Progettazione del Software

Introduzione alla Progettazione del SW

Domenico Fabio Savo

Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti

SAPIENZA Università di Roma

Le slide di questo corso sono il frutto di una rielaborazione di analogo materiale redatto da Marco Cadoli, Giuseppe De Giacomo, Maurizio Lenzerini e Domenico Lembo

Prima Parte

Introduzione alla Progettazione del SW

- **Contesto organizzativo**
- Ciclo di vita del software
- Qualità
- Modularizzazione
- Principi di base dell'orientazione agli oggetti

Il contesto organizzativo

Attori nella progettazione del software:

- Committente
- Esperti del domino
- Analista
- Progettista
- Programmatore
- Utente finale
- Manutentore

Esercizio 1

Il Comune di **XYZ** intende automatizzare la gestione delle informazioni relative alle contravvenzioni elevate sul suo territorio.

In particolare, intende dotare ogni vigile di un dispositivo palmare che gli consenta di comunicare al sistema informatico il veicolo a cui è stata comminata la contravvenzione, il luogo in cui è stata elevata e la natura dell'infrazione.

Il sistema informatico provvederà a notificare, tramite posta ordinaria, la contravvenzione al cittadino interessato.

Il Comune bandisce una gara per la realizzazione e manutenzione del sistema, che viene vinta dalla Ditta **ABC**.

Quali sono gli attori coinvolti in questa applicazione SW?

Classificazione delle applicazioni (1)

- Le applicazioni posso essere classificate in base a differenti fattori, tra cui:
  - Il flusso di controllo
  - Gli elementi di interesse primario

Nel dettaglio…

Classificazione delle applicazioni (2)

Rispetto al flusso di controllo

- **Sequenziali:** un unico flusso di controllo governa l'evoluzione dell'applicazione
- **Concorrenti:** le varie attività necessitano di sincronizzazione e comunicazione
    - Composte da varie attività sequenziali che possono (e devono) essere sincronizzate al fine di garantire la correttezza
    - Il tempo di esecuzione influenza le prestazioni, non la correttezza

      → corsi di *Sistemi Operativi*
- **Dipendenti dal tempo:** esistono vincoli temporali riguardanti sia la velocità di esecuzione delle attività sia la necessità di sincronizzare le attività stesse

Esercizio 2

Considerare i seguenti contesti applicativi:

- sistema di controllo di una centrale nucleare
- sistema di prenotazione dei voli di un aeroporto
- risolutore di sistemi di equazioni
- sistema di gestione di una banca dati
- sistema operativo di un elaboratore elettronico

Per ciascuno di essi, fareste ricorso ad applicazioni sequenziali, concorrenti o dipendenti dal tempo?

Classificazione delle applicazioni (2)

Rispetto agli elementi di interesse primario

- **Orientate alla realizzazione di funzioni:** la complessità prevalente del sistema riguarda le funzioni da realizzare
- **Orientate alla gestione dei dati:** l'aspetto prevalente è rappresentato dai dati che vengono memorizzati, ricercati, e modificati, e che costituiscono il patrimonio informativo di una organizzazione

  → corso di *Basi di Dati*
- **Orientate al controllo**: la complessità prevalente del sistema riguarda il controllo delle attività che si sincronizzano e cooperano durante l'evoluzione del sistema

Applicazioni di interesse per questo corso

- Sequenziali
- Orientate alla realizzazione di funzioni

Sono le applicazioni più tradizionali, e vengono spesso adottate come riferimento per i metodi e le tecniche di base per la progettazione

Prima Parte

Introduzione alla Progettazione del SW

- Contesto organizzativo
- **Ciclo di vita del software**
- Qualità
- Modularizzazione
- Principi di base dell'orientazione agli oggetti

Ciclo di vita del software

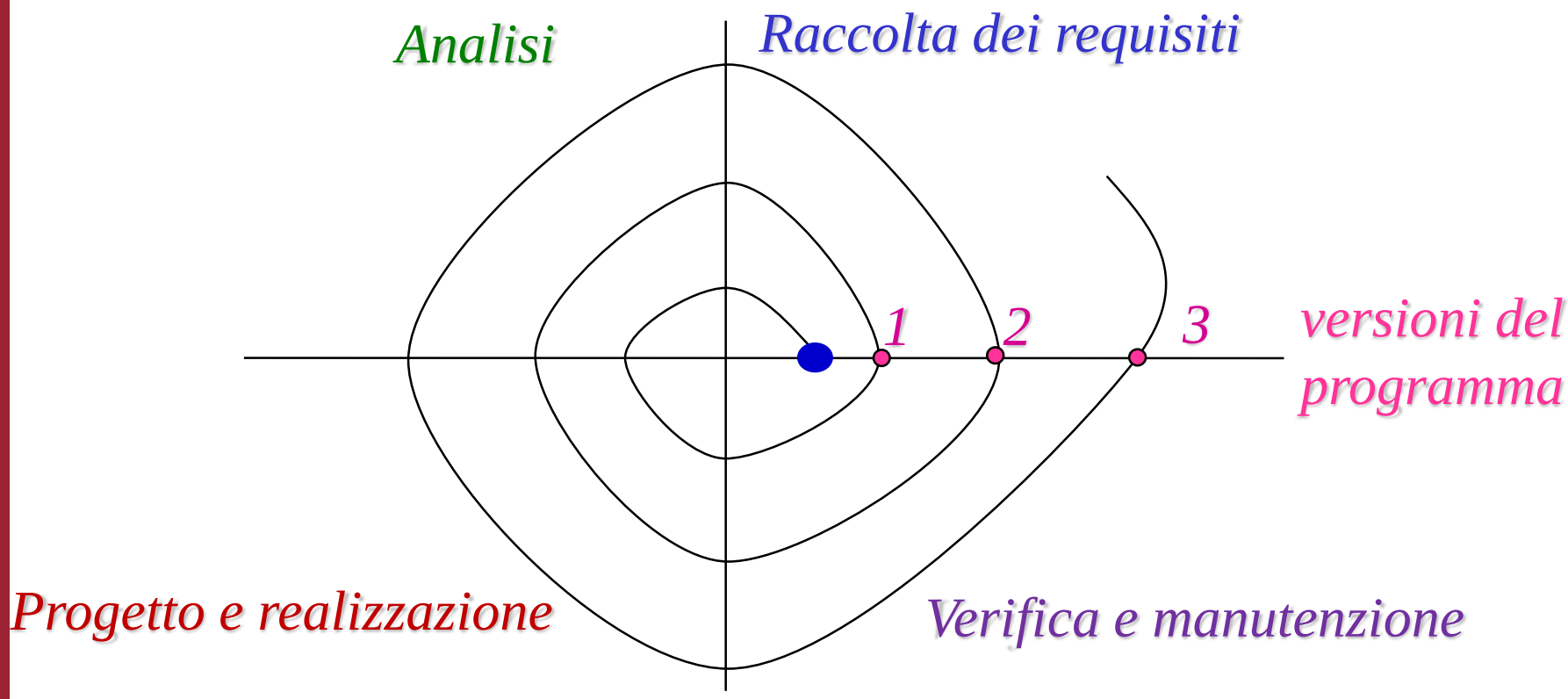
1. Studio di fattibilità e raccolta dei requisiti
   - valutare costi e benefici
   - pianificare le attività e le risorse del progetto
   - individuare l'ambiente di programmazione (hardware/software)
   - raccogliere i requisiti
2. Analisi dei requisiti
   - si occupa del cosa l'applicazione dovrà realizzare
   - descrivere il dominio dell'applicazione e specificare le funzioni delle varie componenti: lo schema concettuale
3. Progetto e realizzazione
   - si occupa del come l'applicazione dovrà realizzare le sue funzioni
   - definire l'architettura del programma
   - scegliere le strutture di rappresentazione
   - scrivere il codice del programma e produrre la documentazione

Ciclo di vita del software

1. Studio di fattibilità
2. Analisi dei requisiti
3. Progetto e realizzazione
4. Verifica
   - Il programma svolge correttamente, completamente, efficientemente il compito per cui è stato sviluppato?
5. Manutenzione
   - Controllo del programma durante l'esercizio
   - Correzione e aggiornamento del programma

Se, necessario, si torna alla fase di raccolta dei requisiti

Ciclo di vita del software (modello a spirale)



Prima Parte

Introduzione alla Progettazione del SW

- Contesto organizzativo
- Ciclo di vita del software
- **Qualità**
- Modularizzazione
- Principi di base dell'orientazione agli oggetti

Fattori di qualità del SW

I fattori di qualità del SW sono classificati in due famiglie:

ESTERNI

Correttezza
Affidabilità
Robustezza
Sicurezza
Innocuità
Usabilità
Scalabilità
Interoperabilità

INTERNI

Efficienza
Strutturazione
Modularità
Comprensibilità
Verificabilità
Manutenibilità
Portabilità
Riusabilità

Qualità esterne ed interne

- Qualità esterne:
  - **Sono le qualità visibili agli utenti del sistema**
  - Percepibili anche da chi non è specialista
  - Non richiedono l'ispezione del codice sorgente

- Qualità interne:
  - **Sono le qualità che riguardano gli sviluppatori**
  - Valutabili da specialisti
  - Richiedono conoscenza della struttura del programma

A volte la distinzione fra qualità esterne ed interne non è perfettamente marcata

Esercizio 3

Considerare le seguenti proprietà di un impianto Hi-Fi e stabilire quali di esse sono esterne e quali sono interne.

1. il tempo dedicato al collaudo,
2. la fedeltà sonora,
3. la probabilità di malfunzionamenti nel primo anno di utilizzazione,
4. la dimensione del trasformatore per l'alimentazione,
5. l'ergonomia dei comandi,
6. la linearità della risposta in frequenza.

Qualità Esterne (1)

- **Correttezza**: il software fa quello per il quale è stato progettato. Questa è una qualità fondamentale.
- **Affidabilità**: quanto l'utente può fare affidamento sulle funzionalità del software (correttezza nel tempo).
- **Robustezza**: comportamento accettabile anche nel caso di situazioni non previste nella specifica dei requisiti.
- **Innocuità**: il sistema non entra in certi stati (pericolosi).
- **Sicurezza**: riservatezza nell'accesso alle informazioni.

Qualità Esterne (2)

- **Usabilità:** è la facilità con cui un utente impara ad operare con un sistema o un componente, a fornirgli gli input e ad interpretarne gli output.

- **Scalabilità:** è la capacità del SW di adattarsi per fare fronte alle necessità e disponibilità senza richiedere la modifica del codice. Come si comporta il sistema al variare della dimensione dell'input o del numero degli utenti? Possiamo utilizzare il SW su macchine con capacità di calcolo differenti? Maggiori risorse = migliori prestazioni?

- **Interoperabilità:** Facilità di interazione con altri moduli al fine di svolgere un compito più complesso.

Qualità Interne (1)

- **Efficienza**: si riferisce al “peso” che il software ha sulle risorse del sistema quali tempo di esecuzione e memoria interna.
- **Strutturazione**: capacità del SW di riflettere con la sua struttura le caratteristiche del **problema** trattato e delle **soluzioni** adottate.
- **Modularità**: grado di organizzazione del SW in parti ben specificate ed interagenti.
- **Comprensibilità**: capacità del SW di essere compreso e controllato anche da parte di chi non ha condotto il progetto. Facilitata da una buona strutturazione e modularità, influenza positivamente la l’analisi della correttezza ed il riuso.

Qualità Interne (2)

- **Verificabilità**: la possibilità di verificare che gli obiettivi di qualità proposti siano stati raggiunti. E' possibile verificare che il SW fa quello per cui è stato progettato? Possiamo sapere quanto il SW è affidabile?

- **Manutenibilità**: capacità del SW di essere modificato dopo il suo rilascio per correggere eventuali errori (bug nel codice ed errori introdotti da specifiche errate), effettuare modifiche adattive o di miglioramento.

- **Portabilità**: capacità del prodotto di operare su diversi ambienti (piattaforme e sistemi operativi). Favorito da linguaggi compatibili con diverse piattaforme o dall'uso di macchine virtuale (es., Java).

- **Riusabilià:** Facilità con cui il SW può essere re-impiegato in applicazioni diverse da quella originaria.

Esercizio 4

Assimilando le qualità di un programma alle proprietà delle automobili, il fatto che i motori di un certo modello possono essere montati su diversi modelli di automobile a quale qualità interne fa riferimento?

Misura della qualità

- Ogni qualità deve essere valutata attraverso alcune proprietà possedute dalle entità e misurabili in modo oggettivo e quantitativo

- Differenti entità possono essere collocate su una scala di valori in funzione dei livelli misurati per questi attributi.

Qualità in Contrasto

- Non tutte le qualità possono essere massimizzate
- Alcune sono intrinsecamente **in contrasto** fra loro.

  Ad esempio:

- È necessario scegliere un adeguato bilanciamento

Esercizio 5

- Le qualità dei programmi non sono necessariamente indipendenti tra loro.
- Ad esempio, all'aumentare della modularità aumenta in genere anche la leggibilità.
- Considerare due qualità alla volta e valutare il loro rapporto reciproco.

Tendenza nello sviluppo di applicazioni SW

Complessità delle informazioni da gestire 

Progetti di medio-grandi dimensioni 

Eterogeneità degli utenti 

Durata media dei sistemi 

Bisogno di interventi di manutenzione 

Costi di produzione e tempi di produzione 

Qualità del prodotto finito 

Principi guida nello sviluppo del software

In base alle tendenze descritte precedentemente, attualmente si considerano fondamentali nello sviluppo del software i seguenti principi:

Rigore e formalità	Lo sviluppo del software è una attività creativa che va accompagnata da un approccio rigoroso (o formale: in logica o matematica) che permette di realizzare prodotti affidabili, controllarne il costo, aumentare la fiducia nel loro corretto funzionamento.
Separazione degli interessi	Affrontare separatamente i diversi aspetti per dominare la complessità
Modularità	Realizza la separazione degli interessi in 2 fasi: • Tratta i dettagli di singoli moduli in modo separato • Tratta separatamente dai dettagli interni dei singoli moduli le relazioni che sussistono tra i moduli stessi
Astrazione	Identifica aspetti fondamentali ed ignora i dettagli irrilevanti
Anticipazione del cambiamento	Per favorire l'estendibilità e il riuso
Generalità	Ricerca di soluzioni generali
Incrementalità	Per anticipare feedback dell'utente per facilitare verifiche di correttezza per predisporsi alla estendibilità e al riuso

Prima Parte

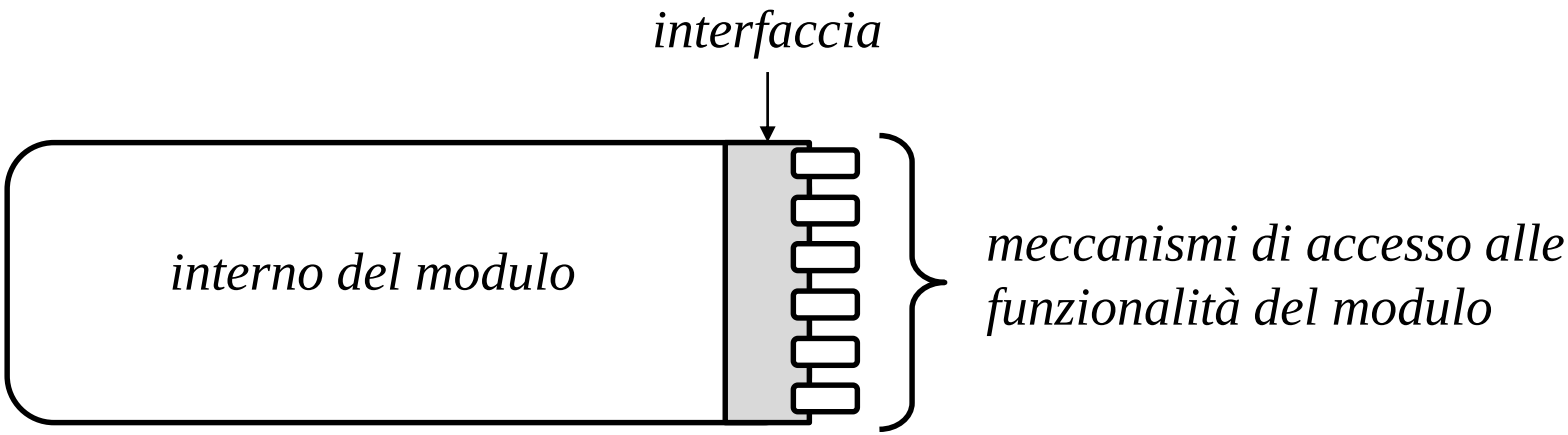
Introduzione alla Progettazione del SW

- Contesto organizzativo
- Ciclo di vita del software
- Qualità
- **Modularizzazione**
- Principi di base dell'orientazione agli oggetti

Modularizzazione

- La **Modularizzazione** è il principio secondo il quale il software è strutturato secondo unità, dette appunto **moduli**
- Un modulo è una unità di programma con le seguenti caratteristiche:
  - ha un obiettivo chiaro;
  - ha relazioni strutturali con altri moduli;
  - offre un insieme ben definito di servizi agli altri moduli (server);
  - può utilizzare servizi di altri moduli (client).
- Principio fondamentale sia per la **strutturazione** sia per la **modularità**
- Concettualmente alla base dell' architettura client/server del software

Modulo



Il modulo è un componente software realizzato con un scopo bene preciso ed in grado di eseguire le funzioni per cui è stato progettato in modo autonomo.

Principi della modularità

- **Principio di unitarietà** *(incapsulamento, alta coesione)*
  un modulo deve corrispondere ad una unità concettuale ben definita e deve incorporare tutti gli aspetti relativi a tale unità concettuale
- **Poche interfacce** *(basso accoppiamento)*
  un modulo deve comunicare con il minor numero di moduli possibile (**quelli necessari**)
- **Poca comunicazione** *(basso accoppiamento)*
  un modulo deve scambiare meno informazione possibile (**quella necessaria!**) con gli altri moduli
- **Comunicazione chiara** *(interfacciamento esplicito)*
  informazione scambiata predeterminata e più astratta possibile
- **Occultamento di informazioni inessenziali** *(information hiding)*
  le informazioni che non devono essere scambiate devono essere gestite privatamente dal modulo

Principi della modularità

In sintesi, i quattro dogmi della modularizzazione sono:

- ***Alta coesione*** *(omogeneità interna)*
- ***Basso accoppiamento*** *(indipendenza da altri moduli)*
- ***Interfacciamento esplicito*** *(chiare modalità d'uso)*
- ***Information hiding*** *(poco rumore nella comunicazione)*

Principi della modularità

- Esempi negativi:
  - *Bassa coesione*

    Allo stesso sportello degli uffici postali si pagano i bollettini di conti corrente e contemporaneamente si ritirano le pensioni (servizi disomogenei!)
  - *Alto accoppiamento*

    Per usufruire di alcuni servizi delle circoscrizioni, occorre acquistare delle marche da bollo dal tabaccaio (accoppiamento tra tabaccaio e ufficio circoscrizionale)
- Esempi positivi:
  - *Interfacciamento esplicito*

    In molte pagine web (ad esempio per prenotazione posti in un teatro), le informazioni da fornire sono chiaramente espresse mediante campi da riempire
  - *Information hiding*

    L'utente non conosce esattamente cosa è memorizzato nella banda magnetica delle carte di credito

Cattiva modularizzazione (esempio 1)

```
public class Server {
    public static int alfa;

    public static bool Cerca(Lista x){

       do {  if (x.info == alfa)  return true;

                x = x.next;
            }
        while (x != null);
        return false;
    }
}
```

Il metodo **Cerca** restituisce true se il valore intero memorizzato nel campo alfa è presente nella lista passata in input (parametro x). Altrimenti, restituisce false.

Cattiva modularizzazione (esempio 1)

```
public class Server {
    public static int alfa;                          // ALTO ACCOPPIAMENTO: impone al client di
                                                     // usare alfa in maniera coerente
    public static bool Cerca(Lista x){               // BASSO INTERFACCIAMENTO ESPLICITO:
                                                     // non è esplicito che alfa è il valore cercato
      do {  if (x.info == alfa)  return true;        // BASSA COESIONE: non effettua il controllo
                                                     // di lista vuota
              x = x.next;
           }
       while (x != null);
       return false;
    }
}
```

Il metodo **Cerca** restituisce true se il valore intero memorizzato nel campo alfa è presente nella lista passata in input (parametro x). Altrimenti, restituisce false.

Cattiva modularizzazione (esempio 2)

```
public class Client {
  static Lista MiaLista() { … }
  public static void main(String[] args) {
      Lista s = MiaLista();
       if (s != null) {

         System.out.print("Inserisci un intero: ");
         Server.alfa = InOut.readInt();

         if (Server.Cerca(s))
                  System.out.print(Server.alfa+"presente");
         else     System.out.print(Server.alfa+" non  presente");
       }
    }
}
```

Il main verifica se un intero letto da riga di comando è presente nella lista s, e stampa un opportuno messaggio.

Cattiva modularizzazione (esempio 2)

```
public class Client {
  static Lista MiaLista() { … }
  public static void main(String[] args) {
      Lista s = MiaLista();
       if (s != null) {                          // BASSA COESIONE: il client
                                                 // deve controllare che la lista non sia vuota
         System.out.print("Inserisci un intero: ");
         Server.alfa = InOut.readInt();          // ALTO ACCOPPIAMENTO: il client deve
                                                 //  definire e usare alfa come vuole il server
         if (Server.Cerca(s))
                  System.out.print(Server.alfa+"presente");
         else     System.out.print(Server.alfa+" non  presente");
       }
    }
}
```

Il main verifica se un intero letto da riga di comando è presente nella lista s, e stampa un opportuno messaggio.

Buona modularizzazione (esempio 1)

```
public class Server {

    public static bool Cerca(Lista x, int alfa){      // ALTO INTERFACCIAMENTO ESPLICITO:
                                                      // è esplicito che alfa è il valore cercato
                                                      // ALTA COESIONE: effettua il controllo di
                                                      // lista vuota
        while (x != null)
        {  if (x.info == alfa)  return true;
            else x = x.next;
         }
         return false;
    }
}
```

Buona modularizzazione (esempio 2)

```
public class Client {
    static Lista MiaLista() { … }
    public static void main(String[] args) {
        Lista s = MiaLista();
        System.out.print("Inserisci un intero: ");
        int alfa = InOut.readInt();
        if (Server.Cerca(s, alfa))              // BASSO ACCOPPIAMENTO: il cliente si
                                                // concentra sulla richiesta del servizio
            System.out.print(Server.alfa+" presente");
        else
            System.out.print(Server.alfa+" non presente");}
}
```

Osservazioni sugli esempi

- Anche semplicissimi programmi possono essere mal modularizzati.
- Esempio:
  - **Cattiva modularizzazione** (server con basso interfacciamento esplicito, bassa coesione, alto accoppiamento; conseguenze negative sul client)
  - **Buona modularizzazione** (riprogettazione del server e del client)
- Tutti i programmi presentati nei precedenti esempi sono **corretti**, ma hanno **diversa qualità.**

Effetti di una buona modularizzazione

- Il progetto, le competenze, ed il lavoro possono essere distribuiti (i moduli sono indipendenti tra loro)
- Rilevare eventuali errori nel software è più semplice (si individua il modulo errato e ci si concentra su di esso)
- Correggere errori e modificare il software è più semplice (si individua il modulo da modificare e ci si concentra su di esso)
- Attenzione: l'efficienza del programma ne può risentire!

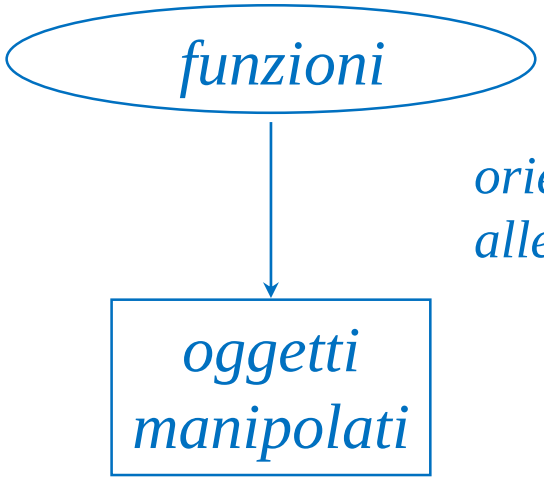
Prima Parte

Introduzione alla Progettazione del SW

- Contesto organizzativo
- Ciclo di vita del software
- Qualità
- Modularizzazione
- **Principi di base dell'orientazione agli oggetti**

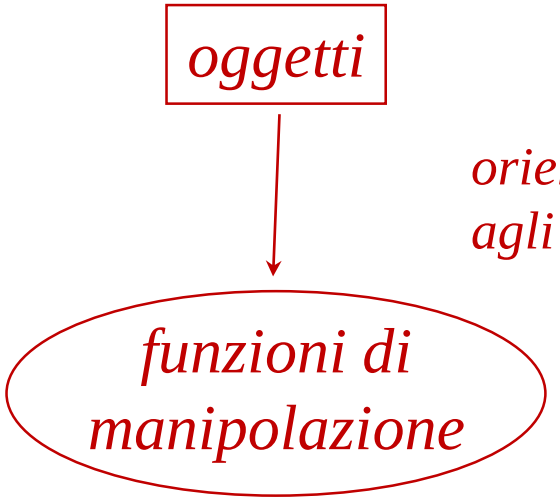
Funzioni o Oggetti ?

Molte delle tecniche tradizionali sono basate sull'idea di costruire un sistema concentrandosi sulle **funzioni**



orientamento alle funzioni

Le tecniche **Object-Oriented (OO)** rovesciano questo rapporto: un sistema viene costruito partendo dalla classificazione degli **oggetti** da manipolare

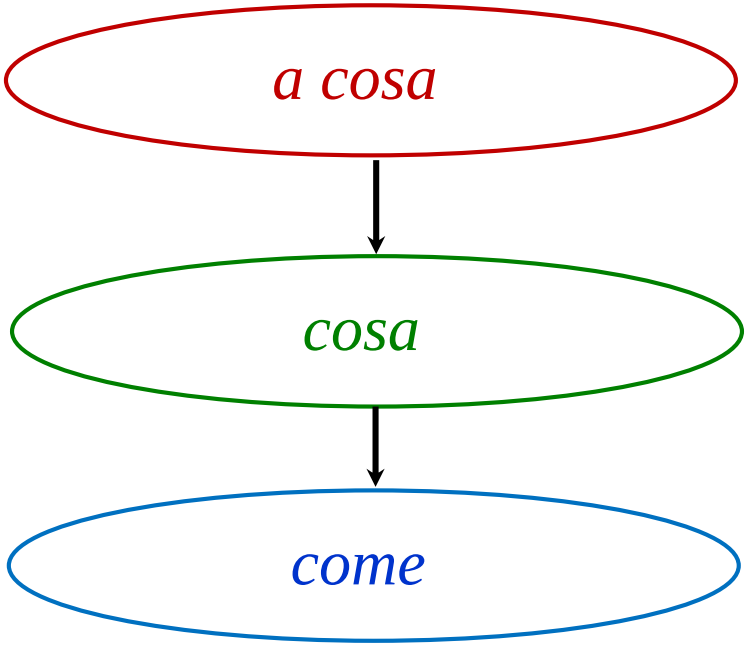


orientamento agli oggetti

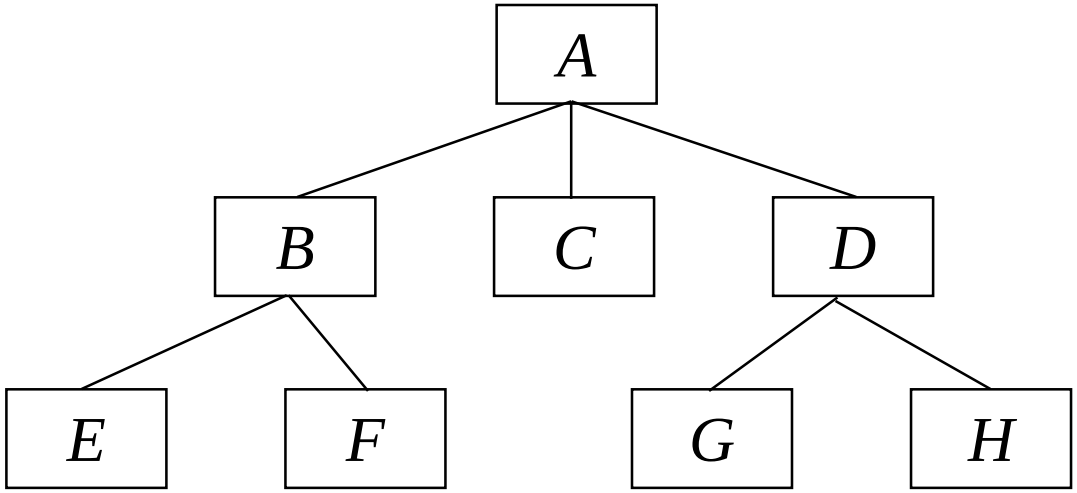
Quindi

La progettazione del SW con l'approccio OO è il metodo che conduce a concentrarsi sugli oggetti che il sistema deve manipolare piuttosto che sulle funzioni che deve realizzare.

*Non chiederti cosa fa il sistema ma **a cosa** serve, di **quali oggetti** è composto, e **come** si opera su di essi!*



Tecnica tradizionale: sviluppo funzionale



- Introduzione di funzioni troppo astratte/innaturali
- Declassamento degli aspetti relativi ai dati
- Attenzione agli aspetti meno stabili (metodi di manipolazione) rispetto a quelli più stabili (informazioni da gestire)
- Creazione di moduli validi solo in un contesto
- Estendibilità e riusabilità (qualità esterne) difficili da raggiungere

Principi di base dell'approccio OO

- Modellare gli aspetti della realtà di interesse nel modo più diretto ed astratto possibile (strutturazione), mediante le nozioni di oggetto, classi e relazioni
- Costruire il programma in termini di moduli, sulla base del principio che ogni classe è un modulo (modularità)
- Legare gli aspetti comportamentali a quelli strutturali
- Proteggere le parti delicate del SW permettendo solo un accesso controllato a dati e funzioni (concetto di interfaccia)

Classi ed Oggetti

Nella realtà, dato un insieme di "cose", spesso astraiamo dalle loro caratteristiche per arrivare a definire un "**concetto**" che le rappresenta tutte

- Mario Rossi, Luisa Bianchi, etc. sono tutte ***Persone***
- Microsoft Word, Open Office, etc. sono tutti ***Programmi "Word processor"***

Classe vs. Oggetto (1)

Il "**concetto**" unificante è la **classe** (anche detta entità o concetto)

- Definisce l'insieme delle caratteristiche comuni a diverse "cose"
- Caratteristiche "strutturali" (attributi) e "comportamentali" (metodi/operazioni)
- Es.: **Persona** e **Programma "Word Processor"**

- La singola "cosa" è l'**oggetto**
  - Istanza di una classe
  - Es.: *Mario Rossi*, *Microsoft Word*

Classe vs. Oggetto (2)

Una **classe** definisce le caratteristiche comuni a tutti gli oggetti

- PERSONA presenta *nome*, *cognome*, *data_nascita* (attributi) e *calcolaEta()* (metodo/operazione)

Gli **oggetti** presentano differenti valori per le caratteristiche strutturali

- L'istanza di PERSONA ***p*** assume i valori *Mario*, *Rossi*, *01/01/1900* (rispettivamente per gli attributi *nome*, *cognome* e *data_nascita*)

Cosa “esiste” realmente ?

- Nella realtà esistono solo gli oggetti.
- Per rappresentare insiemi di oggetti (cioè, i concetti) utilizzo le classi.
  - Non esistono le classi, esistono (i.e., “vivono” nel mondo, “girano” nell’applicazione) solo istanze di classi

Ereditarietà (1)

- E' possibile dichiarare che una classe B ***eredita*** da un'altra classe A
  - Gli oggetti istanza di B presentano tutte le caratteristiche tipiche degli oggetti istanza di A
  - Possono inoltre avere ulteriori caratteristiche (che li rendono più specifici)

- In Java questo si esprime con *extends*

(es., class Student exteds Person)

Ereditarietà (2)

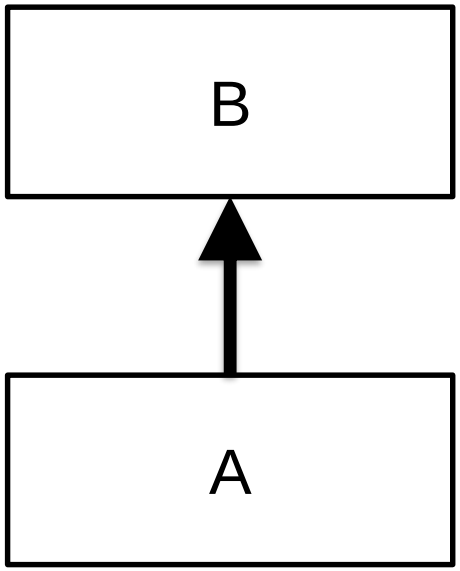
```
class PERSONA {
    public PERSONA(string unNome, string unCognome) { … }
    public int calcolaEtà() { … }
    private string nome;
    private string cognome;
}
class STUDENTE extends PERSONA {
    public void stampaListaEsami() { … }
    private string numeroMatricola;
    private string annoDiCorso;
}
```

Un oggetto di tipo STUDENTE presenta quindi tutti gli attributi e metodi di PERSONA più quelli specifici di STUDENTE, quindi:
- nome
- cognome
- numeroMatricola
- annoDiCorso
- calcolaEtà()
- stampaListaEsami()

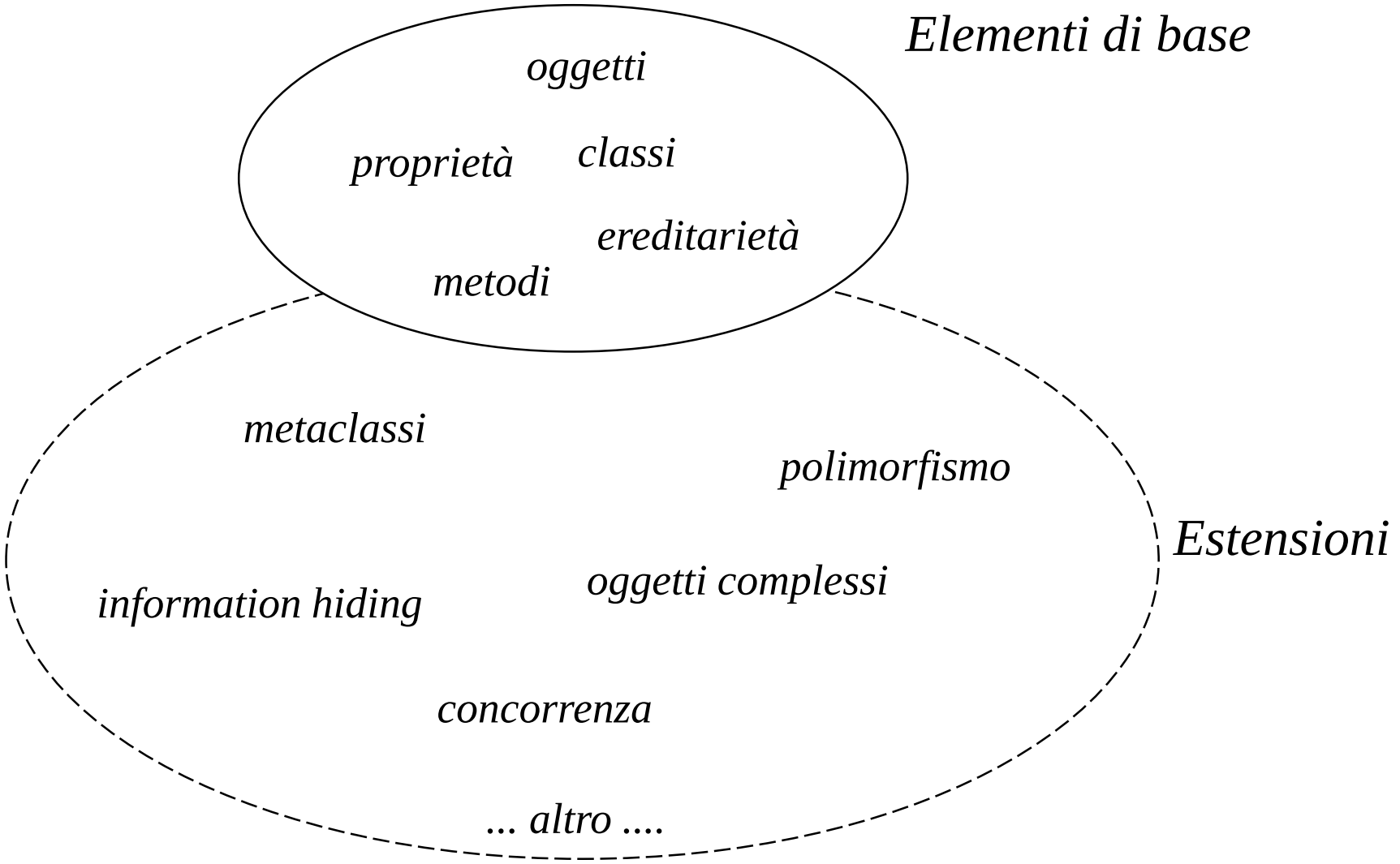
Ereditarietà

A (e.g., classe STUDENTE) *è in relazione di ereditarietà con* B (e.g., classe PERSONA)

- A eredita da B
- A è una sottoclasse di B
- A è una classe derivata da B
- A è una specializzazione di B
- B è una superclasse di A
- B è una classe base di A
- B è una generalizzazione di A



Fondamenti dell'approccio OO



Qualità influenzate dall'approccio OO

Strutturazione: favorita dal concetto di incapsulamento e dalla ereditarietà

Modularità: favorita dal concetto di classe

Comprensibilità: favorita dalla struttura vicina al modo in cui vediamo la realtà

Manutenibilità: favorita dal concetto di incapsulamento

Riusabilità: favorita dalla struttura vicina al modo in cui vediamo la realtà