L'Ingegneria del Software

L'ingegneria del software prevede teorie, metodi e strumenti per gestire e mantenere software di grandi dimensioni.

Un software è un insieme di programmi e della documentazione associata.

Esistono **software generici** sviluppati per essere venduti a una vasta gamma di utenti e **software personalizzati** che sono sviluppati per un utente specifico in base alle sue necessità.

Caratteristiche di un prodotto software

È necessario evolvere il software in rapporto alla modifica dei requisiti (**mantenibilità**). L'utente si deve poter fidare de software in termini di correttezza, robustezza e sicurezza (**affidabilità**).

Il software non deve sprecare risorse (**efficienza**) e deve avere interfaccia e documentazione appropriata (**usabilità**).

I diversi tipi di software:

Software di SISTEMA

Collezione di programmi al servizio di altri programmi.

Software REAL-TIME

Analizza elementi esterni nel momento esatto in cui avvengono

Software GESTIONALE

Creato per elaborare dati aziendali.

Software SCIENTIFICO e per l'ingegneria

Creato per fornire algoritmi di calcolo intensivo.

Software EMBEDDED

Risiede in memoria per sola lettura e ha lo scopo di controllare prodotti e sistemi di consumo.

Software per il PERSONAL COMPUTING

Sono i tipici software per l'elaborazione di testi oppure i fogli elettronici, posta elettronica etc.

Software per l'INTELLIGENZA ARTIFICIALE

Software che utilizza algoritmi non numerici.

Software basato sul WEB

Script CGI, pagine PHP etc.

L'ingegneria del software si occupa di aspetti pratici che riguardano lo sviluppo di software di qualità e ha come oggetto tutti gli aspetti dello sviluppo di un sistema (hardware e software).

Può essere vista come una branca dell'ingegneria dei sistemi poi che gli ingegneri del software collaborano alla specifica del sistema, alla progettazione architetturale e anche all'integrazione con le varie componenti.

Processo della produzione del software

È un insieme di attività il cui fine è di produrre o modificare il software. In tutti i processi di

produzione si svolgono attività generiche come:

- **Specifica:** ossia che cosa deve fare il sistema e quali sono i vincoli per la progettazione.
- **Sviluppo:** produzione del sistema software.
- **Validazione:** verifica che il software soddisfi le richieste del cliente.
- **Evoluzione:** modificare il software in base alle richieste del cliente.

Modello di processo software

Una rappresentazione semplificata di un processo di sviluppo osservata da uno specifico punto di vista, come ad esempio:

- **work-flow**
- **data-flow**
- **role/action**

I modelli di processo sono generalmente:

- **A cascata:** ci sono fasi distinte di specifica e di sviluppo.
- **Modello Evolutivo:** la specifica e lo sviluppo interagiscono.
- **Modello trasformazionale:** un sistema matematico viene trasformato formalmente in un'implementazione.
- **Sviluppo basato sul riutilizzo:** il software è ottenuto cambiando componenti già esistenti.

Problemi nel processo di sviluppo

Durante lo sviluppo del software possono presentarsi problemi come:

- specifiche incomplete o incoerenti
- mancata distinzione fra le fasi di specifica, progettazione e implementazione
- assenza di validazione
- la manutenzione ha il compito di modificare il prodotto in base a nuove esigenze.

Visibilità del processo di sviluppo

È necessario documentare e tenere traccia di ciò che si sta facendo e per questo ogni fase del processo di produzione deve generare una documentazione che rende visibile il processo di produzione.

Costi del processo di produzione

Il 60% dei prodotti dei costi è legato allo sviluppo mentre il 40% sno costi per la validazione; inoltre i costi variano a seconda del tipo di software che deve essere sviluppato.

La distinzione dei costi dipende dal modello di processo adottato.

Sfide dell'ingegneria del software

Lagency Systems

Ci sono sitemi obsoleti che però sono ancora utilizzati e che quindi necessitano di molta manutenzione.

Eterogeneità

Sistemi distribuiti che includono componenti Hardware e Software di nature diversificate.

Tempi di consegna

Sono necessari tempi sempre più rapidi.

Ingegneria di sistema

Un sistema è un insieme di componenti correlate (software, hardware, risorse umane, dati etc.) che sono finalizzate ad un obiettivo comune.

La definizione è: **un insieme di elementi organizzati in modo da raggiungere uno scopo prefissato mediante l'elaborazione di dati.**

Ingegneria di sistema significa quindi **progettare, implementare e installare** sistemi che includono hardware e software oltre che a coordinare il personale.

Modellazione del sistema

Può essere utile per valutare in anticipo le caratteristiche quantitative o qualitative. È divisa in varie fasi:

- **Definire una serie di processi che rappresentano entità della realtà fisica.**
- **Definire il comportamento di ciascun processo.**
- **Definire i dati che guidano il sistema (esogeni o endogeni)**

Simulazione del sistema

Gli strumenti di modellazione e simulazione aiutano a eliminare i problemi su sistemi. Sono applicati durante il processo di ingegneria del sistema.

Affidabilità di un sistema

L'affidabilità di un sistema dipende dall'affidabilità dell'Hardware e del Software oltre che a quella degli operatori. L'affidabilità del software differisce da quella dell'hardware perché il software non è soggetto ad usura. La capacità di **recupero** (resilience) è l'abilità del sistema di continuare ad operare correttamente in presenza di fallimenti.

Proprietà emergenti

Sono le proprietà del sistema nella sua globalità e derivano dalle interazione tra le componenti che possono essere:

- **Peso globale del sistema**
- **Affidabilità**
- **Usabilità (dipende anche dalla formazione dell'utente)**
- **Riparabilità.**

Queste proprietà si distinguono in **funzionali** e **non funzionali**. A causa di alcune interdipendenze all'interno del sistema , gli errori si possono propagare da una componente all'altra.

Requisiti

- **Requisiti funzionali**
- **Proprietà del sistema (non funzionale)**
- **Caratteristiche che il sistema non deve presentare**

Disegno del sistema

Definisce il modo in cui le funzionalità del sistema devono essere fornite dalle diverse componenti..

Integrazione del sistema

Ciascun sottosistema deve essere integrato in una entità completa.

Installazione del sistema

Consiste nella collocazione del sistema nel suo ambiente operativo. A volte succede che l'ambiente operativo differisca da quello che era stato preventivamente ipotizzato.

Il modello architetturale è presentato sotto forma di diagramma a blocchi.

Processi di produzione del software

È l'insieme coerente delle attività per la specifica, il progetto e l'implementazione di sistemi software.

Un modello di progetto è una rappresentazione astratta e descrive il processo in una particolare prospettiva. Modelli di processo generici possono essere:

- **A cascata**
- **Sviluppo evolutivo**
- **Sviluppo formale**

Nel ciclo di risoluzione dei problemi esso assume un aspetto frattale perché si può applicare a diversi livelli (**status quo, definizione del problema, sviluppo tecnico, integrazione**).

Modello a cascata (modello sequenziale lineare)

Si divide in varie fasi:

- **analisi dei requisiti**
- **disegno del sistema**
- **implementazione e test dei sottosistemi**
- **integrazione e test del sistema**
- **operatività e mantenimento**

Il modello a cascata presenta alcuni problemi infatti, dopo l'avvio del processo di produzione, è difficile operare dei cambiamenti.

Il modello a cascata è delegato quando i requisiti sono ben compresi e non sono soggetti a modifiche.

Nel caso in cui i requisiti non siano chiari viene messa in atto una metodologia di lavoro detta **prototipazione**.

Sviluppo evolutivo

Può essere basato su:

Exploratoring Programming

Data una idea di base si lavora in stretto contatto con il committente in modo da arrivare ad evolvere verso un prodotto finale seguendo i requisiti che man mano vengono proposti dal committente. Procedendo in questa maniera si è sicuri che i requisiti sono ben compresi.

Throw-away Prototyping

L'obiettivo è coprire al meglio i requisiti proposti dal cliente per cui il punto di partenza è lo sviluppo dei requisiti che non sono stati ben compresi.

Anche in questo modello si presentano alcuni problemi come la mancanza di visibilità sull'andamento dello sviluppo e la tendenza a creare sistemi poco strutturati.

Può essere applicato per sistemi di piccole dimensioni, per sistemi con un breve ciclo vitale ma soprattuto per lo sviluppo di prototipi.

Modello RAD (Rapid Application Development)

È un modello sequenziale che unta a un ciclo di sviluppo breve. Può essere applicato quando i requisiti sono chiari e può portare allo sviluppo del software in tempi brevi. Ogni parte deve poter essere sviluppata indipendentemente dalle altre.

Il modello è inadatto quando il sistema nono è partizionabile e quando si sviluppano sistemi con tecnologie innovative c'è un notevole aumento del rischio.

Sviluppo formale

Si basa su specifiche matematiche di un sistema in differenti rappresentazioni. Questo sviluppo richiede però conoscenze specialistiche. Può essere applicato per gestire sistemi critici.

Sviluppo incrementale

È la combinazione di sviluppo lineare e sviluppo con prototipi, il sistema non è fornito da subito in modo completo ma viene sviluppato un prototipo che soddisfa parte dei requisiti e ad ogni iterazione con il cliente si sviluppano nuove funzionalità. Nello sviluppo si prende come punto di partenza la funzionalità che ha priorità più elevata.

Il cliente riceve un prototipo funzionante in tempi brevi riducendo i rischi.

Modello a spirale

Il processo di sviluppo non ha una rappresentazione sequenziale ma a spirale dove ogni ciclo è una fase del processo. Al termine di ogni ciclo può risultare un **progetto** o un **prototipo** oppure un **prodotto software**. Non ci sono fasi predefinite. la spirale è divisa a spicchi:

- **comunicazione con il cliente**
- **pianificazione**
- **analisi dei rischi**
- **strutturazione**
- **costruzione e rilascio**
- **valutazione da parte del cliente**

Il modello a spirale tiene conto dei rischi ed è adatto allo sviluppo di sistemi complessi. Prevede l'integrazione co il cliente ad ogni ciclo di spirale. La stima dei rischi prevede competenze specifiche.

Extreme programming (XP)

Prevede una consegna pressoché continua di prototipi che presentano di volta in volta piccoli incrementi di funzionalità. Si basa sul miglioramento costante del codice.

Visibilità dei modelli

Il modello a cascata ha visibilità ottima.

Il modello a spirale e le trasformazioni formali hanno buona visibilità.

Gestione dei progetti

Software project management

È l'insieme delle attività necessarie per assicurare che un prodotto software sia sviluppato rispettando determinate esigenze. È un implementazione tra aspetti economici e tecnici.

Un progetto è un insieme ben definito di attività che hanno un inizio e una fine, che realizzano un obiettivo, e che sono realizzate da un'equipe di persone utilizzando un certo numero di risorse.

L'ingegneria del software deve attenersi a determinati vincoli economici che vengono imposti o contrattati. Un progetto software è intangibile quindi p necessario valutarne i progressi basandosi sulla documentazione.

Un progetto può essere in ritardo a causa di alcuni eventi quali:

- **scadenza non realistica fissata da valutatori esterni**
- **mutamenti dei requisti**
- **rischi dei quali non si era tenuto conto**
- **problemi di comunicazione fra i membri dello staff**

Nel caso in cui non sia possibile rispettare i tempi di consegna pattuiti, le possibilità sono le seguenti:

- aumentare il budget ed aggiungere risorse (soluzione non sempre attuabile)
- eliminare funzionalità non strettamente necessarie
- tentare comunque di rientrare nel tempo stabilito

Principi fondamentali per la pianificazione temporale

Ripartizione

Un progetto deve essere ripartito fra gli sviluppatori in attività e compiti di dimensioni ragionevoli.

Interdipendenza

Occorre determinare le dipendenze reciproche fra le attività e dei compiti. Alcuni di questi vanno sviluppati in sequenza mentre altri in parallelo. È anche da tener presente la propedeuticità di alcune attività ovvero che alcune non possono iniziare prima che altre siano state terminate.

Assegnazione del tempo

Ad ogni compito è necessario assegnare un certo numero di unità di lavoro e una data di inizio e di fine.

Valutazione dell'assegnazione

Ad ogni progetto va assegnato un numero preciso di persone assicurandosi di non averne assegnate un numero maggiore o minore di quello che è necessario.

Responsabilità ben definite

Ogni compito deve essere assegnato ad un membro e questo deve risponderne.

Risultati definiti

Ogni compito deve produrre un risultato coerente alla richiesta.

Punti di controllo

Si dice punto di controllo l'approvazione dell'elaborato in corso che consiste nell'esaminare e nel controllo della sua qualità.

Le figure in gioco...

- **Senior Managers:** definiscono i termini economici del progetto.
- **Project Managers:** hanno il compito di organizzare la stesura del progetto, stimare il costo del progetto e monitorare il progetto, selezionare il personale, stendere e presentare i rapporti dell'andamento lavorativo.
- **Practitiores:** hanno le competenze tecniche che consentono la realizzazione del sistema.
- **Costumers:** specificano i requisiti del software.
- **End users:** utilizzano il software utilizzato.

Il piano del progetto

- **definire gli obiettivi del progetto**
- **organizzare il team di sviluppo**
- **analizzare i rischi e la probabilità che si presentino**
- **definire le risorse hardware e software richieste**
- **suddividere il lavoro**
- **descrivere le dipendenze fra le attività**
- **controllo e rapporto sulle attività**

I fattori più importanti che determinano un fallimento sono: **requisiti incompleti** e **mancato coinvolgimento del cliente**.

Organizzazione delle attività

Le attività del progetto devono produrre dei documenti per i manager. È necessario raggiungere delle **milestones** che sono i punti finali di ciascuna attività. I **deliverables** sono i risultati consegnati al cliente.

Modello a cascata

Consente di definire banalmente i milestones in quanto le attività sono molto ben definite.

Permette inoltre di suddividere il progetto in compiti e stimare il tempo e le risorse necessarie ed evitare ritardi causati da compiti che devono attendere la terminazione di altri.

La realizzazione tra numero di persone e produttività non è lineare.

Tipologie di Team

Democratico decentralizzato

In questa tipologia c'è assenza di un leader permanente ma un consenso di gruppo. È necessario un livello comunicativo ottimale. È una tipologia molto produttiva per la risoluzione di problemi articolati e complicati che però è molto difficile da applicare.

Controllato decentralizzato

C'è un leader che coordina globalmente il lavoro, i problemi vengono risolti e discussi in gruppo mentre l'implementazione è assegnata a sottogruppi. C'è una comunicazione orizzontale con i gruppi e verticale con il leader (al quale ogni sottogruppo deve rendere conto dell'elaborato).

Controllato centralizzato

Il leader decide tutto riguardo all'organizzazione: la comunicazione è sempre di tipo verticale sia fra i leaders dei team che fra i membri.

Diagrammi d GANTT e Activity Network

Sono rappresentazioni grafiche usate per illustrare lo schedule di un progetto suddiviso in tasks.

Le activity network mostrano le dipendenze fra i vari tasks con il loro percorso critico mentre i diagrammi d GANTT mostrano lo scheduling che funge da calendario di lavoro.

Una buona gestione è fondamentale per il successo di un progetto.

I requisiti

L'ingegneria dei requisiti stabilisce quali sono le richieste di un utente dal sistema software. Stabilisce quindi i servizi e i vincoli del sistema.

Esistono due tipi di requisiti:

- **funzionali:** descrivono le funzionalità e i servizi offerti
- **non funzionali:** descrivono i vincoli che il sistema o il processo di sviluppo devono soddisfare

Si possono verificare inconsistenze sui requisiti in quanto utenti diversi possono avere diversi requisiti e priorità, oppure perché gruppi diversi di utenti esprimono allo stesso modo requisiti diversi.

I requisiti non funzionali possono presentarsi anche più critici di quelli funzionali.

Processo di ingegneria dei requisiti

Nello **studio di fattibilità** le esigenze del cliente vengono prese in esame per stabilire se possano essere soddisfatte o meno. Con l'**analisi dei requisiti** si individua cosa viene richiesto.

La fase di **definizione dei requisiti** prevede di descrivere i requisiti in forma comprensibile. I requisiti verranno descritti con maggior dettaglio nella fase di **specifica dei requisiti**.

Documento dei requisiti

Definisce ufficialmente ciò che viene richiesto dal sistema, include sia le specifiche che la definizione dei requisiti.

Validazione dei requisiti

La validazione è molto importante poiché un errore sull'implementazione dei requisiti può avere una ripercussione sui costi molto elevata.

Consiste nel dimostrare che i requisiti definiti coincidono con quello che ha richiesto il cliente.

Controllo dei requisiti

- **Validità**
- **Consistenza** (assenza di conflitti)
- **Completezza**
- **Realismo**
- **Verificabilità**
- **Comprensibilità**
- **Tracciabilità** (è indicata l'origine del requisito)
- **Adattabilità**

I requisiti devono inoltre essere **revisionati periodicamente** in quanto è più semplice risolvere i problemi non appena essi si manifestano.

Esistono **revisioni formali** e **revisioni informali**.

FAST (Facilitated Application Specification Technique)

È una tecnica per la specifica di applicazioni e promuove la formazione di un team congiunto di clienti e sviluppatori che collaborano ad individuare il problema, proporre soluzioni etc.

Viene svolta una riunione a cui partecipano ingegneri del software e clienti in cui vengono stabilite delle regole. Si pone un moderatore e successivamente si applica un meccanismo di definizione (presentazione alla lavagna etc.).

Classi di requisiti

Requisiti duraturi

Derivano dalla natura stessa dell'attività del cliente.

Requisiti volatili

Possono variare durante lo sviluppo del sistema.

Gli scenari detti anche casi d'uso descrivono il modo in cui il sistema verrà usato. La parte degli **attori** corrisponde a coloro che interagiscono con il sistema mentre i **casi d'uso** descrivono il modo in cui gli attori interagiscono con il sistema.

Analisi e definizione dei requisiti

Nell'analisi dei requisiti possono verificarsi problemi in quanto le controparti non sanno esattamente cosa vogliono ed esprimono requisiti con la propria terminologia.

Attività del processo

- **Comprensione del dominio:** gli analisti devono sviluppare una propria comprensione del dominio applicativo.

- **Raccolta dei requisiti:** interazione con i clienti.
- **Classificazione**
- **Risoluzione dei conflitti**
- **Attribuzione delle priorità**
- **Validazione dei requisiti**

VORD (Viewpoint Oriented Requirements Definition)

- **Identificazione dei Viewpoints**
- **Strutturazione dei Viewpoints rappresentabili** (ovvero creare una gerarchia di essi)
- **Documentazione dei Viewpoints**
- **Mapping dei Viewpoints di sistema** (ovvero definire gli oggetti che li rappresentano)

Alternative al linguaggio naturale

- **Linguaggio naturale strutturato** (definisce forme standard o template per esprimere le specifiche)
- **Design description Languages** (una sorta di linguaggio di programmazione)

Tracciabilità dei requisiti

I requisiti in qualche modo associati devono essere collegabili. Questa è una proprità intrinseca dei requisiti che si ottiene numerandoli.

PDL (Program Description Language)

I requisiti possono essere descritti in modo operativo utilizzando una specie di linguaggio di programmazione adatto in due particolari situazioni:

- quando un'operazione deve essere specificata come sequenza in cui l'ordine ha un'importanza rilevante
- quando è necessario specificare le interfacce fra hardware e software

Svantaggi dei PDL

I PDL possono risultare insufficienti per descrivere informazioni ad alto livello.

Specifica delle interfacce

Si possono definire tre tipi di interfacce in un documento di specifica dei requisiti:

- **Interfacce procedurali:** servizi offerti da sottosistemi
- **Interfacce dati:** strutture dati che vengono trasmesse
- **Interfacce di rappresentazione:** pattern visualizzati per descrivere i dati

I requisiti non funzionali possono essere più critici di quelli funzionali e possono essere **requisiti di prodotto** oppure **requisiti organizzativi** o **requisiti esterni**.

Progettazione

La progettazione si divide in due fasi:

- **Diversificazione:** il progetista acquisisce materiale gretto del progetto.
- **Convergenza:** seglie e combina gli elementi disponibili per arrivare ad un prodotto finale.

Il progetto si divide in 3 requisiti:

1. Deve **soddisfare tutti i requisiti espliciti** contenuti nel modello concettuale e tutti i **requisiti impliciti**.
2. Deve essere una **guida** leggibile e comprensibile per chi si occuperà della modifica, del collaudo e della manutenzione.
3. Deve dare un **quadro completo** e coerente del software.

Indicazioni generali

*L'**architettura*** del progetto deve essere creata con **modelli** di presentazione riconoscibili e deve essere implementata in **modo evolutivo.**

Il ***progetto*** deve essere **modulare** e le **interfacce** devono ridurre le difficoltà di comunicazione dei moduli verso l'esterno.

Le tecniche di ***progettazione*** sono basate su quattro tipi di **tool**:

1. **Meccanismi** per tradurre il modello concettuale in progetto.
2. **Notazioni** per presentare i componenti funzionali e le loro interfacce.
3. **Regole** euristiche per la suddivisione dei compiti.
4. **Metodi** per la valutazione della qualità.

Il progetto deve essere sempre riconducibile al modello concettuale e il progetto finale deve apparire uniforme e integrato. Il progetto non è da intendersi come la stesura di codice.

Fasi del progetto

- **Comprensione del problema**
- **Identificare una o più soluzioni**
- **Descrivere in astratto le soluzioni**
- **Ripetere il processo per ciascuna astrazione**

L'**astrazione** è l'atto di dare una descrizione del sistema ad un certo livello trascurando i dettagli inerenti ai livelli sottostanti.

Il **raffinamento** utilizza invece tecniche di scomposizione per passare da astrazioni di *alto livello* a *linee di codice*. Il raffinamento è complementare all'astrazione.

Fasi del disegno

- **Disegno architetturale:** identificare i sottoinsiemi e le relazioni
- **Specifica astratta**
- **Disegno delle interfacce**
- **Disegno delle componenti**
- **Disegno di strutture di dati**
- **Disegno degli algoritmi**

Progettazione Top-Down

È un metodo per affrontare l'attività di progettazione dei sistemi dove il problema viene partizionato ricorsivamente in sotto problemi (si parte dal componente radice per poi andare verso il basso).

Esiste anche la progettazione *Bottom-Up* che consiste dapprima nella composizione delle varie soluzioni e anche la soluzione a *Sandwich*.

Modularità e integrazione

Un programma può essere gestibile se vengono individuati moduli con funzionalità definite.

Criteri di Meyer

Ci sono 5 criteri per valutare un metodo di progettazione di software in base alla loro capacità di produrre sistemi modulari.

- **Scomponibilità:** riduce la complessità
- **Componibilità:** migliora la produttività
- **Comprensibilità:** è di più semplice costruzione e modificabilità
- **Protezione:** l'anomalia ha una propagazione controllata
- **Continuità**

Architettura del software

Descrive la struttura gerarchica dei moduli di un programma ed ha le seguenti proprietà:

- **Proprietà strutturali**
- **Proprietà extra-funzionali**
- **Affinità** (deve permettere di usare strutture simili per progetti simili)

L'architettura può essere presentata usando uno o più modelli

- **Modelli strutturali** (architettura come una collezione organizzata di componenti)
- **Modelli schematici** (individuano schemi progettuali ricorrenti)
- **Modelli dinamici** (indica in che modo il sistema muta a seguito di eventi esterni)
- **Modeli Funzionali** (descrivono la gerarchia funzionale)
- **Modelli di processo** (mettono in evidenza il processo)

La **gerarchia di controllo** descrive l'organizzazione gerarchica dei moduli di un programma.

Ripartizione strutturale

In un sistema ad organizzazione gerarchica la strutturazione di un programma può essere partizionata in senso:

- **orizzontale:** è l'approccio più semplice e definisce tre partizioni (**input, trasformazioni, output**) avendo come vantaggio un software più facile da collaudare, con più semplice manutenzione e propagazione ridotta degli errori.
- **verticale:**

La struttura dei dati

Definisce l'organizzazione i metodi di accesso e le alternative di elaborazione per le informazioni.

Organizzazione e complessità dipendono dal progetista e dalla natura del problema.

La procedura software si concentra sui detagli dell'elaborazione specificando la sequenza degli

eventi.

Information hiding

Il principio dell'information hiding richiede che ciascun modulo sia definito in modo che le sue procedure e le informazioni locali non siano accessibili ad altri moduli. Il vantaggio sta nella facilità di modificare successivamente ciascun modulo senza comprometterne il funzionamento di altri.

Può essere utile nell'identificare il punto di minimo costo per la modularità del sistema.

Strategie di design

Disegno funzionale

Lo stato del sistema è centralizzato e viene condiviso dalle funzioni che operano su di esso.

Disegno objet-oriented

Il sistema viene visto come una collezione di oggetti che interagiscono fra loro. La comunicazione avviene tramite la chiamata di metodi di altri oggetti.

Qualità di un progetto

Dipende dalle specifiche priorità di tipo organizzativo. Durante le modifiche, il progetto dovrebbe rimanere comprensibile e i cambiamenti dovrebbero avere un effetto locale.

Per avere **modularità** i moduli dovrebbero essere del tutto indipendenti e l'indipendenza dovrebbe essere misurata in termine di coesione e di accoppiamento con altri moduli.

Ogni modulo avrà un grado più o meno alto di coesione ovvero esegue un numero di compiti limitato e coerente (caso ideale prevede un compito per modulo).

Sono stati identificati **diversi livelli di coesione:**

- **Coesione incidentale**: le diverse parti di un componente sono semplicemente impacchettate ma non correlate.
- **Associazione logica:** le componenti con compiti simili vengono raggruppate.
- **Coesione temporale:** vengono raggruppate le componenti che vengono attivate nello stesso istante.
- **Coesione procedurale:** sono raggruppati tutti gli elementi che costituiscono una singola sequenza di controllo.
- **Coesione di comunicazione:** elementi che operano sullo stesso input e che producono gli stessi output.
- **Coesione sequenziale:** l'output di una componente diventa l'input di un'altra.
- **Coesione funzionale:** ogni parte di una componente è necessaria per l'associazione di una singola funzione.
- **Coesione di oggetto:** ogni operazione fornisce funzionalità per osservare gli attributi di un oggetto.

Accoppiamento

Misura la forza dell'interconnessione tra le componenti del sistema. Con **accoppiamento debole** si intende che i cambiamenti di una componente difficilmente avranno effetti su un altra componente. Questa situazione si ha quando c'è una decentralizzazione dello stato.

L'uso di variabili condivise porta invece alla situazione di **accoppiamento stretto**.

- **Basso accoppiamento:** moduli diversi si scambiano parametri semplici la cui alternativa è l'accoppiamento a stampo ovvero quando attraverso l'interfaccia di un modulo passa una struttura dati.
- **Accoppiamento di controllo:** un segnale di controllo viene scambiato fra due moduli.
- **Accoppiamento comune:** moduli diversi hanno dati in comune. È il più complicato da gestire.

Nei sistemi object-oriented l'accoppiamento è debole perché non ci sono variabili condivise e gli oggetti comunicano tramite l'**invocazione di metodi**. Una classe è però **strettamente accoppiata** alla sua **superclasse**.

Comprensibilità

Alta comprensibilità implica molte relazioni fra le diverse parti del progetto. Esso è riferito a diverse caratteristiche di una componente (coesione, nomi, documentazione, complessità...).

Adattabilità

Un progetto è adattabile se:

- **Le componenti sono debolmente accoppiate.**
- **È ben documentato.**
- **C'è corrispondenza ovvia fra livelli di progetti diversi.**
- **Ogni componente è auto-contenuta** (forte coesione).

L'ereditarietà migliora l'adattabilità ovvero le componenti possono essere adattate senza cambiamenti attraverso la derivazione di una sotto-componente.

Man mano che la profondità dell'albero di ereditarietà cresce il progetto diventa via via più complesso e meno adattabile (è necessaria una ristrutturazione dell'albero).

Rendere un progetto più modulare

- **Aumentare la coesione e diminuire l'accoppiamento:**
  - esplosione dei moduli (aumento coesione)
  - implosione dei moduli (accoppiamento diminuisce)
- **Diminuire il fan-out ad alto livello gerarchico e diminuire il fan-in a basso livello gerarchico**
- **Mantenere il campo di azione di un modulo dentro al suo campo di controllo**
  - campo d'azione (insieme dei suoi moduli locali)
  - campo di controllo (insieme dei moduli subordinati)
- **Studiare interfacce per ridurre l'accoppiamento**
- **Definire i moduli con funzioni prevedibili**

Specifica del progetto

Documento che descrive il progetto finale nel seguente modo:

- Descrizione globale derivata dalla specifica dei requisiti

- Descrizione del progetto dei dati
- Descrizione dell'architettura
- Progetto delle interfacce interne ed esterne
- Descrizione procedurale dei singoli componenti in linguaggio naturale

In ogni punto vanno inseriti riferimenti alla specifica dei requisiti da cui è stata ricavata quella parte.

Nella descrizione delle interfacce va inclusa una prima versione della documentazione. Alla dine della documentazione vano aggiunti i dati supplementari.

Progettazione Architetturale

L'**architettura del software** deve analizzare l'efficacia del progetto. Valutando tutte le possibili alternative architetturali, riducendo i rischi tecnici nella fase successiva di implementazione. La fase di architettura non riguarda la stesura di codice.

Architettura software = processo dei dati + progetto architetturale.

La rappresentazione dell'architettura è il mezzo tramite cui le parti comunicano. Mette in evidenza le decisioni progettuali e costituisce un modello di come il sistema è strutturato e di come le componenti collaborano tra di loro.

Progetto dei dati

Si concentra sulla rappresentazione delle strutture dati e si basa su alcuni principi:

- **Analisi sistematica sulla funzionalità**
- **Individuazione di tutte le strutture di dati e le operazioni da svolgere**
- **Dev'essere compilato un dizionario dei dati**
- **Decisioni di basso livello devono essere rimandate alle ultime fasi di progettazione**
- **Le strutture dati devono essere accessibili soltanto ai moduli che le utilizzano**
- **Sviluppo di una libreria di strutture dati**
- **Tipi di dati astratti devono essere utilizzati con un progetto del software e su un linguaggio di programmazione**

Stili di architettura

Uno stile è composto da un insieme di componenti che implementano le funzionalità, un insieme di connettori che consentono comunicazione e cooperazione, i vincoli che definiscono i modi in cui i componenti possono essere integrati, modelli semantici che consentono al progettista di comprendere il funzionamento globale del sistema.

Architettura basata sui dati (Repository)

In questo caso il sistema è concentrato su un archivio di dati dove la altre componenti accedono operando in modo indipendente. L'archivio può essere **passivo** (tutte le operazioni vengono eseguite da parte del client) oppure **attivo** (notifica ai clients le variazioni dei dati).

Il vantaggio è dato sostanzialmente dall'indipendenza fra i moduli e l'accoppiamento può avvenire attraverso il meccanismo di **bleckboard**.

Vantaggi

- **Modo efficiente per condividere grandi quantità di dati**
- **I client non necessitano di sapere in che modo i dati sono condivisi poiché la gestione è centralizzata**
- **Lo schema dei dati può essere pubblicato nel repository condiviso.**

Svantaggi

- **I clients devono basarsi sulla stessa struttura dei dati condivisa**
- **Modificare la struttura dei dati risulta al quanto complicato**
- **È difficile impostare nuove politiche di accesso**
- **C'è una bassa scalabilità**

Architettura client-server

È un modello distribuito ed è composto da un insieme di **servers** che offrono servizi, un insieme di **clients** che utilizzano i servizi e una **rete** di comunicazione.

Vantaggi

- **Semplice distribuzione dei dati**
- **La rete permette di collegare fra loro nodi che non hanno un'elevata potenza di calcolo**
- **È semplice aggiungere servers**

Svantaggi

- **Non esiste un unico modello di dati condiviso**
- **Alcune attività di gestione devono essere replicate per ciascun server**
- **Non c'è alcun registro centrale di nomi e di servizi**

Architettura a flusso di dati (pipe-and-filter)

Il sistema è modellato sul flusso che porta i dati da input ad output dove i moduli si comportano da filtri connessi da pipe di dati e ogni filtro si attende solo dati di input con un certo pormato e produce output di un altro determinato formato.

Ogni filtro lavora senza preoccuparsi dei filtri precedenti o che seguono.

Se il flusso si riduce ad una unica catena di filtri allora è detto **Filtro a Batch Sequenziale**.

Vantaggi

- **Semplice costruzione**
- **Ogni filtro può essere riutilizzato**
- **Non c'è condivisione dello stato in filtri ben progettati**

Svantaggi

- **I formati di dati in input e in output di filtri collegati devono essere compatibili**
- **Modello inadeguato per una struttura di controllo non linearizzata**

Architettura a macchina astratta (a livelli)

Utilizzata per modellare l'interfacciamento di sottosistemi. Viene organizzata in un insieme di strati, ciascuno dei quali fornisce servizi e dove ciascun livello comunica con i livelli adiacenti.

Vantaggi

- **Supporta uno sviluppo del sistema in modo incrementale**
- **Se si cambia un interfaccia di un livello ci sono ripercussioni solo sul livello adiacente**

Svantaggi

- **È complicato strutturare sistemi a livelli**
- **È restrittivo il fatto che un livello possa integrarsi soltanto con quelli adiacenti**

Modelli di controllo

Descrivono come fluisce il controllo tra sottosistemi.

Controllo centralizzato

Un sistema ha la responsabilità del **controllo globale** e deve occuparsi di attivare o disattivare sistemi e sottosistemi. Esiste un sottosistema che si prende l'onere di gestire il controllo complessivo.

Modello Call-And-Register

È applicabile a sistemi sequenziali in quanto una routine parte dal nodo radice e si sposta verso il basso. Il programma principale richiama le diverse routines. Quella che inizialmente era stata progettata come un insieme di routines annidate, adesso è considerata un'architettura ad oggetti distribuiti.

Manager Model

Applicato per sistemi concorrenti dove una componente coordina le altre.

Controllo basato su eventi

Eventi esterni attivano i sottosistemi responsabili della gestione di quell'evento.

Modelli basati su broadcast

Lo stesso evento viene inviato a tutti i sistemi contemporaneamente. È utile per integrare diversi sistemi collegati in rete. Ciascun sottosistema si registra per ricevere gli eventi.

Modelli basati sugli interrupt

Sono utilizzati in sistemi **real-time** dove gli interrupt sono raccolti da un gestore e passati direttamente alle componenti del sistema. Bisogna avere una definizione per ogni tipo di interrupt.

Architetture per domini applicativi specifici

Modelli architetturali specifici a determinati domini applicativi.

Modelli generici

Rappresentano astrazioni di un numero di modelli reali, sono solitamente **bottom-up**. Un esempio di questo può essere un compilatore.

Modelli di riferimento

Sono modelli più astratti e idealizzati. Forniscono informazioni su una particolare classe di sistemi e sono utilizzati per confrontare architetture diverse. Solitamente sono **top-down**.

Derivano dallo studio del dominio applicativo e possono essere utilizzati come base per un'implementazione di sistemi diversi. Ad esempio il modello OSI di comunicazione.

Progettazione di interfacce utente

Esistono tre regole che devono guidare alla creazione dell'interfaccia utente:

1. **Controllo nelle mani dell'utente** (l'input deve poter avvenire in più condizioni)
2. **Limitare la necessità che l'utente faccia spesso ricorso all'uso della propria memoria**
3. **Utilizzare una interfaccia uniforme per tutte le applicazioni**

Prospettive sull'interfaccia

Bisogna considerare un'interfaccia a partire da quattro punti di vista:

- **Vista centrata sul progetto (ingegneria del software)**
- **Vista centrata sull'utente (ergonomia)**
- **Vista centrata sulla percezione (utente finale)**
- **Vista concentrata sull'implementazione e sull'immagine del sistema (programmazione)**

Queste viste pongono vincoli contrastanti sul risultato finale e il compito di un progettista di interfacce è quello di arrivare ad un buon compromesso.

Modelli utente

Vanno valutate le caratteristiche tipiche di un utente finale.

Dal punto di vista del background l'utente può essere suddiviso in 3 categorie:

1. **Principiante**
2. **Casuale**
3. **Esperto**

Per **immagine di sistema** si intende la manifestazione intesa dal punto di vista dell'implementatore.

Per **percezione del sistema** si intende l'immagine mentale che l'utente si costruisce attraverso l'interazione con il sistema.

Il massimo beneficio si ha quando l'immagine tende a coincidere con l'immagine del sistema e per fare ciò il progetto deve tener conto delle modalità di input.

Processo di progettazione dell'interfaccia

Segue un andamento a spirale e si può suddividere in quattro fasi:

- **Analisi e modellazione degli utenti**
- **Disegno dell'interfaccia**
- **Implementazione dell'interfaccia**
- **Validazione dell'interfaccia**

Analisi dell'utente e dell'ambiente

Bisogna determinare il livello di abilità dell'utente, le conoscenze generali e la sua disponibilità ad accettare un certo tipo di sistema.

Dell'ambiente si deve invece tener conto di dove dovrà essere collocato fisicamente l'hardware, nel rispetto dei vincoli ambientali.

Attività di progettazione dell'interfaccia

È necessario stailire gli obiettivi di ciascuna operazione e mapparli in una sequenza di azioni. Bisogna indicare lo stato del sistema e definire i meccanismi di controllo.

Definizione degli oggetti e delle azioni

Dalla descrizione delle operazioni si ricava una lista di oggetti e azioni. Gli oggetti possono essere di destinazione o di origine. Una volta individuati gli oggetti bisogna specificare come devono essere disposti.

Problemi di progettazione

I tempi di risposta non dovrebbero essere né troppo lunghi né troppo brevi ma avere una durata constante. Per gestire gli errori vanno evitati messaggi incomprensibili: vanno quindi fornite informazioni utili all'utente attraverso messaggi audio o video.

Strumenti di implementazione

Esistono strumenti per la creazione di prototipi di interfaccia che utilizzano componenti ed oggetti di base per gestire:

- dispositivi di input
- errori e messaggistica
- campi di scorrimento interno
- connessione fra software e interfaccia

Valutazione del progetto

La valutazione passa per un test di utilizzo dell'interfaccia ed è **informale** oppure **formale** (la valutazione informale è data dai report dei singoli utenti).

La fase di valutazione complessiva del progetto può essere accorciata se ci sono valutazioni già effettuate in fasi precedenti.

UML

UML è una notazione di modellazione ma non una metodologia in quanto essa di solito comprende un linguaggio di modellazione e un processo di modellazione.

UML definisce solamente una **notazione** e un **meta-modello**.

- **Notazione:** definisce l'aspetto grafico dei modelli, rappresenta la sintassi del linguaggio di modellazione.
- **Meta-modello:** un insieme di diagrammi che definisce la notazione stessa.

Tipi di diagrammi UML

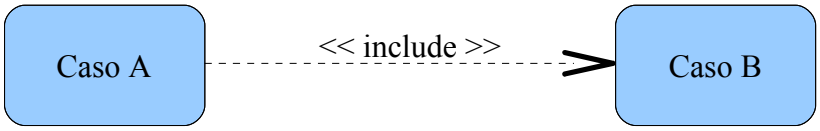
- Diagrammi dei casi d'uso
- Diagrammi di classe, degli oggetti e del package
- Diagrammi di sequenza e di collaborazione
- Diagrammi di componenti e di deployment
- Diagrammi di stato
- Diagrammi di attività

Concetto di scenario

È una sequenza di passi che descrivono l'interazione tra un untente e un sistema. Un **caso d'uso** è un insieme di azioni legate ad un obiettivo comune per l'utente.

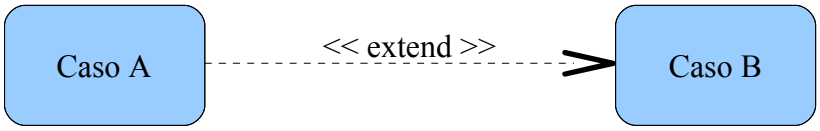
I diagrammi di casi d'uso descrivono ad alto livello l'interazione tra il sistema e uno o più attori che richiedono un servizio.

Un attore è un ruolo interpretato dall'utente e nei confronti del sistema non deve necessariamente essere una persona. Gli attori interpretano i casi d'uso. Un singolo attore può infatti trovarsi in più casi d'uso e un caso d'uso potrebbe essere usato contemporaneamente da più attori.



Rappresenta il **diagramma dell'inclusione** ovvero quando un detrrminato componente si ripete in più casi.

Il **diagramma della generalizzazione** indica che un caso d'uso è simile ad una latro ma è in grado di fare qualcosa in più.



L'**estensione** p simile alla generalizzazione ma il caso d'uso che estende il caso base può aggiungere comportamenti riferiti a determinati punti di estensione specificati nel caso base.

Individuazione degli attori

Gli attori sono agenti esterni al sistema e quindi non possono essere controllati. Interagiscono direttamente con il sistema aiutando così a definire i confini del sistema stesso.

Non rappresentano personoe o cose specifiche ma i ruoli generici che gli stessi possono rivestire.

L'attore può rivestire ruoli diversi nei confronti del sistema e deve essere identificato fa un nome breve e caratterizzato da una descrizione significativa. Le azioni periodiche non prevedono nessun attore.

Diagrammi di classe

Descrivono il tipo di oggetti che compongono il sistema e le relazioni statiche esistenti (associazioni e sottotipi). Mostrano anche gli attributi e le operazioni.

Ogni associazione ha una **molteplicità** ovvero indica quanti oggetti possono prendere parte a un'associazione. Hanno inoltre una **navigabilità** ovvero indicano l'appartenenza ad oggetti di un'altra classe.

Aggregazione e composizione

L'**aggregazione** p la relazione “parte di” mentre la **composizione** è simile ma si aspetta che l'oggetto in causa abbia lo stesso ciclo vitale di tutti gli oggetti da cui è composto. È possibile che più aggregazioni condividano oggetti in comune.

Classi parametriche

È il cosiddetto **template** nei linguaggi fortemente tipizzati. Ed è utile per lavorare con collezioni di tipi diversi ma che devono svolgere uno stesso compito. Vengono indicati come rettangoli tratteggiati posti vicino al nome di una classe.

Diagramma delle istanze

Rappresentano gli oggetti in un determinato istante ma non mostrano le classi.

Ridurre le dipendenze fra i packages

Bisogna dichiarare classi private per le funzionalità interne al package mentre si usano classi pubbliche per la sua comunicazione con l'esterno.

Comportamento dinamico

Diagrammi di collaborazione

Estensione del diagramma delle istanze e mostra come interagiscono tra loro gli oggetti. I messaggi fra gli oggetti sono rappresentati tramite frecce che puntano verso l'altro oggetto e contenenti il nome del messaggio.

Il diagramma delle collaborazioni e il diagramma delle sequenze sono abbastanza simili. La principale distinzione fra i due diagrammi è che data dal fatto che quello delle sequenze segue come costante il tempo, mentre quello delle collaborazioni segue lo spazio come costante.

Diagrammi di sequenza

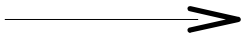
La linea tratteggiata indica la linea di vita di ciascun oggetto e il messaggio è indicato da una freccia tra le linee tratteggiate.

Oggetti sincroni

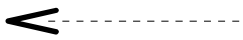
Prima di continuare con qualsiasi altra operazione bisogna attendere un messaggio di risposta.

Sono messi in relazione con la seguente ferccia: 

Oggetti asincroni

Non è necessaria una risposta e usano la seguente freccia: 

Ritorno

Indica il ritorno di una operazione sincrona: 

Si può anche mostrare la chiamata interna di un oggetto ad una sua stessa funzione (ricorsione).

Un messaggio può avere una condizione e un indicatore di **iterazione** che significa che viene inviato più volte verso diversi oggetti!

Diagrammi di stato

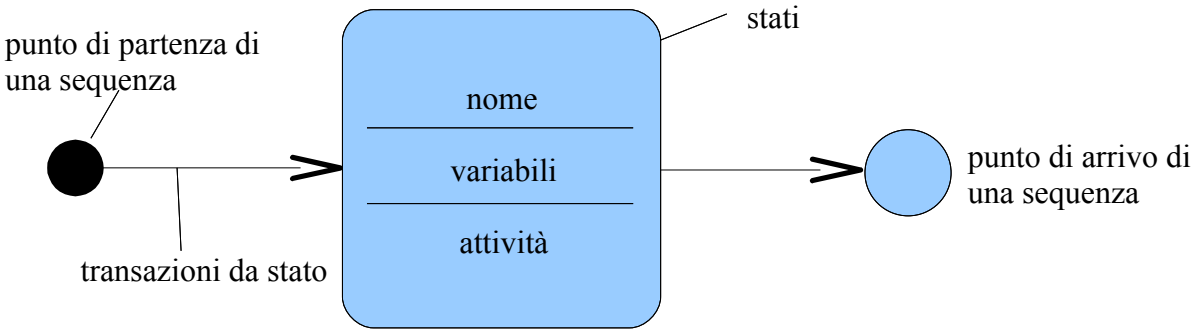
Descrivono l'evoluzione del sistema e sono costituiti da una serie di stati che descrivono delle attività tra le quali ci si muove attraverso le **azioni** associate ai cambiamenti di stato mentre le attività sono associate a stati.

Un evento **trigger** è un evento che causa il verificarsi di una transazione ma non si parla di trigger quando una transazione si verifica perché uno stato completa la sua attività.

Sintassi per una transazione: <Evento> [<Conduzione>] | <Azione>

Sintassi per una attività: do | <Nome Attività>

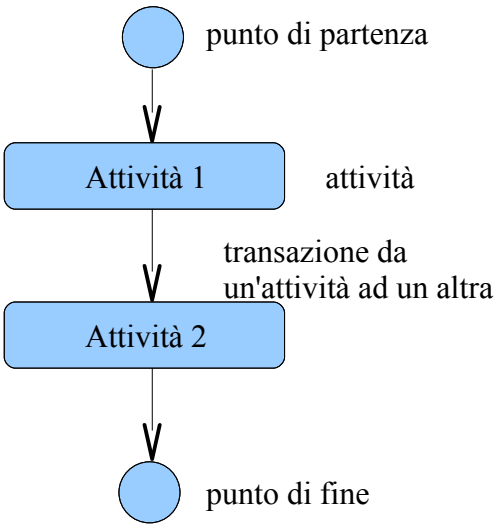
Il **segnale** è un messaggio che causa la transazione e di questo è possibile generare delle gerarchie poiché viene anch'esso visto come un oggetto.



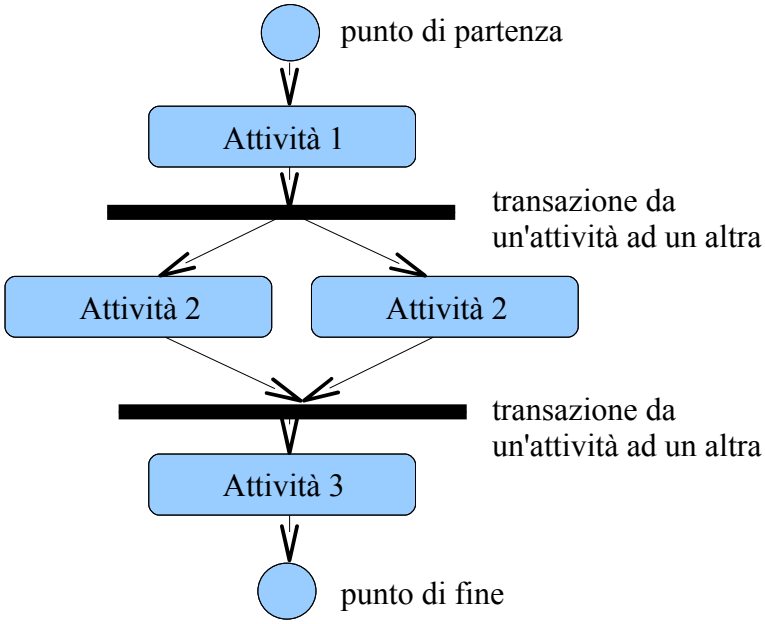
Diagrammi di attività

Descrivono la sequenza delle attività e supportano il comportamento sia **condizionale** che **parallelo**.

Utilizza due costruttori: **branch** (diramazione) e **merge** (unisce).



Si può anche stabilire che siano eseguite due attività **concorrenti** delle quali l'inizio e la fine viene indicato con delle linee in grassetto.

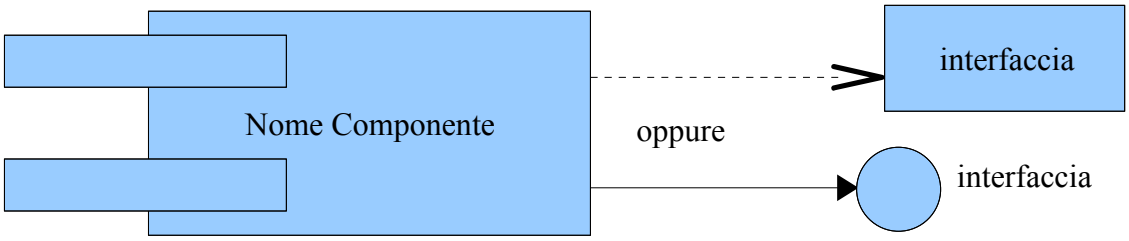


Diagrammi di deployment

Mostrano le relazioni fisiche fra i componenti software e hardware. I **nodi** indicano un'unità computazionale mentre le connessioni mostrano i canali di comunicazione.

Diagrammi dei componenti

Mostrano i vari componenti di un sistema e le loro dipendenze. Un componente è un modulo software mentre le dipendenze mostrano come i combiamenti si ripercuotono sugli altri e sono la **compilazione** e la **comunicazione**.



Diagrammi combinati

Sono l'integrazione fra i diagrammi delle componenti e i diagrammi di deplyment. È possibile mostrare quali componenti si trovano in esecuzione in ogni nodo del diagramma di deployment.

Verifica e validazione

Il collaudo è l'unico passo del processo software che si può considerare distruttivo.

La verifica e la validazione hanno il compito di scoprire i difetti presenti nel sistema (verifica) e poi verificare che il sistema sia utilizzabile secondo le specifiche richieste (validazione).

La **verifica** può essere **statica** (utilizzata in ogni fase del processo di sviluppo) oppure **dinamica** (riguarda la fase di test con la quale si controlla il comportamento dinamico del sistema e richiede l'esistenza di un prototipo).

La **validazione** può essere solamente di tipo dinamico.

Ogni singola prova deve essere riconducibile ai requisiti del cliente, i collaudi devono essere pianificati con anticipo, inizia con il collaudare le singole componendi per poi estendere il collaudo all'intero sistema. Il collaudo non dovrebbe essere effettuato da chi ha scritto il codice.

La **collaudabilità** è la facilità con la quale un programma può essere testato. Migliore è il funzionamento di un programma e più semplice sarà il suo collaudo (operabilità).

Ciò che si collauda è ciò che è visibile (osservabilità).

Quanto più è possibile controllare il software, tanto più il collaudo è ottimizzato (controllalbilità) è inoltre necessario poter collaudare i moduli separatamente (scomponibilità).

Meno sono le cose da collaudare e più velocemente si svolgono i collaudi (semplicità), minori sono le modifiche e minore è il lavoro di collaudo (stabilità). Maggiori sono le informazioni e più adeguati sono i collaudi (comprensibilità).

Tipi di collaudo

Collaudo white-box

È necessaria la conoscenza della struttura interna e mira a controllare se le strutture interne funzionano secondo le specifiche progettuali.

Collaudo black-box

Si parte dalle specifiche del progetto e verifica che tutte le funzionalità richieste siano presenti. Viene effettuato nlla fase finale dell'implementazione.

Collaudo per condizioni

Si concentra sui test di correttezza delle condizioni logiche del programma (Semplici, composte e booleane).

Collaudo per cicli

Si concentra su prove che verifichino la validità dei costrutti di ciclo (cicli semplici, cicli annidati – entrambi partono dal ciclo più interno).

Tsting

Confermare la presenza di errori.

Debugging

Localizzare e correggere gli errori.

Il collaudo finisce quando la probabilità del verificarsi di un guasto software per unità do tempo scende sotto una certa soglia preventivamente prefissata.

Complessità ciclotomica

È una misura quantitativa della complessità logica di un programme e misura il numero di cammini indipendenti di un grafo.

È calcolabile in tre modi:

1. V(G)=R con R corrispondente al numero di regioni.
2. V(G)= E-N+2 con E=numero di lati e N=numero di nodi
3. V(G)=P+1 con P=numero delle diramazioni

Strategie di testing

- **Top-down:** dalle componenti più astratte verso quelle a più baso livello. Può individuare problemi archietetturali.
- **Bottom-up:** dalle componenti più a basso livello a quelle più astratte. È appropriato per OOP.
- **Collaudo per regressione:** testa le modifiche del software.
- **Stress testing:** utilizzato per vedere come reagisce il sistema quando è in sovraccarico.
- **Back-to-back testing:** quando sono disponibili più versioni diverse. I problemi vengono segnalati da differenti output.
- **Smoke-test:** per tempi di consegna brevi si inserisce il collaudo direttamente nella fase di sviluppo.

Prove di validazione

Si fa usare il prodotto ad un insieme ristretto di utenti.

Collaudo Alfa

Gli utenti usano il prodotto sotto il controllo degli sviluppatori.

Collaudo Beta

Gli utenti usano il software per conto loro e creano dei report sull'andamento del software.

Debugging

I metodi principali di debugging sono:

- **dump della memoria**
- **si parte dal punto in cui si presenta l'errore e si procede a ritroso**
- **eliminazione delle cause**

Affidabilità

L'affidabilità è definita come probabilità che il sistema funzioni senza errori per un determinato intervallo di tempo in un certo ambiente per un determinato scopo. Utenti diversi percepiscono una diversa affidabilità del sistema.

Questo parametro può essere difficilmente definito in modo oggettivo.

Un **fallimento** corrisponde ad un comportamento **run-time** inaspettato mentre il **fault** è una caratteristica statica del software che causa fallimento solamente quando viene visualizzata la componente errata.

L'**affidabilità** migliora quando si rimuovono i fault nelle parti più utilizzate.

L'uso di metodi formali per lo sviluppo del sistema può portare ad un sistema maggiormente affidabile perché dimostrano che esso si comporta conformemente ai requisiti in quanto lo sviluppo di una specifica formale obbliga un'analisi dettagliata dei requisiti.

All'aumentare dell'affidabilità l'efficienza diminuisce perché per rendere affidabile il sistema potrebbe essere necessario inserire codice ridondante.

Metriche

POFOD (probability of faliture on demand)

È una metrica adottata per i sistemi critici o non-stop e indica la probabilità che ci sia fallimento quando viene fatta una richiesta al sistema. (Numero).

ROCOF (rate of fault occurrence)

È il tasso di occorrenza e indica la frequenza con cui si verificano comportamenti anomali. È rilevante per i sistemi operativi. (Misura tempo).

MTTF (mean time to failure)

Misura il tempo trascorso tra due fallimenti ed è utilizzato per sistemi in cui le transazionei possono durare a lungo. (Tempo).

AVAIL (availability)

Misura la probabilità che il sistema sia operativo in un certo istante. È utile per sistemi che devono lavorare in continuazione. (Tempo di ripristino).

Requisiti di affidabilità

Per verificare i requisiti di affidabilità occorre definire un **profilo operazionale** ovvero definire in che modo il sistema verrà utilizzato.

L'affidabilità è dinamica.

Classificazione dei fallimenti

I fallimenti possono essere classificati in:

- **Transienti:** si verificano soltanto con certi input.
- **Permanenti:** si verificano per tutti gli input.
- **Recuperabili:** il sistema riparte automaticamente.
- **Non recuperabili:** è necessario l'intervento dell'operatore.
- **Non corruttivi:** il fallimento non provoca la corruzione dei dati del sistema.
- **Corruttivi:** il fallimento corrompe il sistema.

Creare specifiche di affidabilità

Analizzare le conseguenze di possibili fallimenti per tutti i sotto-sistemi.

Bisogna **partizionare** i fallimenti nelle classi appropriate e definirli quantitativamente tramite le metriche opportune.

Ottenere una buona affidabilità è molto costoso quindi spesso è molto più conveniente tollerare possibili falimenti.

Test statici

1. **Determinare il profilo operazionale del sistema** (tipi di input probabili)
2. **Generare i dati di test in base al profilo**
3. **Applicare i test e misurare i tempi fra i fallimenti**
4. **Valutare l'affidabilità**

Il primo punto dovrebbe essere generato automaticamente per quanto possibile in quanto richiede dei costi elevati e possibili incertezze nella generazione

Modelli di crescita dell'affidabilità

Sono **modelli matematici** che mostrano i cambiamenti del livello di affidabilità man mano che i bug vengono risolti.

Ottenere l'affidabilità

Fault Avoidance

Il sistema viene sviluppato in modo da non contenere errori cioè quando il software è conforme alle specifiche.

Fault Detection

Gli errori sono corretti e individuati prima di consegnare il sistema al cliente.

Fault Tolerance

Eventuali errori non causano necessariamente un fallimento totale del sistema.

Sviluppare software privo di errori

1. **Partire da una specifica formale.**
2. **Utilizzare information hiding ed incapsulamento.**
3. **Utilizzare linguaggi di programmazione fortemente tipizzati.**
4. **Effettuare revisioni periodiche.**
5. **Test e collaudi accurati.**

Tecniche

Progettazione strutturata

Programmare senza *goto* e *break*. Utilizzare solamente *do-while* o *repeat until* e applicare un approccio top-down in modo da spezzare il codice in procedure.

Possono esserci problemi generati dall'uso di numeri in virgola mobile, puntatori, ricorsione e interrupts.

Information hiding

Le informazioni devono essere rese disponibili solo alle parti del programma che ne hanno bisogno.

ADT

Le operazioni sull'ADT sono rappresentate da metodi e la rappresentazione dell'ADT è contnute nella parte protetta | privata.

Come tollerare i fallimenti?

Si deve come prima cosa rilevare il guasto e valutarne entità e danni causati. Bisogna riportare il sistema ad uno stato valido ripristinando il guasto. **La maggior parte dei guasti è dovuta ai dati di input.**

Fallimenti dell'hardware

Ci sono tre componenti identiche che ricevono gli stessi input e devono quindi generare gli stessi output. Se uno è diverso dagli altri due viene ignorato e la componente viene considerata guasta.

Analogie software

N-Version programming

Implementa le stesse specifiche in modo diverso e gli input sono passati contemporaneamente. L'output è selezionato in basa a ciò che calcola la maggioranza. Le diverse versioni devnon essere sviluppate e implementate da team differenti.

Recovery Blocks

Esistono diverse versioni di software che vengono provate in sequenza finché non se ne trova una che fornisce la risposta corretta.

Occorre implementare algoritmi diversi fra le versioni.

Eccezioni

Sono errori causati da eventi inattesi e per gestirli esistono meccanismi che evitano vontrolli continui: *try { ... }* | *catch*.

Programmazione difensiva

Approccio dello sviluppo di software in cui si parte dall'osservazione che possono esistere errori non individuati e fornire al programma del codice per recuperare tali errori.

Recuperare da un fallimento

Forward recovery

Riparare lo stato corrotto riportandolo verso uno stato valido.

Richiede conoscenze specifiche sugli stati.

Backward recovery

Ripristina lo stato ad uno stato precedente. È molto più semplice da implementare.

Sistemi in serie e sistemi in parallelo

I **sistemi in serie** funzionano se tutti i sistemi funzionano.

Nei **sistemi in parallelo** il sistema funziona se almeno una delle componenti funziona.

Gestione del personale

Una scarsa gestione del personale è un contributo al fallimento di un progetto.

È importante che ci sia **consistenza** cioè trattare i membri in modo comparabile, **rispetto** ovvero membri diversi hanno capacità differenti e ciò deve essere rispettato, **inclusione** ovvero rendere partecipi tutti i membri del team e **onestà**.

Un ruolo molto importante del manager è quello di motivare le persone (bisogni sociali, stima e auto-realizzazione). Il livello di motivazione è legato anche al tipo di personalità:

- **Task oriented:** la motivazione è il lavoro di per sé.
- **Self oriented:** il lavoro è un mezzo per raggiungere un obiettivo.
- **Interaction oriented:** la motivazione è data dalla presenza dei colleghi di lavoro.

Il gruppo deve essere composto da persone che non abbiano le stesse motivazioni ma tutti devono avere almeno una fra le motivazioni precedentemente indicate. (Composizione).

La **leadership** dipende solamente dal rispetto e non dai titoli e dovrebbe esistere sia un leader tecnico che uno amministrativo.

La **coesione** del gruppo fa si che sia più importate il gruppo che non l'individualità della persone che ne fanno parte ma la coesione è influenzata da fattori come la cultura organizzativa e la personalità.

La **comunicazione** è indispensabile per un buon lavoro di gruppo e un buon livello di comunicazione migliora la coesione.

Ciò dipende comunque dall dimensioni, dalla struttura e dalla composizione e dall'ambiente di lavoro.

L'**organizzazione** dipende dalle dimensioni del progetto infatti per progetti piccoli si possono creare **gruppi informali** dove il gruppo agisce come un tutt'uno, il leader ha il compito di interfacciarsi con l'esterno ma non assegna compiti. Questa cosa è utile dove le persone hanno pressoché le stesse competenze.

I gruppi **XP** (extreme programmi) prevedono alcune decisioni di management e i programmatori

lavorano in coppia.

Nel **Chief Programmer Teams** si ha un gruppo di specialisti assistiti da altri e permette quindi differenze d competenze. È però difficile trovare specialisti che ricoprono il ruolo di chief.

L'**organizzazione dell'area di lavoro** è fondamentale. È necessario che ognuno abbia un posto adeguato dove lavorare senza essere disturbato.

Domande e Esercizi

Descrivere il modello di sviluppo a **cascata,** illustrandone la struttura e descrivendo sinteticamente le attività associate alle varie fasi di cui si compone. Indicare inoltre le caratteristiche di **visibilità** di questo modello di sviluppo.

Risoluzione:

Il modello a cascata è di tipo sequenziale o lineare. Come presupposto si prevede di avere la piena comprensione delle caratteristiche che il software dovrà implementare.
È diviso in varie fasi:

- analisi dei requisiti
- disegno del sistema
- implementazione e test dei sottoinsiemi
- integrazione e test del sistema
- operatività e mantenimento

La visibilità è ampia nel senso che tutte le caratteristiche vengono definite all'inizio e suddivise in tappe da seguire linearmente rispettando un certo numero di scadenze.
L'unica limitazione che prevede il modello è quella di non essere molto elastico e quindi di non poter facilmente operare modifiche al modello quando questo è in corso.

Descrivere cosa sono le tecniche di **fault avoidance, fault detection** e **fault tolerance**, utilizzabili per produrre sistemi software affidabili.
In quali modi è possibile realizzare software affidabile, evitando sin dall'inizio introduzione di errori nel sistema? Dare una breve spiegazione.

Risoluzione:

Sono tecniche che si usano per rendere affidabile un software basate sull'individuazione degli errori.

Fault avoidance: Il sistema viene sviluppato in modo da non contenere errori cioè quando il software è conforme alle specifiche.
Fault detection: Gli errori sono corretti e individuati prima di consegnare il sistema al cliente.
Fault tolerance: Eventuali errori non causano necessariamente un fallimento totale del sistema.

Per evitare del tutto l'introduzione di errori nel sistema è necessario correggere questi direttamente durante la fase di scrittura del software, facendo un continuo e peculiare debugging sui prototipi prodotti.

Descrivere le strategie di testing **Top-Down** e B**ottom-Up**, indicandone i vantaggi e i limiti.

Risoluzione:

Con **Top-Down** ovvero dall'alto verso il basso i moduli a più alto livello sono i primi ad essere esposti a test. In questo modo si testa la logica ad alto livello e il flusso dei dati. Il test risulta piuttosto ostico in caso di problemi perché le componenti a basso livello vengono testate per ultime. Con **Botom-Up** l'approccio è l'inverso: si parte dal testare le componenti a più basso livello fino ad arrivare a quelle a più alto livello. Questo tipo di test ha come solo svantaggio quello di ritardare i test logici di alto livello poiché molto spesso nascono problemi con drivers e conflitti a basso livello. Questa tecnica è consigliata per gli OOP.

Commentate la seguente affermazione: “se un progetto software è in ritardo, possiamo sempre aggiungere del personale per accorciare i tempi”. Secondo voi è vera? Perché? Motivare le risposte e fornire un esempio.

Risoluzione:

L'affermazione è in parte vera nel senso che un maggior numero di personale deve poter essere finanziato economicamente (bisogna considerare se sono maggiori i costi nel fornire il software in ritardo rispetto a quelli del costo di personale oppure se non ci si può assolutamente permettere di aggiungerlo). In secondo luogo dipende anche dalla qualità del personale: se il personale non è estremamente competente con quello che è il suo compito ci vorrà del tempo per spiegare cosa deve fare.

Che cosa si intende con i termini **verifica** e **validazione** di sistemi software? Che cosa cambia il fatto di effettuarla in modo *statico* o *dinamico*?

Risoluzione:

La **verifica** è un processo di ricerca di errori nei sistemi software, può essere di due tipi:

- **statica:** sviluppata durante la fase di sviluppo (viene eseguita direttamente sulle componenti e non richiede l'esecuzione di codice)
- **dinamica:** sviluppata in fasi non contigue e richiede l'esistenza di un prototipo (può essere eseguita anche sul sistema oltre che sulle componenti e prevede l'esecuzione di codice)

La **validazione** è il collaudo che controlla che siano rispettate le specifiche richieste dal cliente e per maggior sicurezza è bene far fare questo tipo di collaudo a qualcuno che non abbia scritto il software.

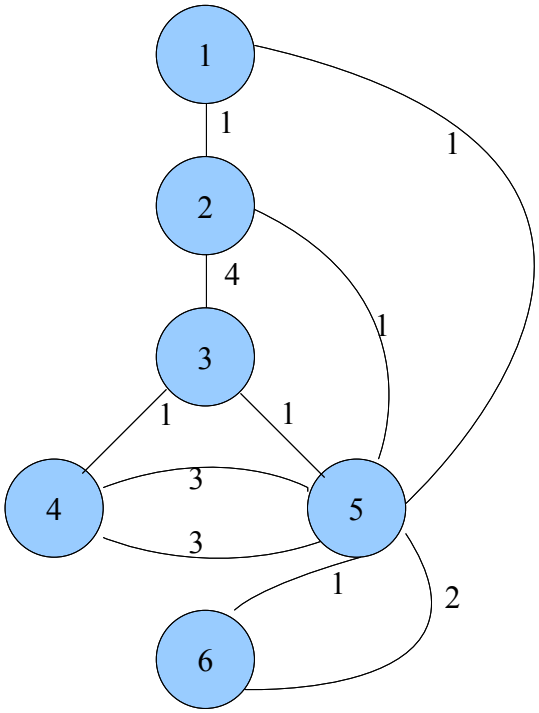
Si consideri il seguente frammento di codice:

```
/*
 * Parametri di input:
 *       vettore v[] ordinato,
 *       dimensione dim del vettore (1<= dim <=10),
 *       valore val da cercare
 * Risultati:
 *       i tale che v[i]==val; -1 se val non è presente nel vettore
 */
INT v[1 .. 10];     /* parametro di input */
INT dim;            /* parametro di input */
INT val;            /* parametro di input */
INT i;              /* parametro di input */
INT a,b;
BOOL f;
a:=1; b:=dim; f:=false;
WHILE ((a<=b) AND (NOT f)) DO BEGIN
    i:=(a+b)/2;
    IF (v[i]==val)
         f:=true;
    ELSE BEGIN
         IF(v[i]<val)
              b:=i-1;
         ELSE
              a:=i+1;
         END;
    END;
IF (NOT f)
    i:=-1;
```

a) Si consideri che cos'è e a che cosa serve la complessità ciclotomatica di un programma.
b) Calcolare la complessità ciclotomica del programma, indicando anche quali sono i cammini indipendenti del grafo di flusso.

Risoluzione:
La complessità ciclotomica è la misura quantitativa della complessità logica di un programma.
Misura il numero di cammini indipendenti in un grafo.
Uno schema per rappresentare il programma descritto sopra può essere il seguente (le condizioni sono rappresentate da dei tondi e sulle relazioni c'è scritta la complessità delle operazioni che sono

necessarie fino al raggiungimento della condizione successiva.



I cammini indipendenti sono i seguenti:

- 1-5-6
- 1-2-5-6
- 1-2-3-5-6
- 1-2-3-4-5-6

Quello con complessità minore è il primo che ha costo minore è 1-5-6 di costo 2 mentre quello con costo maggiore è 1-2-3-4-5-6 con costo 11.

Commentare le seguenti affermazioni relative all'ingegneria del Software. Secondo voi sono vere? Perché? Discutete e giustificate le risposte:

1. “Se un progetto software è in ritardo, si può sempre recuperare aggiungendo personale al team di sviluppo”.
2. “Un'affermazione generica di cosa deve fare un programma è sufficiente per iniziarne a scrivere il codice”.
3. “Una volta messo in opera il programma, il nostro lavoro è finito”.

Risoluzione:
La prima affermazione è vera se sono disponibili somme di denaro e persone per mettere in piedi il team, inoltre è necessario che il team sia sufficientemente qualificato.

La seconda affermazione è completamente sbagliata a meno che non si tratti di un tipo di software molto comune o di chiarissima intuizione. Fidarsi di quello che si pensa sia giusto non si rivela produttivo nella maggior parte dei casi.

Anche la terza affermazione non è del tutto corretta! Di solito un programma operativo non è detto che con il tempo non possa dare alcune noie, in più è molto spesso richiesto di aggiungere nuove funzionalità!

Discutere quali sono i fattori che possono influenzare l'affidabilità di un sistema informatico e in che modo tali fattori lo influenzano.

Risoluzione:

Ricordando che l'affidabilità corrisponde alla percentuale che ha un sistema di essere attivo prefissato un obiettivo temporale, il funzionamento e l'affidabilità è influenzato da fattori distinguibili in tre diverse categorie:

1. software
2. hardware
3. personale

Riguardo al software ciò che influenza l'affidabilità è la probabilità che si verifichi un errore che può essere dovuto a un comportamento inaspettato, una divisione per zero, la mancata gestione delle eccezioni ecc.
Per quanto riguarda l'hardware influenzano i guasti o anche la cattiva gestione o bilanciamento di un sovraccarico.
Riguardo all'errore umano invece si deve prevedere la possibilità di guasti accidentali (ad esempio l'interruzione della corrente perché si è inciampati in un cavo) oppure all'uso improprio del sistema.
Fare in modo di prevedere e gestire tutte queste casistiche aumenta notevolmente l'affidabilità del sistema.

In quali informazioni il sistema di sviluppo a cascata è maggiormente indicato e invece il quali NON è indicato?

Risoluzione:

Il modello a cascata è indicato quando il software da produrre non subirà contrattazioni durante la fase di sviluppo (è molto difficile rimettere mano al modello) e anche quando si ha un'idea precisa sui tempi e sui compiti che dovrà svolgere il software. È adatto anche quando viene richiesto di rispettare tempi di consegna ben precisi.
Questa metodologia di sviluppo non è quindi adatta quando non si hanno idee chiare su come sarà il prodotto finito, se si lavora a diretto contatto con il cliente (il quale vuole sviluppare il software secondo le sue direttive) oppure quando si prevede di dover effettuare numerosi prototipi da mostrare al cliente prima di sceglierne uno definitivo.
La costante prevista da questo modello di sviluppo, basato su previsione, sono le *milestones* ovvero pietre miliari (per cui il quantitativo di software prodotto e il tempo sono costanti da rispettare).

Descrivere in cosa consiste l'architettura software “a flusso dei dati” (*pipe and filter*) e quella basata su *repository* centrale.
Illustrare con un esempio un sistema software che possa essere descritto mediante una di tali architetture.

Risoluzione:

L'architettura *pipe and filter* è progettata per soddisfare un sistema software a controllo lineare. Il software è suddiviso in moduli indipendenti che non condividono uno stato comune. Vengono quindi create delle pipe che richiedono solamente che l'output di un modulo sia coerente con l'input dell'altro modulo agganciato tramite pipe.
La progettazione di un tipo di software come questo risulta piuttosto semplice, è molto scalabile e prevede un forte riutilizzo dei moduli e del codice (orientata alla OOP).

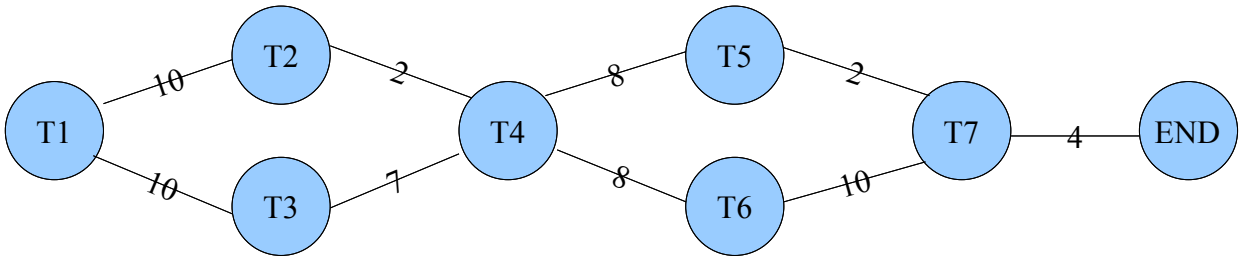
L'architettura su repository centrale, invece, prevede appunto una banca dati comune (repository) a cui tutti i clients accedono per prelevare informazioni. La repository può essere passiva (i clients eseguono cambiamenti in modo indipendente) o attiva (notifica i cambiamenti ai clients).
Questa architettura è indicata se si devono condividere grandi quantità di dati, e se i dati devono essere condivisi.
Non è invece adatta a modificare le politiche di accesso ai dati, non è scalabile, è difficile modificare la struttura dei dati e privatizzare le informazioni.

La tabella seguente mostra le durate e le dipendenze dei task che è necessario completare per realizzare un certo software:

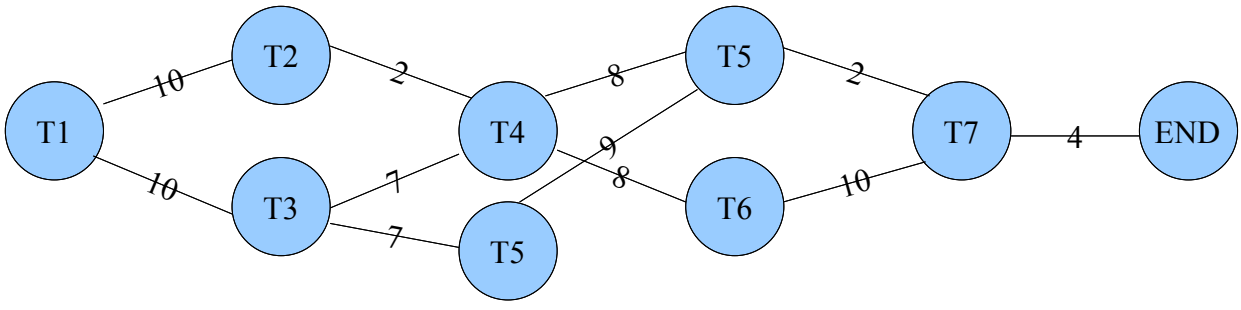
Task	Durata in gg	Dipende da
T1	10	
T2	2	T1
T3	7	T1
T4	8	T2, T3
T5	2	T4
T6	10	T4
T7	4	T5, T6

1. Determinare il cammino critico (cirtical path) del grafo delle dipendenze dato.
2. Supponiamo di voler accorciare la durata del cammino critico. Supponiamo che per fare questo sia possibile ridurre la durata di un solo task; si supponga inoltre che il task, una volta "accordato" non possa avere durata nulla. Si richiede di individuare quali task possono essere accorciati in modo da ridurre la lunghezza del cammino critico di 4 giorni.
3. Riconsiderare la tabella dei task e delle durate di partenza. Supponiamo di aver individuato un nuovo task T8, di durata 9 giorni, che dipende da T3 e supponiamo inoltre che il task T5 dipenda anche da T8, Determinare il cammino critico di questa nuova situazione.

Risoluzione:



1. Il percorso critico è dato da: T1 – T3 – T4 – T6 – T7.
2. I task che si possono accorciare sono soltanto T1 e T4 poiché sono gli unici in comune con tutti i percorsi possibili e che non si annullano (T7 anche se comune ha una durata di soli 4 giorni e quindi non si può annullare). Si possono accorciare anche i task T3 e T6 poiché hanno una durata superiore ai 4 giorni ma sarebbe meno produttivo poiché T1 e T4 sono task obbligati.
3.



In questo caso il percorso critico rimane sempre uguale poiché T4 e T8 nella peggiore delle ipotesi vengono avviati allo stesso momento, T5 deve aspettare al massimo 9 girni per partire per cui T8 allunga di un solo giorno il percorso attraverso T5 nel caso in cui i tasks possano essere portati avanti in parallelo. Se invece devono essere compiuti in modo sequenziale allora il passaggio per T8 e T5 determineranno il nuovo cammino critico!

Si descriva lo sviluppo di un sistema software i cui requisiti non sono ben compresi. Quale modello di sviluppo sceglierestе? Motivare la risposta e illustrare le caratteristiche e il funzionamento del modello scelto.

Risoluzione:

Non c'è un solo modello di sviluppo ma ce ne sono almeno due: il primo è il Throw-away poiché si concentra sulle parti del software ben comprese per poi passare a fare un prototipo usa e getta da mostrare al cliente. Questa metodologia è molto rischiosa per cui si adatta solo a progetti di piccole dimensioni.
La seconda metodologia è l'XP progrmming. Questa evita perdite di tempo e si sviluppa a diretto contatto con il cliente. Prevede molti piccoli step che devono essere visionati dal cliente permettendo ad entrambe le parti di convergere pian piano verso al risultato finale desiderato. In questa seconda tecnica non è necessariamente vincolato lo sviluppo con il cliente sulle parti che sono chiare.

Descrivere il concetto di *proprietà emergente* di un sistema. Dare alcuni esempi di proprietà emergenti di un sistema software.

Risoluzione:

Poiché un sistema è caratterizzato da uno svariato numero di componenti, le proprietà emergenti derivano dalla loro interazione. Sono:

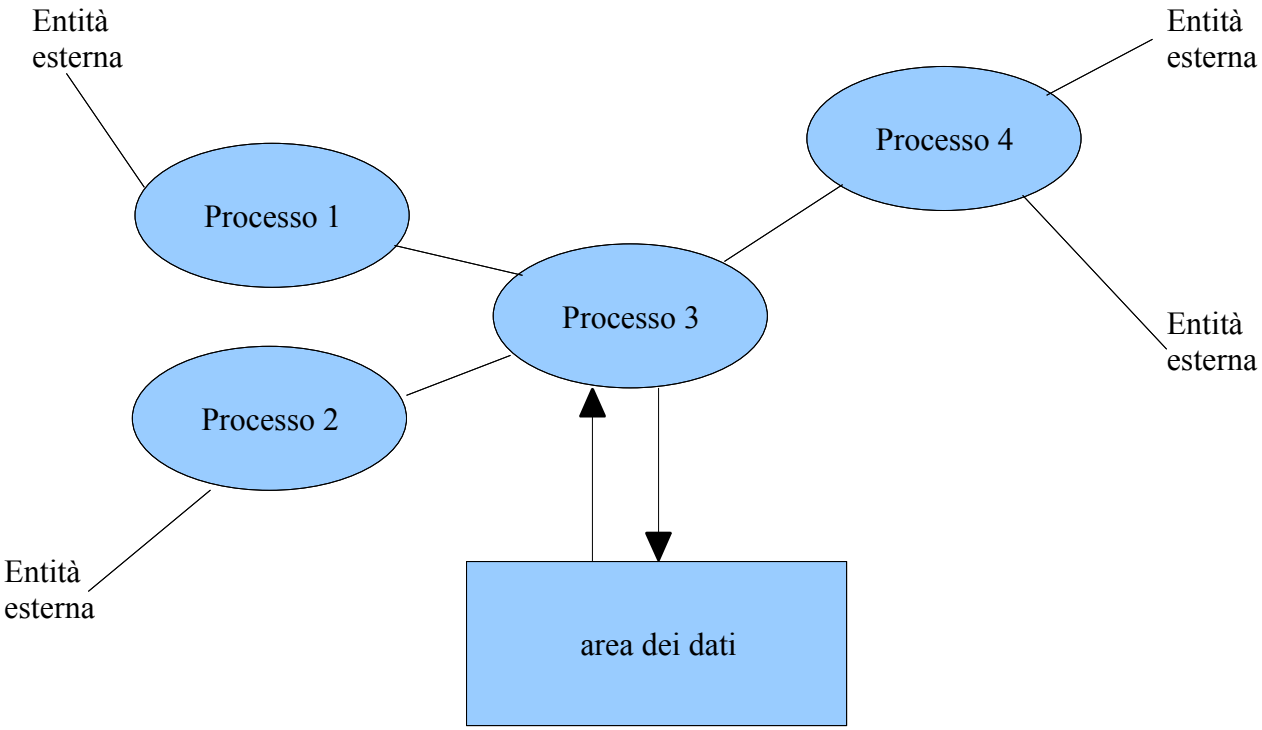
- Peso globale del sistema
- Affidabilità
- Usabilità
- Riparabilità

Queste proprietà si dividono in funzionali e non funzionali. Un errore si propaga nel sistema a seconda delle connessioni che ci sono le componenti.
Le proprietà emergenti possono esser valutate solamente una volta che il sistema è stato assemblato.

Che cos'è e come viene descritto un modello *data-flow* (a flusso di dati) di un sistema software? Dare un esempio di un semplice sistema descritto tramite un modello data-flow.

Risoluzione:

Un data flow serve per mostrare la struttura dei dati e come essi fluiscono nel sistema. È un modello sviluppato come parte intrinseca di molti modelli di analisi. È basato su una notazione semplice che deve essere comprensibile a tutti ed indicano come i dati vengono processati durante la computazione.
Nel semplice diagramma che segue viene indicato come sono organizzate le unità computazionali del sistema e come sono collegate ai generatori e ai ricevitori.



Descrivere il modello organizzativo di un team di sviluppo basato su *chief programmer* (ossia sulla metafora chirurgo copilota). Quali sono i vantaggi e i limiti di una simile organizzazione?

Risoluzione:

Questo gruppo di lavoro è strutturato in modo che ci siano due specialisti (il chirurgo e il copilota) che vengono assistiti da altri programmatori. Questa struttura differenzia le competenze fra i vari programmatori in modo che ognuno sia specialista in qualcosa. La composizione resta comunque gerarchica: programmatori di contorno forniscono supporto ad un programmatore estremamente abile che funge anche da responsabile della lavorazione.

Quando si presenta un gruppo di lavoro di questo tipo, di solito l'elaborato presenta notevole successo ma comunque i problemi non mancano:

- È difficile trovare un programmatore di estremo talento.
- Gli altri membri del team potrebbero non accettare il fatto che ci sia un programmatore leader che si prenda tutto il merito del progetto.
- Se la figura del chief programmer o del copilota vengono a mancare il progetto salta (è incentrato su 2 figure).
- La struttura della compagnia nella quale si lavora potrebbe non essere adatta a questo tipo di team.

Supponete di dover motivare i componenti di lavoro di cui siete i responsabili. In che modo procedete? Quali caratteristiche individuali delle persone tenete in considerazione?

Risoluzione:

Per motivare i componenti del progetto si potrebbe far leva su motivazione a carattere economico, di soddisfazione personale come anche di competizione e affermazione all'interno dell'azienda oppure anche l'obiettivo del progetto in sé.
Bisogna sempre tener conto del bisogno di realizzazione che ha una qualsiasi persona in modo che non perda la sua autostima.
Le caratteristiche personali da tenere in considerazione sono almeno 3 classificazioni:

1. **Task oriented:** la motivazione è il lavoro di per sé.
2. **Self oriented:** il lavoro è un mezzo per raggiungere un obbiettivo individuale (arricchirsi o aumentare il proprio tenore di vita).
3. **Interaction oriented:** l'interazione con i colleghi (alcune persone lavorano perché si divertono e sono contente dell'ambiente in cui si trovano).

Che differenza c'è fra l'affidabilità del software e l'affidabilità dell'hardware? In cosa differiscono le rispettive curve di affidabilità? Motivare la risposta.

Risoluzione:

I fallimenti dell'hardware a differenza di quelli dovuti al software sono dovuti all'usura e al danneggiamento (cose alle quali il software è immune). In caso di guasto le componenti danneggiate possono essere sostituite e ripristinare così il sistema.
I fallimenti del software sono dovuti al modo in cui il software è stato disegnato. È possibile prevenire questi fallimenti gestendo le anomalie ma comunque sia ci sono sempre delle ripercussioni più o meno rilevanti nel sistema.

Si consideri un sistema di frenaggio ABS di un'automobile. Per migliorare la sicurezza del dispositivo, i progettisti hanno deciso di includere te centraline hardware di controllo dell'impianto ABS.

Ciascuna centralina è controllata da un apposito modulo software. Il sistema funziona purché almeno una centralina con il relativo modulo software siano operativi. Si assuma che l'affidabilità di una centralina sia *Rc* e quella di un modulo software sia *Rs.*

Calcolare l'espressione dell'affidabilità del sistema nelle seguenti ipotesi:

a) Il software di controllo di ciascuna centralina è esattamente lo stesso. Quindi quando un modulo fallisce tutti gli atri falliscono contemporaneamente.

b) Il software di controllo di ciascuna centralina è stato implementato indipendentemente fagli altri da parte di team diversi. I software di controllo implementano esattamente gli stessi requisiti funzionali e non funzionali. Quindi i moduli di controllo falliscono indipendentemente gli uni dagli altri.. Si assuma che l'affidabilità di ciascuno dei tre moduli sia di *Rs*.

Risoluzione: