentrato sui concetti che sono manipolati dal sistema, le loro proprietà e le relazioni. Con i class diagram si definiscono classi, attributi ed operazioni.
Modello dinamico: modello che è rappresentato dagli statechart diagram e dai sequence diagram, è incentrato sul comportamento del sistema, permette di assegnare responsabilità alle classi e di individuare nuove classi da aggiungere al modello ad oggetti di analisi. Con i sequence diagram si rappresentano le interazioni tra un gruppo di oggetti durante uno use case, mentre con gli statechart diagram si rappresenta il comportamento di un singolo oggetto o di più oggetti strettamente relati.

Il modello ad oggetti ed il modello dinamico derivano da un raffinamento del modello funzionale (scenari ed use case) svolto durante l'analisi dei requisiti.

Il modello ad oggetti di analisi consiste di:
Oggetti Entity: oggetti che rappresentano un'informazione persistente.
Oggetti Boundary: oggetti che rappresentano le interazioni o interfacce tra attori e sistema.
Oggetti Control: oggetti che si occupano di realizzare gli use case coordinando gli oggetti.

Con tale approccio thre-object-type i modelli ad oggetti di analisi sono più flessibili.

Ereditarietà: proprietà consente di organizzare i concetti in gerarchie.
Generalizzazione: attività di modellazione che identifica concetti astratti da concetti di più basso livello.
Specializzazione: attività di modellazione che identifica concetti specifici da concetti di più alto livello.

Attività che servono per il passaggio da use case ad oggetti:
-identificazione degli oggetti entity
-identificazione delgi oggetti boundary
-identificazione degli oggetti control
-mapping degli use case in oggetti con sequence diagram
-identificazione delle associazioni
-identificazione delle aggregazioni
-identificazione degli attributi
-modellazione del comportamento dipendente dallo stato degli oggetti individuali
-modellazione delle relazioni di ereditarietà
-revisione del modello di analisi

Mapping degli use case in oggetti con sequence diagram

Un sequence diagram costruisce relazioni tra use case e oggetti, dato che mostra come il comportamento di uno use case è distribuito tra gli oggetti èartecipanti allo use case stesso ed illustra la sequenza di interazioni tra gli oggetti necessari per realizzare uno use case.
Durante l'analisi i Sequence Diagram sono usati per individuare nuovi oggetti e comportamenti mancanti.
Disegnare Sequence Diagram è un'attività laboriosa, quindi occorre dare priorità a quelle funzionalità problematiche o non ben specificate; inoltre per le parti ben definite può essere utile solo per evitare di posticipare alcune decisioni chiavi.

Identificazione delle associazioni

Le associazioni tra entity object sono le più importanti poichè rivelano altre informazioni sul dominio di applicazione.
Per costruire un buon sistema bisogna:
-eliminare qualsiasi associazione che può essere derivata da altre associazioni
-evitare troppe associazioni, perché, se troppe, rendono il modello illeggibile
-eliminare le associazioni ridondanti

Identificazione delle aggregazioni

Aggregazione: tipo speciale di associazione tra oggetti che denota relazioni whole-part.

Identificazione degli attributi

Attributo: proprietà individuale di un singolo oggetto.
Per costruire un buon sistema bisogna:
-considerare solo gli attributi che sono rilevanti per il sistema
-identificare tutte le associazioni prima di iniziare l'identificazione degli attributi

Gli attributi hanno:
-un nome
-una breve descrizione
-un tipo che descrive i valori che può assumere

Gli statechart diagram sono caratterizzati da due tipi di operazioni:
Attività: operazioni che richiedono tempe per essere eseguite e sono associate agli stati dell'oggetto.
Azioni: operazioni istantanee e che sono associate con gli eventi dell'oggetto e con gli stati: entry, exit e internal dell'oggetto.

Gli statechart diagram permettono allo sviluppatore di costruire una descrizione più formale dell'oggetto e di conseguenza permettono di identificare casi d'uso mancanti.
Focalizzando l'attenzione sui singoli stati gli sviluppatori possono identificare nuovi comportamenti.
Non è necessario creare uno statechart per ogni classe nel sistema (solo quelli con una lunga vita ed un comportamento dipendente dallo stato), molto spesso sono control object.

Revisione del modello di analisi

Il modello di Analisi è costruito in maniera incrementale ed iterativa.
Una volta che il modello diventa stabile, il modello di analisi viene revisionato prima dagli sviluppatori (revisione interna), poi congiuntamente dagli sviluppatori e dal cliente.
In tal modo si ha una specifica dei requisiti corretta, completa, consistente e non ambigua.

Responsabilità durante l'analisi dei requisiti

Ci sono tre tipi di ruoli:
generatori di informazioni
End user: generano informazioni sul sistema
Client: integra le informazioni del dominio di applicazione e risolve le inconsistenze
integratori
Analyst: modella il sistema e genera informazioni sul sistema da costruire
Architect: integra i casi d'uso e i modelli ad oggetti dal punto di vista del sistema
revisori
Reviewer: valida il RAD

Modeling with UML

Dominio di applicazione: ambiente in cui il sistema sta operando; include l'ambiente fisico, gli utenti ed altre persone, i loro processi di lavoro, ecc.
Dominio di soluzione: tecnologie disponibili per costruire il sistema.
Analisi object-oriented: analisi che tratta il dominio di applicazione.
Disegno object-oriented: disegno che tratta il dominio di soluzione.
Unified Modeling Language: linguaggio per la modellazione di software object-oriented e che risulta dalla convergenza di notazione provenienti da tre linguaggi dominanti per la modellazione: OMT, OOSE e Booch.
Use case diagram: descrive il comportamento funzionale del sistema come visto dall'utente.
Class diagram: descrive la struttura statica del sistema: oggetti, attributi ed associazioni.
Interaction diagram: traccia il comportamento dinamico tra attori, sistema ed oggetti.
Statechart diagram: descrive il comportamento dinamico di alcuni oggetti di una classe.
Activity diagram: modella il comportamento dinamico di un insieme di oggetti interagenti, in particolare il flusso di lavoro (workflow).

Package: meccanismo dell'UML per organizzare elementi in gruppi, solitamente non di concetti del dominio di applicazione.

Use Case

E' una classe di funzionalità fornite dal sistema come flusso di eventi.
Consiste di:
-Nome
-Attori
-Condizioni di entrata
-Flusso di eventi
-Condizioni di uscita
-Requisiti speciali
Ci sono possibili relazioni tra use case:
extend relationship: rappresenta casi eccezionali o rari di uno use case.
include relationship: rappresenta casi d'uso abbastanza comuni ed interni, infatti, al flusso di eventi di un altro use case.
inheritance relationship: rappresenta una specializzazione di uno use case.

Class Diagram

Rappresenta la struttura del sistema. E' utilizzato: durante la raccolta dei requisiti per modellare concetti del dominio del problema, durante il system design per modellare i sottosistemi e le interfacce, durante l'object design per modellare le classi.
Una classe rappresenta un concetto e contiene stato, come attributi, e comportamento, come operazioni.
Ogni attributo ha un tipo ed un nome.
Ogni operazione ha una firma.
Il nome della classe è l'unica informazione necessaria per una classe.
Un oggetto è un'istanza di una classe.
Lo stato di un oggetto è l'insieme delle sue coppie di attributi.

Sequence Diagram

E' usato: durante la raccolta dei requisiti per raffinare la descrizione degli use case e trovare nuovi oggetti partecipanti allo use case, durante il system design per raffinare le interfacce.
Consiste di:
-Classe
-Messaggio
-Attivazione
-Linea di vita

Activity Diagram

Mostra il flusso di controllo all'interno del sistema; è un caso speciale di statechart diagram in cui gli stati sono attività, dette anche funzioni. Due tipi di stato: azione, che non può essere suddiviso ulteriormente ed accade istantaneamente rispettando il livello di astrazione del modello ; attività, può essere suddiviso ulteriormente ed è modellata da un altro activity diagram.

System Design

Il system design si incentra sul dominio di implementazione, a differenza dell'analisi, che si incentra sul dominio di applicazione. Il System design è svolto dagli sviluppatori del sistema.

Attività del system design:
Identificazione degli obiettivi di design: si identificano e definiscono le priorità delle qualità del sistema.
Progettazione della decomposizione del sistema in sottosistemi: si scompone il sistema in parti più piccole usando stili architetturali standard e basandosi sui modelli di analisi e sugli use case.
Raffinamento della decomposizione in sottosistemi per rispettare gli obiettivi di design: si raffina la scomposizione iniziale per far sì che soddisfi gli obiettivi di design.
Come output del system design si ha un documento composto in particolar modo da informazioni riguardanti:
Obiettivi di design: informazioni che descrivono la qualità del sistema.
Architettura software: informazioni sulla decomposizione del sistema, in particolare le responsabilità di ogni sottosistema (in questo modo ogni sottosistema può essere facilmente assegnato ad un team e realizzato a sè), le dipendenze fra i sottosistemi, l'hardware associato ai vari sottosistemi, le decisioni (politiche) in merito al flusso di controllo, al controllo degli accessi e alla memorizzazione dei dati.
Boundary use case: informazioni che descrivono la configurazione del sistema, le scelte per lo startup, per lo shutdown e per la gestione delle eccezioni.

Identificazione degli obiettivi di design

Si identificano le quialità su cui il sistema deve essere focalizzato. I design goals possono essere ricavati dai requisiti non funzionali, o dal dominio di applicazione o dai clienti. Formalizzarli in maniera esplicita è importante perché ogni decisione di design importante deve essere presa seguendo lo stesso insieme di criteri. Questi criteri possono essere di cinque tipi:
Prestazioni: requisiti imposti sul sistema in termini di spazio e tempo, in particolare:
Tempo di risposta: tempo per la soddisfazione di una richiesta immessa.
Throughput: lavoro svolto in una determinata unità di tempo.
Memoria: spazio richiesto dal sistema per il suo funzionamento.
Affidabilità: sforzi per minimizzare i crash e le loro conseguenze, in particolare:
Robustezza: capacità di resistere ad input non validi.
Attendibilità: differenza tra comportamento effettivo e comportamento specificato.
Disponibilità: tempo in cui il sistema può essere utilizzato per svolgere attività.
Tolleranza ai fault: capacità di lavorare sotto errori.
Sicurezza: resistenza ad attacchi che mirano al danneggiamento del sistema.
Fidatezza: capacità di evitare di danneggiare persone pur in stato di errore o fallimento.
Costi: costo dello sviluppo del sistema iniziale, costo del training degli utenti, costo dell'installazione del sistema, costo della conversione dei dati del sistema precedente (nel caso di backward compatibility, cioè di richiesta di compatibilità con un vecchio sistema), costo di manutenzione (costo di correzione errori hardware ed errori software), costo dell'amministrazione del sistema.
Mantenimento: difficoltà della modifica del sistema dopo il suo rilascio, in particolare:
Estendibilità: facilità di aggiunta di nuove classi o nuove funzionalità al sistema.
Modificabilità: facilità di modifica delle funzionalità del sistema.
Adattabilità: facilità di spostamento a diverso dominio di applicazione.
Portabilità: facilità di spostamento a diversa piattaforma.
Leggibilità: facilità di comprensione del sistema con la lettura del suo codice.
Tracciabilità dei Requisiti: facilità di mappare il codice nei requisiti corrispondenti.
Criteri End User: qualità desiderate dagli utenti ma non coperte dai criteri di performance ed affidabilità, in particolare:
Utilità: qualità del supporto del lavoro dell'utente.
Usabilità: semplicità dell'uso del sistema da parte dell'utente.

Trade-off di Design
Spazio vs. velocità (più memoria per velocizzare il sistema o compressione e meno memoria a discapito della velocità).
Tempo di rilascio vs. funzionalità(meno funzionalità di quelle richieste per minor tempo di rilascio).
Tempo di rilascio vs. qualità (il project manager può rilasciare il software nei tempi prefissati con dei bug e successivamente correggerli, o rilasciare il software in ritardo con meno bug).
Tempo di rilascio vs. staffing (aggiungere delle risorse al progetto per accrescere la produttività).

Due proprietà dei sottositemi:
Coupling: dipendenza fra due sottosistemi.
Cohesion: dipendenza fra classi di un sottosistema.

Due tecniche per relare tra loro sottosistemi e ridurre la complessità del sistema:
Layering: organizza un sistema come una gerarchia di sottosistemi.
Partitioning: pone tutti i sottosistemi di un sistema sullo stesso piano e fa sì che ognuno di essi offra diversi servizi agli altri sottosistemi.

Dividere il sistema in sottosistemi riduce la complessità della soluzione, perché ogni piccola parte del sistema può essere svolta da uno sviluppatore o da un team di sviluppatori. Lavorando individualmente, si ha il minimo overhead di comunicazione. Inoltre, sottosistemi complessi possono essere anch'essi decomposti in sottosistemi più piccoli.

Ogni sottosistema offre servizi.
Servizio: iniseme di operazioni fornite dal sistema.
Interfaccia del sottosistema: insieme di operzioni di un sottosistema che sono disponibili ad altri sottosistemi. Tale interfaccia contiene varie informazioni sulle operazioni: loro tipo di valori di riotrno, loro nome, loro tipo, loro tipo di parametri.

Per avere un buon sistema, i suoi sottosistemi dovrebbero essere:
Loosely coupled - perché, se sono strettamente accoppiati, le modifiche su di un sottosistema avranno poca influenza sul sottosistema accoppiato ad esso e viceversa. Al contrario, due sistemi strongly coupled si influenzano molto a vicenda. Diminuendo il coupling, però, spesso si rischia di aumentare molto la complessità del sistema, aggiungendo altri livelli di astrazione che portano i tempi di risposta e quelli di sviluppo ad aumentare; per questo motivo, il coupling deve essere ridotto in questi modi solo quando c'è un elevata probabilità di influenza di un sottosistema su altri sottosistemi.
Altamente coeso – perché, se le classi di un sottosistema sono altamente coese, allora realizzano compiti simili e sono strettamente correlate le une alle altre tramite associazioni. Se la coesione di queste classi è bassa, allora le classi sono poco correlate tra loro oppure nel sottosistema ci sono classi non correlate affatto tra loro.

Layering

Layer: insieme di sottosistemi che forniscono servizi correlati, anche se realizzati utilizzando servizi di sottosistemi di altri layer.
Ogni layer appartiene ad un livello. Un layer può accedere solo ai layer di livelli più bassi al proprio e non ha conoscenza dei layer ai livelli più alti al proprio.
Architettura chiusa: ogni layer può accedere solo al layer del livello immediatamente più basso del proprio.
Architettura aperta: ogni layer può accedere ai layer di ogni livello più basso del proprio.
Di solito l'architettura aperta offre grande efficienza al tempo di esecuzione, mentre l'architettura chiusa offre alta manutenibilità.

I sistemi layered hanno vari vantaggi:
-essendo gerarchici, la loro complessità non è elevata, perché la gerarchia riduce la complessità dei sistemi;
-le architetture chiuse sono più portabili;
-le architetture aperte sono più efficienti;
Se un sottosistema è layer, spesso viene chiamato macchina virtuale.
Macchina virtuale: astrazone che fornisce un insieme di attributi e di operazioni.

Partitioning

Peer: sottosistema paritario rispetto ad altri sottosistemi dello stesso sistema.
Il partitioning prevede che si suddivida il sistema in peer, ognuno dei quali fornisce diverse classi di servizi.

Molto spesso, la decomposizione del sistema in sottosistemi è fatta utilizzando sia partitioning che layering: prima si suddivide il sistema in peer, ognuno dei quali offre specifiche funzionalità; poi, se i peer sono complessi li si suddivide in layer, fino a renderli facilmente implementabili da un singolo sviluppatore o team di sviluppatori.

Ci sono vari stili architetturali che possono fare da base per l'architettura software di un sistema:
Repository Architecture
Model View Controller Architecture
Client Server Architecture
Peer to Peer Architecture

Repository Architecture

I sottosistemi sono loosely coupled ed interagiscono solo attraverso un repository, cioè una struttura dato a cui accedono e che modificano. Il flusso di controllo è dettato dal repository (cambiamento nei dati memorizzati) oppure dai sottosistemi (lock, synchronization primitives).
Esempi di applicazioni che sfruttano questo stile di architettura sono i Compilatori e i Database Management System.
Vantaggi di questo stile: -adatto per applicazioni con task di elaborazione dati che cambiano molto frequentemente.
-è facile definire nuovi sottosistemi per un sistema quando c'è un repository definito.
Svantaggi di questo stile: -il repository può rendere abbassare per il sistema il livello di modificabilità e quello di prestazione.
-il coupling tra sottosistemi e repository è molto alto, quindi il repository influenza molto i sottosistemi del sistema.

Model View Controller Architecture

I sottosistemi sono divisi in tre categorie:
Sottosistema Model: si occupa di mantenere la conoscenza del dominio di applicazione.
Sottosistema View: si occupa di visualizzare agli uteni gli oggetti del dominio di applicazione.
Sottosistema Controller: si occupa di gestire la sequenza di interazioni tra sistema ed utenti.

Lo stile di architettura software Model View Controller Architecture è un caso particolare dello stile di architettura Repository Architecture. Infatti il sottosistema model implementa la struttura dati centrale, mentre il sottositema controller gestisce il flusso di controllo ricevendo input dall'utente e mandando messaggi al modello, mentre il sottosistema view visualizza il modello.

Model, View e Controller sono separati anche perché le interfacce utenti sono più spesso soggette a cambiamenti rispetto alla conoscenza del dominio di applicazione.
Gli svatnaggi e i vantaggi di questo stile di architettura software sono più o meno gli stessi dello stile Repository, anche se sono due stili che sono adatti in situazioni diverse.

Client Server Architecture

Ci sono due tipi di sottosistema:
Client: sottosistema che chiama uno o più sottosistemi server per ottenere risultati di servizi.
Server: sottosistema che risponde ad uno o più sottosistemi client per fornire dei servizi.
I client conoscono l'interfaccia dei server ma i server non conosco l'interfaccia dei client. Molto spesso la risposta dei server ai client è immediata. Gli utenti interagiscono solo con i client.

Questo stile di architettura è utilizzato sopratttutto per i sistemi basati sui database. In tali casi:
Front-end: applicazione utente (lato client).
Back-end: accesso e manipolazione del database (lato server).

Il client fornisce: interfacce utenti customized, elaborazione al front-end delle informazioni, inizio delle chiamate a procedure remote, accesso al database sul server tramite rete.
Il server fornisce: gestione dei dati centralizzata, integrità dei dati, consistenza dei dati, sicurezza dei database, operazioni concorrenti (multiple user access), elaborazione centralizzata (ad esempio archiving).

Peer to Peer Architecture

Ogni sottosistema può funzionare da client e da server. DI conseguenza, la Peer to Peer Architecture è una generalizzazione della Client Server Architecture. La sicnronizzazione sulle richieste rende i flussi di controllo dipendenti tra loro nei sottosistemi, per le altre operazioni il flusso di controllo dei sottosistemi è indipendente per ogni sottosistema.

Three Tier Architecture

I sottosistemi sono organizzati in tre strati:
Interface Layer: strato del sistema composto dai boundary objects che fanno da interfaccia utente
Application Logic Layer: strato del sistema che è composto dagli oggetti di controllo e di entità.
Storage Layer: strato del sistema che si occupa della memorizzazione, del recupero e dell'interrogazione di oggetti persistenti dle sistema.

Per rendere possibile la suddivisione del sistema in sottosistemi, gli sviluppatori devono far fronte a varie scelte quando il sistema viene decomposto:
Mapping Hardware/Software
Gestione dei Dati Persistenti
Controllo di Accesso
Flusso di Controllo Globale
Condizioni Limite

Mapping Hardware/Software

Decidere la configurazione hardware del sistema, le responsabilità dei nodi per le funzionalità, la gestione della comunicazione tra i nodi realizzati, i servizi da realizzare utilizzando componenti già esistenti e come includere queste ultime nel sistema. Spesso questo mapping porta all'aggiunta di nuove componenti addizionali, che permettono di muovere le informazioni da un nodo ad un altro e gestire i problemi di concorrenza. Componenti legacy o off.the-shelf permettono di ridurre i costi della realizzazione di servizi complessi.
-Come realizzare i sottosistemi: Hardware o Software?
-Come mappare gli oggetti sull'hardware scelto e sul software?
    -Mappare gli oggetti su : processori, memoria, Input/Output
    -Mappare le associazioni : connettività

Mappare gli Oggetti
Processori:
    -La computazione richiede più processori?
    -Possiamo velocizzare l'elaborazione distribuendo le mansioni tra più processori?
Memoria:
    -C'è abbastanza memoria per gestire le richieste?
I/O:
    -C'è necessita di uno spazio extra per trattare il tasso di generazione dei dati?
    -Il tempo di risposta eccede la larghezza della banda di comunicazione disponibile tra i sottosistemi?

Durante il system design dobbiamo modellare la struttura statica e quella dinamica:
-Diagramma delle componenti per la struttura statica (mostra la struttura al "design time", o al "compile time").
-Diagramma di deployment per la struttura dinamica (mostra la struttura al "run-time").

Durata di vita delle componenti
-Alcune esistono solo al "design time"
-Altre esistono fino al "compile time"
-Altre esistono solo al "run time"

Component Diagram: grafo delle componenti connesse attraverso relazioni di dipendenza, che mostra le dipendenze tra le componenti software.
Deployment Diagram: grafo di nodi connessi attraverso associazioni di comunicazione, che è utile per mostrare il progetto del sistema dopo che scelte su decomposizione in sottosistemi, concorrenza e mapping Hardware/Software sono state prese; utilizzato per descrivere le relazioni tra le componenti runtime e i nodi hardware.

Gestione dei Dati Persistenti

Decidere quale dovrebbe essere l'informazione persistente, dove memorizzarla e come accedere ai dati persistenti. L'accesso a questi dati dovrebbe essere veloce ed affidabile, dato che molte funzionalità del sistema spesso coinvolgono la creazione e la modificazione di dati persistenti.

Gli oggetti entity identificati durante la fase di analisi sono candidati ad essere persistenti.
Quando gli oggetti persistenti sono identificati, bisogna decidere come devono essere memorizzati:

File

-economici, semplici, memorizzazione permanente
-a livello basso (Read, Write)
-le applicazioni devono aggiungere codice per fornire un opportuno livello di astrazione

Database

-potenti, facili da portare
-supportano letture e scritture multiple

Scegliamo di memorizzare i dati persistenti su file quando: -i dati sono voluminosi (bit maps)
-c'è necessità di tenere traccia delle informazioni solo per un tempo breve?
-la densità dell'informazione è bassa

Scegliamo di meorizzare dati persistenti su database quando: -le informazioni richiedono un accesso ad un livello raffinato di dettaglio attraverso utenti multipli
-le informazioni devono essere portate attraverso piattaforme multiple (heterogeneous systems)?
-più programmi hanno accesso alle informazioni?
-la gestione dei dati richiede molte infrastrutture?

Nel caso di database, possiamo affidare il sistema a database relazionali (basati sull'algebra relazionale) o database object oriented (basati su concetti fondamentali della modellazione objet oriented).

- Le informazioni devono essere distribuite?
- Il database deve essere estendibile?
- Quanto spesso il database è acceduto?
- Quale è il tasso di richiesta atteso (per le query)? E nel caso pessimo?
- Quale è la grandezza della richiesta tipica e di quella pessima?
- Le informazioni richiedono di essere archiviate?
- Il system design tenta di nascondere la locazione dei database (trasparenza della locazione)?
- C'è necessità di una singola interfaccia per accedere le informazioni?
- Quale è il formato delle query?
- Il database deve essere relazionale o object-oriented?

Controllo di Accesso

Decidere chi può accedere alle informazioni, se il controllo di accesso può cambiare dinamicamente e come è specificato e realizzato il controllo d'accesso. Tali scelte dovrebbero essere le stesse per tutti i sottosistemi.

Durante il system design si modellano gli accessi determinando quali oggetti sono condivisi tra gli attori.Questo vuol dire:

-Descrivere i diritti di accesso per classi differenti di attori
-Descrivere come gli oggetti "vigilano" contro gli accessi non autorizzati

- Il sistema richiede un meccanismo di autenticazione?
- Se sì, quel è lo schema di autenticazione (nome utente e password, lista dei controlli di accesso, ecc.)?
- Quale è l'interfaccia utente per l'autenticazione?
- Come un servizio è visto dal resto del sistema?

Bisogna definire per ogni attore quali operazioni può effettuare su ogni oggetto condiviso: uso di matrici di accesso.

Matrici di accesso

Global Access Table: rappresenta esplicitamente ogni cella nella matrice come una tupla: actor,class,operation, ecc. (se una tale tupla non c'è allora l'accesso è negato).

Access Control List: associa una lista di coppie (actor,operation) per ogni classe che può essere acceduta (ogni volta che un oggetto è acceduto, la sua lista degli accessi è controllata per il corrispondente attore e operazione).

Capability: associa una coppia (class,operation) con un attore (consente ad un attore di accedere ad un oggetto della classe descritta nella capability: negare una capability equivale a negare un accesso).

La global access table richiede molto spazio, l'access control list rende veloce la ricerca degli attori che possono accedere ad un determinato oggetto, la capability rende veloce la ricerca degli oggetti a cui un attore può accedere.

Flusso di Controllo Globale

Decidere come gestire la sequenza delle operazioni, se il sistema deve essere guidato da eventi e se il sistema può interagire con più utenti alla volta. Tali scelta influenzano le interfacce delle componenti: se c'è un controllo guidato da eventi, allora i sottosistemi dovrebbero fornire un gestore degli eventi; se c'è un controllo bastao sui thread, allora i sottosistemi devono garantire mutua esclusione nelle sezioni critiche.

Il flusso di controllo globale è la sequenza di azioni nel sistema. Nel caso di sistema object oriented, tale sequenza include anche le decisioni su quali operazioni eseguire e il loro ordine di esecuzione (decisioni prese in base agli eventi esterni generati dagli attori o al trascorrere del tempo).

Ci sono vari emccanismi per il controllo del flusso globale:

Procedure-driven control: controlli risiedenti nel codice del programma, semplice da usare e costruire e adatto a sistemi legacy e quelli scritti con linguaggi procedurali.

Event-driven control: un ciclo principale attende un evento esterno. Quando un evento si verifica, è spedito all'oggetto appropriato, sulla base delle informazioni associate con l'evento. Il controllo risiede in un dispatcher che chiama le funzioni di sottosistema. Flessibile, buono per le interfacce utenti.

Threads: il sistema può creare un numero arbitrario di thread, ognuno risponde ad un differente evento. Se un thread necessita di informazioni addizionali aspetta un input da uno specifico attore.

Identificare i processi concorrenti e trattare i problemi della concorrenza.
Due oggetti sono concorrenti se possono ricevere eventi nello stesso istante senza interagire.
Oggetti concorrenti dovrebbero essere assegnati a differenti thread di controllo.
Oggetti con attività mutualmente esclusive possono essere raggruppati nello stesso thread di controllo.

Condizioni Limite

Decidere come il sistema deve essere avviato, come il sistema deve essere interrotto e come individuare e poi gestire i casi eccezionali. Spesso avvio e interruzione del sistema rappresentano gran parte della complessità di un sistema, soprattutto nel caso di sistemi distribuiti, perché incidono sulle interfacce di tutti i sottosistemi.

Nella fase di system design bisogna determinare le condizioni limite per il sistema che si sta sviluppando:
-**inizializzazione** (descrive come il sistema è portato da uno stato non inizializzato ad uno stato stabile: "startup use cases").
-**terminazione** (descrive quali risorse sono sono rilasciate e quali sistemi sono notificati della terminazione: "termination use cases").
-**fallimento** (molte possibili cause: errori, problemi esterni , come ad esempio alimentazione elettrica; buoni sistemi progettano i fallimenti fatali: "failure use cases").

In generale, gli use case per le condizioni limite vengono identificati esaminando ogni sottosistema e ogni oggetto persistente:
-**Configurazione**: Per ogni oggetto persistente, si esamina in quale use case è creato o distrutto. Per ogni oggetto non creato o non distrutto in uno degli use case, si aggiunge uno use case invocato dall'amministratore di sistema.
-**Avvio e terminazione**: Per ogni componente si aggiungono tre use case: start, shutdown, configure.
-**Gestione eccezioni**: Per ogni tipo di fallimento di componente, si decide come il sistema debba reagire. Documentiamo ognuna di queste decisioni con uno use case eccezionale che estende lo use case di base.

Eccezione: evento o errore che si evrifica durante l'esecuzione del sistema.
Le eccezioni possono avere tre cause principali:
Fallimento hardware: l'hardware "invecchia" e fallisce. Crash dell'hard disk e perdita dei dati. Il fallimento di link provoca la disconnessione di due nodi.
Cambiamento nell'ambiente operativo: L'ambiente influenza il lavoro del sistema. Perdita di connessione per un sistema wireless se un trasmettitore è fuori uso. Una pausa nell'erogazione della corrente elettrica può far andare giù il sistema, finché una batteria di back-up non è attivata.
Fallimento del software: Un errore si verifica perché il sistema o una delle sue componenti contiene un errore commesso durante la fase di progetto. Sebbene scrivere codice senza bug è difficile, sottosistemi individuali potrebbero anticipare errori di altri sottosistemi.

Gestione delle eccezioni
- Un meccanismo attraverso cui il sistema tratta le eccezioni.
- Nel caso di un errore utente, il sistema mostra all'utente un messaggio così che egli possa far fronte all'errore.
- Nel caso di un fallimento di un link di comunicazione, il sistema dovrebbe salvare lo stato temporaneo così che possa essere recuperato quando la rete ritorna ad essere funzionante.
- Si raffinano gli use case in modo da descrivere le situazioni in cui si possono verificare le eccezioni.
- Durante il System design si gestiscono le eccezioni a livello di componente, mentre durante l'Object Design a livello degli oggetti.

Revisione del System Design

Come l'analisi, il Sistem Design è un'attività iterativa. A differenza dell'analisi, non ci sono agenti esterni (come il cliente) per revisionare il lavoro e le scelte fatte. Il manager del progetto e gli sviluppatori devono organizzare un processo di revisione per sostituirsi al cliente.

Il System Design model dovrebbe essere:
Corretto: il modello è corretto se il modello di analisi può essere mappato sul System design model.
Completo: il modello è completo se ogni requisito è stato perseguito.
Consistente: il modello è consistente se non contiene contraddizioni.
Realistico: il modello è realistico se il corrispondente sistema può essere implementato.
Leggibile: il modello è leggibile se sviluppatori non coinvolti nel design possono comprenderlo.

Object Desing

Lo scopo dell'object design è quello di chiudere il divario tra oggetti di applicazione e componenti off-the-shelf per mezzo dell'identificazione di nuovi oggetti di soluzione e la raffinazione di oggetti già esistenti. Serve come base dell'implentazione; durante la fase dell' object design, infatti, si prendono decisioni su come implementare il modello d'analisi. Inoltre si aggiungono dettagli all'analisi dei requisiti.
Include:

-Riuso
-Specifica dei servizi
-Ristrutturazione del modello ad oggetti
-Ottimizzazione del modello ad oggetti

Riuso

Le componenti off-the-shelf identificate durante il system design vengono utilizzate nella realizzazione di ogni sottosistema; vengono selezionati: librerie di classi e componenti utili per struttutre dati e servizi basilari; Design Patterns per proteggere le classi da eventuali cambiamenti futuri e per risolvere vari problemi frequenti.

Specifica dei servizi

Tutti i servizi identificati durante il system design sono "trasformati" in interfacce di classi (argomenti, firme, operazioni ed eeccezioni). Inoltre,si aggiungono alcune operazioni ed oggetti utili per far fronte ai trasferimenti dei dati tra i vari sottosistemi. Ogni sottosistema, alla fine di questa fase di specifica, avrà le proprie interfacce. Tale specifica è spesso chiamata API (Application Programmer Interface)

Ristrutturazione del modello ad oggetti

Al fine di aumentare il riuso e di soddisfare altri obiettivi importanti per lo sviluppo del sistema (design goals), il modello ad oggetti viene modificato (trasformazioni di associazioni n-arie in associazioni binarie, di associazioni binarie in riferimenti, fusione di classi di simile natura in una sola classe e decomposizione di classi complesse in varie classi più semplici, trasformazioni di classi che mancano di uno specifico comportamento in attributi, aumento dell'ereditarietà e del packaging tramite modifica di classi ed operazioni). Inoltre, ci si occupa anche del mantenimento, della leggibilità e della comprensione di tale modello.

Ottimizzazione del modello ad oggtti

Al fine di migliorare il modello di sistema, si fa in modo che quest'ultimo soddisfi anche i requisiti di performance. Avvengono quindi: modifiche agli algoritmi (per soddisfare requisiti di velocità e memoria), riduzione delle associazioni (per rendere le query più veloci), aggiunta di associazioni ridondanti (per aumentare l'efficienza), aggiunta di attributi derivati (per ridurre il tempo di accesso agli oggetti), apertura dell'architettura (concedere l'accesso anche agli starti più bassi del sistema).

Riuso

Durante l'object design oggetti di applicazione (entity e relazioni tra gli entity), identificati nell'analisi, e oggetti di soluzione (boundary e control), identificati nell'analisi e nel system design, vengono raffinati e dettagliati e c'è l'identificazione di nuovi oggetti di soluzione, per far sì che il divario di object design si colmi.

Oggetti di applicazione = oggetti che rappresentano concetti rilevanti per il sistema
Oggetti di soluzione = oggetti che non rappresentano concetti, ma componenti che non hanno nessun riscontro nel dominio di applicazione (oggetti dell'interfaccia utente, dati persistenti, ecc.).

Si ridono le ridondanze e si migliora l'estensibilità tramite due tipi di ereditarietà: di specifica e di implementazione.

Ereditarietà di specifica = ereditarietà utile per classificare i concetti in tipi di gerarchia
Ereditarietà di implementazione = ereditarietà utile per dare la possibilità di riutilizzare il codice in futuro
Delegazione = attività utile a favorire il riuso (può essere anche un'alternativa all'ereditarietà), che consiste nel far sì che una classe implementi una operazione (un metodo) ri-inviando un messaggio ad un'altra classe. La delegazione, quindi, comporta una stretta dipendenza tra classi delegate e classi deleganti.

Principio di sostituzione di Liskov = principio che afferma che se un codice cliente usa i metodi forniti da una superclasse, allora per gli sviluppatori c'è la possibilità di aggiungere nuove sottoclassi senza modificare il codice cliente.
Delegazione ed Ereditarietà = dato in molte situazioni è difficile capire se l'ereditarietà sia più adatta della delegazione o viceversa, i design patterns rappresentano un grande aiuto per l'implementazione.
Design Pattern = template di soluzioni, che sono stati migliorati con il tempo e che sono descritti ognuno da quattro elementi:

-nome, che lo identifica
-descrizione del problema, che descrive le situazioni in cui il pattern può essere utile
-soluzione, che è un insieme di classi ed interfacce
-conseguenze, che rappresentano i trade-off, quindi gli svantaggi e i vantaggi per ogni pattern di fronte ai vari requisiti e agli obiettivi fissati.

Design patterns
Tre categorie: Creazionali (forniscono meccanismi per creare oggetti), Strutturali (separano interfaccia ed implementazione e gestiscono le modalità di composizione degli oggetti), Comportamentali (permettono di modificare il comportamento degli oggetti minimizzando le necessità di cambiare il codice).

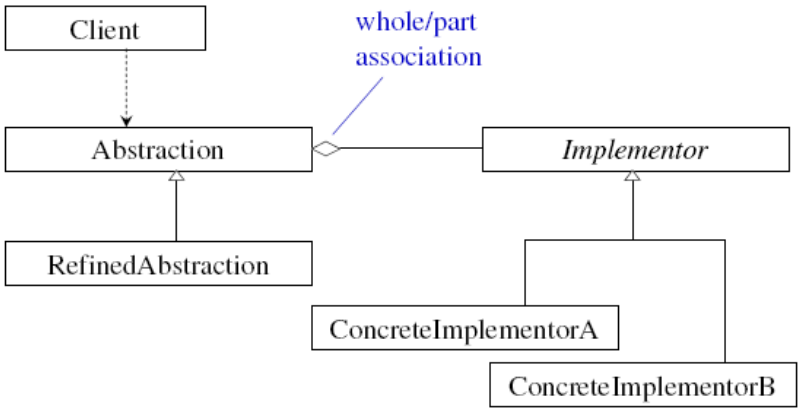
Bridge Pattern

Nome: Bridge design pattern

Descrizione del problema: Separare un interfaccia da una implementazione così che una differente implementazione può essere sostituita, eventualmente a runtime (e.g., testare differenti implementazioni della stessa interfaccia).

Conseguenze : Client è protetto dall'implementazione astratta e concreta.Interfacce e implementazioni potrebbero essere testate separatamente

Soluzione:



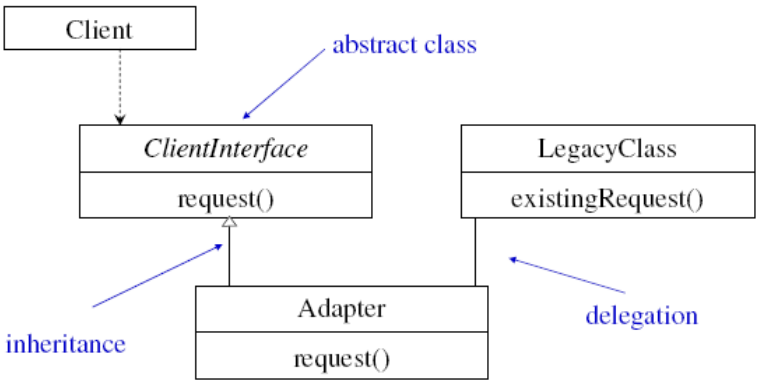
Adapter Pattern

Nome : Adapter Pattern

Descrizione del problema : Convertire l'interfaccia utente di una classe legacy in una interfaccia diversa che il cliente si aspetta, cosi che il cliente e la classe legacy possono lavorare insieme senza cambiamenti.

Conseguenze : Client e LegacyClass lavorano insieme senza modifiche o altro. Adapter lavora con LegacyClass e tutte le sue sottoclassi. Un nuovo Adapter deve essere scritto se Client è sostituito da una sottoclasse.

Soluzione :



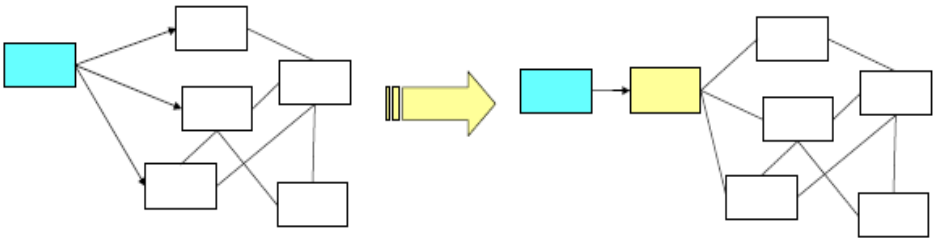
Façade

Nome : Façade pattern

Descrizione del problema : Rendere più semplice l'uso di un sistema. Fornire un'unica interfaccia per un insieme di funzionalità "sparse" su più interfacce/classi

Conseguenze : Promuove un accoppiamento debole fra cliente e sottosistema. Nasconde al cliente le componenti del sottosistema. Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema

Soluzione :



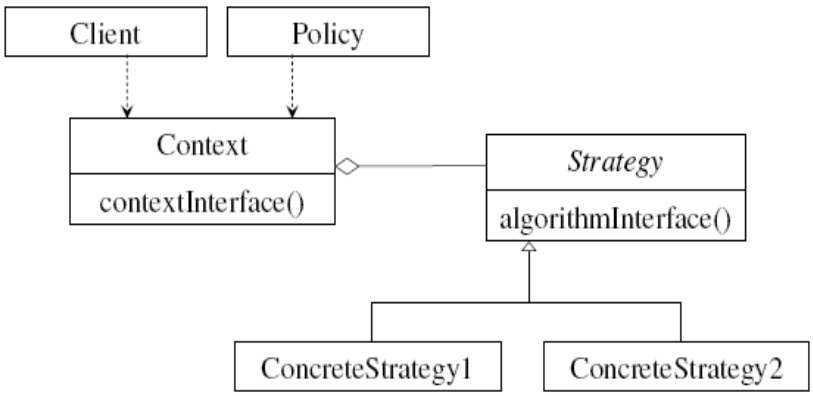
Strategy

Nome : Strategy design pattern

Descrizione del problema : Separare una classe che descrive una decisione da un insieme di meccanismi, cosi che differenti meccanismi possono essere cambiati in maniera trasparente.

Conseguenze : ConcreteStrategy può essere sostituito in maniera transparente da Context. Policy decide quale Strategy è migliore, date le circostanti correnti. Nuovi algoritmi possono essere aggiunti senza modificare Context o Client. Nel Bridge la classe Abstraction crea ed inizializza le ConcreteImplementations. Nello Strategy il Context non ha conoscenza delle ConcreteStrategy, il client crea gli oggetti ConcreteStrategy e configura il Context.
I ConcreteImplementation sono di solito creati durante l'inizializzazione.
I ConcreteStrategy di solito sono creati e sostituiti diverse volte a run-time

Soluzione:



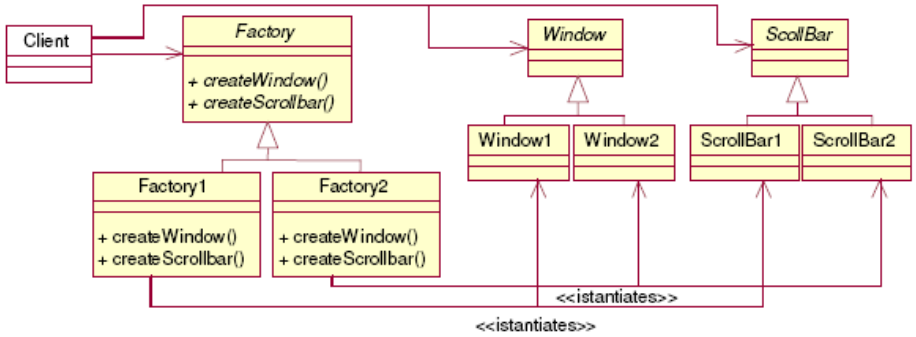
Abstract Factory

Nome : Abstract Factory design pattern

Descrizione del problema : Proteggi il cliente da differenti piattaforme che forniscono differenti implementazioni dello stesso insieme di concetti

Conseguenze : Client è protetto da classi concrete. E' possibile sostituire famiglie di classi a runtime.

Soluzione :



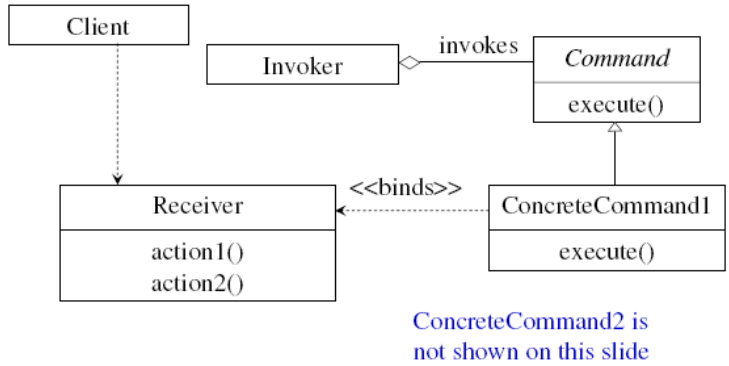
Command

Nome : Command design pattern

Descrizione del problema : Incapsula richieste così che possano essere eseguite, annullate, o accodate indipendentemente dalla richiesta.

Conseguenze : Sono separati l'oggetto del comando (Receiver) e l'algoritmo del comando (ConcreteCommand). Invoker è protetto dal comando specifico. ConcreteCommands sono oggetti. Possono essere creati e memorizzati. Nuovi ConcreteCommands possono essere aggiunti senza cambiare il codice esistente.

Soluzione :



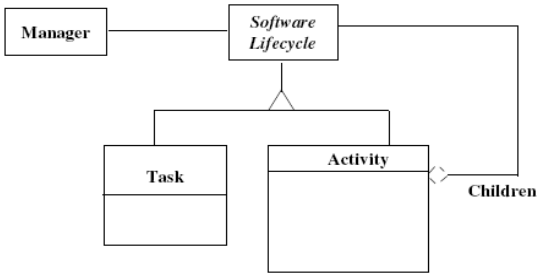
Composite

Nome : Composite design pattern

Descrizione del problema : Rappresenta una gerarchia di ampiezza e profondità variabile, cosi che le foglie e gli oggetti composti possono essere trattati in maniera uniforme attraverso un'interfaccia comune.

Conseguenze : Client usa lo stesso codice per trattare con Leaves o Composites.Comportamenti specifici di un Leaf possono essere cambiati senza cambiare la gerarchia. Nuove classi di Leaves possono essere aggiunte senza cambiare la gerarchia.

Soluzione :



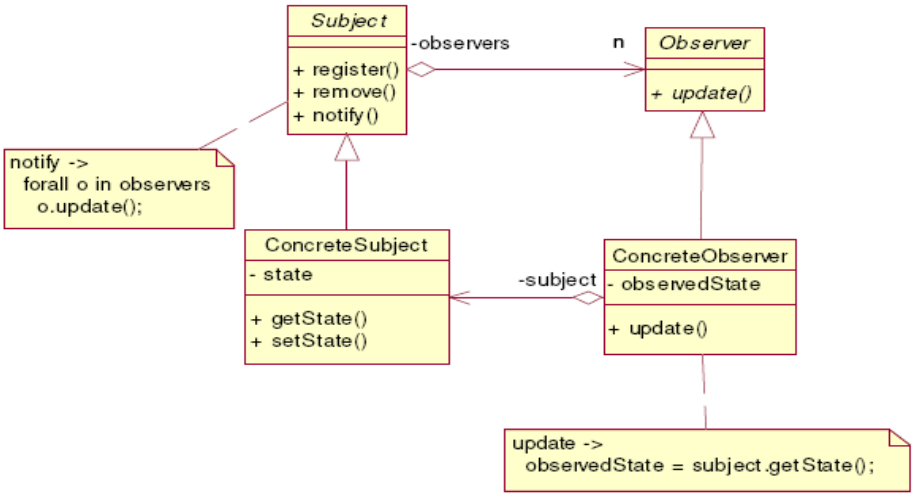
Osservatore

Nome : Observer pattern

Descrizione del problema : Avere oggetti che "osservano" lo stato di un oggetto e sono notificati e aggiornati immediatamente quando lo stato cambia.

Conseguenze : ¨ Gli osservatori devono potersi registrare:
l'oggetto osservato deve fornire un'interfaccia standard per la registrazione: register() e remove().
L'oggetto osservato deve poter notificare: gli osservatori devono fornire un'interfaccia standard per la notifica; update() (e l'oggetto osservato ha notify() che chiama tutti gli update dei registrati).
Chi invoca setState()
-Chiunque
-Uno degli osservatori
Chi invoca notify()?
-Ogni metodo dell'oggetto osservato che modifica lo stato.
-Il cliente, dopo che ha terminato una sequenza di modifiche
E se un osservatore osserva più oggetti, come fa a sapere chi ha variato lo stato?
-Parametro di update() (ad es., this)
Push vs. Pull
-Pull: update() non passa nessuna info sullo stato, l'osservatore usa getState()
-Push: update() passa anche lo stato

Soluzione :



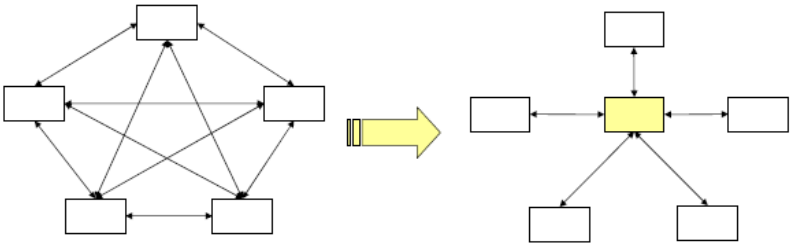
Mediator

Nome : Mediator pattern

Descrizione del problema : Un oggetto incapsula le modalità di interazione fra oggetti evitando a tutti gli oggetti di "conoscere" tutti gli altri.

Conseguenze : Si abbassa l'accoppiamento.Classi piccole sono più facilmente riusabili. Consente di variare le modalità di interazione. Il mediatore rappresenta la comunicazione (la comunicazione come oggetto separato). Contro: Mediator può diventare un oggetto complesso, monolitico, di difficile manutenzione.

Soluzione :



Framework di applicazione : applicazione riusabile ed incompleta che può essere specializzata per ottenere applicazioni custom. Sono sviluppati per particolari tecnologie (data processing, comunicazione cellulare, ecc) o per domini di applicazione (come interfacce utenti o applicazioni aeronautiche real-time)
La riusabilità dei framework evita la ricreazione e rivalidazione di soluzioni ricorrenti. Un framework di applicazione migliora l'estensibilità fornendo dei metodi hook, che sono riscritti dall'applicazione per estendere il framework.
I metodi hook separano l'interfaccia e il funzionamento dalle variazioni richieste dall'applicazione per un particolare contesto.

Framework di infrastruttura
Semplificano il processo di sviluppo del SW (framework per sistemi operativi, debugger, progettazione di interfacce utenti, task di comunicazione, ecc.).
Sono utilizzati all'interno di un progetto SW e non sono rilasciati al cliente.

Framework middleware
Usati per l'integrazione di applicazioni e componenti distribuite.
MFC e DCOM, Java RMI, WebObjects, implementazioni di CORBA, database transazionali.

Framework per applicazioni enterprise
Sono applicazioni specifiche per particolari domini come telecomunicazioni, aeronautica, ambienti di modellazione, ecc.

Whitebox Framework
L'estensibilità è basata su ereditarietà e associazioni dinamiche
Funzionalità esistenti sono estese creando delle sottoclassi per le classi base del framework e facendo l'overriding dei metodi hook predefiniti utilizzando dei pattern.

Blackbox Framework
L'estensibilità è supportata definendo interfacce per componenti che possono essere integrate nel framework
Una funzionalità esistente è riutilizzata definendo componenti conforme ad una particolare interfaccia ed integrando queste componenti con il framework attraverso la delegazione.

Whitebox richiede una precisa conoscenza della struttura interna del framework, ed i sistemi prodotti sono strettamente accoppiati (cambiamenti possono richiedere la ricompilazione). Blackbox sono più facili poiché si basano sulla delegazione. Sono però difficili da sviluppare perché richiedono la definizione di interfacce e hook che anticipano possibili casi d'uso.

Specifica dei servizi

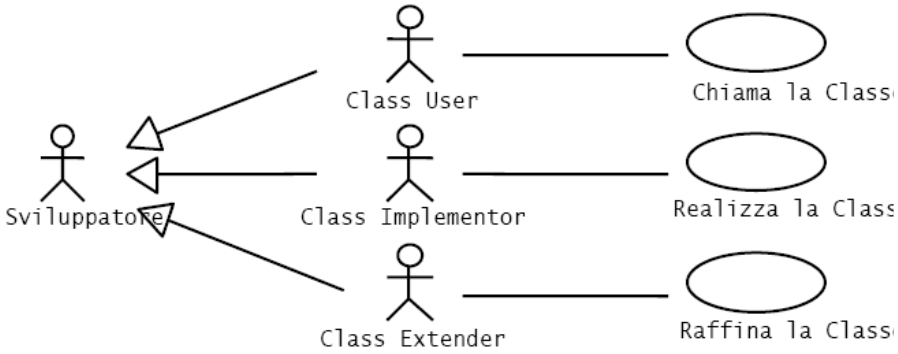
La specifica dei servizi (o delle interfacce) si ha come obiettivi il comunicare chiaramente e precisamente i dettagli di basso livello degli oggetti del sistema ed il descrivere precisamente l'interfaccia di ogni oggetto così che non ci sia necessità di lavoro di integrazione per oggetti realizzati da diversi sviluppatori.

Specifica delle interfacce, attività:

- -Identificare attributi e operazioni mancanti
- -Specificare le signature e la visibilità di ogni operazione
- -Specificare le precondizioni (sotto le quali un'operazione può essere invocata e quelle che determinano un'eccezione)
- -Specificare le postcondizioni
- -Specificare le invarianti

Tutti i modelli sin qui costruiti forniscono una visione parziale del sistema, molti pezzi mancano e altri sono da raffinare:

-Il modello a oggetti di analisi: descrive gli oggetti entity, boundary e control che sono visibili all'utente
-La decomposizione in sottosistema: descrive come questi oggetti sono partizionati in pezzi coesi realizzati da diversi team. Ogni sottosistema fornisce un insieme di servizi (ad alto livello) ad altri sottosistemi
-Il mapping Hardware/software: identifica le componenti che costituiscono la macchina virtuale su cui costruiamo gli oggetti soluzione (es. classi e API definite da componenti esistenti)
-Use case Boundary: descrivono dal punto di vista dell'utente, casi amministrativi e eccezionali gestiti dal sistema
-Design pattern (selezionati durante l'object design reuse): descrivono object design parziali che risolvono questioni specifiche



Tipo di un attributo: specifica l'insieme dei valori che un attributo può assumere e le operazioni applicabili su quest'ultimo. Tale tipo interessa anche i valori di ritorno delle operazioni ed i parametri delle operazioni.

Signature: tupla che informa in merito al tipo dei parametri e del valore di ritorno di un'operazione.

1. Aggiungere informazione relativa alla visibilità

3 livelli di visibilità:

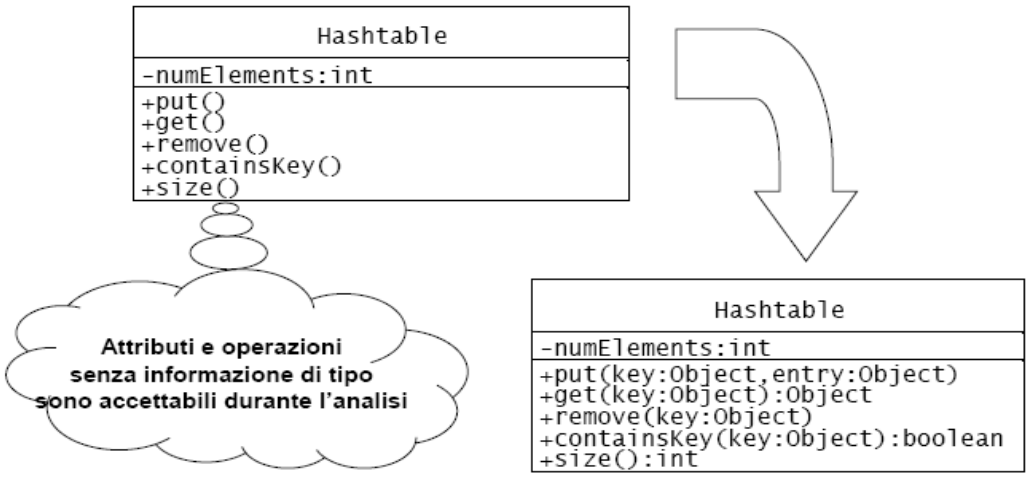
- : **Private** (Solo per Class implementor): Ad un attributo privato può accedere solo la classe in cui è definito. Un'operazione privata può essere invocata solo dalla classe in cui è definita. Ad attributi eoperazioni private non possono accedere sottoclassi o altre classi.

#: **Protected** (Class extender): Ad un attributo o operazione protetto può accedere solo la classe in cui è definito e ogni discendente della classe.

+: **Public** (Class user): Ad un attributo o operazione pubblica possono accedere tutte le classi (interfaccia pubblica).

Euristiche per Information Hiding
-Definire attentamente l'interfaccia pubblica per le classi così come per i sottosistemi.
-Meno un'operazione sa:
-Più bassa sarà la probabilità che sarà influenzata da qualche cambiamento
-Più facilmente la classe potrà essere cambiata
-Principio "Need to know": Solo se qualcuno necessita di accedere all'informazione, rendilo possibile , ma solo attraverso ben definiti canali, in modo che conosci sempre l'accesso.

2. Aggiungere informazione sui tipi e sulle signature



3. Aggiungere contratti

Contratti su una classe consentono a class users, implementors e extenders di condividere le stesse assunzioni sulla classe.
3 tipi di vincoli nei contratti:
Invariante: predicato che è sempre vero per tutte le istanze di una classe: vincoli assocciati a classi o interfacce.
Precondizione: predicato associato con una specifica operazione e verificatosi prima che l'operazione sia invocata.E' usata per specificare vincoli che un chiamante deve soddisfare prima di chiamare un'operazione.
Postcondizione: predicato associato con una specifica operazione e verificatosi dopo che l'operazione è stata invocata. E' usata per specificare vincoli che l'oggetto deve assicurare dopo l'invocazione dell'operazione.

Object Constraint Language: linguaggio che consente di specificare formalmente i vincoli sugli elementi di un singolo modello (attributi, operazioni, classi) o gruppi di elementi di modello (associazioni e classi partecipanti).
Constraint: espressione OCL che ritorna un vaolore vero o falso. OCL non è un linguaggio procedurale (non si può vincolare il control flow).

I constraints possono coinvolgere più di una classe

3 Tipi di Navigazione
1. Attributo locale
2. Classe direttamente relazionata
3. Classe indirettamente relazionata

Tre tipi di collezioni:
OCL sets: sono usati quando si naviga una singola associazione

OCL sequences: sono usati quando si naviga una singola associazione ordinata

OCL bags (multinsiemi): sono usati per accumulare oggetti quando si accede a oggetti correlati in modo indiretto (Se non siamo interessati al numero di occorrenze di ogni oggetto nel bag, allora il bag può essere convertito in un insieme usando l'operatore asSet(collection))

Operazioni OCL per accedere alle collezioni

->: operatore per accedere alle collezioni
size: restituisce il numero di elementi della collezione
includes(object): restituisce True se object è nella collezione
select(expression): restituisce la collezione contenente solo gli elementi della collezione originale per cui l'espressione è True
union(collection): restituisce la collezione contenente sia gli elementi della collezione originale sia quelli della collezione specificata come parametro
intersection(collection): restituisce la collezione contenente solo gli elementi che appartengono sia alla collezione originale sia alla collezione specificata come parametro
asSet(collection): restituisce un insieme contenente gli elementi che appartengono alla collezione

forall:
forALL(variabile |espressione) è vera se l'espressione è vera per tutti gli elementi nella collezione

exists:
exists(variabile |espressione) è vera se esiste almeno un elemento nella collezione per cui l'espressione è vera

Ristrutturazione ed Ottimizzazione del modello ad oggetti

La ristrutturazione del modello ad oggetti è fatta per soddisfare i design goals del sistema e anche per aumentare il riuso. L'ottimizzazione del modello ad oggetti ha il fine di soddisfare requisiti sulle prestazioni richiesti per il sistema.
Esistono 4 tipi di trasformazione:
Trasformazione di modello: trasformazioni interne al modello ad oggetti (come la conversione di un attributo in classe o viceversa, ecc.). E' fatta per ottimizzare o semplificare il modello ad oggetti originale, ma la sua necessarietà è prevedibile solo se si ha molta esperienza.
Rifattorizzazione: trasformazione intera al codice sorgente. E' fatta per migliorare la leggibilità del codice sorgente e la sua modificabilità senza alterare il comportamento del sistema, è focalizzato su un campo oppure un metodo di una classe ed è attuata a piccoli passi sempre più grandi interrotti periodicamente da test.
Forward Engineering: produzione di codice sorgente a partire da un modello ad oggetti con la trasofrmazione (mapping) di molti costrutti di quest'ultimo in costrutti del codice sorgente e l'aggiunta di nuovi elementi (nuovi metodi, corpi dei metodi, ecc.). E' fatta per far corrispondere il più possibile codice sorgente e modello ad oggetti, eliminare quanti più errori possibili emersi durante l'implementazionee semplificare l'implementazione.
Reverse Engineering: produzione di un modello ad oggetti a partire da un codice sorgente. E' fatta per ricreare un modello ad oggetti per un sistema che già esiste, perché il modello precedente è andato perduto o non è ancora stato realizzato o non è più in sincronia con il codice sorgente.

Il mapping consiste di varie attività:
-Ottimizzazione del modello di Object Design
-Mapping delle associazioni in collezioni
-Mapping dei Contratti in Eccezioni
-Mapping dell'Object Model in uno schema di memorizzazione persistente

Ottimizzazione del modello di Object Design

Si trasforma il modello ad oggetti per soddisfare gli obiettivi di design identificati durante il system design (es. tempo di risposta, risorse di memoria, etc.).

1.Ottimizzare i cammini di accesso (access path)
Si introducono nuove associazioni tra quegli oggetti che vengono coinvolti spesso da query: oggetto interrogante ed oggetto interrogato con query molto frequenti cominciano ad avere un'associazione tra loro. Tutto ciò riduce notevolmente i tempi di risposta.
Si prova a ridurre quante più associazioni multiple possibili in associazioni singole (da many a one) o, se questo non conviene, ordinare o indicizzare gli oggetti da lato many.
Si possono eliminare classi che sono abbastanza povere di attributi e metodi, spostando questi attributi in un'altra classe (quella che richiede il valore degli attributi, ad esempio) per non perdere informazioni.

2.Collassare gli oggetti in attributi
Si trasformano classi con pochi attributi e comportamenti e con l'associazione ad una sola altra classe in attributi di quest'ultima.
3.Ritardare le elaborazioni costose
Si cerca di ritardare il più possibile operazioni che costano molto tempo e che potrebbero aumentare molto i tempi di risposta del sistema.
4.Mantenere in una struttura dati temporanea il risultato di elaborazioni costose
Si memorizza il risultato della chiamata ad alcuni metodi molto utilizzati in un attributo privato. Dato che spesso tale risultato è uguale per varie chiamate ai vari metodi, questo attributo può evitare ulteriori chiamate a metodi e far risparmiare tempo, anche se può essere molto oneroso in termini di spazio in memoria.

Mapping delle associazioni in collezioni

Dato che nei linguaggi object oriented non esiste il concetto di associazione, ma sono presenti solo i riferimenti, bisogna realizzare le associazioni del modello ad oggetti tramite riferimenti considerando la molteplicità e la direzione delle associazioni.

Associazioni uno-a-uno unidirezionali

Classe A (1) → (1) Classe B

Attributo in A che referenzia un oggetto di B. Tale attributo per un oggetto A ha valore nullo solo poco dopo la creazione dell'oggeto A in questione.

Associazioni uno-a-uno bidirezionali

Classe A (1) ←→ (1) Classe B

Attributo in A che referenzia un oggetto di B ed attributo in B che referenzia un oggeto di A. Tali attributi hanno valore nullo solo appena viene creato l'oggetto di cui sono parte.

Associazioni uno-a-molti

Classe A (1) ←→ (molti) Classe B

Collezione di riferimenti di oggetti di B in un attibuto di A (tipo Set se riferimenti non ordinati, tipo List se riferimenti ordinati).

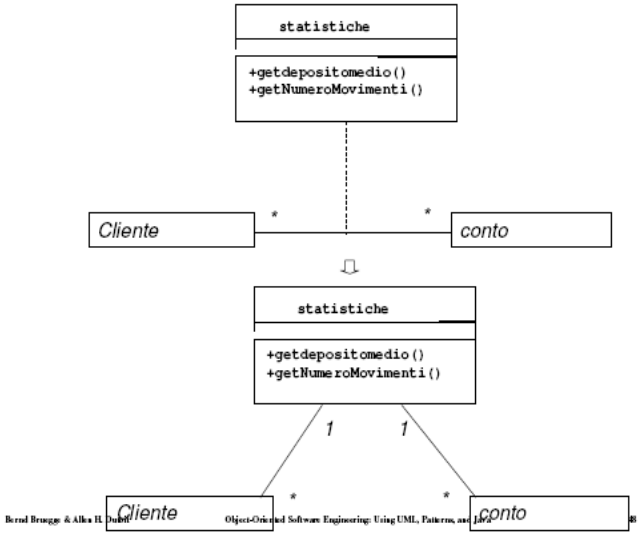
Associazioni molti-a-molti

Classe A (molti) ←→ (molti) Classe B

Collezione di riferimenti di oggetti B in un attributo di A e collezione di riferimenti di oggetti A in un attributo di B (le collezioni sono implementate tramite List con ArrayList).

Classi di associazioni

Sono usate per contenere gli attributi e le operazioni di un'associazione: si trasforma l'associazioni in oggetti separati con associazioni binarie.



Mapping dei Contratti in Eccezioni

Per gestire e segnalare violazioni ai contratti si utilizzano le eccezioni.

-**Controlla precondizione**: Controlla la precondizione prima dell'inizio del metodo con un test che lancia un'eccezione se la precondizione è falsa.

-**Controlla postcondizione**: Controlla la postcondizione alla fine del metodo e lancia un'eccezione se il contratto è violato. Se più di una postcondizione non è soddisfatta, lancia un eccezione solo per la prima violazione.

-**Controlla invarianti**: Controlla le invarianti nello stesso momento delle postcondizioni.

-**Gestire l'ereditarietà**: Incapsula il codice di controllo per precondizioni e postcondizioni in metodi separati che possono essere richiamati dalle sottoclassi.

Dato che gestire tutti i contratti può essere difficile ed oneroso in termini sia di spazio che di tempo, si possono omettere codici di controllo per: invarianti, postcondizioni, metodi provati e metodi protetti; mentre ci si può concentrare sui codici di controllo per oggetti di lunga durata (gli entity) e cercare di fare in modo di poter riutilizzare il codice di controllo in più casi (incapsulamento di codice di controllo in metodi).

Mapping dell'Object Model in uno schema di memorizzazione persistente

Se usiamo database relazionali o flat file per memorizzare gli oggetti persistenti, è necessario mappare il modello degli oggetti in uno schema di memorizzazione.
Dato che i database relazionali non supporano l'ereditarietà, bisogna effettuare un mapping verticale o un mapping orizzontale.
Mapping verticale: una tabella per la superclasse e una per ogni sottoclasse; quella per la superclasse contiene un attributo che descrive la sottoclasse a cui l'oggetto appartiene; tabella superclasse e tabelle sottoclassi sono caratterizzate dalla stessa chiave primaria (query lente, ma facili cambiamenti alle tabelle e facili aggiunte di sottoclassi).
Mapping orizzontale: tutti gli attributi della superclasse vengono inseriti nella tabella di ogni sottoclasse (query più veloci perché c'è maggiore frammentazione, ma i cambiamenti delle tabelle sono difficili e richiedono tempi lunghi).

Core Architet: è responsabile delle trasformazioni periodicamente applicate.
Architecture Liaison: è responsabile della documentazione dei contratti associati alle interfacce dei sottosistemi (la notifica a tutti gli utenti delle classi di cambiamenti dei contratti, ad esempio).
Sviluppatore: è responsabile di seguire le convenzioni stabilite dal Core Architect, dell'applicazione delle trasformazioni, della trasformazione del modello ad oggetti in codice sorgente e dell'aggiornamento dei commenti del codice sorgente.

Le trasformazioni del modello ad oggetti permettono di migliorare aspetti specifici del modello e di convertirlo in codice sorgente, riducendo errori in tale codice e gli sforzi per l'implementazione. Una documentazione di tali trasformazioni permette di riapplicare facilmente nel modello ad oggetti o nel codice sorgente queste trasformazioni in caso di cambiamenti del sistema.

Testing

Il testing ha lo scopo di "scomporre" il sistema per dimostrare che è inconsistente con il modello di sistema ad esso associato e scoprire i suoi difetti ed errori dovuti a svariate cause, per poi correggerli.

Componente: parte del sistema che può essere isolata dal resto del sistema per essere testata (oggetto, gruppo di oggetti, sottosistema o parte di esso, ecc.).

Test Case: insieme di input e di risultati attesi che servono a testare una componente per scoprirne gli errori e i fallimenti.E' composto da:

- -name: nome univoco del test case
- -location: path o url che permette di reperire il test case
- -input: insieme di input o comandi che servono per avviare il test
- -oracle: risultati attesi da una corretta esecuzione del test
- -log: correlazioni del comportamento effettivo e di quello atteso relative a varie esecuzioni del test

Ci sono due tipologie di relazioni per il test case:

- -aggregazione: relazione usata quando un test case può essere decomposto in più subtest
- -precedence: relazione usata tra due test case per far precedere uno dei due all'altro

Due tipi di test case:

Black Box: è incentrato sul comportamento I/O e non ha a che fare con la struttura interna della componente.

White Box: è incentrato sulla struttura interna della componente: vengono testati tutti gli stati nel modello dinamico dell'oggetto e ogni integrazione tra gli oggetti, indipendentemente dall'input o dall'output.

Correzione: cambiamento di una componente del sistema volto a correggere un errore.

Una correzione può portare a nuovi difetti, gestibili con tre tecniche:

Problem Tracking: si documenta ogni fallimento, errore, bug, correzione e revisione delle componenti attuate durante il cambiamento (in tal modo è più facile, in molti casi, trovare nuovi errori).

Regression Testing: si rieseguono alcuni test precedenti a quello che ha portato al cambiamento (in tal modo ci si accerta che il cambiamento non abbia influenzato in negativo funzionalità precedenti a quella che esso ha coinvolto).

Rational Maintenance: si documentano le motivazioni dei cambiamenti e quelle della revisione della componente (in tal modo si evita di introdurre nuovi fault).

Livelli di Testing:

- **Unit Testing**: testing di una singola unità
- **Integration Testing**: testing di più componenti in relazione
- **System Testing**: testing sui requisiti del sistema
- **Alpha Testing e Beta Testing**: testing fatti da utenti prima del rilascio sul mercato
- **Acceptance Testing**: testing fatto dai clienti

Unit Testing
Applicato ad una sola unità o modulo del sistema software, fatto dai programmaori dell'unità stessa, serve per rilevare errori nell'unità che è testata.
Test Stub e Test Driver servono a coprire le parti mancanti del sistema.
Test Stub: implementazione parziale di componente chiamata dalla componente testata e da cui, quindi, quest'ultima dipende.
Test Driver: implementazione parziale di componente che chiama la componente testata e che quindi dipende da quest'ultima.
Il Driver, o modulo guida, simula l'ambiente chiamante e dell'inizializzaione dell'ambiente non locale del modulo esaminato. Lo Stub, o modulo fittizio, ha la stessa interfaccia del modulo simulato, ma più semplice.
L'implementazione di un test stub non è semplice, perché il test stub deve simulare il giusto comportamento: la componente chiamata deve fare un qualche lavoro e non deve restituire sempre gli stessi valori.
Per far sì che un test stub ed un test driver siano accurati e possano essere facilmente sostituito con le componenti reali, bisogna costruirli dopo che la componente ad essi associata è stata realizzata.
Driver e Stub possono essere:
Interattivi: richiedono interventi umani (sono onerosi e rischiosi perché possono portare a molti errori per colpa degli utenti).
Automatici: il driver inizializza automaticamente le costanti dell'ambiente non locale, mentre lo stub calcola valori approssimati e restituisce valori costanti (a volte non sono sufficienti e provocano malfunzionamenti).
Realizzati come prototipi: driver e stub sono realizzati rapidamenti e sia i loro costi che la loro efficienza sono molto ridotti.
Realizzati a partire da specifiche: driver e stub sono caratterizzati dalle stesse metodologie del sistema software da rilasciare.

Basic Unit Testing: test di una singola operazione (metodo) di una classe (inta method testing).
Unit Testing: test di una intera classe nella sua globalità (intra class testing).

Integration Testing
Applicato ad un aggregato di moleplici unità di un sistema, fatto da persone che non hanno prodotto le unità coinvolte, serve per rilevare errori nell'integrazioni di più unità del sistema e nelle funzioni che questo aggregato sistema deve offrire.
I possibili errori di integrazione sono:
Interpretation Error: il modulo si comporta diversamente da come l'utente del modulo si aspetta.
Miscoded Call Error: l'istruzione di chiamata viene inserita nel posto sbagliato o diverso dal richiesto.
Interface Error: l'interfaccia stantard tra due moduli viene violata.

System Testing
Applicato sull'intero sistema, fatto dal team addetto al testing (che è esterno al team di sviluppo), serve per scoprire se il sistema rispetta i requisiti specificati, come quelli di qualità, di prestazione, manutenibilità o usabilità.

Acceptance Testing
Applicato sull'intero sistema, fatto da clienti, collaudatori, enti, utenti particolari o altri, serve per scoprire se il sistema può davvero soddisfare le richieste fatte dal cliente.

Alpha Testing e Beta Testing
Applicati sull'intero sistema, fatti da utenti, servono per capire se il sistema prodotto è pronto per essere immesso sul mercato.

Alpha Testing: testing fatto da utenti all'interno dell'ambiente in cui il sistema è stato prodotto e prima dell'immissione del sistema software sul mercato. Spesso fatto per utenti privilegiati.
Beta Testing: testing fatto da utenti nell'ambiente in cui il sistema sarà utilizzato e prima dell'immissione del sistema software sul mercato. A volte è preceduto da un alpha testing o da un beta testing fatto su un gruppo più ristretto di utenti.

Attività di Testing:

Component Inspection: trova i fault di una componente del sistema tramite l'ispezione del codice sorgente.
Unit Testing: trova i fault di un'unità del sistema utilizzando test driver e test stub ed esercitando l'unità tramite un test case.
Integration Testing: trova i fault di un aggregato di unità.
System Testing: trova le discordanze tra i requsiti funzionali e non funzionali richiesti al sitema e quelli effettivi del sistema.
Usability Testing: trova le differenze tra le funzionalità richieste e quelle offerte dal sistema.

Component Inspection

Si esamina il codice sorgente in meeting formali per trovare per ogni singola componente del sistema i fault che la caratterizzano. Vinene svolta prima o dopo l'unit testing da parte del team degli sviluppatori, dello sviluppatore della specifica componente, un moderatore e uno o più revisori, che cercano i bug.
Moderatore: estraneo al gruppo del progetto, presiede le sedute, ne sceglie i partecipanti e controlla il processo.
Lettore: legge il codice al gruppo e cerca errori in esso.
Autore: risponde alle domande sulla componente da lui sviluppata.

Processo di ispezione software di Fagan
pianificazione
il moderatore scegli i partecipanti e le checklist, pianifica gli incontri
fasi preliminari
fornisce le informazioni necessarie e assegna i ruoli
Overview
L'autore presenta l'obiettivo e lo scope della componente e i goal dell'ispezione
Preparation
I revisori analizzano l'implementazione della componente
Inspection meeting
Un reader legge il codice sorgente e spiega ciò che dovrebbe fare, il team di ispezione analizza la componente e evidenzia eventuali fault, il moderatore tiene traccia. La maggior parte del tempo è passata a discutere, ma le soluzioni per riparare il bug non sono esplorate in questo punto
Rework L'autore rivede la componente
Follow-up Il moderatore controlla la qualità del rework e determina la componente da ispezionare di nuovo.

Usability Testing

Testing che prova la comprensibilità del sistema da parte degli utenti: cerca le differenze tra come il sistema è atteso dagli utenti e come è effettivamente. E' fatto da un insieme di utenti che ben possano rappresentare gli utenti finali, che riscontrano, eventualmente, problemi utilizzando interfacce o simulazioni di queste utime. In un primo momento, gli sviluppatori selezionano degli obiettivi, poi questi vengono valutati con degli esperimenti in cui si richiede agli utenti di eseguire determinate attività e si tiene traccia di informazioni sulla performance degli utenti e sulle opinioni di questi ultimi per identificare problemi o raccogliere idee per migliorare il sistema.
Ci sono tre tipi di usability test:
Scenario Test: uno scenario immaginario, ma più realistico possibile, del sistema viene presentato ad uno o più utenti; gli sviluppatori si accorgono di quanto velocemente gli utenti comprendono lo scenario, quanto bene esso rappresenti il modello di lavoro e come gli utenti reagiscono alla sua presentazione del nuovo sistema. Test economico da realizzare e ripetere, ma non permette agli utenti un'interazione diretta con il sistema e i dati sono fissi.
Prototype Test: una parte del sistema, che implementa aspetti chiave di quest'utlimo, viene presentato agli utenti. Si utilizza un vertical prorotype o un horizontal prototype. Il vertical prototype implementa del tutto uno use case e con esso dei prototipi funzionali sono usati per valutare i requisiti più importanti. L'horizontal prototype presenta l'interfaccia per la maggior parte degli use case, senza fornire funzionalità, quindi solo un singolo layer del sistema viene implementato. Test molto meno economico degli scenari cartacei e che richiede più sforzi, ma che presenta una visione abbastanza realistica del sistema agli utenti; i prototipi, inoltre, possono collezionare informazioni dettagliate.
Product Test: vengono svolte le stesse operazioni del prototype test, ma al posto di prototipi si usa una versione funzionale del sistema. Test che può essere fatto solo dopo che una buona parte del sistema è stata sviluppata e che richiede una facile modificabilità del sistema.

Unit Testing

Testing che si concentra sul comportamento I/O di una unità: se per ogni input dato si può predire l'output, allora il test è superato con successo dall'unità testata. Dato che però considerare tutti gli input è molto difficile e richiede molto tempo, si possono individuare delle classi di equivalenza degli input.

Black-box Testing

Equivalence Testing: Si sceglie, in tal caso, un test case per ogni classe di equivalenza.

Boundary Testing: Si selezionano gli elementi ai limiti delle classi di equivalenza come input con cui testare l'unità in considerazione.
Sull'unità da testare basta applicare il test con un solo elemento di ogni classe di equivalenza per sapere, anche se non con l'assoluta certezza, che il test sia valido o meno anche per il resto della classe.
Se la condizione sulle variabili d'ingresso specifica:

intervallo di valori
una classe valida per valori interni all'intervallo, una non valida per valori inferiori al minimo, e una non valida per valori superiori al massimo

valore specifico
una classe valida per il valore specificato, una non valida per valori inferiori e una non valida per valori superiori

elemento di un insieme discreto
una classe valida per ogni elemento dell'insieme, una non valida per un elemento non appartenente

valore booleano
una classe valida per il valore TRUE, una classe non valida per il valore FALSE

Equivalence Testing e Boundary Testing non permettono di scoprire bug dovuti a combinazioni di input al test, cioè combinazioni di elementi di più classi valide che potrebbero dare luogo a bug.

White-box Testing

Si concentra sulla struttura interna dell'unità testata. Ogni stato nel modello dinamico dell'oggetto ed ogni integrazione tra gli oggetti viene testata, indipendentemente dall'input o dal risultato effettivo.
Ogni statement della componente è eseguita almeno una volta.

Ci sono quattro tipi di Whitebox Testing:
Statement Testing o **Algebraic Testing**: viene testato ogni singolo statement della componente

Loop Testing: provoca l'esecuzione del loop da saltare completamente, loop da eseguire una sola volta, loop da eseguire più volte.
Path Testing: si assicura che tutti i path nel programma siano eseguiti.

Branch Testing: si assicura che ogni possibile uscita da una condizione sia testata almeno una volta.

Differenze sostanziali tra Blackbox Testing e Whitebox Testing:
-il Whitebox Testing può portare al test di infiniti path, testa quello che viene fatto, piuttosto che quello che si dovrebbe fare, non può controllare casi mancanti;
-il Blackbox Testing può portare ad un'esplosione combinatoriale di test case, spesso non chiarisce se determinati test case coprono un particolare errore e non controllano casi dìuso estranei.

Integration Testing

Testing che rileva errori che nascono dall'integrazione di più componenti, anche se queste ultime sono state già testate individualmente. Due o più componenti sono integrate e la loro integrazione viene analizzata ed eventualmente modificata per correzioni, poi vengono aggiunte a tale integrazione altre componenti, per analizzare una più grande integrazione e così via fino ad avere, infine, l'analisi dell'intero sistema.
L'ordine di integrazione ed analisi delle componenti è molto importante, perché può influenzare molto i risultati e gli sforzi per il testing. Tale ordine determina la strategia usata.
Ci sono quattro principali strategie di Integration Testing e quindi quattro tipi di Integration Testing:
Big Bang Testing: testing non incrementale che prevede prima il testing individuale di tutte le componenti e successivamente il testing dell'integrazione di tutte le componenti, come un singolo sistema. E' un test molto semplice e che evita di ricorrere all'uso di test stub e test driver, ma che è sconveniente perché costoso, molto poco adatto a sistema implementati in linguaggi object oriented e perché non permette di distinguere fallimenti derivanti dalle interfacce da fallimenti interni alle componenti.
Top-down Testing: testing incrementale che prevede inizialmente il testing individuale delle componenti del layer più in alto e poi poi il testing dell'integrazione di queste utlime e componenti del layer immediatamente più in basso e così via. Per ogni testing tranne l'ultimo, che è il testing dell'intero sistema, si utilizzano test stub come supporto per simulare componenti di layer più in basso che ancora non sono state testate ed integrate. In tal modo, spesso ci si ritrova ad aver bisogno di stub molto complessi da scrivere e in gran numero, soprattutto se le componenti da simulare hanno molti metodi. Si può evitare il problema degli stub complessi e numerosi con il Modified top-down testing, che prevede che tutte le componenti di tutti i layer siano testate individualmente inizialmente, ma ciò porta alla necessità di test driver.
Bottom-up Testing: testing incrementale che prevede inizialmente il testing individuale delle componenti del layer più in basso e poi il testing dell'integazione di queste ultime con le componenti del layer immediatamente più in alto e così via. Per ogni testing tranne l'ultimo, che è il testing dell'inetro sistema, si ha bisogno di test driver di supporto per simulare le componenti dei layer più in alto che ancora non sono state testate ed integrate. In tal modo, si testano le componenti più importanti per il sistema alla fine e si deve in molti casi ricorreggere molte volte componenti già testate, perché influenzate fortemente dalle componenti più in alto.
Sandwich Testing: testing incrementale che combina top-down strategy e bottom-up strategy. Prevede inizialmente che si testino le componenti del layer più in alto e le componenti del layer più in basso e poi si integrino queste con le componenti del target layer, il layer intermedio, per poi testare tale integrazione. In tal modo si risparmia molto tempo e molti test stub e test driver, perché il testing delle componenti del layer più in basso può essere fatto in parallelo al testing delle compoennti del layer più in alto e non c'è bisogno di test stub e di test driver per testare le componenti del target layer nella loro integrazione con altre componenti. Le componenti del target layer, in tal modo, nono sono, però, mai testate individualmente e questo potrebbe far saltare l'individuazione di alcuni errori di tali componenti. Il Modified sandwich testing evita quest'ultimo porblema eseguendo il testing individuale di tutte le componenti di tutti i layer. In questo modo si ha bisogno di un grande numero di test stub e test driver inizialmente, ma successivamente si può fare a meno di tali supporti sfruttando le componenti già testate.

System Testing

Testing che prova la soddisfazione da parte del sistema dei requisiti funzionali e non funzionali richiesti dal cliente.
Ci sono varie attività che caratterizzano il System Testing:

Functional Testing: testa la funzionalità del sistema cercando le differenze tra i requisiti funzionali ed il sistema effettivo. E' un black-box testing, si usano test case derivanti dai casi d'uso. Si scelgono solo test case che sono rilevanti e che possono portare a fallimenti molto probabilmente.

Performance Testing: testa il sistema cercando le differenze tra i design goals, che derivano dai requisiti non funzionali, del sistema ed il sistema effettivo. Tenta di rompere il sistema spingendolo ai suoi limiti, facendo operazioni non usuali, sovraccaricandolo, minacciando la sua sicurezza e facendo altri vari tentativi per portarlo al fallimento per sperimentare il suo comportamento in tali situazioni.
Vengono svolti vari tipi di testing:
Stress Testing: prova se il sistema può rispondere a più richieste di quelle dovute.
VolumeTesting: tenta di trovare errori dovuti al trattamento di quantità di dati troppo grandi, superiori ai limiti fisici del sistema o di parti di esso.
Security Testing: tenta di assalire il sistema facendo venir meno la sua sicurezza.
Timing Testing: prova i tempi di risposta del sistema, assicurando che siano rispettati i tempi specificati con i requisiti non funzionali.
Recovery Testing: prova la capacità del sistema di ripristinarsi dopo errori e fallimenti vari.
Compatbility Testing: prova la compatibilità con sistemi esistenti.
Configuration Testing: prova le varie configurazioni hardware e le varie configurazioni software.
Human factor Testing: testa con l'utenta l'interfaccia utente.
Environmental Testing: testa la tolleranza alla portabilità e varie caratteristiche dell'ambiente in cui il Sistema sarà messo in funzione.
Quality Testing: testa la manutenibilità e la disponibilità del sistema.

Pilot Testing: testa il sistema una volta installatolo; l'installazione avviene da parte di un gruppo ristretto di utenti, che lo utilizzano e sperimentano come se fosse in versione definitiva, fornendo successivamente i propri commenti. Ci suo due tipi principali di pilot test:
alpha test: un ristretto gruppo di utenti prova il sistema nell'ambiente di sviluppo del sistema, a volte anche con l'intervento degli sviluppatori del sistema..
beta test: un ristretto gruppo di utenti prova il sistema nell'ambiente di uso del sistema; in molti casi il sistema viene diffuso ad un gran numero di persone o addirittura a tutti coloro che vogliono testarlo e si tiene traccia dei bug riscontrati da questi tester.
Acceptance Testing: testa il sistema per fare in modo che sia accettato in tutti i suoi aspetti dal cliente, facendolo provare ai clienti stessi.
Ci sono tre modi in cui il cliente può valutare il sistema:
Benchmark Testing: il cliente sottopone il sistema ad un gruppo di test case che possano rappresentare le condizioni tipiche sotto cui il sistema dovrà lavorare.
Competitor Testing: il sistema viene confrontato con un sistema con comportamento simile esistente già da molto tempo o un prodotto competitore.
Shadow Testing: il sistema viene confrontato con un sistema già esistente, come nel competitor testing, ma in tal caso i clienti eseguono il sistema e quello già esistente da molto tempo in parallelo, per poi confrontare i loro output.
Installation Testing: testa il sistema una volta installatolo nel suo ambiente di utilizzo per riscontrare errori rintracciabili solo in tale ambiente: in molti casi vengono eseguiti test case tipici del functional testing e del performance testing o di altri testing già eseguiti sul sistema, dato che le diverse condizioni dell'ambiente di utilizzo del sistema rispetto all'ambiente di sviluppo del sistema possono portare a risultati diversi per i vari test fatti e si possono riscontrare errori nuovi nel sistema.

Project Management

Un progetto è un insieme ben definito di attività che:

-ha un inizio
-ha una fine
-realizza un obiettivo
-è realizzato da un'equipe di persone
-utilizza un certo insieme di risorse

Il prodotto software è "intangibile": per valutare i progressi ci si deve basare sulla documentazione.

Business Managers: definiscono i termini economici del progetto
Project Managers: pianificano, motivano, organizzano e controllano lo sviluppo
Practitioners: hanno le competenze tecniche per realizzare il sistema
Customers: specificano i requisiti del software da sviluppare
End users: interagiscono con il sistema una volta realizzato

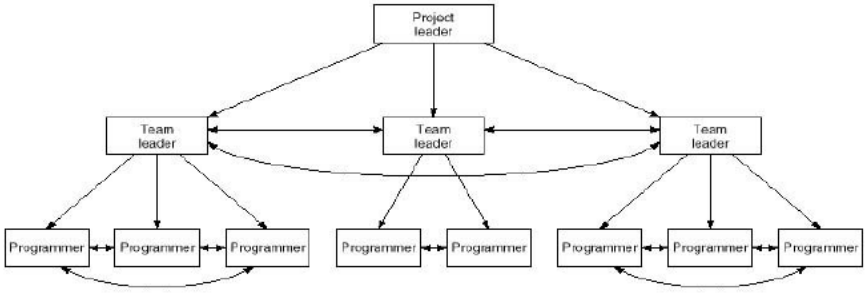
Ci sono varie tipolgoie di team:

Democratico Decentralizzato: Assenza di un leader permanente
Consenso di gruppo sulle soluzioni e sulla organizzazione del lavoro
Comunicazione orizzontale

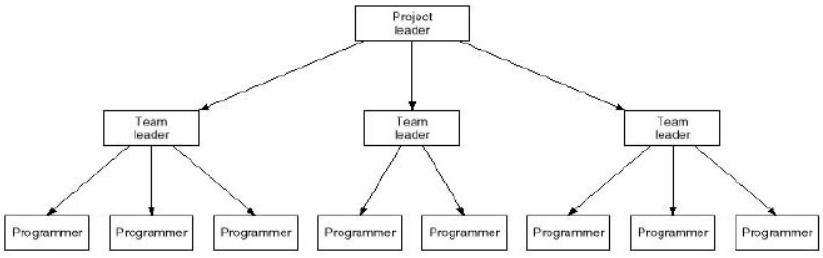
Vantaggi → Attitudine positiva a ricercare presto gli errori
Funziona bene per problemi "difficili" (ad esempio per la ricerca)

Svantaggi → È difficile da imporre…
Non è scalabile...

Controllato Decentralizzato: Un leader riconosciuto, che coordina il lavoro
La risoluzione dei problemi è di gruppo, ma l'implementazione delle soluzioni è assegnata a sottogruppi da parte del leader
Comunicazione orizzontale nei sottogruppi e verticale con il leader



Controllato Centralizzato: Il team leader decide sulle soluzioni e sull'organizzazione
Comunicazione verticale tra team leader e gli altri membri



Il Project manager si occupa di:
-Stesura della proposta di progetto
-Stima del costo del progetto
-Pianificazione (planning) e temporizzazione (scheduling)
-Monitoraggio e revisioni del progetto
-Selezione e valutazione del personale
-Stesura di rapporti e presentazioni

Milestones: punti finali di ogni singola attività di processo.
Deliverables: sono i risultati che sono forniti al committente.

Funzioni Attività o insiemi di attività che coprono tutta la durata del progetto:

- -Project management
- -Configuration Management
- -Documentation
- -Quality Control (Verifica e validazione)
- -Training

Task: Unità di lavoro atomiche, che hanno durata stimabile, necessitano di certe risorse, producono risultati tangibili (come documentazione o codice).

Specifica di un task: Work package

- -Nome e descrizione del lavoro che deve essere fatto
- -Precondizioni per poter avviare il lavoro, durata, risorse necessarie
- -Risultato atteso del lavoro e criteri di accettabilità
- -Rischi

Scheduling di progetto: Divide il progetto in attività e mansioni (tasks) e stima il tempo e le risorse necessarie per completare ogni singola mansione; organizza le mansioni in modo concorrente, per ottimizzare la forza lavoro; minimizza la dipendenza tra le singole mansioni per evitare ritardi dovuti all'attesa del completamento di un'altra mansione.

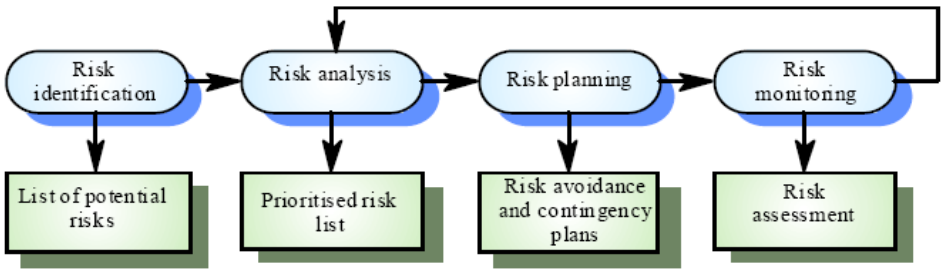
E' difficile stimare la difficoltà dei problemi ed il costo di sviluppo di una soluzione.
La produttività non è proporzionale al numero di persone che lavorano su una singola mansione
Aggiungere personale in un progetto in ritardo può aumentare ancora di più il ritardo
Imprevisti succedono sempre...

Gestione dei rischi

La gestione dei rischi è incentrata sull'identificazione dei rischi e sulla costruzione di piani per minimizzare i loro effetti sul progetto.

Rischio: probabilità che una circostanza avversa possa occorrere.

Ci sono vari tipi di rischio:
Rischio di progetto: rischio che riguarda lo schedule o le risorse.
Rischio di prodotto: rischio che riguarda la qualità o la performance del software che si sta sviluppando.
Rischio di business: rischio che riguarda l'organizzazione che sviluppa o procura il software.



Risk Identification

Identificare la tipologia di rischio:
-Technology risks
-People risks
-Organisational risks
-Tools risks
-Requirements risks
-Estimation risks

Risk Analysis

Valutare la probabilità e la serietà si ogni rischio. Tale probabilità può essere: molto bassa, bassa, moderata, alta, molto alta. Gli effetti dei rischi possono essere: catastrofici, seri, tollerabili, insignificanti.

Risk Planning

Considerare tutti i rischi e sviluppare una strategia per trattarli.
Le tipologie di trattamento sono:
Avoidance strategy: viene ridotta la probabilità che il rischio si presenti.
Minimisation strategy: viene ridotto l'effetto sul progetto o sul prodotto del rischio che eventualmente può verificarsi.
Conitngency plans: i piani di contingenza gestiscono il rischio se il rischio si presenta.

Risk Monitoring

Valutare ognuno dei rischi identificati per capire se il rischio sta diventando man mano più probabile o meno.
Valutare se gli effetti di ognuno dei rischi sono cambiati.
Durante gli incontri per il progetto si dovrebbe discutere sui rischi più importanti e con più gravi conseguenze sul progetto.