Università degli Studi di Perugia
Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Ingegneria del Software

Prof. Alfredo Milani

Princìpi & Qualità

Materiale note, grazie al contributo di: Alfredo Milani, Fabrizio Montecchiani, Carlo Ghezzi, Alessandro Riccardi et al.

Docente

- Alfredo Milani
  - Email: alfredo.milani@unipg.it
  - Studio: DMI 6° piano +39 075 585.5049
- **Ricevimento**
  - su appuntamento per email
  - Mercoledì 14:00-15:00
  - online

Obiettivi

- Le qualità del software
- I principi dell'ingegneria del software

Qualità interne ed esterne

- Sebbene il software sia intangibile, possiamo attribuirgli delle qualità.
- Le **qualità esterne** del software sono visibili agli *utenti* del sistema.
- Le **qualità interne** del software riguardano invece gli *sviluppatori* del sistema.
- Le qualità interne aiutano gli sviluppatori a raggiungere le qualità esterne desiderate.

Qualità interne ed esterne

- Esempio:
  - Gli utenti vogliono che il software sia affidabile.
  - Gli sviluppatori vogliono che il software sia verificabile.
  - Queste due qualità sono strettamente correlate, poiché l'affidabilità si ottiene mediante la verificabilità.

Qualità del processo e del prodotto

- Il **prodotto** dell'ingegneria del software è il sistema software consegnato al committente (codice, manuali…).
- Tale prodotto si realizza mediante un **processo**, attraverso il quale vengono creati diversi *prodotti intermedi* (documenti, dati di test, rilasci intermedi).
  - I prodotti intermedi sono soggetti agli stessi requisiti di qualità del prodotto finale.
- La **qualità del processo** influenza la qualità del prodotto.

Qualità esterne

Correttezza

- Un programma è **corretto** se si comporta secondo quanto stabilito dalle specifiche funzionali.
  - Le specifiche devono essere non ambigue.
  - Tanto più le specifiche sono scritte in maniera rigorosa, tanto più sarà possibile verificare con precisione la correttezza di un software.
- Il limite di questa qualità sta nell'essere di tipo assoluto (sì/no) e nel dipendere dalla correttezza delle specifiche.

Affidabilità

- Un software è tanto più **affidabile** tanto più è alta la probabilità che funzioni come atteso entro un certo intervallo temporale.
- I software corretti (assumendo che le specifiche siano corrette) sono anche affidabili.
- Software non corretti possono essere considerati comunque affidabili, qualora il difetto non abbia un impatto importante.
- L'affidabilità di un software può migliorare col tempo, grazie a interventi di manutenzione.

Robustezza

- Un programma è **robusto** se si comporta in modo accettabile anche in circostanze non previste nella specifica dei requisiti.
  - Ad esempio, in caso di input non corretto o di fallimenti dell'hardware.
- Un programma corretto potrebbe non essere robusto, semplicemente perché la specifica dei requisiti non prevede alcune casistiche.

Legge di Postel:
sii conservatore in ciò che fai, sii liberale in ciò che accetti dagli altri.

Prestazioni

- Le **prestazioni** di un software dipendono dall'efficienza con cui il software utilizza le risorse interne del computer (memoria, potenza di calcolo, rete).
- Possiamo valutare le **prestazioni** di un sistema mediante misure, analisi, e simulazioni.
- Le prestazioni influenzano l'usabilità e la scalabilità di un software.
  - Se un software è troppo lento diventa inusabile.
  - Un software con scarse prestazioni potrebbe essere praticamente usabile solo per dimensioni dell'input non troppo grandi (ovvero non scalabile).

Usabilità

- Un software è **usabile** quando gli utenti lo reputano semplice da utilizzare (*user-friendly*).
  - L'usabilità è dunque una qualità piuttosto soggettiva, può dipendere dai gusti e dall'esperienza.
- L'interfaccia grafica influisce molto sull'esperienza che l'utente percepisce nell'usare il software.
  - Un sistema che presenta un'interfaccia coerente e prevedibile è tipicamente usabile.
  - La standardizzazione delle interfacce è un fenomeno in crescita e tende a migliorare l'usabilità dei sistemi che adottano componenti e interfacce standard.

Qualità interne

Verificabilità

- La **verificabilità** di un software misura la facilità con cui è possibile verificarne la correttezza e le prestazioni.
  - Metodi di programmazione modulare, norme sistematiche di codifica, e uso di linguaggi di programmazione appropriati alla scrittura di codice ben strutturato aiutano a garantire tale qualità.
- In alcuni casi può essere vista come qualità esterna.
  - Ad esempio, quando la sicurezza di un software è un aspetto critico e deve essere verificabile.

Manutenibilità

- La **manutenibilità** di un software indica la facilità con cui tale software può evolvere, al fine di rimuovere errori, implementare nuove funzionalità, migliorare funzionalità esistenti, ecc.
- I costi di manutenzione incidono tipicamente per il 60 % dei costi totali, pertanto rendere un software mantenibile è di grande importanza.

Manutenibilità

- La **manutenzione correttiva** riguarda la rimozione di errori esistenti sin dal primo rilascio del software, o introdotti successivamente.
- La **manutenzione adattiva** riguarda le modifiche dell'applicazione in risposta a cambiamenti dell'ambiente (ad esempio, un nuova versione dell'hardware, del sistema operativo, o del DBMS).

Manutenibilità

- La **manutenzione perfettiva** riguarda i cambiamenti necessari per migliorare alcune qualità (modifica o aggiunta di funzioni, migliorare le prestazioni o l'usabilità, ecc.).
- Tipicamente, la manutenzione correttiva pesa per il 20% dei costi di manutenzione, così come quella adattiva, mentre il restante 60% è dovuto a costi di manutenzione perfettiva.

Manutenibilità

- Il **software legacy** (o software ereditato) si riferisce a software presente in un'organizzazione da lungo tempo, di valore strategico poiché incorpora molti processi importanti.
- Tale software spesso si poggia su tecnologia desueta, ed è pertanto difficile da modificare e manutenere.
- Tecniche di reverse engineering e reengineering possono essere adottate per scoprire la struttura del software legacy e per ristrutturarlo.

Manutenibilità

- La manutenibilità di un software può essere vista come la somma di due qualità.
  - **Riparabilità**: facilità con cui si eliminano difetti dal software.
  - **Evolvibilità**: facilità con cui si apportano cambiamenti al software.

Nota: se le specifiche sono poco chiare, può non essere evidente se un'attività è da considerarsi una riparazione o una evoluzione.

Riparabilità

- La riparabilità è promossa da software modulari, in cui le interazioni tra i vari moduli sono contenute e prevedibili.
- Linguaggi di programmazione evoluti e strumenti quali i debugger aiutano a riparare un software più facilmente.

Evolvibilità

- I prodotti software di successo hanno spesso una lunga vita, con molti rilasci.
- Se ogni modifica viene analizzata, progettata, e implementata correttamente, allora il software evolve in maniera ordinata e controllata.
- La capacità di evolvere, anch'essa promossa da architetture modulari, tende tuttavia a diminuire mano a mano che il software viene modificato.
  - La struttura si complica e tende ad allontanarsi dal progetto iniziale.
  - Il sistema andrebbe quindi progettato in maniera che anticipi possibili cambiamenti.

Riusabilità

- Singole routine, librerie, moduli, o prodotti completi possono essere **riusati** per dare vita ad altri prodotti.
- Alcuni produttori di software sono specializzati nella produzione di librerie e componenti altamente riusabili in vari contesti.
- I linguaggi orientati agli oggetti sono pensati per favorire la riusabilità del codice.
- Anche alcuni documenti possono essere riusati in altri progetti (ad esempio, la specifica dei requisiti).

Portabilità

- Il software è **portabile** se può essere eseguito in ambienti diversi (piattaforme hardware, sistemi operativi, ecc.).
  - Il sistema operativo consente alle applicazioni una buona portabilità rispetto all'hardware del computer.
- La portabilità può essere ottenuta isolando le dipendenze in pochi moduli del software, facilmente modificabili in base all'ambiente (ad esempio, le librerie grafiche, o l'accesso al filesystem).

Comprensibilità

- Software semplici sono tipicamente anche facili da comprendere, mentre i software più complessi potrebbero esserlo meno.

Legge di Eagleson:
qualsiasi codice sorgente che hai scritto e che non è più stato guardato da sei o più mesi potrebbe benissimo essere stato scritto da qualcun altro.

Comprensibilità

- Internamente, un software comprensibile semplifica le attività di verifica e manutenzione.
  - Spesso chi verifica una parte di software è diverso da chi l'ha implementata, e da chi dovrà poi manutenerla.
- Esternamente, un software prevedibile è tipicamente facile da comprendere per l'utente, favorendo dunque l'usabilità.

Interoperabilità

- L'**interoperabilità** è la capacità di coesistere e cooperare con altri sistemi.
  - Ad esempio, da un elaboratore di testi ci si aspetta la possibilità di incorporare diagrammi prodotti da un pacchetto grafico.
- L'utilizzo di interfacce standard semplifica l'interoperabilità.

Qualità di processo

Produttività

- La **produttività** è una qualità del processo di produzione del software che ne indica l'efficienza e le prestazioni.
- Dipende sia dai singoli sviluppatori, che dal management, che dagli strumenti usati.
- Il **riuso del software** favorisce la produttività, sebbene lo sviluppo di moduli riusabili è più dispendioso e quindi va valutato in prospettiva.

Tempestività

- La **tempestività** di un processo è la capacità di rendere disponibile un prodotto al momento giusto.
- Sebbene il software andrebbe consegnato solo se in possesso di tutte le altre qualità attese, consegnare versioni preliminari può aiutare a raccogliere critiche e suggerimenti.
- Un'attenta **pianificazione** del processo, un'accurata **stima** delle attività, e una specifica chiara degli obiettivi intermedi (*milestone*), sono fondamentali per conseguire la tempestività.

Tempestività

- La specifica dei requisiti è inoltre fondamentale per evitare la produzione di un software che sarà già obsoleto nel momento in cui verrà rilasciato.
  - I requisiti vanno formulati avendo in mente il sistema nel momento in cui verrà consegnato e non nel momento stesso in cui i requisiti vengono raccolti.
- La **consegna incrementale** del software, ovvero la consegna incrementale di set di funzionalità parziali ma significative, può favorire la tempestività.

Visibilità

- Un processo di sviluppo del software è **visibile**, o **trasparente**, se tutti i suoi passaggi sono documentati in modo chiaro.
- Un processo visibile consente ai vari attori di avere chiaro lo stato del progetto, potendo così soppesare le loro scelte, lavorando tutti nella stessa direzione.

Visibilità

- In grandi progetti, il turnover del personale può rappresentare una criticità e rallentare la produttività, se lo stato del progetto non è facilmente reperibile (ad esempio, se è tramandato oralmente o con documenti molto informali).
- Un prodotto è visibile se è ben strutturato come una collezione di componenti, con funzioni ben comprensibili e con un'accurata documentazione disponibile.

Aree applicative

Requisiti di qualità in diverse aree applicative

- Il software viene costruito al fine di automatizzare una specifica applicazione e può essere caratterizzato sulla base dei requisiti dell'area applicativa.
- Principali aree applicative:
  - Sistemi informativi
  - Sistemi in tempo reale
  - Sistemi distribuiti
  - Sistemi embedded

Sistemi informativi

- Lo scopo primario di un **sistema informativo** è quello di gestire le informazioni di una organizzazione.
  - Sistemi bancari, sistemi bibliotecari, Enterprise Resource Planning, ecc.
- Il cuore di un sistema informativo è una base di dati, utilizzata mediante transazioni che creano, ricercano, modificano, o cancellano dati.
- Molti di questi sistemi offrono inoltre una interfaccia web per operare sulle informazioni gestite.

Sistemi informativi

- I sistemi informativi sono applicazioni orientate alle gestione dei dati e possono dunque essere caratterizzati in base al modo con cui elaborano tali dati.
- Di seguito alcune qualità caratteristiche.

Sistemi informativi

- **Integrità dei dati:** capacità di garantire la non corruzione dei dati anche a fronte di determinati malfunzionamenti.
- **Sicurezza:** capacità di fornire un opportuno livello di protezione rispetto all'accesso ai dati.
- **Disponibilità dei dati:** capacità di limitare le condizioni e gli intervalli temporali in cui i dati non sono accessibili.
- **Prestazioni delle transazioni:** capacità di eseguire più transazioni simultaneamente per unità di tempo.

Sistemi informativi

- Anche l'usabilità è essenziale per un sistema informativo, poiché dovrà essere usato da utenti anche inesperti o con scarsa predisposizione all'utilizzo di strumenti tecnologici.
  - Menu semplici e uniformi.
  - Navigazione semplice e guidata.
  - Verifica della correttezza dell'input.
  - Possibilità di personalizzare l'interfaccia e alcune funzioni del programma.

Sistemi in tempo reale

- I **sistemi in tempo reale** sono caratterizzati dalla necessità di dover rispondere a determinati eventi entro un tempo prefissato e limitato.

Sistemi in tempo reale(esempi)

- In un sistema di monitoraggio industriale il software deve rispondere a cambiamenti improvvisi di temperatura, attivando certi dispositivi e inviando segnali di allarme.
- Il software di controllo del volo di un aereo deve monitorare le condizioni ambientali e la posizione corrente dell'aereo, e controllare la traiettoria di volo in funzione di queste.
- Il software che gestisce il mouse di un computer deve rispondere rapidamente in modo da distinguere tra singolo click e doppio click.

Sistemi in tempo reale

- I sistemi in tempo reale si dicono orientati al controllo, e si basano su un pianificatore (**scheduler**) in grado di ordinare le azioni del sistema.
  - Gli scheduler basati su priorità ordinano le azioni in base alla loro priorità ed eseguono in ogni istante l'azione a priorità massima.
  - Gli scheduler basati su deadline hanno invece un tempo associato entro il quale devono iniziare o completare l'azione corrispondente.

Sistemi in tempo reale

- Il **tempo di risposta** è dunque una qualità caratterizzante per i sistemi in tempo reale.
- Anche l'affidabilità è importante, poiché spesso adottati in contesti critici.
  - Monitoraggio di pazienti, sistemi di difesa, sistemi di controllo di processo, ecc.
- Spesso si parla di **safety**, ovvero la capacità del sistema di evitare rischi inaccettabili (ciò che non dovrebbe mai capitare durante l'esecuzione di un sistema).
  - Ad esempio, la radiazione applicata da un sistema di raggi X deve sempre mantenersi al di sotto di un certo limite.

Sistemi distribuiti

- I **sistemi distribuiti** sono composti da macchine indipendenti o semi-indipendenti collegate da una rete di telecomunicazione.
- L'ambiente di sviluppo deve supportare lo sviluppo dell'applicazioni su molteplici computer, su cui gli utenti devono essere in grado di compilare, collegare, e testare il codice.
- Linguaggi interpretati come Java e C# sono particolarmente adatti a questi ambienti eterogenei.

Sistemi distribuiti

- Alcune caratteristiche importanti per questi sistemi:
  - Il **livello di distribuzione**, il quale indica se è possibile distribuire i dati, l'elaborazione, o entrambi.
  - La possibilità di **tollerare il partizionamento** (in presenza ad esempio di un collegamento di rete non funzionante che divide il sistema in più sottosistemi).
  - La possibilità di **tollerare uno o più computer non funzionanti** senza far venir meno l'operatività del sistema.

Sistemi distribuiti

- In un sistema distribuito, l'affidabilità e le prestazioni possono essere aumentate replicando i dati su più macchine.
  - La replicazione comporta però la gestione della consistenza dei dati, che richiede tecniche specifiche e spesso complesse.
- Un ulteriore beneficio in termini di prestazioni si ottiene rendendo il codice mobile, in grado cioè di migrare durante l'esecuzione, ad esempio verso il nodo che memorizza i dati da elaborare.

Sistemi embedded

- I **sistemi embedded** sono sistemi nei quali il software è solo uno dei componenti e spesso non ha interfacce rivolte all'utente finale, ma solo verso altri componenti del sistema che esso controlla.
  - Usato in aerei, robot, elettrodomestici, automobili, telefoni cellulari, ecc.
- In questo caso le interfacce tra componenti possono essere rese più complicate, se questo aiuta ad esempio a semplificare i dispositivi collegati.

Sistemi con più caratteristiche

- Spesso i sistemi presentano più caratteristiche.
  - Ad esempio, i sistemi embedded sono spesso anche in tempo reale.
  - I sistemi informativi spesso inglobano componenti embedded e componenti in tempo reale.

Misurare il software

- Mentre alcune qualità e proprietà del software sono facilmente misurabili, es: prestazioni, per altre non esistono metriche universalmente riconosciute, es: manutenibilità.
- Esempi di **metriche** (unità e modalità di misura) riconosciute:
  - **Lines of code** e **function points** per la dimensione del software.
  - **Cyclomatic** complexity per la complessità di un software.
  - **Mean Time To Failure** e **Mean Time Between Failure** per l'affidabilità.

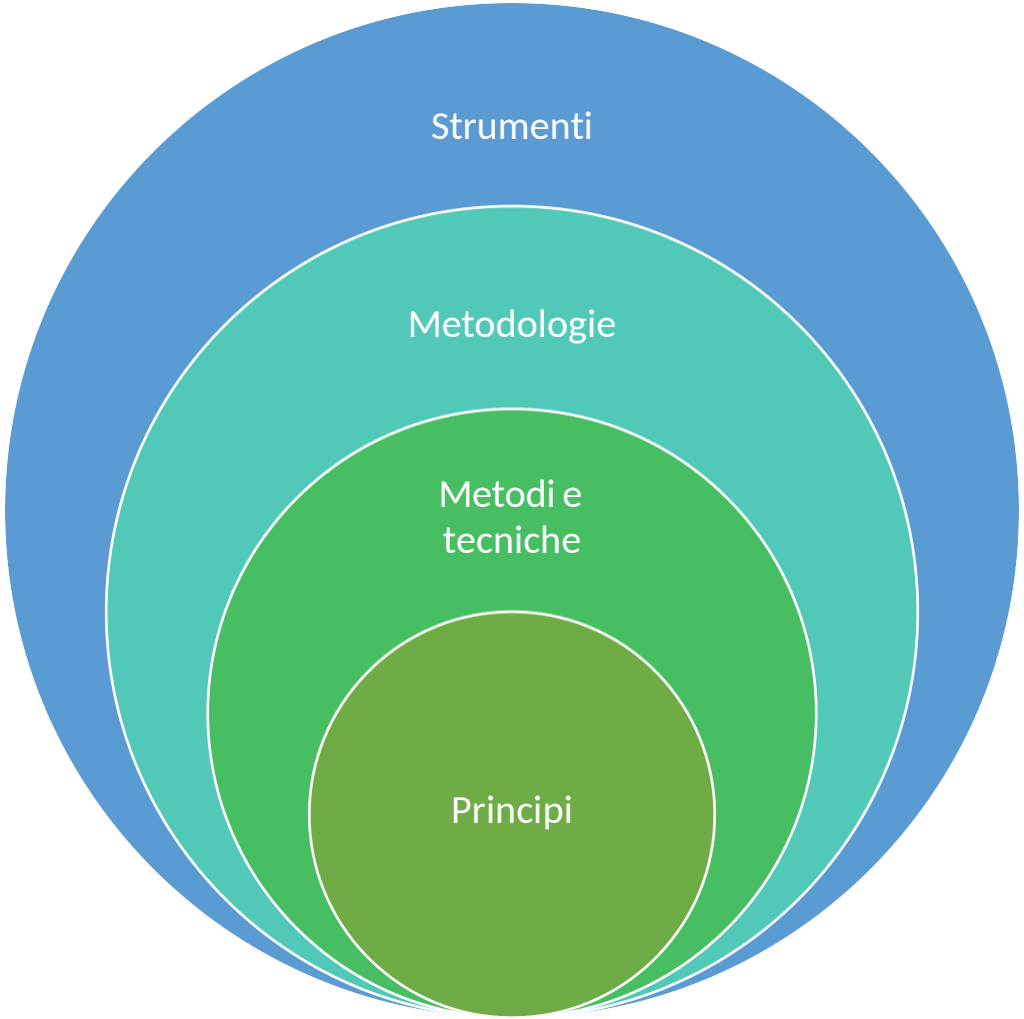
Obiettivi

- Le qualità del software
- I principi dell'ingegneria del software

Principi, metodi, tecniche, metodologie

- I **principi** dell'ingegneria del software descrivono proprietà desiderabili del processo e dei prodotti che riguardano lo sviluppo del software.
- Per poterli applicare, l'ingegnere del software deve disporre di **metodi** appropriati e di **tecniche** specifiche.
- Le **metodologie** coordinano un insieme di metodi e tecniche consistenti tra loro secondo un approccio comune.
- Gli **strumenti** supportano l'applicazione di una Metodologia.

Principi, metodi, tecniche, metodologie



Rigore e formalità

- Sebbene lo sviluppo del software sia un'attività creativa, non va perseguita istintivamente in maniera scarsamente strutturata.
- Il **rigore** e il **formalismo** non limitano la creatività ma la complementano, come in ogni attività ingegneristica.
  - Ad esempio, possiamo dimostrare formalmente la correttezza di un algoritmo, mentre possiamo testare sistematicamente la correttezza di un programma.

Rigore e formalità

- Occorre scegliere il livello di rigore e formalità da raggiungere, in funzione della difficoltà concettuale e della criticità del compito che si sta affrontando.
- Questo livello può differire a seconda delle diverse parti di uno stesso sistema.
  - Ad esempio, lo schedulatore di processi del kernel di un sistema operativo in tempo reale può richiedere una descrizione formale del funzionamento richiesto, e una dimostrazione formale che questo funzionamento sia effettivamente ottenuto dal componente progettato.

Rigore e formalità

- La fase di codifica utilizza un approccio formale, in quanto i linguaggi di programmazione hanno una sintassi e semantica definite.
  - I compilatori sono in grado di verificare la correttezza formale di un programma e trasformarlo in un'altra rappresentazione equivalente (ad esempio in linguaggio macchina).
  - Queste operazioni automatiche, rese possibili dal formalismo dei linguaggi di programmazione, migliorano la verificabilità e l'affidabilità di un programma.

Rigore e formalità

- Rigore e formalismo possono essere applicati anche ad altre fasi lungo il ciclo di vita del software.
  - Ad esempio, una documentazione rigorosa (anche se non formale) può avere effetti benefici sulla manutenibilità, riusabilità, portabilità, comprensibilità e interoperabilità.
  - Una documentazione rigorosa può facilitare il riuso di parti del processo di sviluppo.

Separazione degli interessi

- Il **principio di separazione degli interessi** ci consente di affrontare differenti aspetti del problema, concentrando la nostra attenzione su ciascuno di essi in maniera separata.
  - In questo modo possiamo suddividere i compiti e le responsabilità nel progetto, parallelizzando alcune attività.

Separazione degli interessi

- Le decisioni da prendere durante lo sviluppo del software possono riguardare:
  - Le caratteristiche del prodotto (funzionalità, affidabilità, efficienza, relazioni con ambiente esterno, interfacce utente).
  - Il processo di sviluppo del prodotto (ambiente di sviluppo, struttura organizzativa del gruppo di lavoro, tempistiche di sviluppo, procedure di controllo, strategie di progettazione, meccanismi di gestione di eventuali malfunzionamenti).
  - Aspetti di tipo economico e finanziario.
  - …

Separazione degli interessi

- I modelli di sviluppo del software tipicamente separano le attività in diversi periodi temporali.
- Un altro tipo di separazione degli interessi riguarda le qualità che devono essere considerate separatamente.
  - Ad esempio, efficienza e correttezza potrebbero essere trattate separatamente, pensando prima alla correttezza e poi al miglioramento dell'efficienza.
- Un altro tipo di separazione riguarda la scomposizione del progetto in parti (moduli) disgiunti.

Separazione degli interessi

- Un'importante applicazione del principio di separazione degli interessi riguarda la separazione tra aspetti relativi al dominio del problema da quelli relativi al dominio dell'implementazione.
  - Ad esempio, nel dominio del problema sappiamo che un impiegato dipende da un dirigente, mentre nel dominio dell'implementazione parleremo di oggetti che possiedono puntatori ad altri oggetti (struttura dati).

Modularità

- Un sistema complesso può essere suddiviso in parti più piccole chiamate **moduli**.
- Un sistema composto da moduli è detto **modulare**.
- In questo modo è possibile applicare il principio della separazione degli interessi in due fasi:
  - Una fase prevede la trattazione dei dettagli di ogni singolo modulo separatamente.
  - Un'altra fase prevede la trattazione delle relazioni tra i moduli al fine di integrarli coerentemente.

Modularità

- L'approccio **bottom-up** prevede la progettazione del software trattando prima i singoli moduli separatamente e poi componendoli insieme.
- L'approccio **top-down** studia prima la scomposizione del problema in moduli, e poi studia i singoli moduli separatamente.

Modularità

- I moduli devono essere progettati in maniera che abbiano **alta coesione** e **basso accoppiamento**.
  - Tutti gli elementi di un modulo devono essere strettamente connessi e devono cooperare per realizzare la funzione richiesta per il modulo.
  - Moduli diversi devono dipendere poco l'uno dall'altro, così da poter essere analizzati, capiti, modificati, testati o riusati separatamente.

Modularità

- La modularità permea l'intero processo di sviluppo del software e dà luogo a quattro principali benefici:
  - Capacità di scomporre un sistema complesso in parti più semplici (*divide et impera*).
  - Capacità di comporre un sistema complesso a partire da moduli esistenti.
    - Favorisce il riuso e aumenta la velocità di costruzione.
  - Capacità di capire un sistema in funzione delle sue parti.
  - Capacità di modificare un sistema modificando soltanto un piccolo insieme delle sue parti.
    - Favorisce la riparabilità e l'evolvibilità.

Astrazione

- L’**astrazione** ci consente di identificare gli aspetti fondamentali di un fenomeno e di ignorare i suoi dettagli.
- Ciò che è importante e ciò che è un dettaglio dipende dallo scopo dell’astrazione.

Astrazione

- Esempi:
  - Quando si analizzano e specificano i requisiti di una nuova applicazione, gli ingegneri del software costruiscono un modello della potenziale applicazione, astraendo da numerosi dettagli poco importanti.
  - La stima dei costi per una nuova applicazione tipicamente passa per l'identificazione di fattori chiave, in modo da poter rendere confrontabile la nuova applicazione con applicazioni prodotte in passato.

Anticipazione del cambiamento

- **Anticipare il cambiamento** significa cercare di prevedere quali possano essere i futuri cambiamenti che saranno richiesti al software, e progettare opportunamente il software al fine di rendere tali cambiamenti più semplici possibile.
  - I probabili cambiamenti dovrebbero essere attribuibili a specifiche porzioni del software (moduli) e il loro effetto dovrebbe essere ristretto a tali porzioni.

Anticipazione del cambiamento

- Anticipare il cambiamento significa rendere i singoli moduli o componenti facilmente adattabili ad eventuali cambiamenti.
- Anche il processo di produzione del software dovrebbe anticipare possibili cambiamenti (ricambio del personale, stimare i costi dei cambiamenti anticipabili, prevedere e articolare le fasi di manutenzione,…).

Generalità

- Il principio di **generalità** ci dice che ogni volta che si deve risolvere un problema, si dovrebbe cercare di scoprire prima qual è il problema più generale che si nasconde dietro lo specifico problema da risolvere.
  - Spesso il problema generale è più semplice.
  - Una soluzione per il problema generale è più riusabile.
  - Una soluzione al problema generale potrebbe già esistere (ad esempio, un componente commerciale).

Generalità

- Una soluzione più generale potrebbe però essere più costosa (velocità di esecuzione, occupazione di memoria, tempo di sviluppo, ecc.).
  - Vanno valutati pro e contro.
- Il mercato offre numerosi prodotti di uso generale (*off-the-shelf*), molto spesso disponibili anche come servizi web.
  - Processamento testi
  - Fogli elettronici
  - Posta elettronica
  - Agende
  - …

Incrementalità

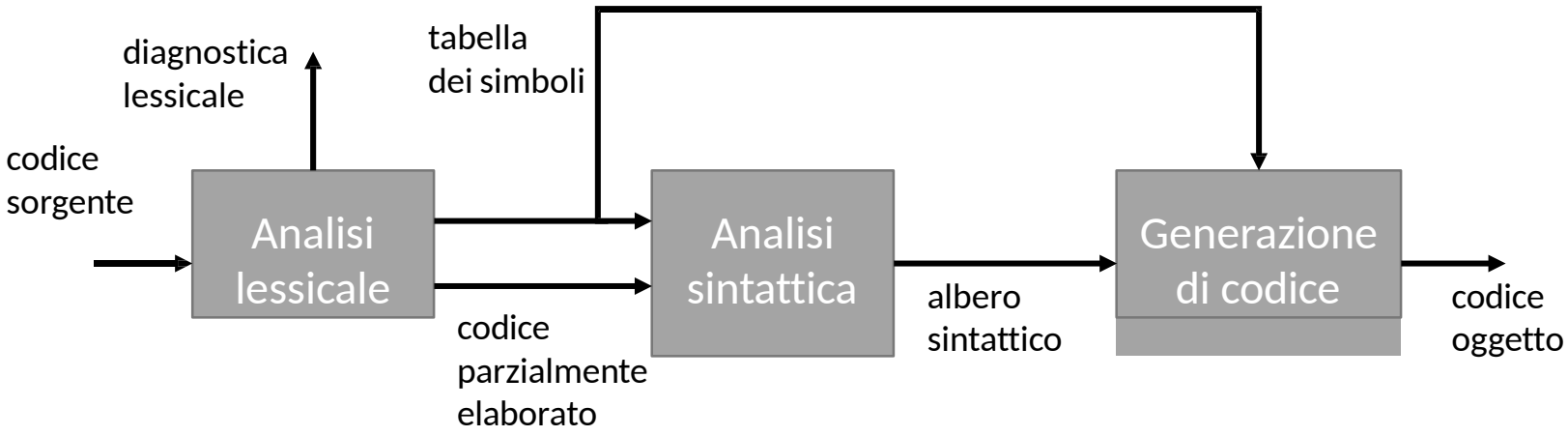
- Il principio dell'**incrementalità** prevede che il processo di produzione del software proceda attraverso una serie di approssimazioni successive.
  - Ad esempio, è possibile identificare da subito dei sottoinsiemi delle funzioni di un'applicazione che possano essere sviluppati subito e consegnati ai committenti (*prototipo*) al fine di ottenere dei feedback immediati.
    - Utile quando i requisiti iniziali non sono stabili o pienamente compresi.
    - Spesso i requisiti emergono solo quando l'applicazione è disponibile.
    - Favorisce l'anticipazione del cambiamento.

Esempio: compilatore

- Rigore e formalità: un compilatore è un prodotto critico, poiché se scorretto produrrebbe applicazioni scorrette.
  - Per tale motivo è fondamentale definire la sintassi di un linguaggio di programmazione in maniera formale (teoria degli automi e linguaggi formali).
- Separazione degli interessi: è opportuno e utile separare aspetti molto importanti quali la correttezza, l'efficienza, l'amichevolezza dell'interfaccia (i quali non necessitano di essere analizzati insieme).

Esempio: compilatore

- Modularità: ad esempio, è possibile realizzare un modulo per ogni fase principale (analisi lessicale, analisi sintattica, generazione di codice).



Esempio: compilatore

- Astrazione: esistono diversi tipi di astrazione in un compilatore.
  - Si distingue tra sintassi concreta e astratta al fine di ignorare alcuni dettagli sintattici ininfluenti.
  - In alcuni casi si passa attraverso la produzione di un linguaggio intermedio che astrae dalla macchina specifica su cui avviene la compilazione, al fine di supportare la portabilità del linguaggio.
- Anticipazione del cambiamento: i compilatori sono progettati avendo in mente modifiche frequenti quali nuovi processori, nuovi dispositivi di I/O, estensioni del linguaggio sorgente.

Esempio: compilatore

- Generalità: spesso i compilatori sono parametrici rispetto all'architettura hardware (si pensi al bytecode per il linguaggio Java).
- Incrementalità: spesso i compilatori vengono rilasciati in maniera incrementale, ampliando ad ogni rilascio la copertura rispetto al linguaggio sorgente.

Approfondimenti

- Ghezzi, Jazayeri, Mandrioli: «Ingegneria del software», 2° Edizione, Pearson (2004)
  - Capitoli 2 e 3