DIETI UNINA UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

Corso di Ingegneria del Software

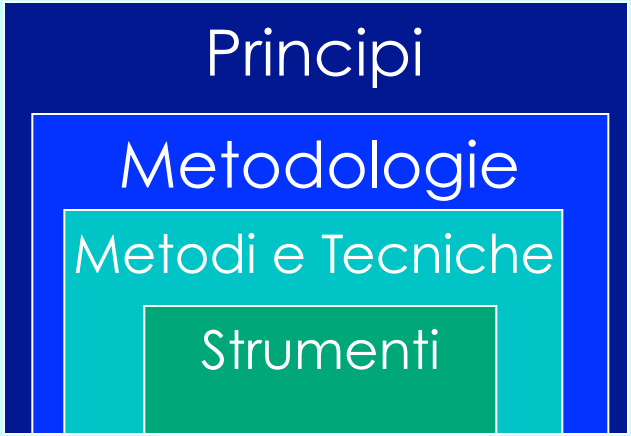
Principi della Ingegneria del Software

Sommario

- Principi dell'Ingegneria del software
  - Rigore, formalità
  - Astrazione, Separazione degli interessi
  - Differimento delle decisioni
  - Modularità, Incapsulamento, *Information Hiding*
  - Relazione tra moduli. Coesione e Accoppiamento
  - Anticipazione del cambiamento, Generalità, Incrementalità

Principi dell'ingegneria del software

- Si tratta di principi generali ed astratti, che ispirano il processo software e/o metodi e tecniche adottati in singole fasi
- Necessari, ma non sufficienti a garantire la qualità
- L'individuazione di **metodi** e **tecniche** inseriti in un processo adeguato alla ingegnerizzazione di un prodotto software è obiettivo di una **metodologia di Ing. del Sw**
- Metodologie, metodi e tecniche si servono di **strumenti**, che ne facilitano l'applicazione



Rigore e Formalità (1/2)

- **Rigore**: necessario a strutturare le attività di sviluppo, fornendo precisione e accuratezza
- Esistono diversi livelli di rigore
  - La **formalità** è il livello più alto del rigore: il processo di sviluppo formale è guidato da **procedimenti matematici**
- Non è sempre possibile o conveniente essere formali. Bisogna capire quando e dove esserlo
  - *Ad es., parti critiche di un sistema possono richiedere una descrizione formale del funzionamento, o una dimostrazione formale di correttezza. Tuttavia, ciò costa*

Rigore e Formalità (2/2)

- La formalità favorisce l'**automazione**
- Si applica in tutte le fasi: **specifica** dei requisiti, **progettazione**, **codifica**, **verifica**
  - Requisiti specificati in modo formale possono consentire la generazione automatica di codice, o la verifica formale
  - La codifica è tradizionalmente formale: i programmi sono oggetti formali, scritti in un linguaggio con sintassi e semantica ben definite.
- *Rigore e formalità hanno benefici sulla affidabilità, verificabilità, manutenibilità, riusabilità, comprensibilità, portabilità, interoperabilità*

Separazione degli interessi (1/2)

(separation of concerns)

- Consente di affrontare diversi aspetti del problema focalizzandosi su ciascuno separatamente
- Essenziale per **dominare la complessità**
- **Separazione temporale:** è alla base dei modelli di ciclo di vita, che definiscono le fasi e gli artefatti di un processo di produzione del software
- **Separazione dei fattori di qualità**: ad es. separare correttezza ed efficienza, progettando prima per assicurare la correttezza e focalizzandosi poi sull'efficienza
- **Separazione di viste diverse:** ad es. concentrarsi separatamente sul flusso di dati e sul flusso di controllo

Separazione degli interessi (2/2)

- **Separazione di parti del sistema:** progettazione di diverse parti del sistema separatamente → definizione di sottosistemi/componenti/moduli
- **Separazione di domini:** separare aspetti del dominio del problema da quelli del dominio della soluzione e da quelli dell'implementazione
- Può determinare la separazione di **ruoli** e **responsabilità** in un'azienda

Differimento delle decisioni

- Ogni decisione va presa al momento giusto, senza anticipare il momento della decisione rispetto a quando prenderla è effettivamente improcrastinabile
- Esempi:
  - La scelta del linguaggio di programmazione non deve essere anticipata alla fase di analisi (a meno che non rappresenti un vincolo, per es. dato dal committente)
  - La scelta degli strumenti di sviluppo va differita alla fase di programmazione
  - Le scelte di progetto (architettura software) non devono essere prese in fase di specifica dei requisiti

L'Astrazione

- L'**astrazione** è il processo che porta ad identificare le proprietà rilevanti di un'entità (o di un fenomeno), ignorando i dettagli inessenziali
  - Le proprietà così astratte definiscono una **vista** dell'entità (o del fenomeno)
  - Una stessa entità può dar luogo a viste diverse
- *Esempio: un'automobile*
  - *vista dal venditore:*
    prezzo, durata della garanzia, colore, ...
  - *vista dal meccanico:*
    tipo di motore, cilindrata, tipo di olio, ...

La Modularità

La **modularità** è l'organizzazione in parti (per moduli) di un modello o di un sistema, in modo che esso risulti più semplice da comprendere e manipolare

Gran parte dei sistemi complessi sono modulari

Esempio: Un'automobile è suddivisa in più sottosistemi:

- *Motore*
- *Trasmissione*
- *...*

Modulo software

Un modulo di un sistema software è un componente che:

- Realizza una astrazione
- È dotato di netta separazione tra ***Interfaccia e Corpo***

L'**interfaccia** specifica il ***cosa*** (l'astrazione realizzata dal modulo).
Il **corpo** descrive ***come*** essa è realizzata.

Un modulo può realizzare tanti tipi di astrazione
- ***realizzare una funzionalità***
- ***fornire un servizio***
- ***gestire una risorsa***
- ***rappresentare una classe di oggetti***
- ***…***

Incapsulamento e *information hiding*

L'**incapsulamento** consiste nel **nascondere** e **proteggere** alcuni aspetti (dati, algoritmi) di un modulo

- L'accesso alle informazioni nascoste è controllato, ad es. tramite le operazioni descritte dall'interfaccia
  - Se l'interfaccia non cambia, le informazioni nascoste possono essere modificate senza che questo influisca sulle altre parti del sistema di cui l'entità fa parte
- *Esempio: un'autoradio*
  - *L'interfaccia consiste dei controlli e dei connettori tramite i quali è collegata all'automobile*
  - *I dettagli di come funziona sono nascosti*
  - *Per installarla e usarla non è necessario conoscere alcunché della sua struttura interna*

Tipi di astrazione

Le astrazioni più tipicamente realizzate da un modulo sono:

ASTRAZIONE SUL CONTROLLO

- Consiste nell'astrarre una data **funzionalità** dai dettagli della sua implementazione;
- È ben supportata dai linguaggi di programmazione tradizionali tramite il concetto di **sottoprogramma**.

ASTRAZIONE SUI DATI

- Consiste nell'astrarre le ***entità*** (oggetti) costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa;
- Può essere realizzata con un uso opportuno delle tecniche di **programmazione modulare** nei linguaggi tradizionali;
- È supportata da appositi costrutti nei linguaggi di **programmazione ad oggetti.**

Categorie di Moduli (1/2)

- Le tipologie di astrazioni danno luogo a diverse categorie di moduli:
- **Procedura o Funzione**. Astrazione procedurale (**sul controllo**):
  - Fornisce una procedura o una funzione che implementa delle operazioni
  - Viene incapsulato un algoritmo
- **Libreria**. Gruppo di astrazioni procedurali correlate:
  - Insieme di funzioni
  - *Ad esempio, una librerie di funzioni matematiche*

Categorie di Moduli (2/2)

- **Oggetti** (**astrazione sui dati**):
  - *Strutture dati* + *operazioni*
  - Rispetto alle librerie, hanno una struttura dati permanente, visibile solo alle funzioni interne
  - La struttura dati fornisce loro uno ***stato***
- **Tipi di dati astratti**
  - Insieme alle operazioni possibili sui dati, esporta un ***tipo***
  - Gli esemplari del tipo sono ***oggetti astratti***

Altre possibili categorie di moduli

- ***Pool* comune di dati**
  - *Ad es. raggruppare tutte le costanti di configurazione in un pool di dati cui accedere durante la fase di configurazione*
  - E' un modulo di livello basso (non "nasconde" nulla; non fornisce alcuna astrazione)
- **Moduli generici**
  - Sono **template** di moduli
  - Sono parametrizzati rispetto ad un tipo
  - Per essere usati, devono essere prima istanziati fornendo parametri reali (tipi)

Benefici della Modularità

- **Scomposizione** di un sistema in parti
  - Utile per dominare la complessità
- **Separazione degli interessi**
  - Trattare i dettagli dei *moduli* separatamente
- **Suddivisione del lavoro**
  - Persone/team diversi, anche in parallelo
- **Composizione** di un sistema (*bottom-up*)
- **Riuso** di moduli esistenti
- **Analisi** di un sistema in relazione alle sue parti
- **Manutenzione**

Coesione e Accoppiamento (1/2)

- Per massimizzare i benefici, bisogna progettare moduli con:
  - **alta coesione**
  - **basso accoppiamento**
- **Alta coesione**: se tutti gli elementi del modulo (istruzioni, procedure, dichiarazioni) sono logicamente ben coesi
  - Gli elementi sono raggruppati per *un motivo logico* e cooperano per la realizzazione della astrazione fornita
  - È una proprietà interna al modulo

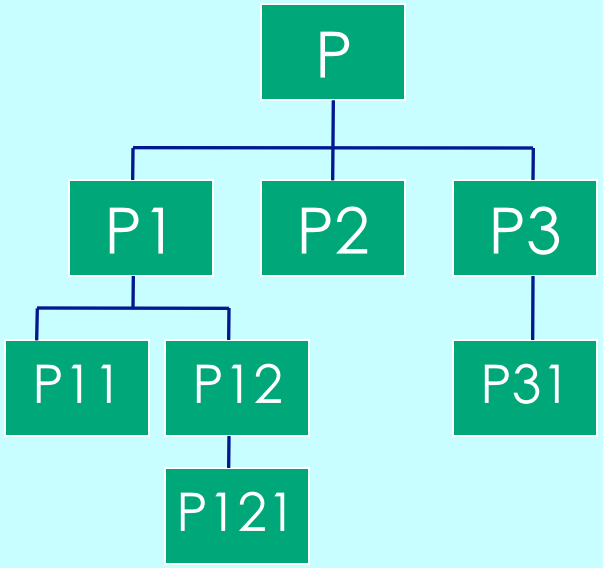
Coesione e Accoppiamento (2/2)

- **L'accoppiamento** misura l'interdipendenza tra moduli
  - Basso accoppiamento -> *buon livello di separazione tra i moduli* (è più semplice analizzare, capire, modificare, testare e riusare i singoli moduli)
  - Alto accoppiamento -> indice di *poor design*

Metodologie di progetto: *top-down* e *bottom-up* (1/2)

La metodologia discendente o "**top-down**" è basata su un approccio di **decomposizione funzionale** nella definizione del sistema software, cioè sull'individuazione delle funzionalità del sistema da realizzare e su raffinamenti successivi, da iterare finché la scomposizione del sistema individua sottosistemi di complessità accettabile.

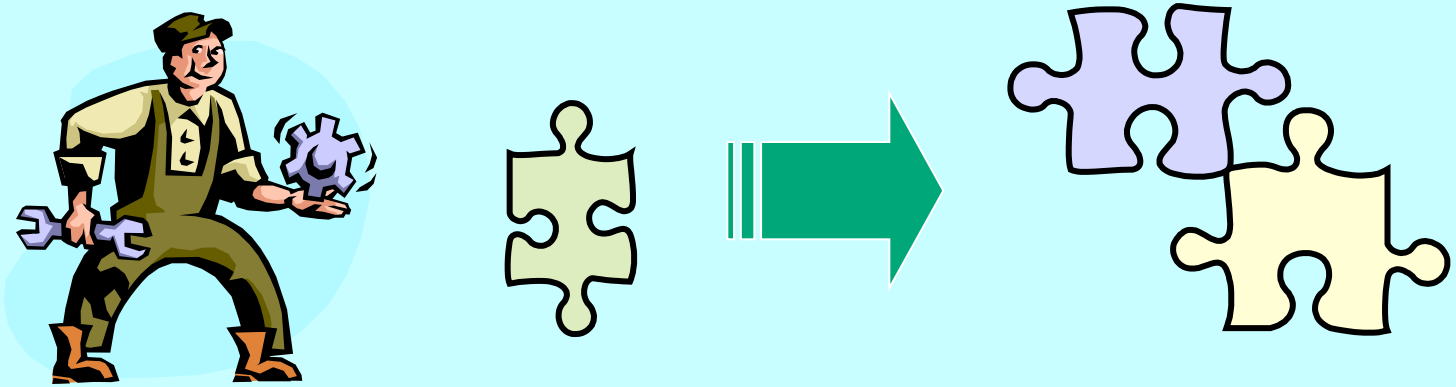
Le foglie dell'albero sono sottosistemi già esistenti (riusabili) o di complessità accettabile



Metodologie di progetto: *top-down* e *bottom-up* (2/2)

La metodologia ascendente o "**bottom-up**" è basata sull'individuazione delle entità (classi e/o oggetti) facenti parte del sistema, delle loro proprietà e delle interrelazioni tra di esse.

Il sistema viene costruito assemblando componenti con un approccio "dal basso verso l'alto"



Relazioni tra moduli (1/2)

- **Relazione *Uses*:** $\mathbf{M_i}$ **USES** $\mathbf{M_j}$ se richiede la presenza di M_j per portare a termine il suo compito. E' rappresentabile tramite un grafo
  - Valutando il **fan-out** (archi uscenti da un nodo) e il **fan-in** (archi entranti) si hanno indicazioni sulla bontà della modularizzazione
  - Sono desiderabili un basso *fan-out* e un alto *fan-in* (i.e., astrazione molto usata)
- **Relazione *IsComponentOf*:** $\mathbf{M_i}$ **IsComponentOf** $\mathbf{M_j}$ se M_j è realizzato aggregando uno o più moduli, tra cui M_i

Relazioni tra moduli (2/2)

- Partendo dalla relazione **IsComponentOf** è possibile definire anche le seguenti relazioni
  - **Comprises**: M_i **Comprises** M_j se e solo se M_j **IsComponentOf** M_i (relazione inversa di IsComponentOf)
  - **IsComposedOf**: sia S un insieme di moduli e $M_{s,i} = \{M_k \mid M_k$ è in **S** e M_k **IsComponentOf** $M_i\}$;

    M_i **IsComposedOf** $M_{s,i}$ se i moduli dell'insieme $M_{s,i}$ forniscono tutti i servizi che devono essere forniti da M_i
    - ✓ Sono il risultato della scomposizione di M_i
  - La relazione inversa è $M_{s,i}$ **Implements** M_i

Anticipazione del cambiamento

- **Anticipazione del cambiamento**: la capacità di far evolvere un prodotto software deve essere **pianificata** con cura
- Il software è soggetto a cambiamenti molto più di altri prodotti ingegneristici
  - Deve essere progettato per facilitare i cambiamenti
- I cambiamenti dovrebbero essere attribuibili a specifiche porzioni del software: questo principio richiede una **corretta modularizzazione**

Generalità

- **Generalità**: a partire da una esigenza specifica, è spesso opportuno valutare se conviene risolvere un problema più generale, facilitando ad es. il riuso
- **Vantaggi**:
  - È possibile che sia già risolto da componenti commerciali
  - Il risparmio non è mai guadagno … ☺
- **Svantaggi**: può essere più costosa rispetto a una soluzione specializzata
- Dal punto di vista commerciale, la tendenza a creare soluzioni generali (*ad es.* ***componenti Commercial Off-The-Shelf*** o **COTS**) è molto marcata

Incrementalità

- **Incrementalità**: raggiungere l'obiettivo attraverso una serie di passi incrementali
  - Un modo è identificare sottoinsiemi del prodotto da consegnare subito per avere *feedback*: consente una **evoluzione controllata**
- Spesso i requisiti emergono contemporaneamente alla disponibilità dell'applicazione o del suo utilizzo

  => prima si ha un *feedback*, prima si incorporano i cambiamenti desiderati
- Si applica anche al miglioramento dei fattori di qualità: *ad es., una prima versione può trascurare le prestazioni in favore di altri fattori*
- **E' alla base di modelli di ciclo di vita evolutivi**

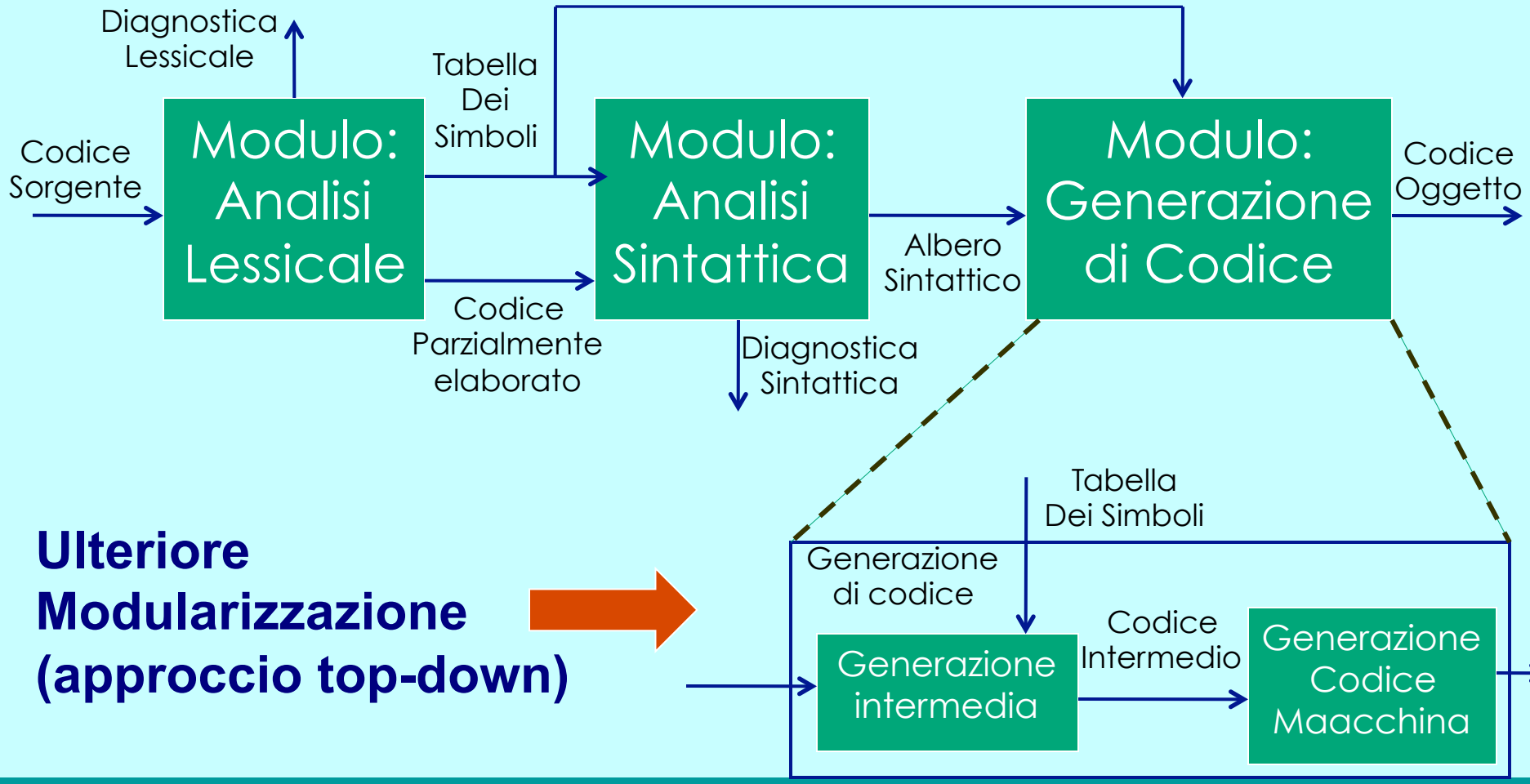
Esempio

- Applicazione dei principi descritti allo **sviluppo di un compilatore**
- **Rigore e formalità**: definire la sintassi in **modo formale** (solitamente tramite la forma di *Backus-Naur,* BNF)
- **Separazione degli interessi**: affrontare correttezza (produrre codice oggetto consistente con codice sorgente) e efficienza (riduzione del tempo di compilazione) o amichevolezza (completezza e comprensibilità dei messaggi diagnostici)

  separatamente

Esempio

- **Modularità:** es., una modularizzazione *funzionale:*



Ulteriore Modularizzazione (approccio top-down)

Esempio

- **Astrazione/Generalità:** ad es., per la sintassi si è soliti distinguere **tra sintassi astratta e concreta**
- *Ad es., una istruzione condizionale è composta di una condizione, di un blocco di istruzioni da eseguire se la condizione è vera, e un altro blocco se è falsa. Ciò è valido indipendentemente dai dettagli del linguaggio (ad esempio, il C non usa "then", il Pascal sì)*
- Si potrebbe avere come obiettivo la produzione di una **famiglia di compilatori**
- Una "estremizzazione" è la creazione di **generatori di compilatori**
  - Prende in ingresso la definizione dei linguaggi sorgente e oggetto, e genera un compilatore (ad es. **Lex e Yacc**)

Esempio

- **Anticipazione del cambiamento**
- Si dovrebbe ad esempio prevedere che:
  - Possono avvenire nuovi rilasci del processore
  - Siano introdotti nuovi dispositivi di I/O, che richiedono nuove istruzioni
  - I comitati di standardizzazione potrebbero estendere il linguaggio sorgente
- Il linguaggio *Pascal* è stato un esempio di errata anticipazione del cambiamento
  - Si tentò di congelare le istruzioni di I/O con definizioni rigide, ma dipendenze tipiche dalla macchina causarono la nascita di numerosi dialetti che differivano per gli aspetti di I/O

Esempio

- **Incrementalità**
- Si può consegnare una prima versione di compilatore per un sottoinsieme del linguaggio sorgente
- Si possono aggiungere capacità diagnostiche e ottimizzazioni solo in versioni successive