Ingegneria del Software – Riassunto

SLIDE 2

Software: Insieme programmi, documentazione e procedure operative mediante le quali gli elaboratori possono svolgere le funzioni richieste.
Ingegneria del software: Applicazione pratica dei metodi scientifici per la specifica, il disegno e l'implementazione di programmi. Si occupa degli aspetti pratici dello sviluppo e rilascio di software utilizzabile. Si limita allo sviluppo dell'infrastruttura software.
Computer science: si occupa di teoria e conoscenze di base.
System engineering: si occupa di tutti gli aspetti del calcolatori (anche hardware, cosa che non fa l'ingegneria del software).

Processo produttivo del software: Insieme di attività il cui scopo è lo sviluppo (o l'evoluzione) del software.
Si divide in:
Specifica: Cosa dovrebbe fare e vincoli di sistema.
Sviluppo: Produzione del software.
Validazione: Controllo che il software faccia quello che il cliente chiede
Evoluzione: Cambiamenti del software in caso di nuove/diverse esigenze.

Modello di processo produttivo di software: Rappresentazione semplificata del processo di sviluppo, presentata da una prospettiva specifica, x es:
Work Flow prospective → Sequenza di attività
Data Flow prospective → flusso di informazioni
Role Action prospective → Chi fa cosa.
I più conosciuti sono **Waterfall**, **Iterative Development**, **Component Based** software engineering.

Costi ingegneria del software: 60% sviluppo, 40% validazione. Se il software è su misura spesso i costi di validazione superano quelli di sviluppo. La distribuzione dei costi dipende dal modello utilizzato.
Costi Waterfall: 15% Specifica 20% Design 15% Development (Sviluppo) 45% Integrazione e testing.
Costi Iterative Development: 10% Specifica, 60% Iterative Development, 30% System Testing.
Costi Component Based: 20% Specifica, 40% Development(Sviluppo), 40% Integrazione e testing.

CASE: Computer aided software engineering. Sono sistemi che aiutano in attività dei processi di sviluppo software fornendo supporti all'automazione di tali attività. Si dividono in:
Upper Case → destinati ad attività ad alto livello (analisi dei requisiti e design).
Lower Case → destinati alle attività di basso livello (programmazione, debugging e testing).

Buon Software: Se ha le funzionalità e performance richieste dal cliente e dovrebbe essere mantenibile, affidabile ed accettabile.
Mantenibile: Il software dovrebbe poter evolvere senza dover essere riscritto.
Affidabile: Il software dovrebbe dare fiducia a chi lo usa.
Efficiente: Il software dovrebbe usare le risorse di sistema senza sprechi.
Accettabile: Il software dovrebbe essere accettato dal cliente per cui è stato fatto (essere usabile).

Maggiori sfide in Ingegneria del Software:
Eterogeneità: Software multi piattaforma.
Rilascio: Sviluppare tecniche che permettano rilasci in tempi rapidi del software.
Fiducia: Sviluppo di tecniche che dimostrino agli utenti (o clienti) che si possono fidare.

Ciclo di vita del software e processo di sviluppo.
Software = prodotto industriale.
Il software è tuttavia diverso dai prodotti normali, essendo immateriale, facilmente modificabile e non soggetto a standard qualitativi.
Tuttavia il software è uguale agli altri prodotti considerando che deve soddisfare le esigenze del cliente e del produttore, deve svolgere le funzioni richieste, deve essere realizzato e consegnato nei limiti di tempo e costo previsti.

Ciclo di vita del software: Inizia quando si manifesta la necessità o utilità del prodotto, attraversa la progettazione, la produzione, la verifica e la consegna, prevede manutenzione e termina con il ritiro.

Analisi → Progettazione → Scrittura → Convalida

Processo di Sviluppo: Particolare modo di organizzare attività costituenti il ciclo di vita, cioè di assegnare

risorse alle varie attività e fissarne le scadenze.
Fase: Intervallo di tempo in cui si svolgono certe attività.
Modello di Processo: descrizione generica di una famiglia di processi simili.

Tipi di processi di sviluppo: Code and Fix, cascata, spirale, incrementale (Unified Process).

Code & Fix: Scrivi codice, correggi, ritorna al primo punto.

Modello A Cascata (o modello a fasi):
Analisi → Progettazione → Scrittura → Convalida.
Definizione di Requisiti → Modellazione e design → Sviluppo del codice → Test e Rilascio.
Lo sviluppo è scomposto in fasi, ogni fase termina prima dell'inizio della successiva, ogni fase produce qualcosa (documento o programma).
Analisi: Definisce i requisiti di sistema, e i risultati devono essere capibili sia dai clienti che dagli sviluppatori.
Progettazione: I requisiti di sistema vengono scomposti in software e hardware, viene definita l'architettura del sistema.
Scrittura: tutte le componenti del sistema vengono realizzate ed assemblate, vengono effettuati test unitari sui componenti.
Convalida: Le componenti vengono integrate e vengono fatti i test di integrazione, dopodiché il sistema viene rilasciato.
Assunti modello a cascata:
Requisiti conosciuti fin dall'inizio, requisiti non cambiano, progettazione può essere effettuata in maniera astratta, si pensa che tutte le componenti si integrino correttamente alla fine del processo.
Problemi modello a cascata:
I requisiti sono tipicamente non definitivi e imprecisi, lo sviluppo unidirezionale porta ad una mancanza di feedback e i problemi vengono rimandati ad una fase successiva, lo sviluppo del tutto in fase avnzata implica rischi tecnologici, concettuali e i test arrivano solo in fondo.

Modello Incrementale:
Lo sviluppo avviene gradualmente e viene gestito tenendo conto dei prodotti parziali. La manutenzione è considerata come una nuova versione del prodotto esistente. Ampli aggiornamenti dei requisiti sono gestiti come nuove versioni. Si scrive soprattutto codice per arrivare a un sistema eseguibile.
Analisi → Progettazione → Sviluppo e test → Rilascio → Relase 1
Analisi → Progettazione → Sviluppo e test → Rilascio → Relase 2 …
C'è comunque comunicazione tra le varie fasi di analisi, le varie fasi di progettazione etc.
I gruppi di analisi, progettazione, sviluppo possono lavorare in contemporanea (pipeline).

Modello a spirale:
Si rappresenta mediante una spirale in cui ogni voluta rappresenta un raffinamento od un completamento del sistema rispetto alla voluta precedente.
La spirale può essere divisa in quattro spicchi, che rappresentano la definizione degli obiettivi, l'analisi dei rischi, lo sviluppo ed il test e la pianificazione della fase successiva (od il rilascio).
Il raggio della spirale indica invece i costi, che crescono mano a mano con l'aumento della complessità del progetto.
Software prodotto in modo ciclico (iterativo ed incrementale), producendo versioni sempre più raffinate del prodotto riducendo i rischi.
Si ha un insieme di prodotti intermedi per garantire che il prodotto è fattibile e soddisfacente.
Il punto di partenza può anche essere un foglio di carta con uno schema presentato al cliente, il quale viene poco a poco fatto evolvere.

Unified Process:
Combina processi a cascata ed iterativo, usa estensivamente UML. Prima di progettare ed implementare un sistema occorre descriverne chiaramente i requisiti. (plan a little, design a little, code a little).
I vari requisiti sono descritti con i casi d'uso, l'implementazione ed il test devono portare alla corretta esecuzione dei casi d'uso.
E' un processo iterativo ed incrementale, cioè ogni stadio di lavorazione può essere eseguito più volte e il sistema viene diviso in mini progetti, tra i quali vengono eseguiti subito quelli a maggior rischio.
Ha 5 stadi di lavorazione fondamentali: Raccolta dei requisiti, Analisi, Progettazione, Implementazione e Test.
Ogni stadio viene eseguito una volta per ogni iterazione e produce un determinato prodotto.
Fasi:
Quattro fondamentali: Inizio, Elaborazione, Costruzione e Trasformazione o transizione.

Inizio: Raccolta requisiti principali.
Elaborazione: Correzione requisiti, Analisi ordinata, Progettazione, Prime implementazioni.
Costruzione: Completamento progettazione + dettagli, Implementazione, Verifiche e test di aggregazione.
Trasformazione: Test finali di pre produzione, Messa in opera, Manutenzione, Archiviazione documentale e attivazione procedure di gestione.

SLIDE 3
Studio di fattibilità: Serve a stabilire se un prodotto può essere realizzato, se è conveniente realizzarlo e nell'eventualità le quantità di risorse necessarie, i tempi necessari e i costi relativi.
Dovrebbe contenere una descrizione del problema in termini di obiettivi e vincoli,
un ipotesi di soluzione sull'analisi dello stato dell'arte (conoscenze e tecnologie disponibili) e una stima costi e tempi richiesti.
Prodotto oggetto dello studio: può essere destinato alla stessa organizzazione di cui fa parte il produttore, ad un committente esterno oppure al mercato. Lo studio di fattibilità può essere venduto come prodotto finito, indipendentemente dall'eventuale prosecuzione del progetto.
Esempio su slide – realizzazione di uno studio di fattibilità.
Contenuto documento dello studio di fattibilità: Descrizione del problema che deve essere risolto dall'applicazione (in termini di obiettivi e vincoli), sia della situazione attuale che di un eventuale cambiamento. **Un'ipotesi di soluzione** sulla base delle conoscenze e delle tecnologie disponibili.
Una stima dei costi e dei tempi richiesti.
Fasi di realizzazione del progetto: Studio di fattibilità, realizzazione del prototipo, test di integrazione, realizzazione dell'applicativo, test di carico e affidabilità, messa in produzione.
Analisi requisiti → L'analisi serve per descrivere i requisiti che dovrà avere il sistema oggetto del progetto.
Requisiti → I requisiti descrivono ciò che l'utente si aspetta dal sistema, e specificarli significa esprimerli in modo
Piramide requisiti: I requisiti sono organizzati in maniera gerarchica, divisi in dominio del problema e dominio della soluzione. "Needs" → Dominio del problema, "Features e Software requirements → Dominio della soluzione. I più importanti sono quelli di "needs" che non possiamo modificare.
Requisiti: Può essere usato indifferentemente in una richiesta di offerta (deve essere interpretabile) oppure in un contratto (deve definire i dettagli). I requisiti si dividono in:
Requisiti utente: Descritti in linguaggio naturale comprensibile all'utente, descrivono i servizi le funzioni ed i vincoli operativi del sistema anche con diagrammi.
Requisiti di sistema: Descrivono in maniera dettagliata e strutturata i servizi, le funzioni ed i vincoli, operativi del sistema e definiscono precisamente cosa deve essere realizzato in base al contratto di realizzazione.
Inoltre i **requisiti** si distinguono in:
Requisiti funzionali: Descrivono cosa deve fare il sistema (generalmente esprimibili in modo formale). Specificano le azioni che il sistema deve essere in grado di effettuare, senza vincoli fisici. Le azioni vengono descritte con il modello use-case Model. Specificano i comportamenti in ingresso e uscita del sistema.
Se i **requisiti utente funzionali** possono essere definiti in alto livello, i **requisiti di sistema funzionali** devono esprimere anche i dettagli.
Requisiti non funzionali: Esprimono vincoli o caratteristiche di qualità (più difficile da esprimere in modo formale). Per esempio la sicurezza, cioè capacità di funzionare senza arrecare danni a persone o cose, oppure la capacità di impedire accessi non autorizzati al sistema e alle informazioni contenute. Inoltre deve essere in grado di funzionare in modo accettabile anche in situazioni non previste (guasti, ingresso dati errati). Un altro esempio sono le prestazioni (uso efficiente x es della memoria e del tempo di esecuzione) oppure la disponibilità (Servizio continuo per lunghi periodi).
Schema:
Requisiti non funzionali si dividono in → requisiti di prodotto, di organizzazione ed esterni.
Requisiti di prodotto si dividono in → usabilità, efficienza, affidabilità (performance e uso risorse) e portabilità.
Requisiti di organizzazione → Standards, delivery e Implementazione.
Requisiti esterni → Etici, Interoperabilità e Legislativi.

FURPS+
Requisiti → Functionality, Usability, Reliability, Performance, Supportability. (FURPS).
Il più indica → Design constraints, Implementation requirements, Interface requiriments, physical requiriments.
Usability → Contiene ergonomia, estetica, eventuale help contestuale e wizard, documentazione.
Affidabilità → Frequenza e gravità malfunzionamenti, predicibilità, accuratezza, Mean time between failures (Ogni quanto si pianta).

Performance → Efficienza, disponibilità, Throughput, tempi di risposta, tempi di recovery.
Supportabilità → Adattabilità, estensibilità, manutenibilità, configurabilità, installabilità, facilità dei test, facilità di internazionalizzazione.
Requisiti di implementazione (vincoli nello sviluppo di sistema) **→** Standard, linguaggi di programmazione, basi di dati, ambienti operativi e limiti di risorse.
Requisiti di interfaccia → Specifica elementi esterni con cui il sistema deve interagire, vincoli di formato dei dati, tempo o altri fattori.
Requisiti fisici → Caratteristiche fisiche del sistema (peso, dimensione, forma, materiale).

Problemi analisi dei requisiti: Clienti hanno raramente idea precisa di quello che vogliono, usano una terminologia non semplice da tradurre in specifiche, possono avere richieste in conflitto tra loro, i requisiti possono cambiare durante l'analisi.
Processo analisi requisiti: Interazione con i clienti per la scoperta dei requisiti (comprensione del dominio applicativo). Classificazione dei requisit (Organizzazione del raggruppamento dei requisiti). Assegnamento delle priorità (ai requisiti e risoluzione dei conflitti). Documentazione dei requisiti.
Il tutto può essere rappresentato tramite il modello a spirale sempre divisa in 4 spicchi, che sono (in ordine) la scoperta dei requisiti, la classificazione dei requisiti, l'assegnazione delle priorità e la documentazione.
Esempio su slide – Bancomat.

SLIDE 4 – DOCUMENTO DI VISUAL
Esempio documento visual – Scenario – Caso d'uso
Documento di Vision → Documento introduttivo sul progetto, individua le principali caratteristiche del sistema evidenziando i problemi e le possibili soluzioni. Documento corto di altissimo livello.
Contenuti → Introduzione e obiettivi, panoramica, requisiti e concetto operativo
Introduzione ed obiettivi:
– Obiettivi → obiettivi del progetto, sinteticamente e con linguaggio naturale.
– Glossario → linguaggio naturale = ambiguità. Devo definire con precisione l'entità coinvolte nel sistema del mondo reale in linguaggio comune. Il glossario evita le ambiguità.
Panoramica:
– Problema → Descrizione in linguaggio naturale del problema che va ad affrontare. Schema chiarificatore contenente CHI, PER, PRODOTTO, CHE, DIVERSAMENTE DA(altri esempi)
– Perché usare questo prodotto → descrizione dei motivi.
– Parti interessate → Stakeolder e interesse che possono avere nel sistema.
– Attori → Che faranno parte dei requisiti funzionali, con attribuzione agli stakeholder.
Requisiti:
– Necessità di businnes → Identificazione dei requisiti di altissimo livello (“NEEDS”) ovvero se non vengono realizzati non c'è la ragione del progetto.
– Requisiti utente → Identificazione dei requisiti ad alto livello del progetto (“BUSINNES NEEDS”).
Concetto operativo:
– Concetto operativo → Diagramma che descrive in termini comprensibili dal Cliente il funzionamento del sistema.

Esempio di uso – pagina 24 → 38

Scenario → esempio tratto dal mondo reale di come il sistema può essere usato. Cioè una descrizione della situazione iniziale, una descrizione di flusso normale di informazioni, una descrizione di errore che può capitare, Una descrizione del sistema quando ci sono attività concorrenti.
Caso d'uso → è una tecnica basata su scenari, esprimibile come diagramma UML. L'insieme dei casi d’uso dovrebbe descrivere tutte le possibili interazioni con il sistema. Il caso d'uso descrive l’interazione fra due entità che interagiscono fra loro e consente di stabilire Servizi forniti e richiesti e utenti abilitati.
Caso d'uso → Il primo scopo `e quello di trovare un confine preciso (boundary) per il sistema/ sottosistema/ componente che si sta analizzando.
Una volta stabilito il confine si può stabilire cosa fa il sistema rispetto all'esterno e identificare attori e use case.
Caso d'uso definizione formale → SEQUENZA DI TRANSAZIONI, eseguita da un ATTORE in interazione col SISTEMA, la quale fornisce un VALORE MISURABILE per l’attore.

Attore (omino) + Relazione di interazione → (sistema(use case)) ← Confine del sistema opzionale. Questo è il nome del caso d'uso.

Attore → rappresenta un’entità esterna al sistema (una persona, un altro sistema software, un componente hardware) che interagisce col sistema, individua un ruolo piuttosto che un’entità fisica.

Use Case → Uno use case rappresenta una situazione tipica di utilizzo del sistema e comprende in sé vari flussi possibili di esecuzione. Uno use case rappresenta un'importante parte di funzionalità, completa dall'inizio alla fine.

Documento di caratteristiche → Caratteristiche del sistema, elenco dei requisiti utente, delle caratteristiche del sistema, dei requisiti architetturali e dei vincoli di sistema realizzato a partire dai requisiti utenti e dal documento di Vision.
Documento di caratteristiche contenuto → Introduzione e obiettivi, Architettura, Requisiti, Specifiche sulle interfacce esterne, Standard e Documentazione a supporto.

Contenuto documento di caratteristiche:

Introduzione ed obiettivi:
– Obiettivi → del progetto, sinteticamente e con linguaggio naturale.
– Glossario → di progetto, o riferimento al glossario ed altri documenti del progetto.
Architettura:
– Modello logico → Diagramma anche blocchi del modello che rappresenti le componenti del sistema.
– Modello fisico → Disegno schematico delle componenti fisiche del sistema e delle interazioni tra esse.
– Tecnologie → usate nella realizzazione del sistema (Sistema operativo, middleware, linguaggi di programmazione).
Requisiti:
– Requisiti funzionali → Identificazione dei casi d'uso. Tipicamente un diagramma dei casi d'uso dell'intero sistema. Spesso viene aggiunto un paragrafo "Funzioni per l'utente" con i vari casi d'uso per tutti gli attori del sistema.
– Requisiti non funzionali → Come specificati prima.
Specifiche sulle interfacce esterne:
– Specifiche sulle interfacce esterne → Descrizione dei flussi e protocolli di interscambio di dati con altri sistemi o con l'ambiente. Tipicamente **input e output.**
Standard e Documentazione a supporto:
– Standard e Documentazione a supporto → Definizione di eventuali standard di riferimento, tecnologici o di dominio. Definizione di quale documentazione si intende fornire a supporto del prodotto.

Versione (Pagine 57 → 80)

SLIDE 5 – SPECIFICA DEI CASI D'USO
Documento di Visual → Documento di caratteristiche (Scelta nostra) → Analisi requisiti funzionali.

Analisi funzionale dei casi d'uso → devono essere individuati con precisione gli scenari d'uso del sistema e gli scenari di interazione fra sistema e attori.
Definizione use case → SEQUENZA DI TRANSAZIONI, eseguita da un ATTORE in interazione col SISTEMA, la quale fornisce un VALORE MISURABILE per l'attore.

Attore → modella un'entità esterna che interagisce con il sistema diviso in tipologia utente e sistema esterno. Un attore ha un nome univoco e opzionalmente una descrizione.
Caso d'uso → modella un servizio fornito dal sistema. Un caso d'uso ha un nome univoco, almeno un attore collegato, un flusso di eventi e un assunto in entrata e uscita.
Contenuto caso d'uso → Un attore principale che ne motiva l'esistenza, Delle condizioni in ingresso e uscita, che descrivono i vincoli che il sistema soddisfa prima e dopo che il caso d'uso sia terminato.
Un caso d'uso descrive un interazione, non come avviene (non interfaccia utente → dovrebbe essere derivata dai casi d'uso).
Include → Serve per riassumere comportamenti comuni (caso d'uso **→ includes →** altri casi d'uso).
Extends → Serve per descrivere comportamenti particolari.
Generalizzazione → È analoga alla estensione tra classi. (caso d'uso → altro caso d'uso) (Amministratore → Operatore).

Identificazione dei casi d'uso → Definizione esatta dei confini.
Si individuano i vari scenari d'uso e interazione fra sistema e attori (I singoli casi d'uso vengono identificati dalle singole ellissi nel diagramma, inoltre descrivono cosa si vuole che il sistema faccia, non come il comportamento deve essere implementato.)
Definizione delle interazioni entro i singoli casi d'uso.
Esame dei diagrammi e delle loro descrizioni per potere procedere alla raccolta a fattore comune facendo uso delle relazioni "extends" ed "include".

Raccolta a fattore comune facendo uso delle relazioni "extends" ed "include" →
Il passo può essere iterato più volte → occorre tenere conto:
Della granularità del problema, del grado di definizione e precisione che si vuole raggiungere, un singolo caso d'uso spesso dà origine ad una singola maschera e da un caso d'uso deriverà anche un caso di test.
C'è una relazione tra i test funzionali e i casi d'uso (vanno di pari passo).

Identificazione di casi d'uso → Il prodotto di questo passo è l'insieme completo degli Use Case inseriti entro uno o più case diagram. → ognuno corredato di adeguata descrizione, considerando sia il percorso principale che gli eventuali percorsi alternativi.
Esame dei diagrammi e delle loro descrizioni per potere procedere alla raccolta a fattore comune facendo uso delle relazioni "extends" ed "include".
Gli use case diagram non esprimono direttamente relazioni di flusso logico /temporale (prima dopo) fra i loro componenti → Do un minimo di consequenzialità tramite gli assunti.
Le relazioni "prima e dopo" vengono espresse da un altro diagramma chiamato activity diagram.
Esame dei diagrammi e delle loro descrizioni per potere procedere alla raccolta a fattore comune facendo uso delle relazioni "extends" ed "include".
I diagrammi da soli non sono sufficienti e si devono aggiungere ulteriori descrizioni, che rendano più preciso il tutto.

Documento di specifica dei casi d'uso → Use Case specification , Documento di dettaglio che descrive, per ogni Use Case, il flusso base, i flussi alternativi e gli step che li compongono.
Contiene → Introduzione ed obiettivi, funzioni per l'utente e specifiche su ogni caso d'uso.
Introduzione e obiettivi:
- **Obiettivi →** del progetto, sinteticamente e con linguaggio naturale.
- **Definizione, acronimi e riferimenti** → Glossario del progetto, o riferimento allo stesso con link a documenti.

Funzioni per l'utente:
- **Use case generale →** diagramma contenente tutti gli attori e tutte le maggiori funzioni offerte dal sistema.

Specifiche per ogni caso d'uso:
- **Un capitolo per ogni attore del sistema →**
  Tipicamente un diagramma dei casi d'uso con un attore e tutte le interazioni che lo coinvolgono, seguito da, per ogni caso d'uso:
  Descrizione → Una descrizione in linguaggio naturale del comportamento del sistema nel caso d'uso.
  Flusso principale → normalmente un diagramma.
  Precondizioni all'ingresso → Una descrizione in linguaggio naturale dei vincoli che devo essere soddisfatti all'inizio del caso d'uso.
  Postcondizioni all'uscita → Una descrizione in linguaggio naturale dei vincoli che devo essere soddisfatti al termine del caso d'uso.

Esempio slide 40 → 56
Documento di vision → Documento di caratteristiche → Documento di analisi dei requisiti.

SLIDE 6 – GESTIONE DI PROGETTO

Progetto → Si definisce progetto la gestione sistematica di un'impresa complessa, **unica e di durata determinata**, rivolta al raggiungimento di un **obiettivo chiaro e predefinito** mediante un processo continuo di pianificazione e controllo di risorse differenziate e con vincoli indipendenti di **costi, tempi e qualità.**
Pmbok 2008 → project management book of knowledge

Un **progetto** è uno sforzo **temporaneo e organizzato** intrapreso allo scopo di creare **un prodotto, un servizio od un risultato unico.**

Deliverables → sono i risultati di un **progetto**. Possono essere un prodotto od un manufatto, quantificabile, che costituisce un prodotto finale o uno stato di lavorazione, un documento od un file.

Caratteristiche progetto:
Obiettivo → Esiste un obiettivo specifico, unico, raggiungibile ed eventualmente interconnesso con altri obiettivi o progetti.
Unicità → L'obiettivo o scopo, non è la ripetizione di esperienze già fatte, ma semmai deve esplicitare
Temporaneità →

Risorse →
Multidiscipilinarietà →
Programmazione →

Definizioni:
Metodologie e tecniche per la realizzazione di un progetto si chiama **Project Management.**
Project Management(1) →
Project Management(2) →

Progetto != Project
In italiano utilizziamo il termine progetto per indicare dei concetti che sono ben distinti in inglese:
→ Design →
→ Engineering →
→ Drawing →
→ Project → Una serie di attività mirate al raggiungimento di un obiettivo. Queste possono comprendere anche una parte di sviluppo progettuale di componenti/prodotti.

Esempi di project (slide 37)

Processo → Un insieme di attività coordinate rivolte ad un compito specifico per produrre un risultato che direttamente o indirettamente crea un valore per uno stakeholder.
In un progetto sono quindi distinguibili:
→ Processi esecutivi (es scrittura del codice)
→ Processi di gestione, che comprendono **processi di avviamento, di pianificazione, di controllo e di chiusura.**

<u>Gestione di progetto:</u>
Avviamento → Pianificazione → Controllo → Chiusura

Avviamento comprende: studio di fattibilità, definizione obiettivi e lancio del progetto.
Pianificazione comprende: Decomposizione del progetto, definizione dei tempi, definizione risorse e analisi dei rischi → formalizzazione di buon senso.
Controllo comprende: Rilevazione (+ criteri misurazione sensati), consolidamento, verifica, eventuale ri-pianificazione e gestione dei rischi.
Chiusura comprende: Verbale di rilascio (collaudo), analisi dei risultati (com'è andato il progetto) e storicizzazione.

Gestione di un progetto, correlazione delle attività.
→ Avvio, pianificazione, controllo, chiusura → Analista.
→ Esecuzione → Capo progetto.

Controllo ↔ Esecuzione:
I processi di controllo e monitoraggio, nella loro relazione con i processi esecutivi seguono un modello teorico chiamato PDCA (o ciclo di deming).

Ciclo di deming:
Il ciclo di deming è una metodologia di miglioramento continuo della qualità totale in un sistema aziendale. Il ciclo di Deming consiste in una sequenza di fasi (Plan - Do - Check - Act) e, per questo motivo, è spesso conosciuto anche con la sigla PDCA.
Deming ha reso sistematico un processo mentale a cui facciamo ricorso, spesso, in modo inconscio.

Ciclo di deming – plan → Nella fase di plan si pianificano e di programmano tutte le attività necessarie al raggiungimento dell'obiettivo, che deve essere misurabile e quantificabile.
Ciclo di deming – Do → Nella fase successiva, quella di Do (fare), si esegue ciò che si è pianificato.
Ciclo di deming – Check → Alla fase di Do segue quella di Check, cioè di controllo e verifica. In questa fase si prendono in considerazione i risultati ottenuti.
Ciclo di deming – Act → A obiettivo raggiunto, quindi, si consolida il procedimento seguito, per poterlo riprodurre tutte le volte che si presenterà la stessa situazione. Altrimenti, dovremo modificare le varie fasi del ciclo.

<u>Processi di chiusura</u>

Chiusura Amministrativa della Documentazione → Sviluppo e distribuzione della Documentazione

relativa al completamento delle diverse fasi.
Chiusura dei contratti → Chiusura dei termini contrattuali e risoluzione punti in sospeso.
Verbale di accettazione → Documento per la formalizzazione dell'esito del Test di Accettazione Utente. Molto importante perché è quello legalmente vincolante.
Contiene **funzioni rilasciate, limiti o vincoli conosciuti, firma del responsabile del committente.**
Verbale di accettazione = Verbale di rilascio → Cambia solo che c'è la firma di accettazione!

SLIDE 7 – GESTIONE DI PROGETTO
Progetto → Si definisce Progetto la "gestione sistematica di un'impresa complessa, unica e di durata determinata rivolta al raggiungimento di un obiettivo chiaro e predefinito mediante un processo continuo di pianificazione e controllo di risorse differenziate e con vincoli interdipendenti di **costi, tempi, qualità.**
Un progetto è uno sforzo temporaneo ed organizzato intrapreso allo scopo di creare un prodotto, un servizio o un risultato unici.

Processo → un insieme di attività coordinate rivolte ad un compito specifico per produrre un risultato che direttamente o indirettamente crea un valore per uno stakeholder.
In un progetto sono presenti **Processi esecutivi** (Scrittura codice) **e Processi di gestione** (Processi di avviamento, pianificazione, controllo e chiusura).

Inizio reale slide
Deliverables → Risultati di un'attività. Possono essere un prodotto od un manufatto, quantificabile, che costituisce un prodotto finale od uno stato di lavorazione, un documento od un file.
Work package → Pacchetti o insieme di deliverables.
Milestone → Momento particolarmente importante, quali revisioni del lavoro e produzione di deliverable, al termine di un processo.

"Work breakdown structure" - WBS
La WBS rappresenta il lavoro specificato nella descrizione dello scopo del progetto, approvata. I componenti costitutivi della WBS sono d'aiuto agli stakeholder per visualizzare i deliverable del progetto.
La WBS è una decomposizione gerarchica, orientata ad i deliverable, del lavoro che deve essere eseguito dal team di progetto per raggiungere gli obiettivi del progetto stesso e creare i deliverable richiesti.
Ciascun livello discendente del WBS rappresenta una definizione sempre più dettagliata del lavoro del progetto.
La WBS è uno strumento che può essere concettualmente applicato a qualunque tipologia di progetto al fine di fornire una rappresentazione scalare e ben strutturata del progetto, concordata da tutti gli stakeholder.
La WBS è **una rappresentazione gerarchica ad "albero"** che rappresenta graficamente la scomposizione del lavoro da svolgere per costruire appunto i deliverables del progetto.
In tal senso è un documento molto importante perché ha come obiettivo il concordare / formalizzare ciò che è dentro l'ambito del progetto e ciò che ne resterà fuori.

Descrizione WBS → L'adozione della WBS ha la funzione di segmentare le attività in pacchetti di lavoro al fine di consentire un controllo più sistematico, articolato e coerente durante tutto il progetto, in cui è possibile identificare:
obiettivi specifici per ciascun pacchetto di lavoro, prestazioni, caratteristiche tecniche, preventivi costi e tempi, milestone di programma e responsabilità.

Realizzazione della WBS → Il processo di creazione della WBS consiste nella suddivisione del lavoro previsto e dei deliverable del progetto in componenti più piccoli e quindi maggiormente gestibili.
Non esiste un modello unico, predefinito, per la creazione di una specifica WBS.
Esiste un approccio standard alla creazione della WBS.

Passi di realizzazione della WBS:
1. Acquisizione degli obiettivi, dei deliverables, delle attività principali di progetto
2. Scelta dei criteri di disaggregazione e definizione della WBS
3. Definizione del primo livello di WBS
4. Scomposizione, secondo i criteri definiti al punto 2, dei livelli successivi al primo (in genere sono sufficienti 4 livelli)
5. Identificazione dei Work Package
6. Identificazione delle attivit`a, dei deliverables, delle milestone.
7. Verifica di congruenza bottom-up della WBS

Schemi vari

Per fare completa la WBS applico la regola del 100%.
La regola del 100%... precisa che la WBS debba includere il 100% del lavoro definito dal progetto e includere tutto il necessario - interno, esterno e appaltato - alla realizzazione del progetto, inclusa la gestione del progetto stesso.
La decomposizione e la valutazione della WBS e si applica a tutti i livelli della gerarchia: la somma del lavoro dei livelli "figli" deve essere uguale al 100% del lavoro rappresentato dal loro "padre" e la WBS non dovrebbe includere alcun lavoro al di fuori dai limiti del progetto, ovvero non può includere più del 100% del lavoro.

Principi basi (riassunto) della WBS → Programmazione dei risultati, non delle azioni. Se il progettista della WBS tenta di comprendervi ogni dettaglio relativo alle azioni probabilmente includerà troppe azioni o troppo poche. Troppe azioni eccederanno il 100% dei limiti del nodo superiore, mentre troppo poche non arriveranno a quella percentuale. In aggiunta alla regola del 100%, è importante che non ci siano sovrapposizioni nella definizione dei limiti tra due elementi della WBS.

Pianificazione a finestra mobile →
Un quesito fondamentale da porsi durante la progettazione di ogni WBS è quando smettere di dividere il lavoro in elementi più piccoli. Il livello di dettaglio dei Work Package varia in base alle dimensioni e alla complessità del progetto e non sempre è possibile effettuare, in fase di prima pianificazione, la scomposizione di tutti i rami di WBS.
Questo perché di alcuni non si ha un adeguato livello di conoscenza, oppure la loro esecuzione è molto lontana nel tempo.
Il team di project management, in questi casi, può in prima analisi fermarsi al primo o al secondo livello di WBS e aspettare fino a che i deliverable o le attività da realizzare siano più chiari per sviluppare i dettagli della WBS.
Tale tecnica viene a volte denominata "pianificazione a finestra mobile" o **Rolling wave planning.**

Definizione pianificazione a finestra mobile:
È una forma di pianificazione a elaborazione progressiva, secondo la quale il lavoro da completare a breve viene pianificato nel dettaglio fino al livello più di dettaglio della WBS, mentre il lavoro da svolgere a lunghissimo termine viene pianificato per i componenti della WBS ad un livello relativamente alto.
Ne consegue che all'interno del ciclo di vita del progetto le attività pianificate possono esistere a diversi livelli di dettaglio.

Errori e malintesi:
Una WBS non è una pianificazione del progetto e non è una lista in ordine cronologico.
Una WBS non è una lista di lavori bensì una classificazione degli scopi di progetto.
È sconsigliabile e considerato controproducente pianificare un progetto prima di progettare una WBS appropriata, sarebbe simile al voler pianificare le attività di un cantiere edile prima ancora di completare il progetto dell'edificio.
Senza concentrarsi sui risultati del progetto, è molto difficile seguire la regola del 100% a ogni livello della gerarchia della WBS.
Non è possibile recuperare una WBS con definizioni improprie senza rifarla dal principio, per cui vale sempre la pena di completarla e controllarla bene prima di iniziare qualunque altra pianificazione.
Una WBS non è una gerarchia organizzativa. Talvolta, si commette l'errore di creare una WBS che fedele alla struttura organizzativa e non agli obiettivi del progetto.
Se i risultati e le azioni si confondono a vicenda, il controllo dei cambiamenti può essere troppo rigido per le azioni e troppo informale per i risultati.

Passo fondamentale dopo l'analisi dei requisiti fare la WBS.
OBS (organization breakdown structure) → L'Organization Breakdown Structure (OBS) o Struttura di Scomposizione dell'Organizzazione è una decomposizione gerarchica delle responsabilità del progetto, al fine di individuare la persona o l'ufficio competente per ogni pacchetto di lavoro.

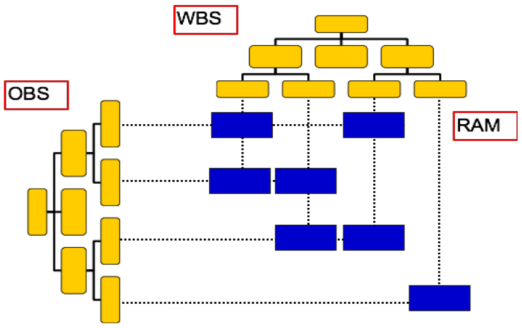
L'OBS somiglia ma non coincide con l'organigramma dell'Azienda, in quanto:
→ Include soltanto le funzioni coinvolte nel progetto, siano persone, uffici o sezioni organizzative.
→ comprende i responsabili esterni (appaltatori, consulenti etc).
→ Il progetto si scompone in sviluppo, sistemi, grafica per esempio. Di solito la grafica viene appaltata esternamente.

RAM (matrice di assegnazione delle responsabilità)
→ Il metodo **LRC** (linear responsability charting) prevede una specificazione dei ruoli e delle strutture nei processi, attraverso una visione tabellare della responsabilità organizzativa che integra l'organigramma ed

incrocia le attività (o le fasi) con le strutture organizzative. Questo metodo ha condotto alla Matrice delle Responsabilità.

→ Una volta definita la WBS è possibile correlare gli elementi di WBS con le unità organizzative aziendali mediante la costituzione di una matrice bidimensionale che pone in relazione i compiti e le attività da svolgere (che cosa) con le unità organizzative interne (es. centri di costo), i subcontractors o i fornitori (chi).

→ Tale matrice definita Responsibility Assigment Matrix o Matrice di assegnazione delle Responsabilità (RAM) nasce dall'unione della WBS, che rappresenta il disegno (strutturato) globale del progetto, con la struttura organizzativa dell'aziendale del progetto, OBS - Organization Breakdown Structure.



La forma più diffusa di matrice RAM è detta matrice **RACI**, definita dallo standard PMI: aiuta a chiarire e concordare chi fa che cosa.
La denominazione "RACI" deriva dall'acronimo composto dalle iniziali delle parole inglesi: Responsible, Accountable, Consulted ed Informed.
Sulla matrice, le diverse persone (o ruoli) appaiono nelle colonne, mentre le attività sono elencate sulle righe. Nei punti di intersezione viene indicato il livello di responsabilità di ogni persona su ogni attività, apponendo la lettera R, A, C o I.

RACI:
R (RESPONSIBLE) con la lettera "R" viene indicato il RESPONSABILE DELLA REALIZZAZIONE, cioè colui che esegue materialmente un'attività mediante una responsabilità di tipo operativo (le R possono essere condivise).
A (ACCOUNTABLE) la lettera "A" indica colui che viene riconosciuto come l'accentratore della responsabilità finale di una certa attività. È la persona che ha l'ultima parola ed il potere di veto;
il project manager, ad esempio, è l'unico vero accountable di un progetto; il successo o il fallimento dello stesso ricadrà, in ultima analisi, sulle sue spalle (ci può essere una sola A per ogni attività).
C (CONSULTED) la "C" di CONSULTATO viene associata alla persona consultata prima di eseguire l'attività o prima di prendere decisioni esecutive (le C possono essere più di una) .
I (INFORMED) è identificato con la "I" di INFORMATO chi viene informato, di solito successivamente, della decisione o dell'azione intrapresa (le I possono essere molteplici).
Esempio tabella slide 69

Dal ruolo svolto in relazione ad una certa attività o si può ricavare una indicazione utile sulla occupazione di tempo della persona nella attività.
Se una persona è R (RESPONSIBLE) su una attività sarà probabilmente impegnato al 100% su tale attività.
Se una persona è C (CONSULTED) su una attività sarà probabilmente impegnato al 10% su tale attività.

SLIDE 8 – SCHEDULAZIONE DI PROGETTO
La creazione della **Schedulazione di progetto** permette di correlare e rappresentare graficamente tutti gli elementi di un progetto in un unico piano temporale.
Schedulazione di progetto → Deve basarsi sulla WBS, deve essere completa, deve rispecchiare i vincoli contrattuali, includere gli eventi chiave (milestone) e d'interfaccia con altri progetti.

Nella schedulazione di progetto le tecniche utilizzate sono chiamate **tecniche reticolari,** che sono strumenti per realizzare lo scheduling ed il controllo dell'avanzamento del progetto.
Consentono inoltre di identificare i percorsi critici e si basano sulla teoria dei grafi.

Il reticolo di progetto con metodo CPM:

Il reticolo di progetto → si individuano tre componenti fondamentali: Evento, attività e durata.
Evento → Stato rappresentativo dell'evolvere del progetto. È un obiettivo da raggiungere e può rappresentare l'inizio o la fine di un'attività. Non presuppone l'impiego di tempo. Inoltre gli eventi di inizio o fine hanno solo attività uscenti od entranti.
Attività → È lo svolgersi di un'azione, ed è inoltre un elemento di lavoro all'interno del progetto. Ogni attività richiede l'impiego di tempo e/o risorse e/o mezzi. Ha inoltre un codice identificativo (allineato alla WBS). È legata attraverso vincoli ad altre attività. Ha una durata; tutte le attività hanno durata maggiore di zero, ad eccezione dell'inizio e la fine del progetto, che hanno durata nulla. Anche le milestone possono avere durata nulla. Ogni attività è inoltre associata ad una milestone.
Durata → Rappresenta una stima del tempo necessario al completamento dell'attività. Comprende sia i tempi tecnici che quelli passivi (ovvero i tempi di attesa). L'unità di misura può essere quella più congeniale (ore, giorni, ore/uomo...).

Identificazione dell'ordine delle attività:
Attività in serie:
(FS) Finish to start → Attività B non può iniziare prima che sia finita A.
Attività in parallelo:
(SS) start to start → (B non può iniziare prima che sia iniziata A).
(FF) finish to finish → (B non può finire prima che sia finita A).
(SF) start to finish → (B non può finire prima che sia iniziata A).
Sfasamento tra attività:
Anticipo (leads) → (B può iniziare di un tempo “t” prima della fine di A).
Ritardo (lag) → (B non può iniziare prima di un tempo “r” dopo la fine di A).

Metodo del reticolo di precedenza (PDM):
PDM → Metodo di costruzione di un reticolo di schedulazione del progetto che utilizza rettangoli (chiamati nodi) per rappresentare le attività che connette con frecce per mostrare le dipendenze. Questa tecnica è chiamata anche attività su nodo (AON). Inoltre la relazione di dipendenza maggiormente utilizzata è quella di fine-inizio.

Esempi pg 44 45

Reticolo delle precedenze → Evidenzia quali sono i legami logici (le precedenze) tra le attività. Fornisce aiuto per calcolare la durata di un progetto o di un gruppo di attività, riconoscere i cammini critici, ottimizzare la pianificazione del progetto in termini di riduzione totale dei tempi di progetto e livellamento delle risorse

Temporizzazione delle attività:
Per ogni attività: Si determina la durata prevista della sua esecuzione, la cui stima può essere deterministica o probabilistica. Inoltre si colloca temporalmente l'attività, individuandone la data di inizio e fine.

CPM (Critical Path Method) → È adatto ai progetti che sono costituiti da una serie di attività singole, facilmente identificabili e definibili, in cui avremo individuato la logica sequenziale e le precedenze, divenendo una complessa rete di attività.

Il CPM può aiutare a capire: La durata totale del progetto, i percorsi critici e il tempo necessario per completare le attività che sono critiche, senza che questo comporti uno slittamento dell'intero progetto. Può inoltre aiutare a capire lo slittamento massimo che possono avere le attività non critiche.

Elementi caratterizzanti del reticolo CPM:
Early Start (ES) → È la data in cui “al più presto” un'attività può cominciare (con disponibilità di risorse, attività precedente completata senza ritardi etc). Per convenzione, l'ES della prima attività è uguale a 1.
ES = MAX(EF attività precedenti) + 1

Early Finish (EF) → È la data in cui “al più presto” una attività può essere completata, se le attività che la precedono non hanno ritardi.
EF = ES + durata dell'attività – 1

Late Start (LS) → È la data in cui “al più tardi” un'attività può iniziare senza causare ritardi sull'intero progetto.
LS = LF – durata delle attività che seguono + 1

Late Finish (LF) → È la data in cui "al più tardi" un'attività può essere conclusa senza causare ritardi sull'intero progetto.
LF = MIN(LS attività che seguono) – 1

Late Finish Time (LFT) = Early Finish Time (EFT) = Durata Totale

Procedimento:
Una prima fase in cui si sommano tutte le durate delle attività e si calcolano le date di inizio (ES) e fine (EF) al più presto e la durata del progetto. Si procede in avanti sommando da sinistra verso destra (dall'inizio verso la fine del progetto).

Slittamenti:
Le attività, così come sono state definite, possono subire degli slittamenti temporali, dovuti a molteplici fattori. Individuate le date di ES – EF – LS – LF per ciascuna attività è possibile individuare se c'è un margine di slittamento dell'attività rispetto alla durata complessiva del progetto e/o rispetto all'attività che segue. Non si tratta di ritardi, in quanto l'attività comunque dovrà terminare entro la LF.

Total Float (TF) → Margine temporale all'interno del quale un'attività può subire ritardi senza influenzare la data di completamento del progetto.
TF = LS – ES = LF –EF
Il Margine Totale o Scorrimento Totale (Total Float) rappresenta il massimo di periodo di tempo di cui può essere ritardata un'attività dall'Early Start (o dall'Early Finish), senza ritardare la durata totale del progetto.
Per le Attività critiche: LS = ES per cui TS = 0
Per queste attività un minimo ritardo nella data di fine determina un ritardo analogo sull'intero progetto (a meno di dispendiosi recuperi). Nelle attività non critiche: **LS = ES + TF.**

Free Float (FF) → Margine temporale all'interno del quale un'attività può subire ritardi senza influenzare l'ES delle successive.
FF = MIN(ES delle attività che seguono)–(EF dell'attività corrente)–1.

SLIDE 9 – GESTIONE DI PROGETTO

Il diagramma di Gantt → È uno strumento che permette di modellizzare la pianificazione delle attività necessaria alla realizzazione del progetto (1917 – Ingegnere statunitense).
È uno strumento di facile lettura universalmente usato da quasi tutti i capi progetto in tutti i settori.

Diagramma → Il diagramma è costruito partendo da un'asse orizzontale (a rappresentazione dell'arco temporale totale del progetto, suddiviso in fasi incrementali (giorni settimane mesi)) e da un asse verticale (a rappresentazione delle mansioni o attività che costituiscono il progetto)

In un diagramma di gantt ogni attività (o task) è rappresentata da una barra, mentre le colonne rappresentano i giorni o settimane o i mesi del calendario secondo la durata del progetto.

Diagramma di Gannt + WBS → È possibile associare ad ogni attività sull'asse verticale un riferimento alle WBS.
Le attività possono susseguirsi in sequenza o essere eseguite in parallelo.
Nel caso si susseguano in sequenza, si possono modellizzare delle relazioni di legame, così come visto per i diagrammi reticolari, attraverso una freccia che parte dall'azione a monte fino a quella finale.
Normalmente vengono aggiunte delle milestone sia di progetto che contrattuali.

Diagramma di Gannt – Specificazioni:
Il diagramma di Gannt rappresenta graficamente l'avanzamento del progetto stesso, ma è anche un buon mezzo di comunicazione tra i differenti attori di un progetto.
Questo tipo di modellizzazione è particolarmente facile da realizzare anche tramite excel ma esistono anche strumenti ad hoc come Microsoft Project.
È usato in tutte le fasi del progetto, in pianificazione perché fornisce una visione immediata e chiara del progetto, durante il monitoraggio e controllo perché permette di verificare l'avanzamento e lo slittamento delle attività in modo semplice.

Diagramma di Gannt – Vantaggi:
Un diagramma di Gantt permette la rappresentazione grafica di un calendario di attività.
E' utile al fine di pianificare, coordinare e tracciare specifiche attività in un progetto;

Fornisce una baseline della pianificazione;
Consente di evidenziare il percorso critico;
Consente di introdurre le Milestone del progetto;
Da una visione grafica dei possibili slittamenti delle attività;
Da una chiara illustrazione dello stato d'avanzamento del progetto rappresentato, confrontabile facilmente con la baseline;
Ad ogni attivit`a possono essere in generale associati una serie di attributi: durata (o data di inizio e fine), predecessori, risorsa, costo...
Rispetto ai diagrammi reticolari ha la calendarizzazione delle attività.

Diagramma di Gannt – Limiti:
Uno dei limiti di questo tipo di rappresentazione risiede nella difficoltà di individuare le interdipendenze tra le attività (graficamente può risultare difficile legare attività molto distanti).
Nei progetti di grande dimensione non `e immediata la lettura delle attività.

Documento di PROJECT PLAN → Documento di descrizione della pianificazione in cui vengono presentate le attività, i ruoli, il cronoprogramma di realizzazione.

Contiene:
Introduzione ed obiettivi;
Definizioni acronimi e abbreviazioni, riferimenti.
Organizzazione del progetto.
Piano di rilascio.
Cronoprogramma (diagramma di Gannt).

Organizzazione del progetto → Attività principali, matrice di responsabilità, reticolo di progetto.
Piano di rilascio → Quali deliverables (es manuale utente) verranno rilasciate nelle varie attività di progetto: Esegubili, database, documentazione.
Cronoprogramma → Le attività del progetto declinate temporalmente, meglio se anche con un diagramma di Gantt.

Documento di PROJECT PLAN → Documento molto importante! Destinato alla committenza! Esempio SLIDE 44 – 55

Infrastruttura tecnica di supporto
In ogni progetto è importante installare e mantenere una infrastruttura dove esistono:
Sorgenti, Documenti, Eseguibili.

Un ambiente riproducibile in cui lo sviluppatore:
può accedere privatamente ad una copia delle risorse di progetto, gestire sorgenti, creare eseguibili, eseguire test unitari.

Workspace → Può essere una semplice struttura di cartelle: Sorgenti, risorse, file di configurazione, documenti ed eseguibili.

Repository (SACRO) → Rappresenta il luogo dove vengono custoditi tutti i file di progetto riguardanti: Sorgenti, risorse (es immagini), file di configurazione, documenti, Non può essere una semplice struttura di cartelle, deve garantire il controllo delle versioni ed il lavoro collaborativo.
Dovrebbe essere su un server dedicato, ovvero il server con i repository non dovrebbe essere usato per compilazioni, esecuzioni e testing, od eseguire altri compiti simili a mail server o web server.

Ambienti di rilascio → Sono gli ambienti dove il prodotto viene testato e rilasciato.
Devono essere dimensionati considerando: Necessità di memoria, necessità di input/output da disco, capacità di banda di rete e necessità di spazio su disco.
Sono gli ambienti dove il prodotto viene testato e rilasciato:
Test → Ambiente interno alla organizzazione ove avverranno i test di integrazione e gli unit test.
Collaudo → Ambiente interno alla organizzazione ma visibile dal cliente, ove i responsabili potranno verificare lo stato di avanzamento dei lavori. Tale ambiente sarà analogo a quello di produzione, ovvero con le medesime caratteristiche di software e sistema operativo solo con un insieme di dati non necessariamente uguale a quello di produzione.
Produzione → Ambiente ove risiede il sistema che viene usato da tutti gli utenti.

Ambiente di rilascio:
Il codice prodotto dai singoli sviluppatori verrà portato su repository di progetto ad intervalli regolari, possibilmente entro i 3 giorni.
Il codice prodotto verrà compilato ed assemblato nell'ambiente di test dove verranno effettuate le prove di integrazione.
Dall'ambiente di test il sistema verrà trasferito, ad ogni significativo rilascio, all'ambiente di collaudo ove sarà a disposizione del cliente per i test di accettazione utente.
Ove i test diano esito positivo, si procederà al rilascio in ambiente di produzione.

SLIDE 10 – GESTIONE DEL RISCHIO
Il processo di sviluppo del software prende in considerazione principalmente elementi conosciuti.
Normalmente si descrive, progetta, pianifica quello che si sa che deve essere prodotto.
La gestione del rischio si occupa di quanto è sconosciuto.
L'idea alla base è prevenire i rischi prima che questi comportino un danno al progetto.

Risk denial mode → diverse organizzazioni lavorano in modalità "risk denial", ovvero ignorando l'esistenza di fattori imponderabili e pianificano e stimano come se questi non esistessero ad esempio su assunti quali.

Definizioni:
Rischio → Variabile che può assumere dei valori tali da compromettere il successo di un progetto.
Successo → si ottiene in un progetto quando si soddisfano l'insieme di tutti i requisiti ed i vincoli di progetto.
Rischio diretto → è tale se il progetto ha un forte controllo su di esso.
Rischio indiretto → è tale se il progetto ha un debole o nullo controllo su di esso.

Attributi di un rischio:
Probabilità che l'evento occorra e **Impatto** che si avrà sul progetto se l'evento occorre.
Grandezza rischio = **Probabilità X Impatto.**

Processo di gestione del rischio:
Analisi del rischio → Identificazione dei vari rischi per il progetto. Per ognuno stima la probabilità e la gravità del danno per ogni rischio.
Decisione su come gestire → Una volta determinata la grandezza di un rischio. Per ognuno decidere se evitarlo, trasferirlo, accettarlo.
Esecuzione delle azioni di gestione → Identificazione dei responsabili per il compimento delle varie attività, Esecuzione delle azioni.
Monitoraggio → Deve essere eseguito il controllo, altrimenti il rischio non è gestito. Il processo non termina, si devono controllare e gestire i rischi fino alla fine del progetto.

L'analisi del rischio:
Identificazione di tutti i rischi del progetto → Si inizia con una lista di possibili aree di rischio. Poi si analizza ogni area di rischio per evidenziare quanto riguarda il progetto.
Stima della probabilità → Frequente (0,7 → 1) , probabile (0,4 → 0,6) , improbabile (0,0 → 0,3).
Stima della dannosità → Elevata 100, significativa 80, media 50, minore 30, trascurabile 10.

Tipologie di rischio:
- **Risorse →** Rischi di organizzazione, rischi finanziarie, rischi relativi alle persone.
  Rischi di Organizzazione → C'è sufficiente interesse nel progetto da parte di tutte le parti interessate?
  Questo è il progetto più grande affrontato finora?
  C'è un processo di sviluppo ben definito?
  Rischi Finanziari → C'è copertura finanziaria per tutto il progetto? Ci sono vincoli economici (con rischio di cancellazione) sul progetto? Sono affidabili le stime dei costi?
  Rischi relativi alle persone → C'è abbastanza personale per svolgere tutto il progetto? Chi lavorerà al progetto dispone di tutte le conoscenze che servono? Chi lavorerà al progetto ha già lavorato assieme. Chi lavorerà al progetto ha fiducia nella riuscita? C'è disponibilità di esperti di dominio e rappresentanti degli utenti?.
- **Businnes →** Cosa succede se un concorrente arriva sul mercato prima di noi? Con il finanziamento attuale del progetto, quello che verrà prodotto ha (ancora) un valore tangibile? Il valore che porterà il progetto è maggiore del costo di produzione?.
- **Tecnica →** Ambito, tecnologia, dipendenze esterne

Ambito → Come si può misurare se il progetto ha avuto successo? Sono tutti gli stakeholder concordi su come determinare se il progetto ha avuto successo? I requisiti sono stabili e facilmente comprensibili? L'ambito del progetto si espande col tempo o rimane stabile?
Tecnologia → Si sta usando una tecnologia consolidata? Sono ragionevoli gli obiettivi di riuso? Sono ragionevoli, con la tecnologia scelta, i requisiti di numerosità e frequenza delle transazioni o di volume dei dati? Il successo dipende dall'utilizzo di prodotti o tecnologia mai provate prima? Se si usano tecnologie o prodotti di terze parti, sono disponibili le interfacce verso questo? Ci sono vincoli di affidabilità o sicurezza *estremamente inflessibili?* Il progetto richiede il multilinguismo? Il sistema è davvero realizzabile?
Dipendenze esterne → Lo sviluppo del progetto dipende da altri progetti sviluppati in parallelo? Il successo dipende dall'utilizzo di tecnologie o prodotti di terze parti?.

- **Tempo →** La pianificazione è realistica? Quanto è critica la data di rilascio? C'è tempo per fare le cose bene? C'è la possibilità di semplificare le funzioni per rispettare la schedulazione? In alcuni progetti la data di rilascio è *critica.* Per esempio un sistema che deve essere preparato per un evento, oppure qualcuno sta sviluppando la stessa cosa in concorrenza con te. Raramente accade, di solito ha impatto sui costi di progetto. Statisticamente, l'85% dei rischi ha un impatto diretto o indiretto sul tempo di rilascio (e quindi sui costi), circa il 5% ha un impatto sui costi e la rimanenza sulla qualità del software.

<u>Strategie di gestione del rischio</u>

Evitare il rischio → Riorganizzare le attività in modo che il progetto non sia soggetto a tale rischio. Non è sempre attuabile, ma è una buona strategia. Spesso molti rischi infatti derivano da una inadeguata delimitazione del progetto. Eliminando o ridimensionando i requisiti non essenziali si tolgono automaticamente dalla lista molti rischi, non ultimo la mancanza di risorse per completare il progetto.
Trasferire il rischio → Riorganizzare le attività in modo che a tale rischio sia assoggettato qualcun altro. Per esempio la polizza assicurativa.
Accettare il rischio → Decidere di convivere con il rischio. Quando un rischio viene accettato si può:
Mitigare il rischio → Effettuare delle azioni preventive ed immediata per ridurre la probabilità o l'impatto del rischio. Oppure eseguire delle azioni che tendono a portare a zero la probabilità che l'evento si verifichi. Se il rischio è di tipo "X potrebbe non funzionare" si affronta X il prima possibile. Se il rischio è del tipo "La tecnologia Y non `e conosciuta dal gruppo di lavoro" bisogna fare il prima possibile formazione strutturata su Y.
Definire un piano di emergenza → Che si metterà in atto se il problema si verifica. Anche se si è pianificato di mitigare un rischio, è necessario definire quali azioni devono essere eseguite se il rischio si materializza. Viene anche comunemente conosciuto come "Piano B".

Il piano di emergenza → dovrebbe considerare il rischio (descrizione del rischio), gli indicatori del rischio (descrizione di come ci si accorge che l'evento dannoso si è verificato, come si sa che il rischio è diventato realtà?), le azioni (descrizione di cosa fare se il rischio si è verificato).

<u>Documento di RISK LIST:</u>
Documento di descrizione dei rischi, della politica di gestione dei medesimi, delle azioni per la mitigazione e dei piani di emergenza.
Normalmente contiene → Introduzione ed obiettivi, Lista dei rischi, Gestione del rischio (per ogni rischio).
Introduzione ed obiettivi:
Obiettivi → Obiettivi del progetto, sinteticamente e con linguaggio naturale.
Glossario → Glossario di progetto, o riferimento al glossario ed altri documenti del progetto.
Lista dei maggiori rischi:
I dieci rischi più pericolosi.
Nome del rischio (nome identificativo del rischio), Grandezza (la grandezza del rischio (probabilità x danno)), Descrizione (descrizione in linguaggio naturale del rischio).
Gestione del rischio (per ogni rischio):
Per ogni rischio identificato:
Nome del rischio, Grandezza, Descrizione, Impatto, Strategia di mitigazione, Contingency plan.

Esempio documento di rischio slide 58 → 64

<u>Gestione dei Cambiamenti</u>
La necessità di cambiamento è connaturata con la creazione e la manutenzione di un software che viene usato. I requisiti cambiano nel tempo, si aggiungono nuove esigenze, quanto rilasciato non corrisponde esattamente a quanto richiesto.

Il processo di gestione del cambiamento si occupa di organizzare queste situazioni.

In piccoli progetti la **gestione dei cambiamenti** in piccoli progetti si può mantenere su di una semplice lista o foglio elettronico. In sistemi complessi si usano sistemi appositi → defect-tracking system.

la **gestione delle richieste di cambiamento** permette di assicurare che i cambiamenti avvengano in maniera controllata.
l'elemento base di questo processo viene chiamato "**Change Request**".

> **Enhancement Requests** vengono usate per richiedere nuove features al prodotto.
> **Defects Reports** vengono usate per richiedere sistemazioni di anomalie o difetti nel prodotto rilasciato (Bug, Mancanze, Usabilità).

Verbali di riunione → Verbali di condivisione con utente. Necessari a formalizzare la ripianificazione delle eventuali nuove richieste o variazioni.

Change request → Documento in cui formalizzare le segnalazione di nuove richieste o la variazione di richieste già espresse, piuttosto che la segnalazione di bugs o difetti.

SLIDE 11 – SCALABILITÀ
La scalabilità → La scalabilità rappresenta la capacità di gestire un aumento del carico di lavoro, applicando ripetitivamente una strategia efficace di aumento delle risorse per ottenere un incremento nella capacità di servizio. La caratteristica principale di un'applicazione scalabile è costituita dal fatto che un carico aggiuntivo richiede solamente risorse aggiuntive anziché un'estesa modifica dell'applicazione stessa.

Scalabilità != Prestazioni → Nonostante il livello di prestazioni influisca sulla definizione del numero di utenti che l'applicazione è in grado di supportare.
In effetti, le operazioni effettuate per migliorare le prestazioni possono talvolta infulire negativamente sulla scalabilità.

Scalabilità di due tipi:
Scalabilità verticale → Scaling up → viene utilizzato per indicare il processo in cui la scalabilità viene realizzata tramite l'upgrade dell'hardware migliore, più veloce e più costoso.
Scalabilità orizzontale → Scaling out → l'aggiunta di server, vista la convenienza offerta dall'utilizzare hardware per PC di largo consumo per distribuire il carico di elaborazione tra più server.

Scalabilità → La scalabilità dell'applicazione richiede un'equilibrata combinazione dei due distinti domini del software e dell'hardware.
Si possono fare grandi progressi e aumentare la scalabilità in uno di questi domini per poi vanificarli a causa degli errori commessi nell'altro.
Ad esempio, la creazione di una Web farm di server con bilanciamento del carico non fornirà alcun beneficio per un'applicazione progettata per essere eseguita su un singolo computer.
Analogamente, progettare un'applicazione a elevata scalabilità e distribuirla su computer connessi a una rete a bassa larghezza di banda non consentirà di gestire carichi elevati in modo efficiente quando l'ingente traffico determina una saturazione della rete.

Impatto sulla scalabilità: Progettazione → ottimizzazione codice → ottimizzazione middelware → ottimizzazione hardware. Più saliamo, minore è l'impatto della scalabilità.

<u>Principi della progettazione finalizzata alla scalabilità:</u>
Eliminazione delle attese → Un processo non deve restare in attesa oltre il necessario. Ogni intervallo di un tempo in cui il processo utilizza una risorsa rappresenta un intervallo di tempo in cui quella risorsa non è disponibile in un altro processo. Un modo per ottenere la scalabilità consiste nell'eseguire in modo asincrono . Quando vengono eseguite in modo asincrono, le operazioni ad esecuzione prolungata vengono accodate per essere completate successivamente da un processo distinto.

Eliminazione della contesa delle risorse → La contesa delle risorse è la causa principale di tutti i problemi di scalabilità. L'insufficienza di memoria, cicli di processore, larghezza di banda o connessioni di database rispetto alle esigenze, ostacola infatti la scalabilità dell'applicativo. Le risorse devono essere acquisite il più tardi possibile e rilasciate il prima possibile.

Progettazione della commutabilità → La progettazione della commutabilità rappresenta uno dei

meccanismi di riduzione della contesa delle risorse più sottovalutati. Due o più operazioni vengono definite commutabili quando determinano lo stesso risultato indipendentemente dall'ordine in cui vengono eseguite. In genere questo meccanismo può rivelarsi opportuno per le operazioni che possono essere eseguite senza transazioni.

Progettazione dell'intercambiabilità → Quando può essere generalizzata, una risorsa può essere intercambiabile. Quando invece si aggiunge uno stato dettagliato a una risorsa, questa risulta meno intercambiabile. Una configurazione che richiede il mantenimento dello stato di parte dei componenti tra le chiamate del metodo impedisce l'intercambiabilità e influisce negativamente sulla scalabilità.

Partizione delle risorse e dell'attività → La partizione delle attività consente di ridurre il carico assegnato alle risorse più costose. Un'altra opportunità di partizionare le attività viene fornita dalle transazioni. Ad esempio, separando i metodi che richiedono transazioni da quelli che non ne richiedono.

Riuso del software:
Il riuso è il tipico approccio ingegneristico alla costruzione di sistemi. I Sistemi sono spesso il risultato di integrazione di sottosistemi spesso non "banali".

Vantaggi del riuso:
Riusare software esistente può consentire di → Ridurre i costi di sviluppo, ridurre i costi di testing, ridurre i costi di manutenzione, ridurre i tempi di rilascio, ridurre il rischio nella produzione, aumentare la qualità del software.

Il riuso è applicabile ai manufatti → Riuso di intere applicazioni, riuso di componenti, riuso di oggetti e librerie.
Ma anche a livello concettuale → Riuso della documentazione di analisi, riuso di documentazione di pianificazione, riuso di test.

Problemi legati al riuso:
Maggiori costi di manutenzione → Se si riusa software di cui non si possiede il codice, il sistema complessivo potrà risultare meno flessibile e meno manutenibile.
Mancanza di strumenti CASE → Spesso i CASE non forniscono funzionalità che consentono un efficace riuso delle librerie di componenti software.
Sindrome NIH → "not invented here" → diffidenza verso software prodotto da altri → Spesso si preferisce ri sviluppare per avere un maggior controllo sul progetto.
Difficoltà nel mantenere e creare librerie di componenti riusabili → Popolare librerie di componenti riusabili può essere costoso, e le tecniche disponibili per classificare, catalogare e ricercare componenti software non ancora mature.
Difficoltà nel trovare, comprendere ed adattare componenti riusabili → Non sempre si è disposti a spendere tempo per cercare, comprendere ed eventualmente adattare un componente riusabile.

Fattori di cui tener conto nel pianificare il riuso → Tempistica richiesta per lo sviluppo, Durata prevista per la vita del software, Background, capacità ed esperienza del team di sviluppo, Criticità del software e altri requisiti non funzionali, Piattaforma sulla quale eseguire il sistema.

Diversi Approcci possibili per il Riuso
Design patterns → COTS Integration → Component-based development → Generatori di programmi.

Design pattern:
Pattern → descrive un problema che ricorre spesso e propone una possibile soluzione in termini di organizzazione di classi/oggetti che generalmente si è rilevata efficace.
Rispetto ai componenti riusabili → Non è un oggetto fisico, non può essere usato così come è stato definito, ma deve essere contestualizzato all'interno del particolare problema applicativo. Due istanze/contestualizzazioni di uno stesso pattern (ad esempio in problemi diversi) tipicamente sono diverse proprio per la contestualizzazione in domini differenti.

Design Pattern **sono caratterizzati** da quattro elementi principali:
Nome → riferimento mnemonico che permette di aumentare il vocabolario dei termini tecnici e ci permette di identificare il problema e la soluzione in una o due parole.
Problema → descrizione del problema e del contesto a cui il pattern intende fornire una soluzione.
Soluzione → descrive gli elementi fondamentali che costituiscono la soluzione e le relazioni che intercorrono tra questi.
Conseguenze → specifica le possibili conseguenze che l'applicazione della soluzione proposta può

comportare.

Caratteristiche design pattern → Un Design Pattern Nomina, Astrae, e Identifica gli aspetti chiave di una struttura comune di design che la rendono utile nel contesto del riuso in ambito object-oriented.
Design pattern identifica → Le classe e le istanze partecipanti, le associazioni ed i ruoli, le modalità di collaborazione tra le classi coinvolte, la distribuzione delle responsabilità nella soluzione del particolare problema di design considerato.

Riuso basato su Generatori → Un generatore è un software che genera software parametrizzato in base alle specifiche fornite dell'utente. I generatori possono essere utilizzati nell'ambito di quei problemi per i quali esistono soluzioni ben consolidate che però dipendono notevolmente dai dati in ingresso. I dati in ingresso al generatore di programmi vanno a descrivere la conoscenza relativa al dominio per il quale debba essere utilizzato il programma da generare.

Esempi di generatori di codice : Generatori di applicazioni per gestione dati aziendali; Parser e analizzatori lessicali per analisi di codice: l'input è una grammatica del linguaggio da analizzare; l'output è l'analizzatore del linguaggio; ORM: Object Relational Mappers ; Lex&Yacc (per codice C) e JavaCC (per codice Java).

Vantaggi e svantaggi dei Generatori:
Il riuso basato su generatori riduce notevolmente il costo di sviluppo e produce codice molto affidabile. Tramite generatori di codice si possono ottenere programmi più versatili e performanti di quanto si possa ottenere limitandosi a leggere direttamente dal database, a tempo di esecuzione, i dati relativi alla personalizzazione. Non tutti i generatori di codice, però, sono in grado di generare codice che sia anche efficiente. Scrivere una descrizione di dominio per un utente programmatore è più semplice che sviluppare programmi da zero (anche se lo skill richiesto non è minimo). La loro applicabilità si limita a poche tipologie di problemi.

SLIDE 12 – RIUSO, COMPONENTI
Application Framework → Rappresentano modelli astratti di progetto di sottosistemi. Le applicazioni si costruiscono integrando e completando una serie di framework. Per esempio gli object oriented framework, che sono composti da una collezione di classi astratte e concrete e di interfacce tra loro. Un framework object oriented è una struttura generica, per realizzare un software bisogna riempire le parti di progetto instanziando le classi astratte necessarie, ed implementando le classi mancanti (per esempio Android).
Gli application framework inoltre si differenziano dai design patterns per il fatto di essere astrazioni a livello più alto, cioè a livello architetturale invece che di design.

Classificazione di framework:
Infrastruttura di sistema → Supportano lo sviluppo di infrastrutture come comunicazione, interfacce utente e compilatori.
Integrazione di middleware → Composti da standard e classi di oggetti per la comunicazione e lo scambio di informazioni tra oggetti (es corba, java beans, com+).
Applicazioni aziendali → Integrano la conoscenza per specifici domini di applicazione (per esempio telecomunicazioni o sistemi finanziari) e supportano lo sviluppo di nuove applicazioni.
I primi due tipi rientrano nella categoria di dei Framework orizzontali, mentre le applicazioni aziendali si considerano framework verticali.

Framework
Lo sviluppatore non scrive codici per coordinare le componenti.
Lo sviluppatore deve determinare le componenti che aderendo alla logica collaborativa del framework verranno coordinate da quest'ultimo.
I framework assumono il controllo dell'applicazione e non il contrario.
L'utilizzo di un framework da parte di programmatori e progettisti comporta il raggiungimento di una notevole abilità riguardante la conoscenza del framework stesso e delle opportunità messe a disposizione. Spesso è necessario molto tempo prima di poter maturare tali conoscenze.

Esempio – model view controller
È un framework usato per la progettazione GUI.
Permette presentazioni diverse di uno stesso oggetto e fornisce interazioni separate con queste presentazioni. **MVC** richiede l'instanziazione di vari design patterns (per esempio Observer, strategy, compose).

Utilizzo del model view controller:
L'utente interagisce con la UI scatenando eventi (x es click su di un elemento della UI). Il Controller interagisce gestendo l'evento tramite un Handler od un Callback. Il Controller modifica la richiesta utente al Model, causando un eventuale cambiamento di stato del Model. La View interroga il Model per generare una nuova interfaccia utente.

Riutilizzo di intere applicazioni
Consiste nel riutilizzo (previa eventuale riconfigurazione o personalizzazione) intere applicazioni.
Le applicazioni possono essere riutilizzate direttamente od essere integrate tra di loro come componenti all'interno di più ampi sistemi.
Esistono due approcci principali → Integrazione di **COTS** (Commercial Off-The-Shelf- Software) e **Sviluppo di linee di prodotti.**

COTS → Commercial Off-The-Shelf- Software che può essere usato dai suoi acquirenti senza modifiche (per esempio soluzioni desktop o prodotti server). Un esempio lampante sono di DBMS.
Si tratta di applicazioni complete che spesso offrono un API (Application program interface) per permettere ad altri componenti software di accedere alle proprie funzionalità.
Nella pratica i COTS sono quasi sempre composti da un insieme di classi astratte di interfaccia visibili all'esterno e di un insieme di classi astratte e concrete non visibili.

Esempio di riuso di COTS → Sviluppo dei sistemi (es sistema di e-commerce + sistema di fatturazione + posta elettronica) e tramite adapters posso collegarli gli uni agli altri in modo di creare un sistema composto da più sottosistemi.

Riuso di COTS – problemi:
Mancanza di controllo sulle funzionalità e sulle prestazioni.
Problemi di interoperabilità tra sistemi COTS.
Nessun controllo sull'evoluzione del sistema.
Supporto dei produttori COTS.
I software open source sono COTS?

Linee di prodotti software → Si intendono famiglie di applicazioni con funzionalità generiche che si prestano ad essere configurati o adattate in modo da poter essere utilizzate in contesti specifici.
Esempio di adattamento:
Specifiche configurazioni di sistema e dei componenti.
Aggiunta di nuovi componenti.
Selezioni di componenti nell'ambito di una libreria.
Modifiche ai componenti per adattarsi alle esigenze del contesto.

Linee di prodotti software: gli ERP → Enterprise resource planning. È un sistema generico che supporta comuni processi aziendali (quali gestioni di ordini, fatture, inventari, paghe) es SAP e BEA.
Il processo di configurazione degli ERP si basa sull'adattamento di un core generico attraverso l'inclusione e la configurazione di moduli, ed incorporando conoscenza sui processi e le regole aziendali del cliente specifico in un database di sistema.
Molto usati in grandi aziende, costituiscono la forma di riuso più comune.

Progettazione a componenti
Nel contesto delle architetture software esistono più accezioni per il termine “componente software” tra cui:
Accezione astratta e generica → Nelle architetture software ci sono due tipi principali di elementi software: Componenti (responsabili dell'implementazione di funzionalità e gestione dei dati) e connettori (responsabili delle interazioni tra i componenti).
Accezione tecnologica → Le tecnologie a componenti rappresentano un'evoluzione delle tecnologie ad oggetti distribuiti.
Accezione metodologica → Lo sviluppo di un software basato su componenti è un approccio alla costruzioni di grandi sistemi software basato sullo sviluppo e sull'integrazione di componenti software.

Componente → Un programma che è capace di svolgere qualche compito ed è concepito in modo da poter funzionare in diversi ambienti ed essere agganciato ad altri componenti per creare delle applicazioni complete.
Per ottenere questo risultato i componenti devono poter funzionare assieme ed essere compatibili, adattarsi cioè ad uno standard comune, quello che in termini tecnici si chiama un'architettura di componenti.

Componente:
Enità software di runtime.
Implementa un insieme di funzionalità.
Offre i suoi servizi mediante un sistema di interfacce con nome (interfacce fornite).
Può richiedere servizi ad altri componenti (sempre sulla base di interfacce note, le interfacce richieste).

Software basato su componenti → Applicazione formata da un insieme di componenti che vengono composti al momento del rilascio sulla base delle loro interfacce.

Tecnologie a componenti → Rappresentano un'evoluzione della tecnologia ad oggetti distribuiti. Un componente è un'entità software di runtime → Implementa un insieme di funzionalità, offre i suoi servizi mediante un insieme di interfacce con nome (interfacce fornite), può richiedere servizi ad altri componenti, sempre sulla base di interfacce con nome (interfacce richieste).
In pratica un'applicazione è formata da un insieme di componenti che vengono composti (al momento del rilascio) sulla base delle loro interfacce. Per esempio CORBA, COM, DCOM, .NET, JAVA EE.

Progettazione a componenti:
Un componente si definisce grazie alle sue interfacce, quelle richieste e quelle fornite.
Ci sono diverse nozioni legate ai componenti. La parola "**componente**" può essere riferita ad alcune di queste nozioni, i cui significati sono correlati ma diversi:
Standard per i componenti,
Specifica di componente,
Interfaccia di componente,
Implementazione di componente,
Componente installato,
Componente oggetto.

Standard di componenti (o modelli di componenti):
Un modello di componenti si definisce uno standard da seguire per:
L'implementazione di un componente, documentazione e impiego (deploy).

Piattaforma per componenti:
Un ambiente, o piattaforma, di esecuzione fornisce ai componenti i servizi definiti dal relativo standard. L'ambiente di esecuzione funge da contenitore per i componenti eseguibili. Per esempio un SO Windows per .NET.

Specifica e interfaccia di componenti → La specifica di un componente definisce in modo preciso il comportamento di un certo componente. Quindi, le modalità di interazione del e con il componente. Gran parte della specifica di un componente consiste nella definizione delle interfacce del componente.

Implementazione di un componente → Un'implementazione di un componente è la realizzazione software di una specifica di componente.
Si tratta di un componente autonomo, che può essere distribuito e installato in modo (possibilmente) indipendente da altri componenti. La separazione netta tra specifica e implementazione è una caratteristica fondamentale dei componenti. Per una certa specifica di componente, possono esistere più implementazioni di componenti diverse che la realizzano. Può essere codice sorgente, codice binario, un file di distribuzione quale un archivio compresso.

Componente installato → È una copia di implementazione di un componente che è stata installata / rilasciata (deployed) su di un particolare computer / ambiente di esecuzione. L'installazione avviene su di un ambiente di esecuzione (l'installazione comprende la configurazione e la registrazione del componente in tale ambiente). Un'implementazione di componente può dare origine a più componenti installate.

Componente oggetto → Per componente oggetto si intende un'istanza creata a partire da componente installato, nel contesto di un ambiente di esecuzione. I componenti oggetto esistono solo durante l'esecuzione (quindi a runtime). Sono solo i componenti oggetto ad essere fisicamente in grado di offrire servizi concretamente. I componenti oggetto hanno un'identità univoca nonché un proprio stato, ovvero dei dati. Un componente installato può avere zero, uno o più componenti oggetto da esso creati.

Componenti e contenitori → Le tecnologie a componenti sostengono lo sviluppo e la gestione di componenti software che possono essere rilasciati, composti ed eseguiti entro ambienti di esecuzione specializzati, chiamati contenitori.

Contenitore → Ambiente runtime lato server per la gestione di componenti. Un contenitore estende le capacità di un broker.
Componenti → Per essere eseguiti devono essere configurati, assemblati e rilasciati in un contenitore.
Il contenitore → Si occupa del ciclo di vita dei componenti in esso rilasciati. Fornisce inoltre ai suoi componenti servizi come sicurezza, transazioni, persistenza e sostiene qualità come sicurezza, affidabilità, prestazioni e scalabilità.

SLIDE 13 – UML (UNIFIED MODELING LANGUAGE)
UML → una notazione / standard industriale OMG (Object management group) per:
Modellare un ambito aziendale, Esprimere i requisiti software, Esprimere l'architettura software, Esprimere la struttura ed il comportamento del software, Documentare l'operatività del software, Riferimento industriale internazionale, Tecnologia di riferimento per IBM.

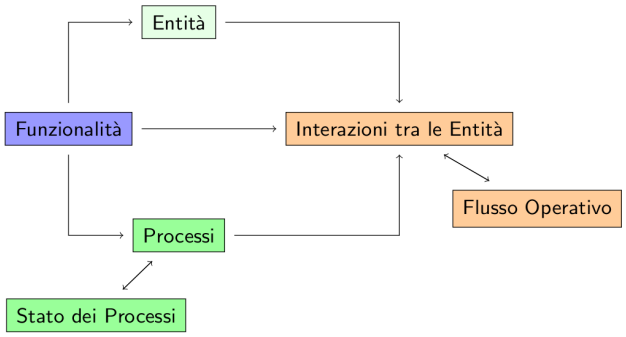
UML → È un sistema di notazioni grafiche (con sintassi semantica e pragmatica predefinite) per la modellazione Object Oriented di sistemi software. UML non è un processo né è una notazione proprietaria. È uno standard OMG definito mediante un metamodello detto infrastruttura UML. UML include:
Viste
Diagrammi
Elementi di modellazione

UML → È un linguaggio unificato per la modellazione di concetti, entità, funzionalità, processi e relazioni che intercorrono tra essi. UML serve a descrivere in modo grafico e compatto:
I requisiti utente, Le componenti dei sistemi, I dati in essi contenuti, Le azioni da essi svolte, Le relazioni che tra loro intercorrono (il processo in cui essi operano).

UML → È un linguaggio di modellazione semigrafico interpretabile da tecnici, non tecnici e macchine. Un modello è una semplificazione della realtà che si ottiene riducendo le caratteristiche in esame e considerando solo quelle utili al fine dell'analisi in corso. Garantisce una forte potenza espressiva nella documentazione.

UML diagrammi principali → I principali tipi di diagrammi componenti una specifica UML, in tutte le versioni, sono:
Diagramma dei casi d'uso (Per capire cosa il sistema deve fare)
Diagramma delle classi (Per definire le entità fondamentali)
Diagramma degli stati(Per definire i processi fondamentali (fortemente imparentati con USE CASE PROGRAM)
Diagramma di Sequenza (Per definire la sequenza delle interazioni fra entità)
Diagramma di collaborazione (per definire le interazioni tra entità fondamentali).

UML – Il legame tra i diagrammi principali.



Diagrammi comportamentali

Definizione formale di USE CASE → Sequenza di transazioni eseguita da un attore in interazione con il sistema, la quale fornisce un valore misurabile per l'attore.

Use case diagram → Praticamente:
Sequenza di transazioni in dialogo con il sistema, Comporta sempre uno o più attori, Rappresenta COSA (non come) il sistema offre l'attore, Mappato alle attività di businnes.

ESEMPIO USE CASE DIAGRAM SLIDE 34

Diagramma delle attività → I diagrammi delle attività(activity diagram) possono essere usati per descrivere il comportamento nel tempo di un particolare elemento come:
Rappresentano una procedura od un workflow.
Mostrando l'evoluzione di un flusso di attività
Ogni attività è definita come un'evoluzione continua, non necessariamente atomica, di un'attività.
Sono un'evoluzione di flow -chart.

Diagramma delle attività → Enfasi posta sulle attività non su chi le compie.
Enfasi sulla sequenza di azioni di una particolare procedura.
Vengono evidenziati vincoli di precedenza o di concorrenza.

Simboli diagramma di attività (slide 42)
Esempio SLIDE 42 → 47

Diagramma degli stati → StateChart → possono essere usati per descrivere il comportamento nel tempo di un particolare elemento come:
Un oggetto, Un intero sottosistema, l'evoluzione di un iterazione, descrivono sequenze di stati ed azioni attraverso cui l'elemento considerato passa durante la propria vita.
Si possono considerare al contrario dell'Activity Diagram. L'enfasi è posta sugli stati e non sulle azioni.

Simboli diagramma degli stati. Esempio slide 54 → 58
Diagramma delle classi :
Rende evidenti gli aspetti statici del modello di codice. Modella la relazione fra entità del sistema rappresentate come classi. Le classi possono avere relazioni fra loro, rappresentate con le associazioni. Per default, un'associazione è bidirezionale, anche se può essere resa monodirezionale.

Le classi → Una classe ha delle caratteristiche che la descrivono (attributi), ha delle elaborazioni che vengono eseguite (metodi).
Una classe rappresenta una entità di businnes coinvolta nell'elaborazione
L'individuazione delle classi consente di identificatore chi fa cosa e come all'interno del sistema. La struttura delle classi relative ad un processo può essere confrontata con la definizione di strutture efficienti per la realizzazione (pattern).
Una classe può rappresentare un oggetto concreto (un tornio) oppure immateriale (Iva) o un'attività (spedizione).
Una classe rappresenta una definizione applicabile ad un insieme di elementi omogenei (oggetti).

Simboli diagramma delle classi:
Classe generica → Un quadrato / rettangolo col titolo in grassetto.
Attributo privato → Un meno davanti l'attributo. (vale anche con i metodi).
Attributo pubblico → Un più davanti l'attributo. (vale anche con i metodi).
Classe astratta → Un quadrato / rettangolo con titolo in italico.

Associazioni nel diagramma delle classi:
Si indicano con una linea che collega le due classi.
La cardinalità di un'associazione esprime il numero di oggetti di una certa classe che prendono parte all'associazione.
Si esprime con un numero od un range disegnati vicino alle classi.
L'associazione può avere un nome.
Simile alla cardinalità di una relazione in basi di dati.
La specializzazione si indica con una freccia verso la classe specializzata.

Esempio di diagramma delle classi → Slide 73 / 74

Diagramma dei package: Il diagramma dei package è un diagramma di struttura che mostra i packages e le relazioni tra questi. Un package è uno spazio dei nomi utilizzato per raggruppare gli elementi che sono in relazione semantica tra loro che possono subire cambiamenti insieme.
Un Package è uno strumento di organizzazione che permette di raggruppare elementi per meglio strutturare il modello del codice.
Un Package può a sua volta essere incluso in un altro package.
Un Package ha elementi che hanno un nome univoco all'interno del package se sono del medesimo tipo; elementi posso avere lo stesso nome se sono di tipo diverso.

Un Package viene rappresentato con un rettangolo con etichetta, questa in alto a sinistra.

Esempio slide 79/80

Diagramma delle sequenze:
I diagrammi delle sequenze (sequence diagram) descrivono le interazioni fra gli oggetti organizzate in sequenza temporale.
Uno use case contiene al suo interno vari diagrammi di sequenza.
Salvo casi banali, non si rappresentano all'inizio tutte le possibili sequenze, ma solo le principali.
Gli elementi costitutivi di un diagramma di sequenza sono gli Oggetti e i Messaggi attraverso cui essi interagiscono.
Lo scambio di messaggi è rappresentato da frecce con un nome.

Esempio slide 86/87

SLIDE 14 – DESIGN PATTERN
Design pattern:
La programmazione ad oggetti può essere anche più ad alto livello di quella procedurale, ma ci si è resi conto subito che è difficile non tanto perché sia difficile scrivere "buoni" oggetti, quanto perché la comunicazione tra gli oggetti può creare delle situazioni complicate e nuove rispetto alla programmazione procedurale.

Problemi comuni:
Un gruppo di 4 programmatori (la banda dei 4, GoF) ha cercato di formulare una lista dei problemi più comuni e delle relative soluzioni e così nel 1994 sono nati i Design Pattern.

Design pattern:
Sono suddivisi in base al loro scopo:
Creazionali: Propongono soluzioni per creare oggetti.
Comportamentali: Propongono soluzioni per gestire il modo in cui vengono suddivise le responsabilità delle classi e degli oggetti.
Strutturali: Propongono soluzioni per la composizione strutturali di oggetti e classi.

Creazionali:
Singleton → Una classe contiene delle variabili globali e deve essere accessibile da parte di tutti gli altri oggetti che vogliono accedere a queste variabili globali. Questo è un problema ricorrente chiamato **singleton.** Per esempio java.lang.Runtime → getRuntime() , java.awt.Desktop → getDesktop(). Di questo oggetto nessuno può fare la new, posso ottenere una referenza tramite una funzione get ma non crearne una nuova. Esempio slide 26!

Prototype → Specifica i tipi di oggetti da creare, utilizzando un'istanza prototipo, e crea nuove istanze tramite la copia di questo prototipo.
Il "prototype" pattern si basa sulla clonazione di oggetti utilizzati come prototipi.
Questi oggetti devono implementare un'interfaccia che offre un servizio di copia dell'oggetto, e che sarà quella di cui il framework avrà conoscenza al momento di dover creare nuovi oggetti.
In Java la implementazione del metodo clone() della interfaccia Cloneable.
Per esempio:

```
ClasstimeClass = objectT ime.getClass();
ConstructortimeClassConstr = timeClass.getConstructor(newClass[]);
return(T ime)timeClassConstr.newInstance(newObject[]);
```

Si ottiene un oggetto della classe java.lang.Class che rappresenta il tipo di oggetto, si crea un'istanza di java.lang.reflect.Constructor associato, si fa un'invocazione al metodo new Instance sull'oggetto java.lang.reflect.Constructor.

Factory → (Detto anche Simple o Concrete Factory) è una semplificazione molto diffusa di Abstract Factory.
Problema → Chi deve essere responsabile di creare gli oggetti quando la logica di creazione è complessa e si vuole separare la logica di creazione per una coesione maggiore?
Soluzione → Si crea un oggetto "Pure Fabrication" chiamato Factory che gestisce la creazione.
Esempio pg 34 slide

Abstract Factory →
Problema → Come creare diverse famiglie di classi tra loro correlate che:
nascondono le loro classi concrete , facilitando l'aggiunta di nuove famiglie , assicurano la creazione e gestione di famiglie consistenti di oggetti?
Soluzione → Si crea una interfaccia per ogni tipo di classe Le classi concrete implementano tali interfacce Il Client usa solo le interfacce dei prodotti . Ogni famiglia ha una Factory usata dal Client per creare le istanze. Tutte le Factory implementano una interfaccia comune (Low Coupling tra Client e le specifiche Factory).
Esempio slide 36 → 38.

Builder:
Problema → Separare la logica della costruzione di un oggetto complesso dalla costruzione stessa: nascondere le loro classi concrete , facilitando l'aggiunta di nuove parti invece di distribuire la responsabilità della creazione tra i vari oggetti (un oggetto crea un altro oggetto che crea altri oggetti ecc.) è più conveniente localizzare in un unico punto la logica di creazione dell'intera struttura.
Soluzione → Si crea una classe astratta che definisca il "Builder" , le classi concrete implementano i singoli "Builder" una classe "Director" che definisce quali siano i "Builder" concreti da i stanziare.

Il "Builder" è un design pattern simile all' Abstract Factory pattern. Nel caso dell' Abstract Factory, il client usa i metodi della factory per creare i propri oggetti. Nel Builder, la classe "Builder" viene istruita su come creare un oggetto e poi viene invocata, ma il modo in cui crea l'oggetto è specifico della classe "Builder" medesima, ed è questo il dettaglio che definisce la differenza tra i due pattern.

<u>Strutturali:</u>
Adapter → Permette a una classe di supportare un'interfaccia anche quando la classe non implementa direttamente quell'interfaccia . Per risolvere il problema si definisce una classe intermedia, detta Adapter, che serve ad accoppiare l'interfaccia con la classe facendo da tramite tra le due. I partecipanti al pattern dell'Adapter sono target (è l'interfaccia da implementare), Adaptee (è la classe che fornisce l'implementazione), Adapter (è la classe intermedia che implementa Target e richiama Adaptee).

Gli oggetti grafici usano l'idioma del Listener per notificare l'occorrenza di una certa azione e l'applicazione o applet deve implementare i metodi dell'interfaccia Listener.

Un esempio Java particolare in questi casi è l'uso di inner-classes anonime come Adapters tra l'interfaccia (Target) che riceve le azioni grafiche e l'applicazione o applet (Adaptee) che ne fornisce il comportamento.
Esempio pg 46

Decorator → Scopo: aggiungere a RunTime delle funzionalità ad un oggetto nella programmazione ad oggetti, per aggiungere delle funzionalità ad una classe viene utilizzata l'ereditarietà. Se in sede di definizione della struttura delle classi non vengono previste delle specifiche funzionalità, queste non saranno disponibili a RunTime.
Esempio → si vuole conoscere il tempo di esecuzione di un metodo ma tale funzionalità non è prevista nel metodo di nostro interesse. Come fare? Creiamo una classe "Decorator" da invocare al posto della classe originaria e che si occuperà di monitorare il tempo trascorso nell'invocazione del metodo originario. Come? Mantenendo una associazione alla classe originaria e calcolando il tempo di esecuzione del metodo.
Esempio pg 49

Proxy → Scopo: fornire un surrogato o un segnaposto per un altro oggetto al fine di controllare l'accesso all'oggetto stesso. Allo stesso tempo può essere utile per rimandare la creazione di un oggetto ad un momento successivo per ridurre uso di risorse.

Motivazione: motivo può essere differire il costo di creazione ed inizializzazione dell'oggetto ad un momento successivo.

Applicabilità: remote proxy, virtual proxy, protection proxy.

Esempio: Java RMI introduce lato client un oggetto Proxy, detto Stub nella terminologia RMI, che fornisce la stessa interfaccia del server e incapsula la complessità della comunicazione remota.

Esempio: Un programma di visualizzazione di testi potrebbe essere in grado di visualizzare il nome di un file, il testo completo, o trovare e visualizzare una singola riga.
Il "Proxy" pattern suggerisce l'implemetazione di una classe (ProxyFileHandler) che offra la stesa interfaccia della classe originale (FileHandler), e che sia in grado di risolvere le richieste più "semplici"

pervenute dall'applicativo, senza dover utilizzare inutilmente le risorse (ad esempio, restituire il nome del file).
Solo al momento di ricevere una richiesta più "complessa" (ad esempio, restituire il testo del file), il proxy andrebbe a creare il vero FileHandler per inoltrare a esso le richieste.
Esempio pg 52

Facade → Fornisce una interfaccia unificata per un insieme di interfacce di un sottosistema, rendendo più facile l'uso di quest'ultimo.
Il "Facade" pattern suggerisce la creazione di un oggetto che presenti una interfaccia semplificata al client, ma in grado di gestire tutta la complessità delle interazioni tra gli oggetti delle diverse classi per compiere l'obbiettivo desiderato.
Esempio pg 54

SLIDE 15 – DESIGN PATTERN PARTE 2
Design pattern comportamentali → Observer, Iterator, Command, Memento, Strategy.

Observer:
Problema → Più oggetti Observer, anche di tipo diverso, sono interessati ad i cambiamenti di stato o agli eventi di un oggetto Observable e vogliono regire in maniera autonoma quando l'Observable genera un evento. Il publisher (l'Observable)non deve essere accoppiato ai subscriber (Observer).
Soluzione → Si definisce una interfaccia Observer o "Listener" (ascoltatore). I vari Observer implementano questa interfaccia. l'Observable può registrare dinamicamente i vari Observer che sono interessati a un evento e avvisarli quando l'evento si verifica.
Esempio pg 12

Iterator → Fornisce un metodo per accedere agli elementi di una collezione di oggetti in modo sequenziale, senza esporre la struttura interna della collezione. In Java questo pattern è presente con l'idioma Enumeration nelle collezioni Vector e Hashtable.
Esempio pg 15

Command → Serve a trattare le richieste di operazioni come oggetti eliminando la dipendenza tra l'oggetto che invoca l'operazione e l'oggetto che ha la conoscenza di come eseguirla e permettere gestioni sofisticate di accodamento, sincronizzazione, gestione delle priorità, ecc.
Esempio è il caso in cui si vogliono disciplinare le invocazioni concorrenti a un oggetto in modo da gestire in modo sofisticato la coda di priorità delle richieste. Un oggetto scheduler può decidere l'ordine di esecuzione e la politica di confitto delle richieste.
Il pattern del Command risolve il problema trasformando un'invocazione in un oggetto per permettere una gestione sofisticata di accodamento e esecuzione delle richieste.
Esempio pg 20

Memento → Viene utilizzato quando si ha necessità di ripristinare lo stato di un oggetto ad un suo precedente stato. Ciò richiede di memorizzare gli stati pregressi di un oggetto per poterli eventualmente ripristinare. Questo pattern è composto dai seguenti partecipanti:
Caretaker: è interessato allo stato dell'Originator e detiene l'oggetto Memento .
Originator: memorizza il proprio stato transitorio attraverso l'utilizzo delle proprietà interne mentre memorizza il proprio stato esterno attraverso l'utilizzo del Memento .
Memento: è l'oggetto che si occupa di detenere lo stato dell'Originator e consente solo a questi di accedervi.
Esempio pg 24 – 25

Strategy:
Problema → Come bisogna progettare per gestire un insieme di algoritmi o politiche variabili ma correlate? Come bisogna progettare per modificare tali algoritmi?
Soluzione → Si incapsula ogni algoritmo (politica, strategia) in una classe separata, e ogni classe ha un'interfaccia comune. Gli algoritmi sono incapsulati in oggetti ConcreteStrategy. Gli oggetti ConcreteStrategy implementano un'interfaccia comune Strategy da cui dipende l'algoritmo (Context) deve:
Possedere i metodi per implementare una Strategy (es setStratey(Strategy S)).
Passare i parametri necessari o sé stesso (this) a Strategy per consentire l'applicazione della strategia.
Esempio pg 28

Principi di SOLID Design → Principio "Single responsibility" , Principio "Open close" , Principio di Liskov di sostituzione , Principio "Interface segregation" , Principio "Dependency inversion".

Alcuni elementi per capire se siamo nel caso di "Bad Design" → Software Rigidity, Software Fragility, Software Immobility, Software Viscosity.
Software Rigidity → Un singolo cambiamento coinvolge molte parti del sistema.
Software Fragility → Un singolo cambiamento coinvolge parti inaspettate del problema.
Software Immobility → è difficile fare riuso di tutto o parte del software.
Software Viscosity → è difficile fare la cosa giusta, ma è facile sbagliare.

Principio importante → È inutile fare di più ciò che si può fare con meno. (Rasoio di Ockham).

Principio di singola responsabilità → Una classe dovrebbe avere una sola ragione per cambiare → Un oggetto fa una sola cosa. Es la classe rettangolo può sia disegnare che calcolare l'area. Sbagliato, perché se agisco sulla classe rettangolo le modifiche agiscono su entrambe le classi, quando magari volevo modificarne solo una.

Principio di apertura / chiusura → Le entità (classi, moduli, funzioni, etc) dovrebbe essere aperto per le estensioni, ma chiuso alle modificaioni.
Il segreto sta nell'utilizzo delle interfacce (o di classi astratte).
Ad un'interfaccia immutabile possono corrispondere innumerevoli classi concrete che realizzano comportamenti diversi.
Un modulo che utilizza astrazioni non dovrà mai essere modificato, dal momento che le astrazioni sono immutabili (il modulo è chiuso per le modifiche).
Potrà cambiare comportamento, se si utilizzano nuove classi che implementano le astrazioni (il modulo è aperto per le estensioni).

Principio di sostituzione di Liskov (LSP) → Sottotipi dovrebbero essere sostituibili per i supertipi.
Oppure le classi figlie non devono mai rompere la definizione delle classi genitrici.
Esempio slide 46 – 47.
Adeguarsi a LSP → "Require no more, promise no less". L'autore di una classe specifica il comportamento di ogni metodi in termini di pre-condizioni e post-condizioni. Le sottoclassi rispettano due regole:
Le precondizioni dei metodi ridefiniti devono essere uguali o più deboli di quelle della superclasse (non si forzano e non si assumono ulteriori vincoli rispetto alla superclasse).
Le postcondizioni dei metdi ridefiniti devono essere uguali o più forti di quelle della superclasse (si assumono tutti gli ulteriori vincoli della superclasse o eventualmente se ne aggiungono).

Il principio di segregazione delle interfacce → "Clients should not be forced to depend upon interfaces that they don't use". Cioè quando faccio l'implements di una classe, ho metodi che devo inserire per coerenza (implements) ma che in realtà non uso. Il principio di segregazione delle interfacce dice di tenere le interfacce minime per non avere metodi inutilizzate nelle classi figlie.

Il principio di inversione delle dipendenze → Un modulo ad alto livello non dovrebbe dipendere dai moduli a basso livello. Entrambi dovrebbero dipendere dalle astrazioni.
Le astrazioni non dovrebbe dipendere dai dettagli, sono i dettagli che devono dipendere dalle astrazioni.
Quindi:
Le astrazioni contengono pochissimo codice (in teoria praticamente nullo) e quindi sono poco soggette a cambiamenti. I moduli non astratti sono soggetti a cambiamenti ma questi cambiamenti sono sicuri perché nessuno dipende da questi moduli. I dettagli del sistema sono stati isolati, separati da un muro di astrazioni stabili, e questo impedisce ai cambiamenti di propagarsi (design for change).
Nel contempo i singoli moduli sono maggiormente riusabili perché sono disaccoppiati fra di loro (design for reuse).

SLIDE 15 – SOFTWARE TESTING
Testing:
Verifica → Consente di stabilire/controllare se un software esegue le funzioni per le quali è stato realizzato in maniera corretta senza malfunzionamenti.
Validazione → Consente nel valutare se il software soddisfa i requisti e le specifiche per esso stabilite. Rientra nella validazione anche la valutazione del soddisfacimento dei requisiti di qualità.

Definizione:
Errore → Incomprensione umana nel tentativo di comprendere o risolvere un problema, o nell'uso degli strumenti.
Difetto o anomalia (fault o bug) → Manifestazione nel software di un errore umano, e causa del fallimento del sistema nell'eseguire la funzione richiesta.

Definizioni x2:
Testing → Il testing o collaudo è un processo di esecuzione del software allo scopo di scoprirne i malfunzionamenti. Osservando i malfunzionamenti possiamo dedurre la presenza di difetti.
Debugging → Il debugging è il processo di scoperta dei difetti a partire dai malfunzionamenti rilevati.
Ispezione → L'ispezione è un processo di analisi del software per scoprirne i difetti.

Definizioni x3:
Un test T **rileva un malfunzionamento** se il programma P non è corretto per T, ovvero se produce risultati diversi da quelli attesi.
T si dice **aver successo** per un programma P se rileva uno o più malfunzionamenti presenti in P.
Un insieme di test T **è inadeguato** se esistono dei malfunzionamenti in P che il test T non è in grado di rilevare.

Tesi di Dijkstra: Il problema della assenza di errori nel software è non decidibile → Non posso trovare tutti gli errori in un software.

Definizioni x4:
Un test è **ideale** se l'insuccesso del test implica la correttezza del programma.
Un test è **esaustivo** se contiene tutte le combinazione dei dati di ingresso del programma.
Un test **esaustivo** è un test **ideale**, tuttavia un test **esaustivo** non è pratico e quasi sempre non è fattibile.

Tipologie di Testing:
White Box Testing → Presuppone che il codice sorgente sia disponibile il codice sorgente. L'idea generale è di trovare dei dati di ingresso che causino l'esecuzione di tutte le operazioni previste del programma, nelle varie sequenze possibili. Il White Box Testing è un **test detto anche strutturale** poiché utilizza la struttura interna del programma per ricavare i dati di test. Implica una conoscenza approfondita del codice che si sta testando e la capacità di metterlo alla prova.
Può essere applicato → Allo Unit Test, Integration Test, System Test. Anche se di solito viene usato per gli Unit Test.
I dati di test vengono scelti tramite **test di copertura:** definiscono l'insieme di sequenza di operazioni che devono essere eseguite nel corso del test.
In caso di Test Strutturale, assume importanza il concetto di **copertura**, ovvero quanta parte del codice è soggetta a test.
Questo è possibile poiché si ha accesso al codice sorgente e si può decidere cosa sottoporre al test.

Black Box Testing → È un metodo di software testing che ha il compito di testare la funzionalità di un applicativo senza conoscere la struttura interna, si differenzia per questo dal white box testing.
Si esercita quindi il sistema immettendo input e osservando i valori di output.
Nel black box testing non si conosce (o non si tiene conto di):
Codice sorgente, lo stato interno dell'applicazione, il funzionamento interno dell'applicazione.
Viceversa, conosciamo e utilizziamo la progettazione dei casi di test:
L'interfaccia di sistema e la documentazione.
I test case relativi al black box testing sono scritti in base alle specifiche funzionali e specifiche tecniche fornite insieme alla documentazione e sulla base delle quali viene sviluppata una applicazione.
Nel Black-box testing sono usati i documenti relativi alla fase di analisi, tra essi Use Case Specifications.
Obiettivi principali del testing:
La copertura degli scenari di esecuzione definiti nei casi d'uso.
La copertura delle funzionalità previste nella specifica dei requisiti.
Difficoltà principali:
La ricerca dei casi di test che coprano i dati scenari/funzionalità.
La verifica dell'avvenuta copertura.
La soluzione più spesso adottata è quella di:
Generare casi di test con un opportuno criterio.
Valutare l'effettiva copertura di scenari/funzionalità.

Classi di equivalenza:
Occorre dividere i possibili input in gruppi i cui elementi si ritiene che saranno trattati similarmente dal processo elaborativo.
Questi gruppi saranno chiamati **classi di equivalenza:**
Una possibile suddivisione è quella in cui classe di equivalenza rappresenta un insieme di stati validi o non validi per una condizione sulle variabili d'ingresso.
Chi esegue il testing eseguirà almeno un test per ogni classe di equivalenza:

Ogni classe di equivalenza deve essere coperta da almeno un caso di test.

Esempio
Una condizione di validità per un input password è che la password sia una stringa alfanumerica di lunghezza compresa fra 6 e 10 caratteri.
Una classe valida CV1 → Quella composta da stringhe di lunghezza tra 6 e 10 caratteri.
Due classi non valide CV2 e CV3 → CV2 stringhe con lunghezza < 6 e CV3 stringhe con lunghezza >10.

Unit Test – continua:
Una buona norma, (best pratice) che ogni sviluppatore dovrebbe effettuare è quella di eseguire test di unità per assicurarsi la singola unità di sviluppo assolva le sue funzioni seguendo i requisiti.
Il testing a livello di unità dei comportamenti di unca classe dovrebbe essere progettato ed eseguito dallo sviluppatore della classe, contemporaneamente allo sviluppo della classe.
Vantaggi → Lo sviluppatore conosce esattamente le responsabilità della classe che ha sviluppato ed i risultati che da essa si attende. Lo sviluppatore conosce esattamente come si accede alla classe.
Svantaggi → Lo sviluppatore tende a difendere il suo lavoro e troverà meno errori di quanto un altro faccia.

Unit Test:
Nei test di unità, il test di ciascun modulo richiede di moduli ausiliari che simulano i moduli, che, nel sistema reale, interagiscono inl modulo in esame.
I moduli ausiliari che simulano i moduli client sono detti **driver**, quelli che simulano i moduli server si dicono **stub.**
I driver e gli stub devono simulare degli interi sottosistemi, ma ovviamente devono essere realizzati in maniera semplice.
In Java un framework standard de facto è Junit.

Integration Test:
Il test di integrazione avviene nella costruzione del sistema a partire dai suoi componenti per scoprire problemi che nascono dall'interazione dei vari componenti.
Richiede l'identificazione di gruppi di componenti che realizzano le varie funzionalità del sistema e la loro integrazione mediante dei componenti che li fanno lavorare insieme.
La strategia del test di integrazione generalmente ricalca la strategia di sviluppo, e quindi può essere top-down o bottom-up, o una combinazione delle due.
Con la **strategia top down** non c'è bisogno di driver, poiché i moduli sviluppati e testati in precedenza fanno da driver per i moduli integrati successivamente. Con la strategia bottom-up non c'è bisogno di stub.
La **strategia top-down** permette di avere presto dei prototipi, mentre la strategia bottom-up permette di testare subito i moduli terminali, che generalmente sono i più critici.

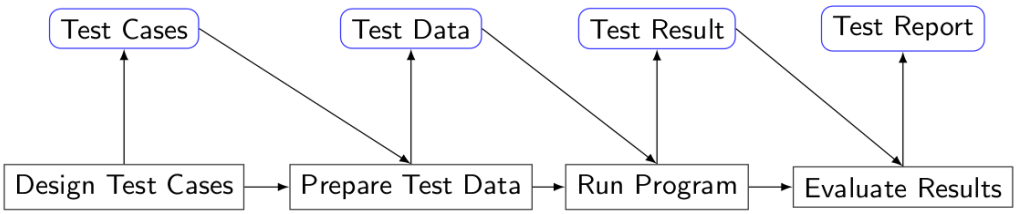
System test → il test di integrazione non può verificare le proprietà globali del sistema, che non si possono riferire a qualche particolare modulo o sottosistema.
Alcune di queste proprietà sono le performances, la robustezza e la riservatezza (security).
Performance test → I test di prestazione in genere riguardano il carico e si pianificano test in cui il carico viene incrementato progressivamente finché le prestazioni diventano inaccettabili. Il carico deve essere progettato in modo da rispecchiare le normali condizioni d'utilizzo.
Stress test → Consiste nel sottoporre il sistema ad uno sforzo superiore a quello previsto delle specifiche, per assicurarsi che il superamento dei limiti non porti a malfunzionamenti incontrollati, ma solo ad una degradazione delle prestazioni. La grandezza che definisce lo sforzo dipende dall'applicazione: per esempio potrebbe essere il numero di utenti collegati contemporaneamente ad un sistema multiutente, od il numero di transazioni al minuto per un database.
Robustness test → Il test di robustezza consiste nell'inserire i dati in ingresso scorretti. Anche in questo caso ci si aspetta che il sistema reagisca in modo controllato, per esempio stampando messaggi di errore e, nel caso di sistemi interattivi, rimettendosi in attesa di ulteriori comandi dell'utente.
Security test → Il test di sicurezza consiste nel verificare le vulnerabilità del sistema. Queste possono essere suddivise in base al **tipo di rischio:**
Attacchi di autenticazione/autorizzazione → Tentativi di bypass dell'autenticazione, intromissione in sessioni autorizzate, escalation dei privilegi, attacchi di forza bruta.
Dipendenza da sistemi → Risorse o componenti cruciali per il sistema possono essere identificati ed usati per un attacco. Questo può avvenire con trojans in files cookies e chiavi di registro. Attacchi man in the middle e a sottoprogrammi: Joomla e Wordpress.
Input attacks → Tentativi di ottenere buffer overflow con stringhe molto lunghe, SQL injection, command injection, LDAP injection.
Design attacks → Attacchi che sfruttano "aperture" dovute al design, quali attacchi usando API interne

non protette, rotte alternative a quelle sottoposte al controllo, porte TCP aperte.
Information disclosure attacks → Attacchi che forzano l'applicazione a emettere più informazioni di quante dovrebbe, quali ad esempio messaggi di errori generati automaticamente oppure directory non protette e direttamente accessibili.

Regression Test → Il test di regressione serve a verificare che una nuova versione di un prodotto non produca nuovi errori rispetto alle versioni precedenti e sia compatibili con esse.
Man mano che si integrano i vari componenti, sarà anche necessario rieseguire i precedenti test.
Il test di regressione si esegue anche dopo un intervento di manutenzione (per verificare che il sistema non sia "regredito").

Test di accettazione → È un test di sistema eseguito dal committente (invece che dal produttore), che in base al risultato decide se accettare o no il prodotto. Il test di accettazione ha quindi una notevole importanza legale ed economica, e la sua pianificazione può far parte del contratto.
È un test black box basato sulle specifiche del software, i tester non non accedono al suo codice.
Un tipo di test di accettazione è quello usato per prodotti destinati al mercato.
In questo caso il test avviene in due fasi: nella prima fase (α-test) il prodotto viene usato all'interno dell'organizzazione produttrice, e nella seconda (β-test) viene distribuito in prova ad alcuni clienti selezionati.

Il processo di Test:



SLIDE 17 – TEST E VERSIONING

<u>Documenti di test:</u>
Test plan → Pianificazione dei test di ogni iterazione che include elenco attività, calendario e risorse coinvolte (Gantt).
Test case → Descrizione del singolo caso di test.
Test result → Documento che contiene una sintesi dei test eseguiti e delle informazioni derivanti dall'analisi dei test log.

Documento di test plan contiene:
Introduzione ed obiettivi (Obiettivi, Definizioni Acronimi ed Abbreviazioni, Riferimenti dei test)
Organizzazione dei test (Caratteristiche da testare: funzionalità, prestazioni, vincoli di progetto, sicurezza, Allocazione del personale).
Piano di Test (Quali deliverables verranno rilasciati nelle varie attività del test: Casi di test, Rapporto finale, diario del test e rapporto di copertura).
Cronoprogramma (Le attività di test declinate temporalmente, meglio se anche con un diagramma di Gantt).

Esempio pg 19 → 25

Documento di Test Case:
Normalmente contiene per ogni use case significativo:
I dati di input,
i risultati attesi,
una descrizione dell'ambiente di esecuzione.

Documento di Test Result:
Normalmente contiene per ogni use case significativo:
I dati di input,
I risultati attesi,
I risultati ottenuti.

Documento di Test Report contiene:
Sommario dei malfunzionamenti (top 10).
Riepilogo del test → Numero totale casi di test eseguiti, numero e tipo malfunzionamenti, numero e tipo difetti.
Diario del test.
Rapporti sul test → descrive i dettagli del test per come si è svolto effettivamente, la specifica dei casi di test può essere completata e usata come diario.

Version Control System:
Equivalente a Version Control, Source Code Management (SCM), Source Control System.

Forniscono supporto alla memorizzazione dei codici sorgenti,
Forniscono uno storico di ciò che è stato fatto,
Può fornire un modo di lavorar in parallelo su diversi aspetti dell'applicazione in sviluppo,
Può fornire un modo di lavorare in parallelo senza intralciarsi a vicenda,
Usare un sistema di Version Control anche per lo sviluppo personale a volte torna utile.

Concetti di base – Repository:
Tipicamente si trova su una macchina remota affidabile e sicura. Per tutti gli sviluppatori condividono lo stesso repository.

Concetti di base – Working Folder:
Ogni sviluppatore ne ha una collocata sulla propria macchina , contiene una copia del codice sorgente relativo al progetto.

Sia il Repository, sia la Working Folder sono una gerarchia di cartelle o directory.

Concetti di base:
Il codice che viene memorizzato sul repository centrale deve portare il progetto in uno stato che consenta a chiunque (nel team) di proseguire nello sviluppo.

Repository → È un archivio di ogni versione di ogni file di codice sorgente.
Contiene la storia del progetto.
Rende possibile navigare indietro nel tempo e recuperare versioni vecchie dei file:
Capire perché sono state fatte certe scelte (e chi le ha fatte),
Capire perché sono stati introdotti bug (e chi li ha introdotti).

Problemi da evitare → Sovrascrittura involontaria su la stessa versione del progetto da parte di due persone diverse (scrivono sulle stesso file senza sapere che c'era stato un commit prima).
Modi per evitarlo:

Modello pre-empitive → Uno sviluppatore deve effettuare un checkout prima di modificare un file .
Tutti sanno chi sta modificando cosa .
I checkout sono effettuati con lock esclusivi.
Solo uno sviluppatore alla volta può modificare un file.
Problemi modello preempitive:
Problemi di amministrazione: un utente pone un lock poi se ne dimentica = possibili ritardi e tempo perso .
Serializzazione non necessaria: un utente blocca un file e ne modifica una parte → impedisce ad un altro utente di modificare lo stesso file in una parte scorrelata dalla prima modifica .
Falso senso di sicurezza: un utente blocca un file e lo modifica, un altro utente blocca un altro file che dipende dal primo e lo modifica → manca la certezza che le modifiche effettuate saranno compatibili.

Modello Collaborativo → Non esistono lock e tutti i file sono writable.
L'utente copia i file dal repository nella sua working folder (checkout).
Modifica i file nella sua working folder.
Richiede un update della propria working folder (nel frattempo, qualcun altro potrebbe aver modificato il repository).
I file modificati vengono "fusi" (viene fatto un merge) con quelli contenuti nel repository nella versione finale del file.
L'operazione di merging è tipicamente fatta in modo automatico ed il risultato è tipicamente positivo: dopo un merge occorre verificare il corretto funzionamento dell'applicazione (testing).
In alcuni casi il merge non può essere fatto automaticamente (conflitti o modifiche della stessa linea di codice da parte di diversi sviluppatori); in questo caso i conflitti vengono risolti con strumenti che aiutano

ad evidenziare le modifiche proprie e di altri.
Il salvataggio nel repository consente di pubblicare le modifiche effettuate (commit).
Modello collaborativo regole:
Nessuno usa i lock quindi non si sa chi stia facendo cosa .
Il sistema si accorge se è necessario un merge oppure se è possibile salvare il file direttamente
È buona norma effettuare un update del contenuto della propria working folder prima di cominciare a lavorare .
Quando uno sviluppatore effettua un commit è sua responsabilità assicurarsi che i cambiamenti siano stati effettuati sull'ultima versione contenuta nel repository.

Funzionamento merge:
Tre file in gioco:
1 File contenuto nel repository (ultima versione)
2 File con modifiche dello sviluppatore
3 File originale prima delle modifiche dello sviluppatore (in working folder) .
Se 1. e 3. sono uguali non serve il merge – si copia la nuova versione sul repository
Come funziona il merge?
Si considera 3. come il "file originale", 1. e 2. come file modificati rispetto all'originale.

Dopo un merge il codice NON si compila sempre .
I cambiamenti di diversi sviluppatori, pur non entrando in conflitto secondo l'algoritmo di merge, possono generare inconsistenze finali .
Dopo il merge e prima di fare un commit, compilare e far girare la suite di test.

Trunk, Branch, Tag.
Un modo ragionevole per organizzare un repository è fare in modo che contenga:
Un tronco principale di sviluppo → Trunk.
Un luogo dove memorizzare le linee di sviluppo alternative → Branch, Branches.
Un luogo dove memorizzare le release stabili → Tag, Tags.
Branch:
Relativamente ad un progetto, un branch è una linea di sviluppo indipendente dalle altre .
Viene inizialmente generato come copia completa → condivide parte della storia .
Consente di iniziare lo sviluppo di una nuova release quando la precedente è ancora in fase di consolidamento .
Terminato il consolidamento è possibile effettuare il merge fra il branch ed il trunk .
ATTENZIONE: questo può diventare un "bagno di sangue".
Tag:
Viene memorizzato separatamente in modo da avere a portata di mano tutti i sorgenti relativi ad una certa release .
In questo modo non è necessario andare a ripescare dal main trunk i sorgenti andando indietro con le versioni dei file.