13. Principi di progettazione e qualità di un progetto software

IS 2024-2025

Laura Semini, Jacopo Soldani
Corso di Laurea in Informatica
Dipartimento di Informatica, Università of Pisa

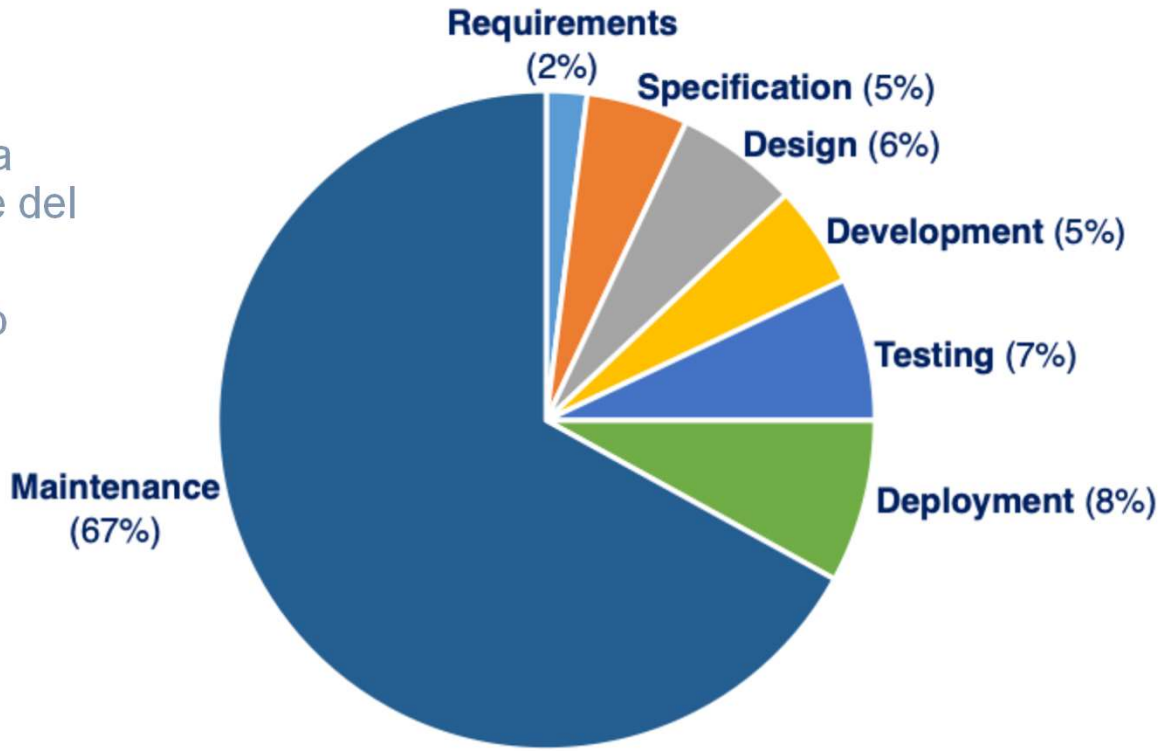
PROGETTARE SOFTWARE

Le **tecniche di progettazione** e le **good practice** mirano a produrre un sistema che realizzi

- **requisiti funzionali**
- **requisiti di qualità**

ma anche che sia

- facilmente **manutenibile** (ovvero sia facile/economico fare manutenzione del sistema)
- **riusabile** (parti del sistema possono essere riutilizzate in altri sistemi)



LA MANUTENZIONE DEL SOFTWARE (RECAP)

La **manutenzione** include tutti i cambiamenti ad un prodotto software, anche dopo che è stato rilasciato in produzione

Diversi **tipi** di manutenzione:

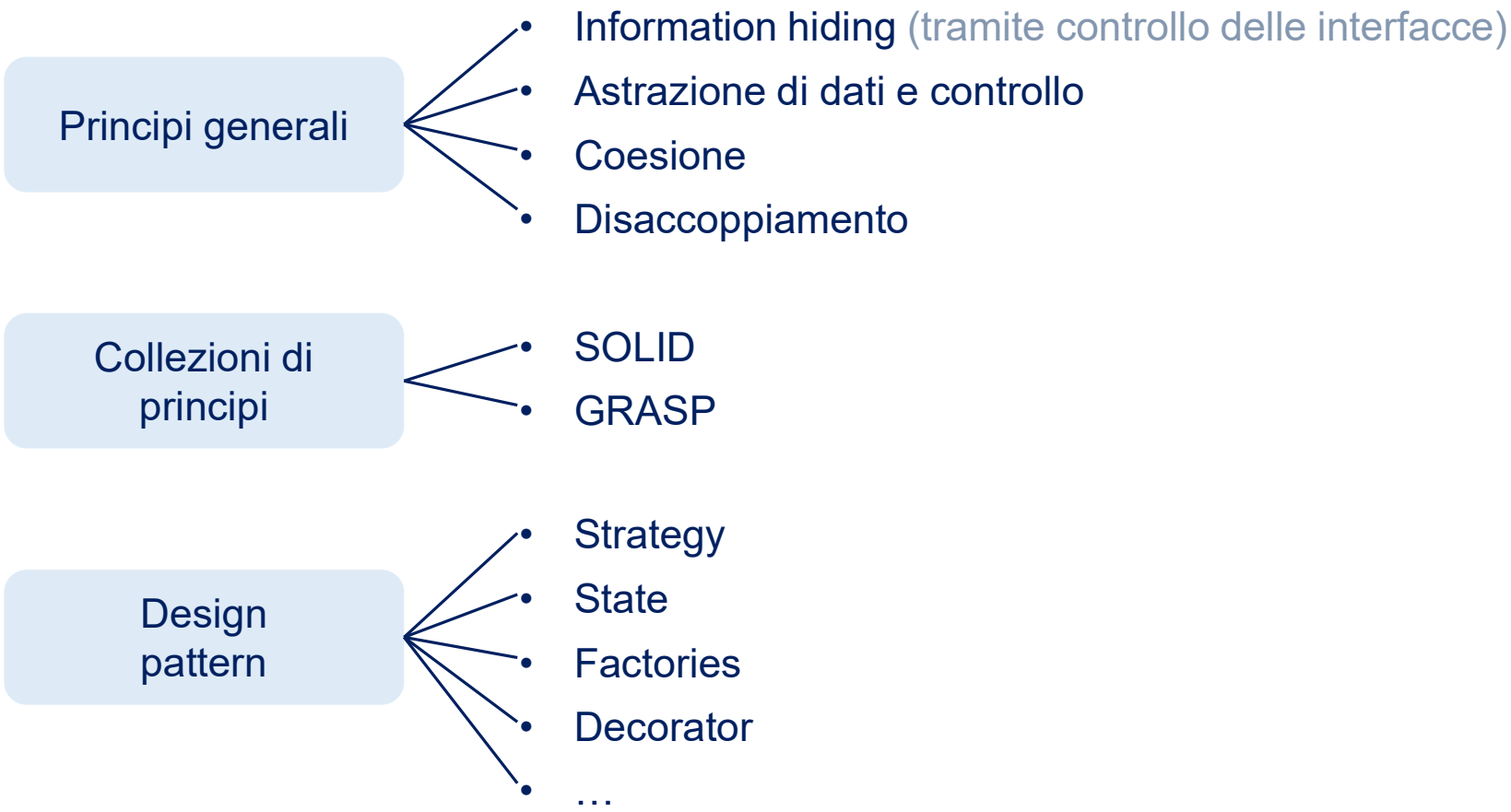
- **correttiva** (~20%): rimuove gli errori/bug lasciando invariata la specifica
- **migliorativa**: consiste in cambiamenti di specifica e implementazione, ovvero
  - manutenzione **perfettiva** (~60%): modifiche applicate per migliorare le qualità del software, fornire nuove funzionalità, o migliorare funzionalità esistenti
  - manutenzione **adattativa** (~20%): modifiche a seguito di cambiamenti nel contesto del software (ad esempio, cambiamenti legislativi, nel hardware, o nel sistema operativo)

Esempio: IVA dal 20% al 22%

⇒

```
float aliquota=22;
…
prezzotot = prezzo + (prezzo*aliquota)/100
```

PRINCIPI E PATTERN DI PROGETTAZIONE

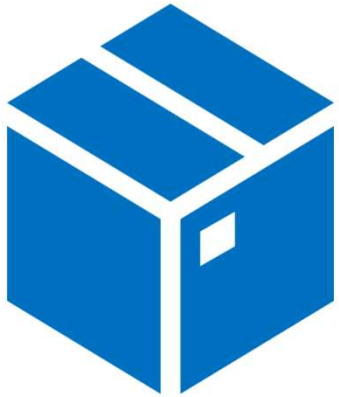


INFORMATION HIDING

Separazione tra **interfaccia** (visibile)
e **implementazione** (privata/nascosta)

Componenti o moduli come **black box** (scatole nere)

- Forniscono e richiedono funzionalità
- Nota solo la loro **interfaccia** // nascosti algoritmi e strutture dati utilizzati internamente



INFORMATION HIDING (CONT.)

Si separano interfaccia e corpo di un'**unità di progettazione**

- L'**interfaccia (visibile)** esprime ciò che l'unità offre/richiede
- Il **corpo (nascosto)** implementa gli elementi dell'interfaccia e realizza la semantica dell'unità

Vantaggi:

- **Comprensibilità** → non servono i dettagli implementativi di un'unità per usarla
- **Manutenibilità** → si può cambiare il corpo di un'unità senza modificare le altre
- **Lavoro in team** → corpi di unità diverse possono essere sviluppati da team indipendenti
- **Sicurezza** → dati di un'unità modificabili solo dal corpo della stessa (e non dall'esterno)

INFORMATION HIDING != INCAPSULAMENTO

Incapsulamento

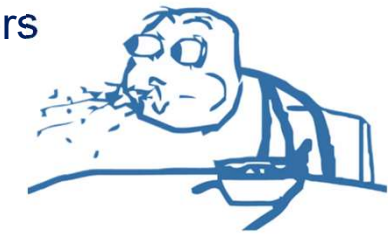
- Capacità degli **oggetti** di mantenere al loro interno **attributi e metodi** // ovvero di incapsulare il loro stato e comportamento
- Si tratta di una proprietà dei **linguaggi di programmazione** object-oriented

L'incapsulamento **permette** – ma non garantisce – **information hiding**

- **Interfaccia** data da attributi e metodi **pubblici**
- **Corpo** costituito da attributi e metodi **privati** // e quindi «nascosti»

⇒ information hiding dipende da cosa viene messo pubblico/privato, e come

Già visto e rivisto inconsapevolmente: variabili private con setters e getters

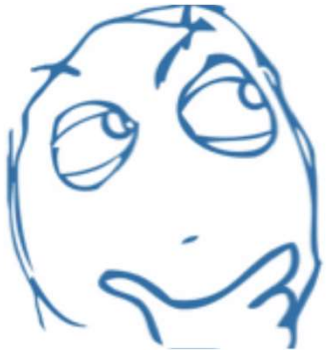


INFORMATION HIDING: ACCESSORS & MUTATORS

(AKA. GETTERS) (AKA. SETTERS)

Standard de-facto per l'accesso agli attributi (privati) di una classe

- **Nasconde** la rappresentazione dei dati
- Accesso solo mediante interfaccia **getter-setter**
  - **getter** → **restituisce** un attributo come **valore** e **senza side effects**
    (non cambia lo stato dell'oggetto)
  - **setter** → **modifica** lo **stato** dell'oggetto, **cambiando il valore** dell'attributo
    (è possibile fare controlli prima dell'effettiva modifica)



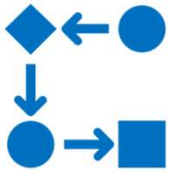
FAQ: *Gli IDE generano automaticamente i getter-setter per gli attributi di una classe. Dobbiamo lasciarli fare?*

A: Solo se realmente necessari! Una volta aggiunti

- devono essere manutenuti e
- possono essere usati per accedere a dati (che magari invece volevamo privati)

ASTRAZIONE

Il principio di **astrazione** si applica a controllo e dati



Procedura come modulo di astrazione del flusso di **controllo**

- **Nasconde l'algoritmo** utilizzato (es. algoritmi di ordinamento)
- Organizzate in classi di moduli, costituiscono **libreria**

Struttura dati astratta per rappresentare dati regolamentando accesso/modifica

- **Interfaccia stabile** anche se cambia l'implementazione della struttura dati
- **Non puramente funzionale** (come nel caso delle librerie)

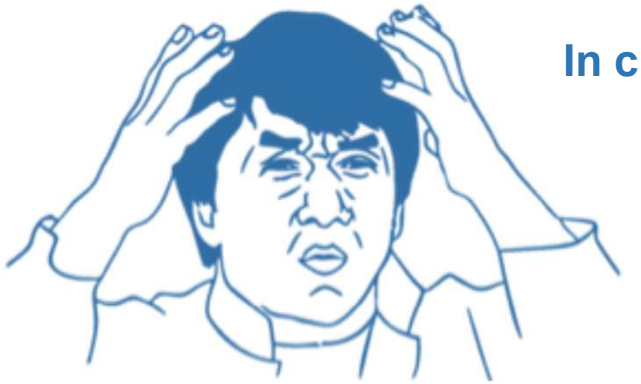
  → Le operazioni offerte possono modificare il dato rappresentato (lo stato)

COESIONE

Quanto strettamente correlate siano le funzionalità offerte da un modulo/componente

Grado con cui un'unità di progettazione realizza «uno e un solo concetto»

- funzionalità «vicine» devono stare nella stessa unità (vicinanza intesa in termini di tipo, algoritmi, dati in ingresso e in uscita)
- un sistema è coeso se tutti gli elementi di ogni unità di progettazione sono strettamente collegati tra loro



In che senso?

UNA CLASSE PER NULLA COESA

```
public class Activities {

	public void PrintDocument(Document doc) {
		. . .
	}

	public void SendEmail(string rcpnt, string sbj, string txt) {
		. . .
	}

	public void ComputeDistance(Point p1, Point p2) {
		. . .
	}
}
```

TIPI E GRADI DI COESIONE

- **Coesione funzionale**: gli elementi collaborano per realizzare una funzionalità
  → Situazione ideale

- **Coesione comunicativa**: gli elementi operano sugli stessi dati in input e/o contribuiscono agli stessi dati in output (es. aggiornare il record nel database e inviarlo alla stampante)
  → Non è un buon modo di raggruppare e non favorisce il riuso

- **Coesione procedurale**: gli elementi realizzano i passi di una procedura (es. ritagliare immagini e applicare dei filtri)
  → Azioni sono debolmente coese e difficilmente riutilizzabili

TIPI E GRADI DI COESIONE (CONT.)

- **Coesione funzionale**: gli elementi collaborano per realizzare una funzionalità
  → Situazione ideale

- **Coesione temporale**: gli elementi sono azioni che devono essere fatto in uno stesso arco di tempo (es. azioni da fare all'apertura dell'anno accademico)
  → Azioni sono debolmente coese e difficilmente riutilizzabili

- **Coesione logica**: gli elementi sono correlati logicamente ma non funzionalmente (es. raccolta dati da sensori, analisi dei dati raccolti e generazione report)
  → Operazioni correlate ma funzioni significativamente diverse
  → Operazioni debolmente connesse e difficilmente riutilizzabili

- **Coesione accidentali**: gli elementi non sono correlati ma piazzati assieme (es. «una classe per nulla coesa»)
  → Peggior forma di coesione

DISACCOPPIAMENTO (AKA. DECOUPLING)

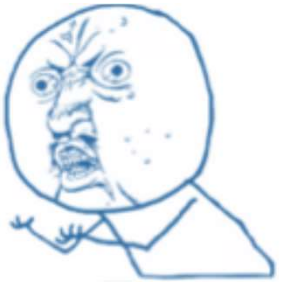
Quanto sono **«slegate»** le unità di progettazione

Con accoppiamento si intende

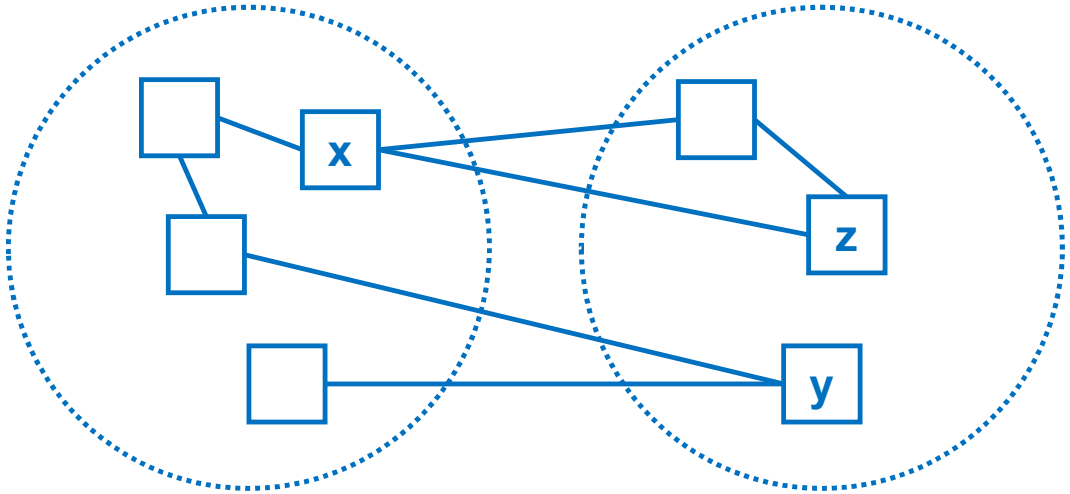
- il grado con cui un'unità di progettazione è **«legata»** (coupled) ad un'altra
- per esempio, in termini di dipendenze funzionali o scambio di messaggi

⇒ meglio creare sistemi **disaccoppiati**!

Perché?

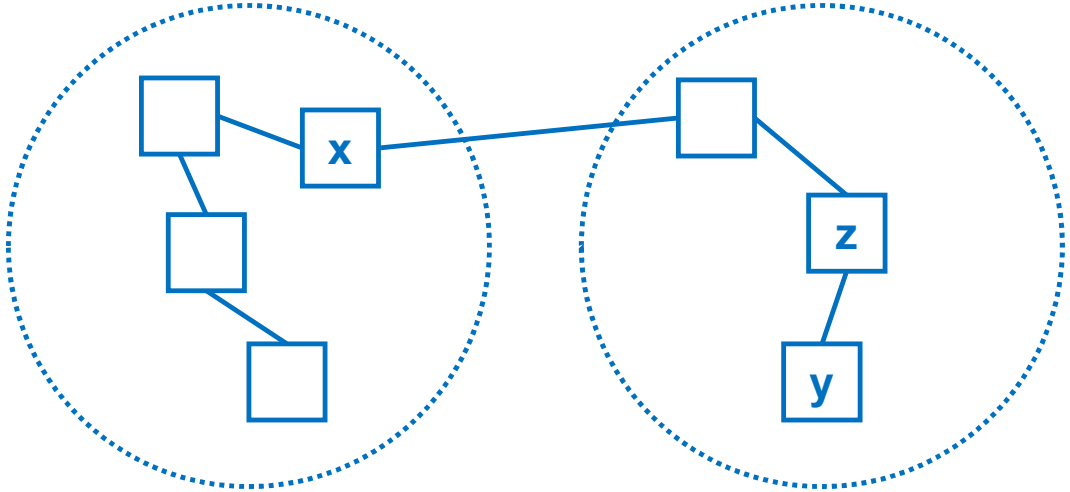


DISACCOPPIAMENTO: PERCHÉ?



Sistema molto accoppiato (**high coupling**)

- modifiche a **x, y, z** impattano su più componenti, anche lontani tra loro
- poco manutenibile



Sistema poco accoppiato (**low coupling**)

- impatto modifiche limitato per ogni componente
- migliore manutenibilità

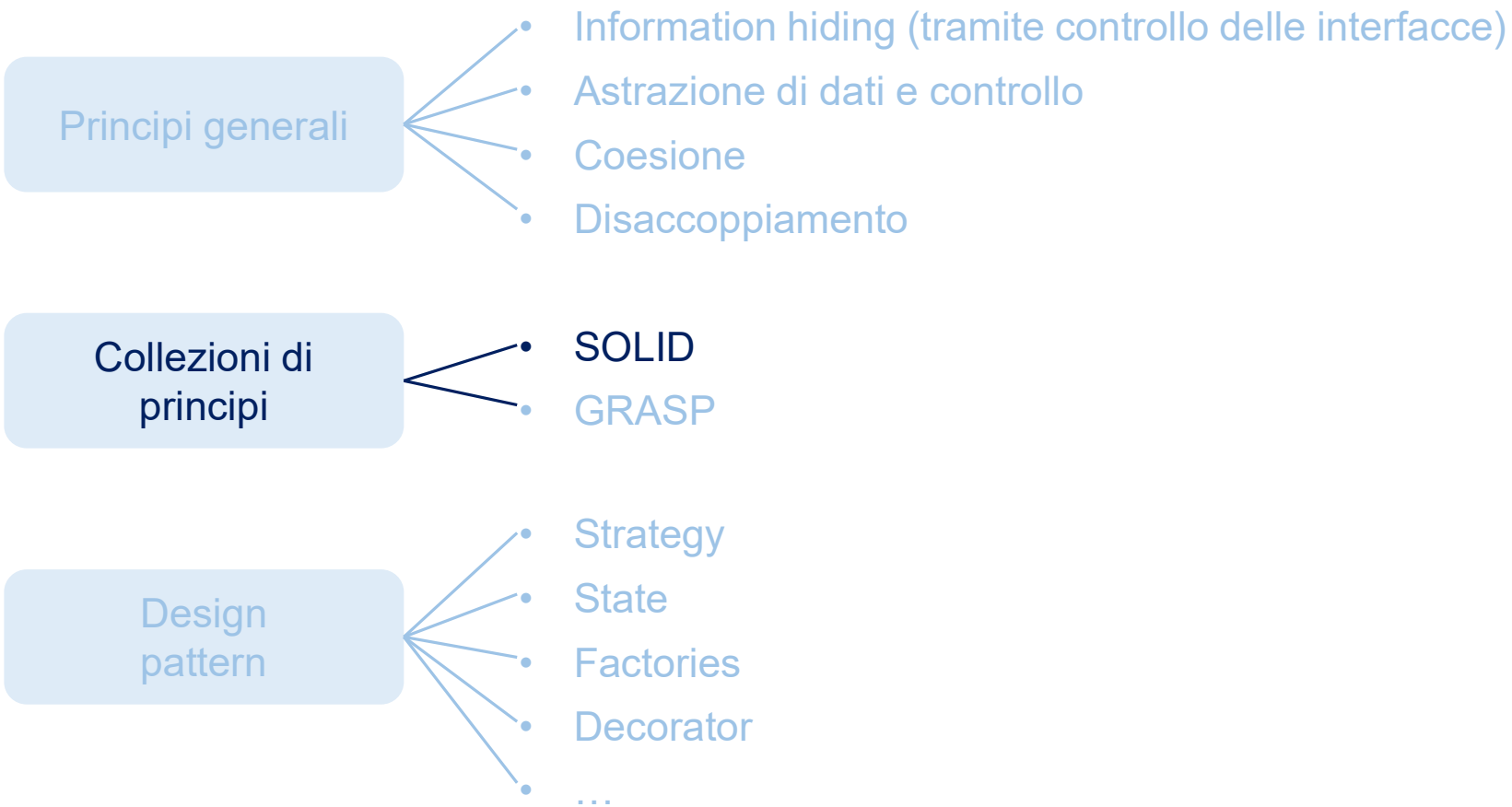
HIGH COHESION, LOW COUPLING

Un **mantra**, specie nei sistemi moderni ☺

- Facilita **riuso** e **manutenibilità**
- Riduce le **interazioni** tra (sotto)sistemi
- Migliora la **comprensibilità**

Un altro grado di coesione contribuisce a ridurre il grado di accoppiamento

PRINCIPI E PATTERN DI PROGETTAZIONE



SOLID

Cinque principi di base per progettazione e sviluppo object-oriented (autore: Robert C. Martin)

- **S**ingle Responsibility: una classe (o metodo) dovrebbe avere solo un motivo per cambiare
- **O**pen Closed: estendere una classe non dovrebbe comportare modifiche alla stessa
- **L**iskov Substitution: istanze di classi derivate possono sostituire istanze della classe base
- **I**nterface Segregation: interfacce a grana fine e specifiche per ogni cliente
- **D**ependency Inversion: programmare guardando le interface e non l'implementazione

Si applicano principalmente in fase di progettazione di dettaglio

SOLID: SINGLE RESPONSIBILITY PRINCIPLE

«just because you can, doesn’t mean you should»

SOLID: SINGLE RESPONSIBILITY PRINCIPLE

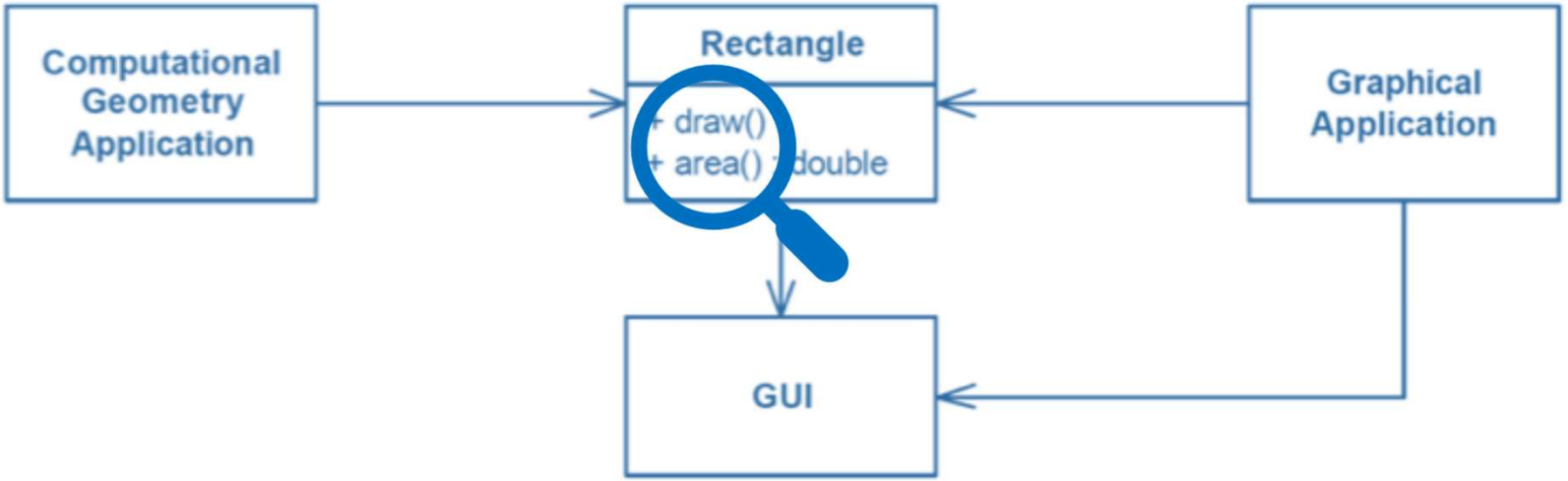
Una classe (un modulo, un metodo) dovrebbe avere solo un motivo per cambiare

Responsabilità intesa come **motivo per cambiare**

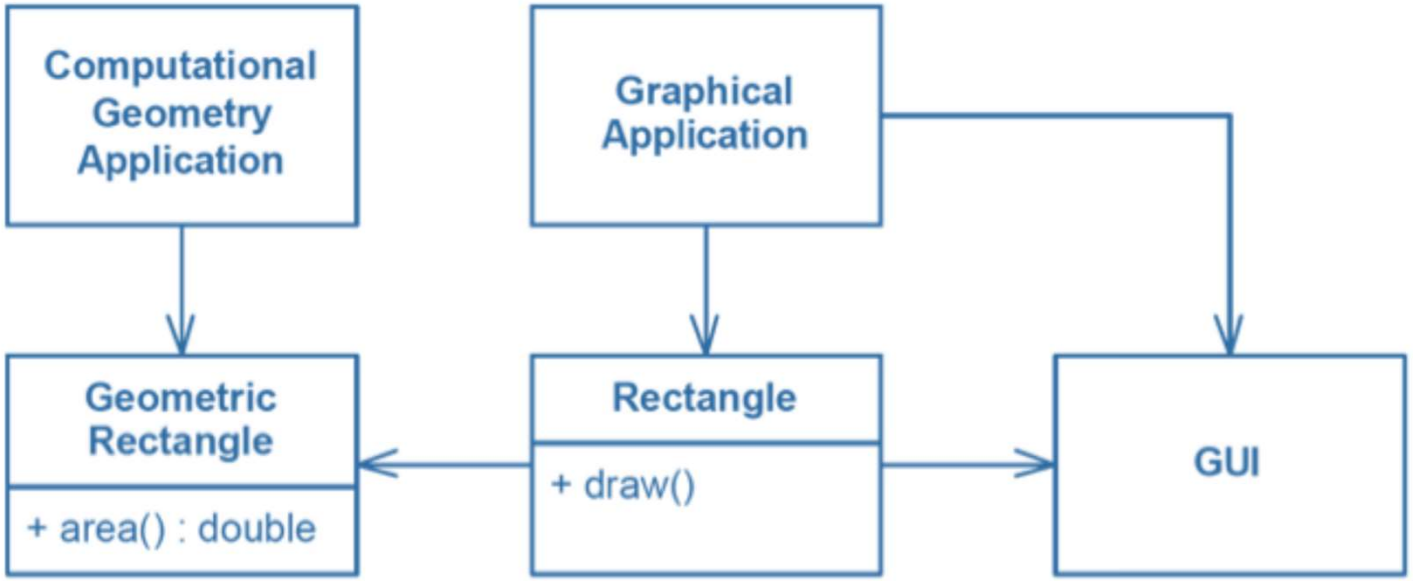
- Un cambiamento deve impattare **solo** la classe che realizza la funzionalità
- Due diversi motivi per modificare una classe ⇒ dobbiamo dividerla in due classi
- Se dovessimo fare un cambiamento in una classe che ha più responsabilità, le modifiche potrebbero influenzare altre funzionalità della classe e (in cascata) i moduli che le usano

⇒ una classe deve essere **funzionalmente coesa** (e realizzare una sola funzionalità)

UN SOLO MOTIVO PER CAMBIARE



UN SOLO MOTIVO PER CAMBIARE (CONT.)



COME DECIDERE SE DIVIDERE RESPONSABILITÀ?

Un'interfaccia con due responsabilità

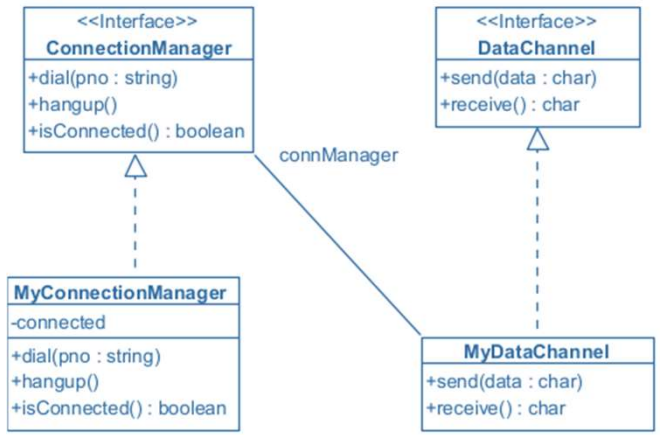
```
public interface Modem {
        public void dial(string pno) {…}        // connection
        public void hangup() {…}                // connection
        public void send(char c) {…}            // data
        public void recv() {…}                  // data
}
```

Soluzione **rigida!**

- Se cambia la segnatura dei metodi per la connessione, le classi che chiamano `send` e `recv` devono essere ricompilate e ricontrollate (più spesso di quanto realmente vorremmo)

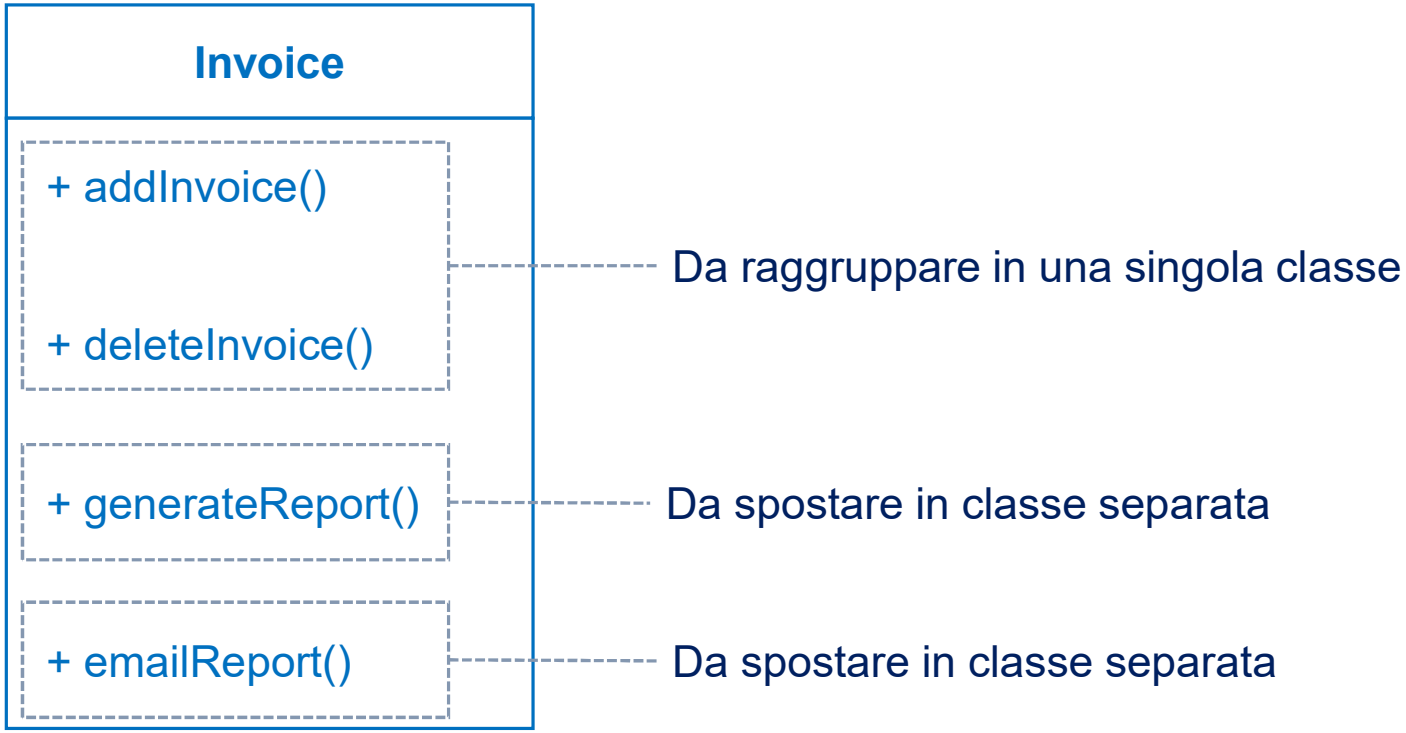
COME DECIDERE (CONT.)

```java
// Responsabile della connessione
public class MyConnectionManager {
    private boolean connected;
    public void dial(pno) {
        //connect to pno
        connected = true;
    }
    public void hangup() {
        // disconnect
       connected = false;
    }
    public boolean isConnected() {
       return connected;
    }
}
```




```java
// Responsabile della gestione dei dati
public class DataChannel {
  private ConnectionManager connManager;
  public DataChannel(ConnectionManager connManager) {
    this.connManager = connManager;
  }
  public void send(data) {
    if (connManager.isConnected()) {
      // send data
    } else {
      System.out.println("Cannot send: disconnected");
    }
  }
  public char receive() {
    if (connManager.isConnected()) {
       // receive data
       return "Some received data";
    } else {
      System.out.println("Cannot receive: disconnected");
      return null;
    }
  }
}
```

UN ALTRO ESEMPIO



ECCEZIONE ALLA REGOLA

Non si può cambiare una delle due responsabilità senza cambiare contestualmente anche l'altra

- Separarle introdurrebbe complessità non necessaria

- Un motivo di cambiamento è tale se è una reale possibilità di cambiamento del sistema

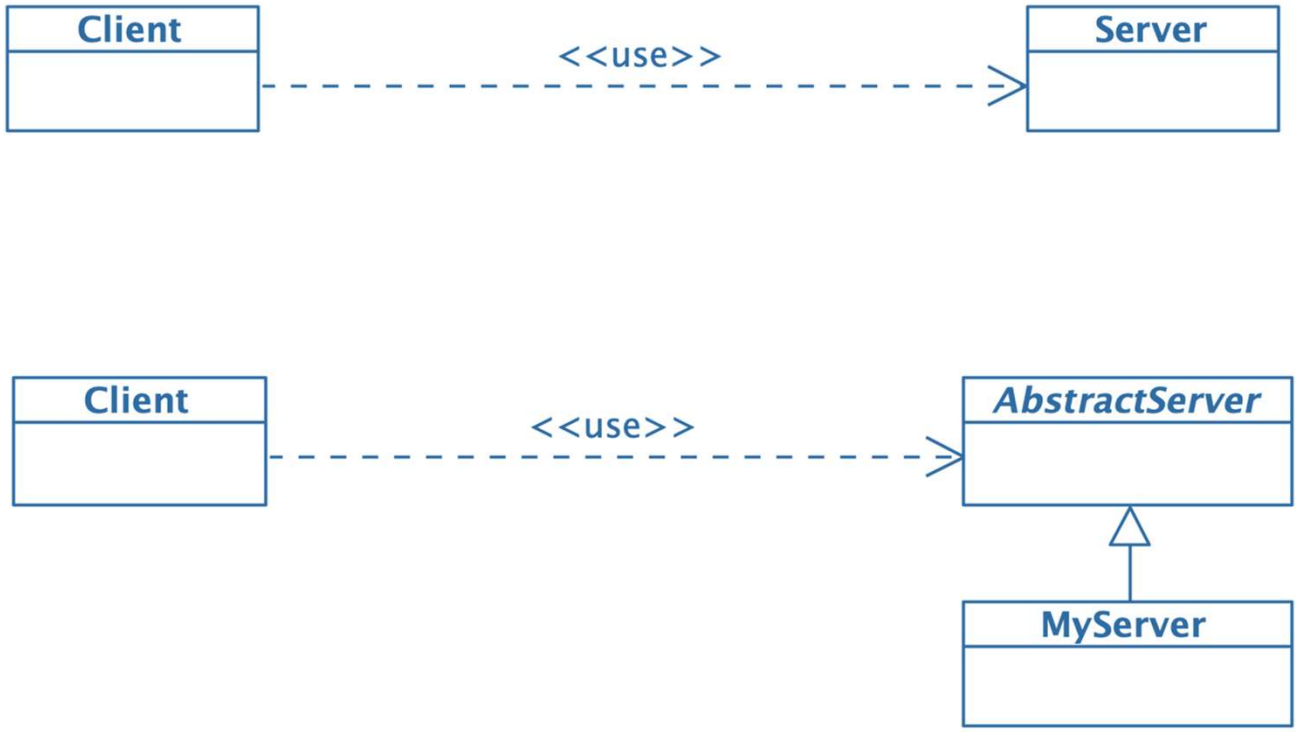
SOLID: OPEN CLOSED PRINCIPLE

Le entità software devono essere **aperte per estensione** ma **chiuse per modifiche**

Disegnare classi/moduli che **non cambiano**

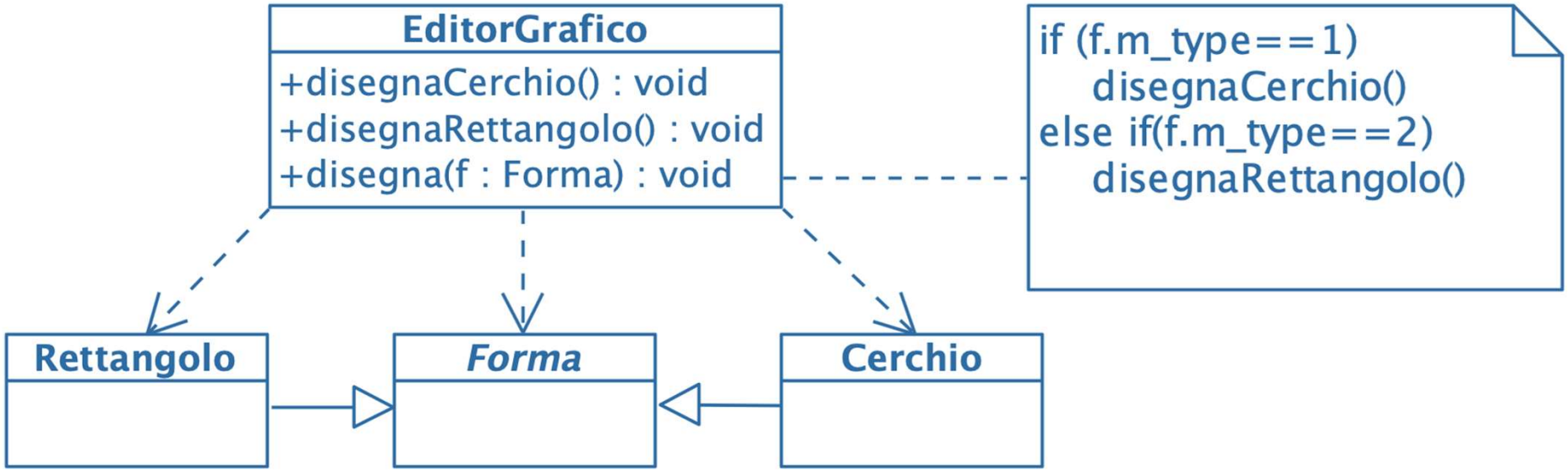
- Quando i requisiti cambiano, estenderne il comportamento aggiungendo nuovo codice (senza cambiare quello esistente e funzionante)
- Disegnare classi in modo che sia possibile estenderle ma senza cambiarle
- Possibile, per esempio, sfruttando
  - classi astratte e classi concrete
  - delega
  - plugin (aggiunta di codice senza ricompilare l'esistente)

CLASSI ASTRATTE E CLASSI CONCRETE



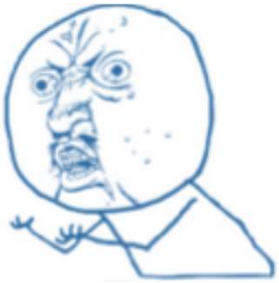
OPEN CLOSED PRINCIPLE: ESEMPIO

Ad ogni una nuova forma,
disegnaForma va cambiato



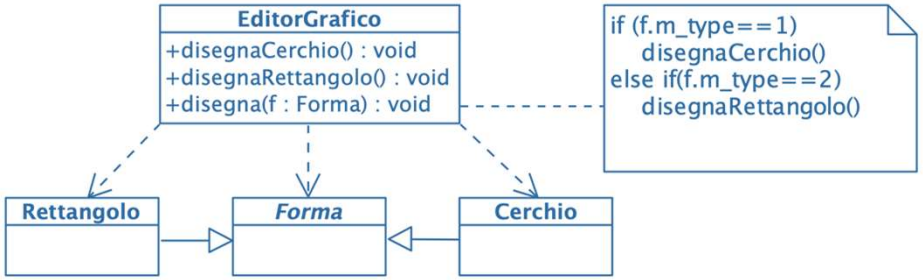
Lo rispetta?

NOOOO!



OPEN CLOSED PRINCIPLE: ESEMPIO (CONT.)

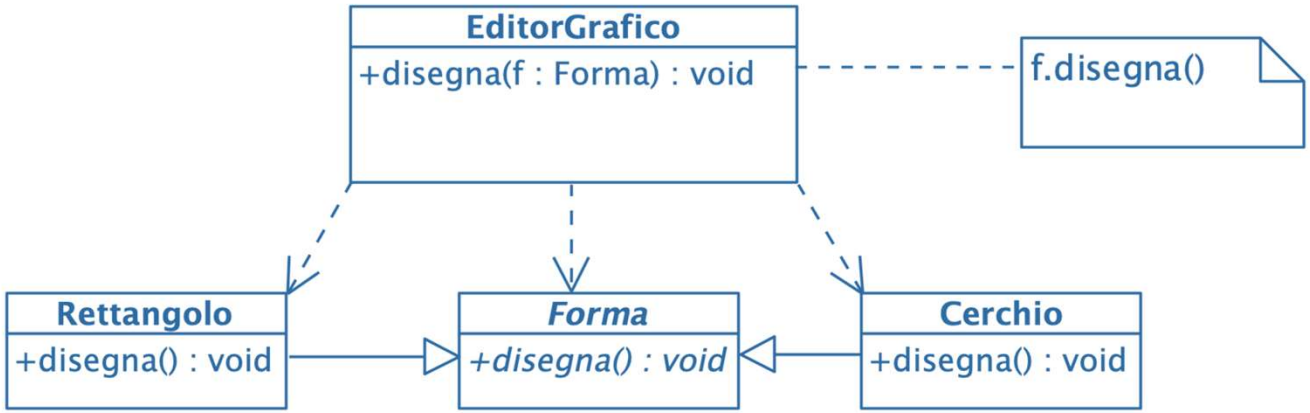
```
public abstract class Forma {int m_type; }
public class Rettangolo extends Forma {
  public Rettangolo() { super.m_type=1; }
}
public class Cerchio extends Forma {
  public Cerchio() {super.m_type=2; }
}
public class EditorGrafico {
  public void disegnaForma(Forma s) {
    if (s.m_type==1) disegnaRettangolo(s);
    else if (s.m_type==2) disegnaCerchio(s);
  }
  public void disegnaCerchio(Cerchio c) {....}
  public void disegnaRettangolo(Rettangolo r) {....}
}
```

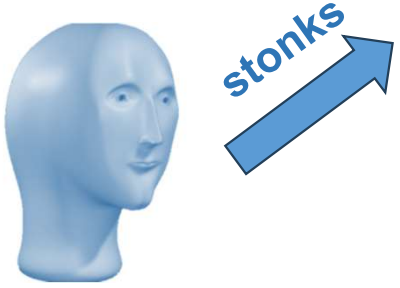


OPEN CLOSED PRINCIPLE: ESEMPIO (CONT.)

Secondo l'**Open Closed Principle**

- **Aperti** alle **estensioni** → usare **astrazioni** (interfacce, classi astratte)
- **Chiusi** al **cambiamento** → usare **delega** e **astrazioni**





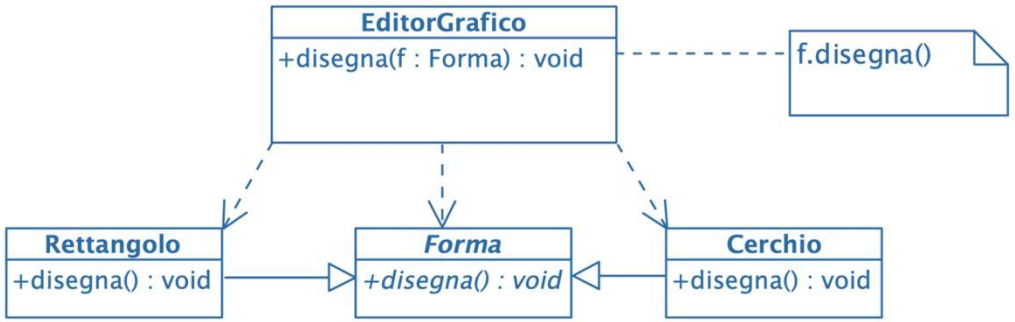
OPEN CLOSED PRINCIPLE: ESEMPIO (CONT.)

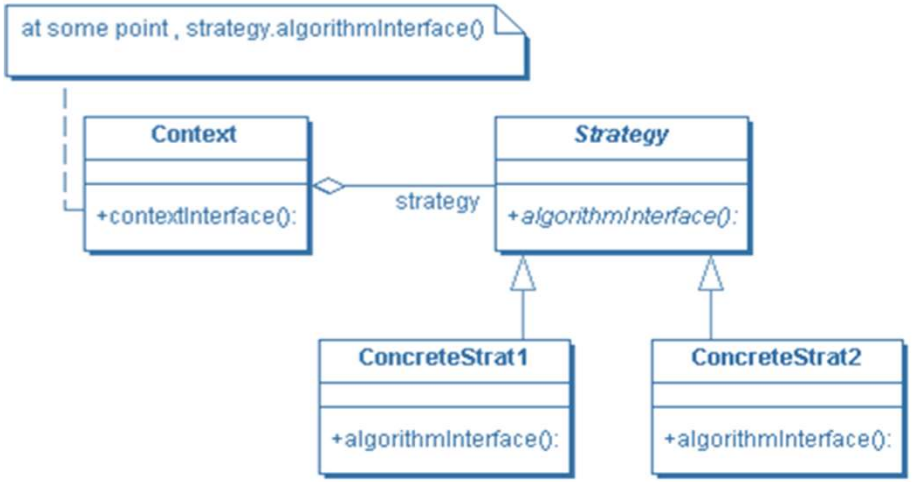
```
public abstract class Forma { public abstract void disegna(); }
public class Rettangolo extends Forma {
  public Rettangolo() {…}
  public void disegna() {…}
}
public class Cerchio extends Forma {
  public Cerchio() {…}
  public void disegna() {…}
}
public class EditorGrafico {
  public void disegnaForma(Forma s) {
    f.disegna()
  }
}
```

Si sposta il codice che dipende dalle classi concrete nelle classi concrete stesse

Si opera per «delega» applicando (di fatto) il design pattern Strategy

COLLEGATO ALLO STRATEGY PATTERN?





SOLID: LISKOV SUBSTITUTION PRINCIPLE

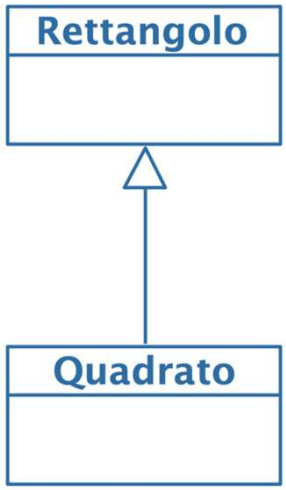
Le **classi derivate** devono poter
sostituire le **classi base**

Deriva dal **principio di sostituzione** definito da Barbara Liskov:

- **S** è sottotipo di **T**
- $\mathbf{o_s}$ e $\mathbf{o_t}$ sono oggetti di tipo **S** e **T**, rispettivamente
- **P** è un qualsiasi programma definito in termini di **T**

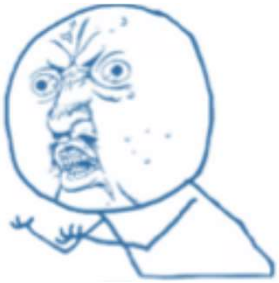
⇒ il comportamento di **P** è immutato quando $\mathbf{o_s}$ è usato al posto di $\mathbf{o_t}$

UN ESEMPIO




```
public class Rettangolo {…}
public class Quadrato extends Rettangolo {
  public Quadrato(int l) { super(l,l); }
  public setBase(int l) { super.setBase(l); super.setAltezza(l); }
  public setAltezza(int l) { super.setBase(l); super.setAltezza(l) }
}

RettangoloFactory rf = new RettangoloFactory();
Rettangolo r = rf.getRettangolo(); // questo può restituire un quadrato
r.setBase(10);
r.setAltezza(5);
r.getArea(); // ci si aspetta 50, ma si ottiene 25 se quadrato
```

Ma il quadrato è (logicamente) un rettangolo!!!!

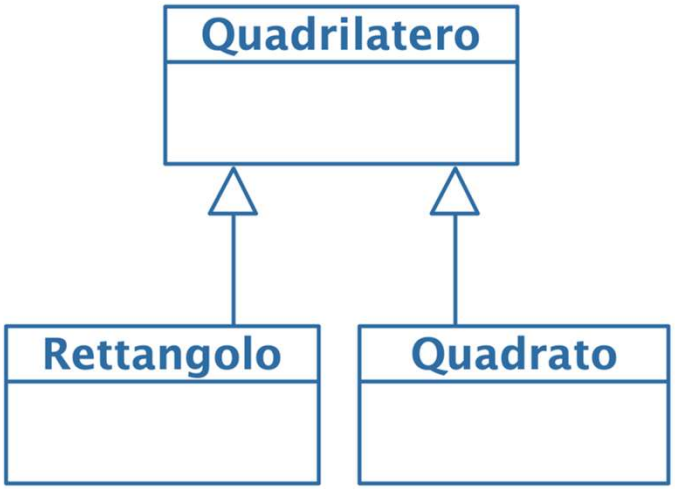


UN ESEMPIO (CONT.)

Problema: `Quadrato` sovrascrive i metodi di larghezza e altezza della classe padre `Rettangolo`

Soluzioni?

Ad esempio



SOLID: INTERFACE SEGREGATION PRINCIPLE

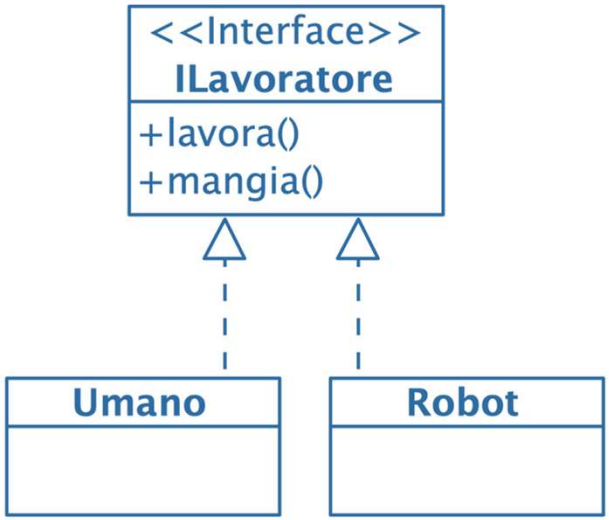
Interfacce a **grana fine** e
specifiche per ogni cliente

Occorre prestare attenzione al modo in cui si scrivono le interfacce

- I client **non devono dipendere** da interfacce che **non usano**
- Mettere solo i **metodi necessari**
  (ogni metodo che si aggiunge, anche se inutile, deve essere implementato)

UN ESEMPIO

Capita di creare classi-sottoclassi o interfacce-implementazioni con metodi che **non hanno senso**

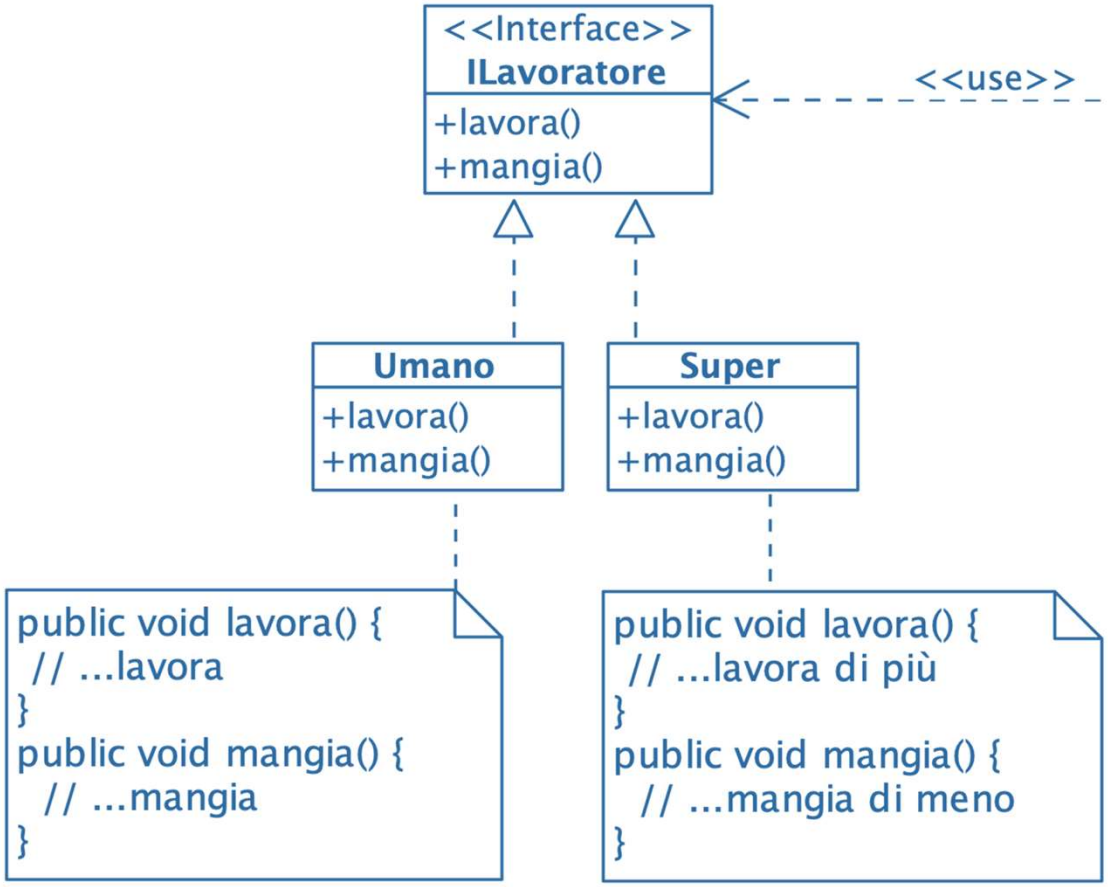


`mangia` deve essere **sempre implementato**

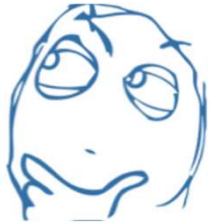
- anche dai `Robot` ?

Meglio evitare interfacce con metodi non specifici (**polluted** o **fat interfaces**).

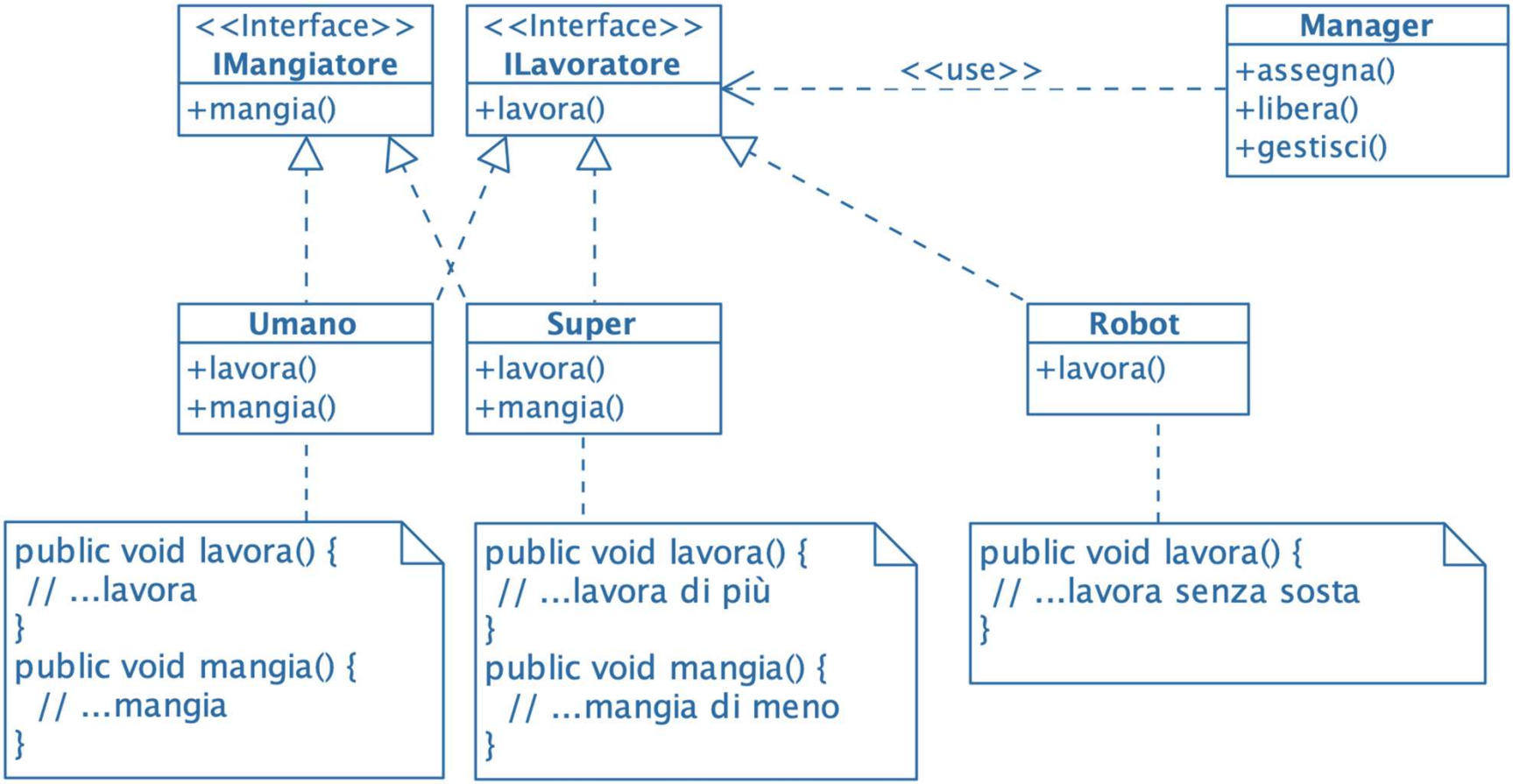
RICOMINCIAMO, SENZA ROBOT



Come aggiungere Robot?



ESEMPIO RIVISITATO



SOLID: DEPENDENCY INVERSION PRINCIPLE

«would you solder a lamp directly to the electrical wiring in the wall?»

SOLID: DEPENDENCY INVERSION PRINCIPLE

Programmare **per l'interfaccia**,
non per l'implementazione

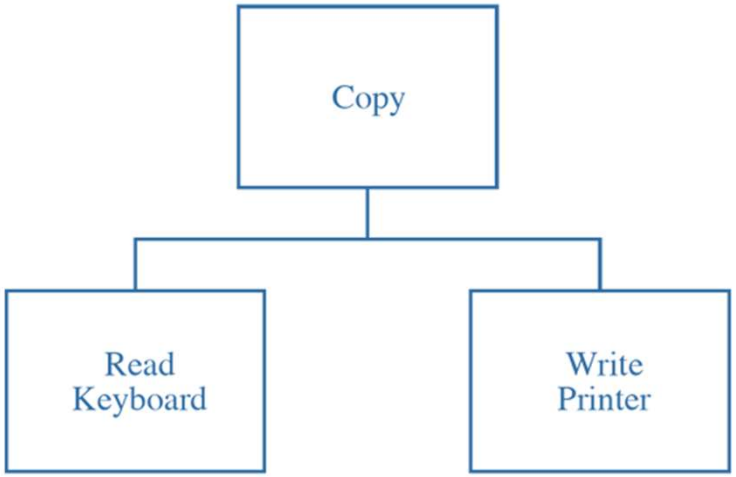
- I moduli di alto livello non devono dipendere da quelli di basso livello
  (entrambi devono dipendere da astrazioni)

- Le astrazioni non devono dipendere dai dettagli
  (sono i dettagli che dipendono dalle astrazioni)

⇒ In sostanza, non ci si deve mai basare su implementazioni concrete ma solo su astrazioni

UN ESEMPIO DI DIPENDENZA DA IMPLEMENTAZIONE

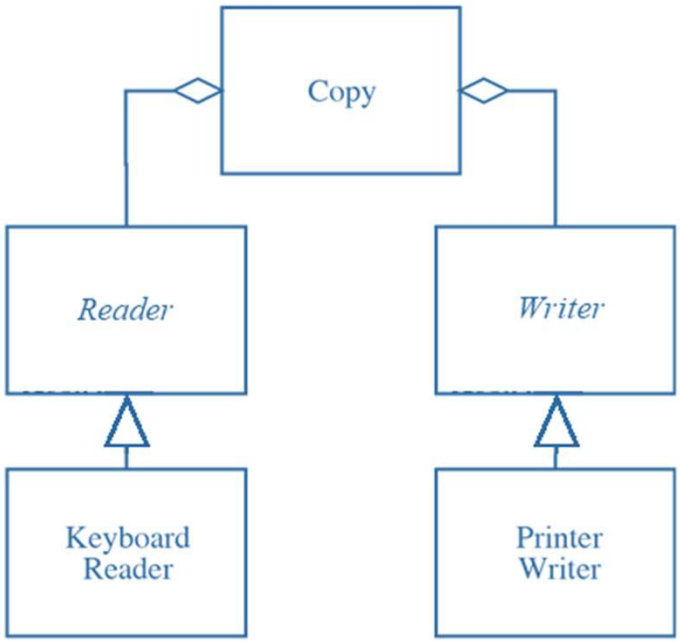
Tre moduli per leggere un carattere da tastiera e stamparlo sulla tastiera




```
void Copy() {
  int c;
  while (( c = ReadKeyboard()) != EOF )
    WritePrinter(c);
}
```

`Keyboard` e `WritePrinter` possono essere riutilizzati, mentre `Copy` no (es. serve un `if` per copiare dalla tastiera e scrivere sul disco)

APPLICHIAMO L'INVERSIONE DELLE DIPENDENZE




```
class Reader
{
  public: virtual char Read() = 0;
};
class Writer
{
  public: virtual void Write(char) = 0
};
void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

UN ALTRO ESEMPIO

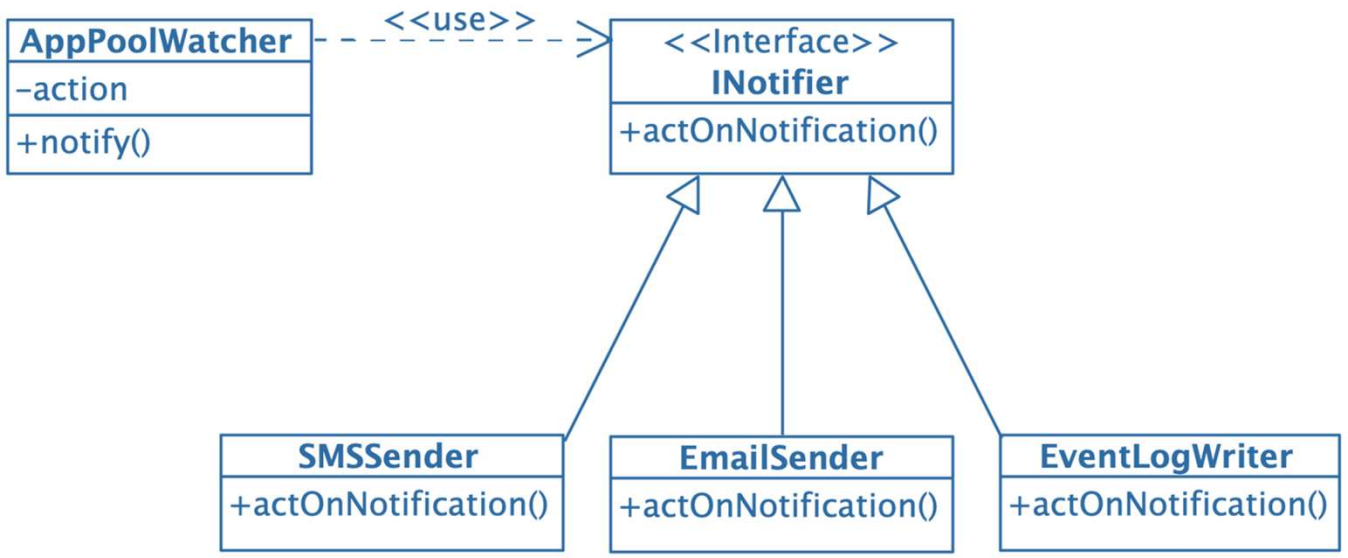
```
class AppPoolWatcher { // Handle to EventLog writer to write to the logs
  EventLogWriter writer = null;
  public void Notify(string message) { // Function called when the app pool has problem
    if (writer == null) {
      writer = new EventLogWriter();
    }
    writer.write(message);
  }
}

class EventLogWriter {
  public void write(string message) {
    // ...write to event log
  }
}
```

E se poi dovessimo mandare altri tipi di notifiche?
(es. mail e sms all'amministratore del sistema)

UN ALTRO ESEMPIO (CONT.)

Applichiamo il principio di inversione delle dipendenze per disaccoppiare il sistema

- il modulo di alto livello (`AppPoolWatcher`) deve dipendere da un'astrazione
- astrazione concretizzata da classi che definiscono le operazioni concrete



DEPENDENCY INJECTION

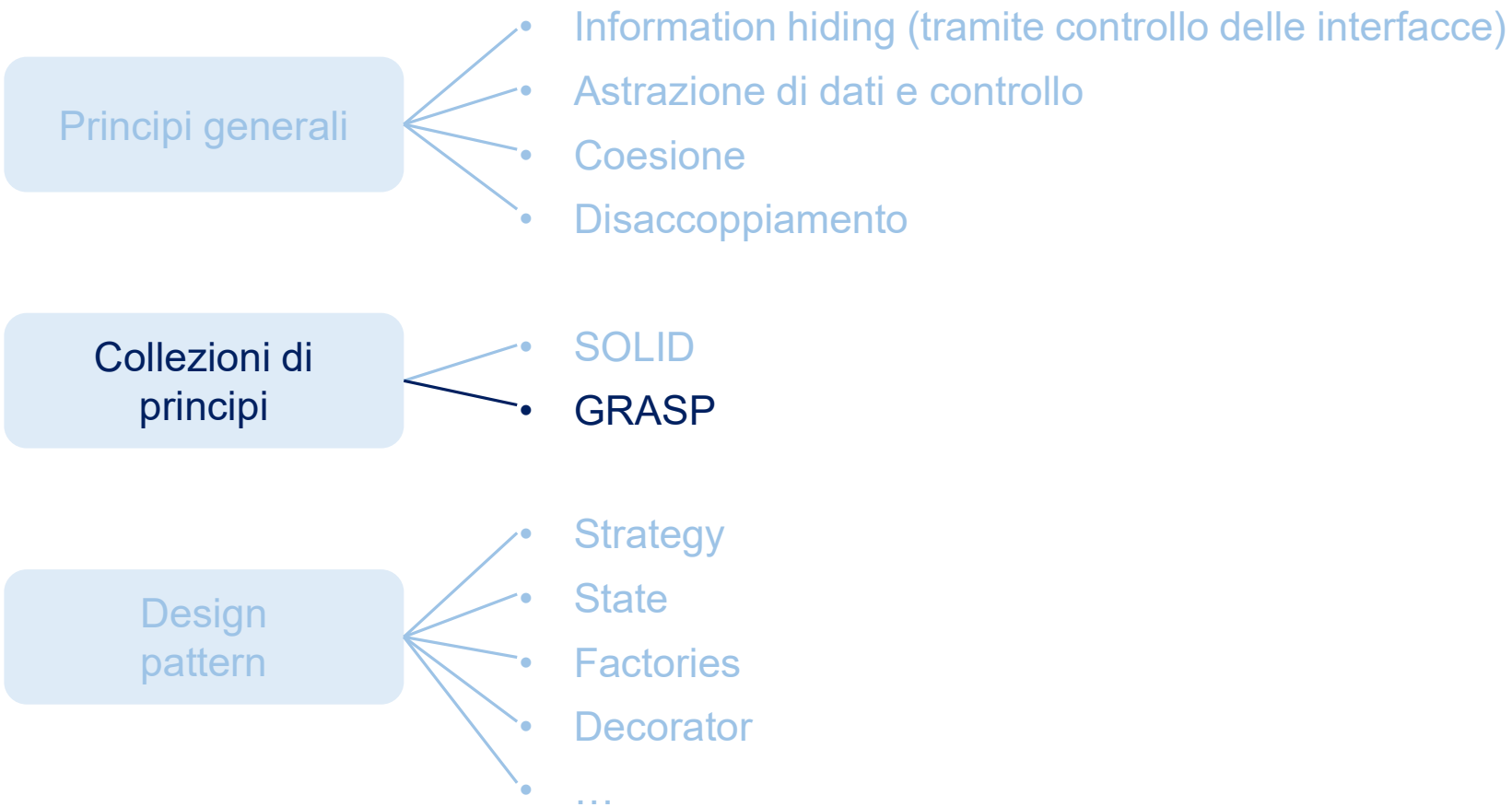
La **dependency injection** è una forma di inversione delle dipendenze

- un cliente non deve sapere «come costruire» i servizi che vuole chiamare
- il cliente **delega ad un «iniettore»** (codice esterno)
- l'iniettore passa i servizi (esistenti o costruiti dall'iniettore stesso) al client
- il client quindi usa i servizi

Il cliente non ha bisogno di conoscere iniettore e servizi (come costruirli, quali siano)

- Deve conoscere **solo le interfacce** dei servizi (che definiscono come il cliente può usarli)
- Questo **separa la responsabilità** di «uso» dalla responsabilità di «costruzione»

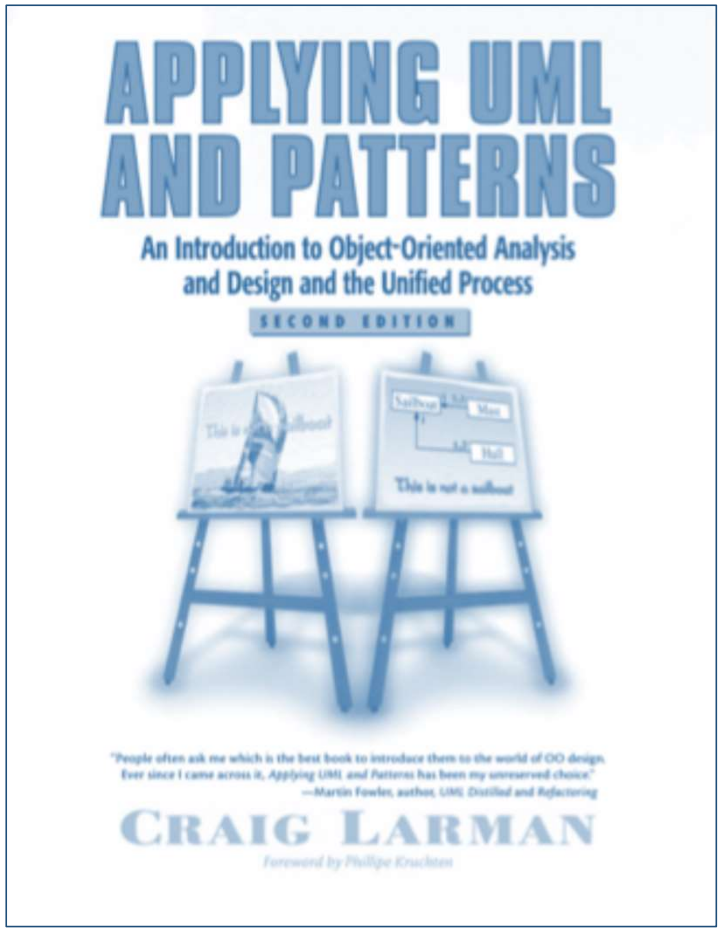
PRINCIPI E PATTERN DI PROGETTAZIONE



GRASP

Un’altra famiglia di principi di progettazione

- **G**eneral
- **R**esponsibility
- **A**ssignment
- **S**oftware
- **P**atterns



PROGETTAZIONE OBJECT-ORIENTED

Nella fase di **analisi** sono stati definiti:

- I casi d'uso
- Il dominio (in termini di classi e associazioni tra classi)

Durante la **progettazione** si devono:

- **Assegnare i metodi** alle classi
- **Dire come gli oggetti collaborano** per realizzare i casi d'uso

GRASP: PROGETTAZIONE GUIDATA DAI CASI D'USO

Realizzare un caso d'uso

- Descrivere come è realizzato nel progetto, in termini di **oggetti collaborativi**
- Si usano **diagrammi di interazione** e **pattern**
- Si assegnano **responsabilità** alle classi

Nota: Realizzare casi d'uso è un'attività di progettazione (il progetto cresce con ogni nuova realizzazione di caso d'uso)

ASSEGNARE RESPONSABILITÀ

- Le responsabilità sono **legate al dominio** del problema

- Le responsabilità sono gli obblighi che un oggetto ha, definiti in termini di comportamento

- **Due tip**i principali di responsabilità:
  - **Fare**
    - Fare qualcosa, come creare un oggetto o fare un calcolo
    - Iniziare l'azione di altri oggetti
    - Controllare e coordinare le attività di altri oggetti
  - **Conoscere**
    - Conoscere i dati privati
    - Conoscere gli oggetti correlati
    - Conoscere dati che possono derivare o calcolare
    - Questo tipo di responsabilità si può normalmente dedurre dal modello di dominio, dove sono illustrati gli attributi e le associazioni

RESPONSABILITÀ VS METODO

- La traduzione delle responsabilità del dominio del problema in classi e metodi è influenzata dalla **granularità** della responsabilità

- Una **responsabilità non è un metodo**

- I metodi sono **implementati per soddisfare** le responsabilità

GRASP

L'approccio GRASP si basa sull'assegnazione delle responsabilità:

- Si definiscono così gli oggetti e i loro metodi

- Guidati da **pattern** (schemi) di assegnazione delle responsabilità
    - Creator
    - Information Expert
    - High Cohesion
    - Low Coupling
    - Controller
    - Polymorphism
    - Indirection
    - Pure Fabrication
    - Protected Variations

PRINCIPI E PATTERN DI PROGETTAZIONE

Principi generali
- Information hiding (tramite controllo delle interfacce)
- Astrazione di dati e controllo
- Coesione
- Disaccoppiamento

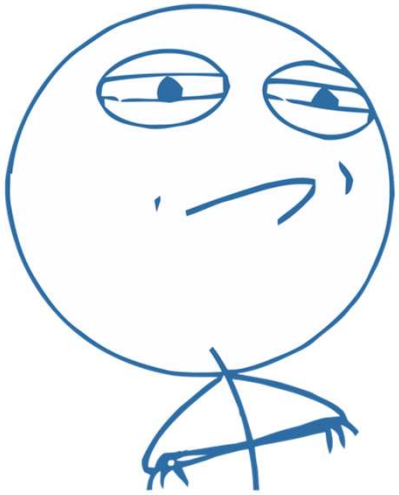
Collezioni di principi
- SOLID
- GRASP

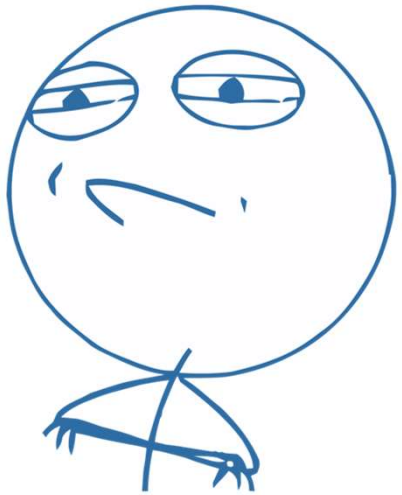
Design pattern
- Strategy
- State
- Factories
- Decorator
- …

Ci serve la «delega», finalmente

DIGRESSIONE

DELEGA VS EREDITARIETÀ





RELAZIONI TRA CLASSI E INTERFACCE

Realizzazione

- Dichiarare la conformità di un'implementazione a un'interfaccia
- Da classe a interfaccia

Estensione di tipo

- Dichiarare pubblicamente la compatibilità di due tipi
- Da interfaccia a interfaccia (ma anche da classe, poiché anche le classi sono interfacce)

Riutilizzo del codice

- Ereditare i metodi dalla classe genitore
- Da classe a classe

EREDITARIETÀ VS DELEGA

Entrambe permettono di definire una nuova classe A a partire da una classe B

Ereditarietà

- La classe A **estende il tipo** definito dalla classe B
- **utilizzando l'interfaccia** di B e
- aggiungendo campi/metodi specifici

Delega

- La classe A **estende il comportamento** della classe B
- **incorporando le funzionalità** di B
- usando un'istanza di B e chiamandone i metodi

Ma quindi è meglio che
A erediti da B o che
A deleghi a B?

PROBLEMI DELL'EREDITARIETÀ

L'ereditarietà è una cosa meravigliosa, ma **non** è sempre l'opzione migliore

Ad esempio, può accadere quanto segue:

- Si definisce una nuova classe B estendendo la classe A

  ↓

- Si scopre che molte operazioni di A non sono realmente applicabili a B

  ↓

- L'interfaccia di B non riflette realmente ciò che fa B
  (oppure si stanno ereditando molti dati che non sono appropriati per B)

ESEMPIO DI EREDITARIETÀ INAPPROPRIATA

Defininiamo una pila (`MyStack`) come estensione di un vettore (`Vector`)

- I metodi offerti dalla pila sono `push`, `pop`, `size` e `isEmpty`
- `push` e `pop` sono definiti in `MyStack`
- `size` e `isEmpty` sono ereditati da `Vector`

```
class MyStack extends Vector {
  public void push(Object element) { insertElementAt(element,0); }
  public Object pop () {
    Object result = firstElement();
    removeElementAt(0);
    return result;
  }
}
```

ESEMPIO DI EREDITARIETÀ INAPPROPRIATA (CONT.)

Il problema principale di `MyStack` è che la sua **interfaccia non è coerente** con quella attesa

- L'interfaccia dovrebbe essere limitata ai metodi `push`, `pop`, `isEmpty`, e `size`
- Estendendo `Vector`, **eredita** anche tutti i suoi **metodi**
- Alcuni metodi ereditati non sono rilevanti (o, peggio, **non adatti**) ad una pila

  Ad esempio,
  ```
  get(int index)
  set(int index, Object)
  ```
  sono parte dell'interfaccia di un vettore, ma non hanno senso nel contesto di una pila

- Questo genera confusione, perché l'interfaccia della pila espone metodi che non fanno parte del suo comportamento specifico.

ESEMPIO DI EREDITARIETÀ INAPPROPRIATA (CONT.)

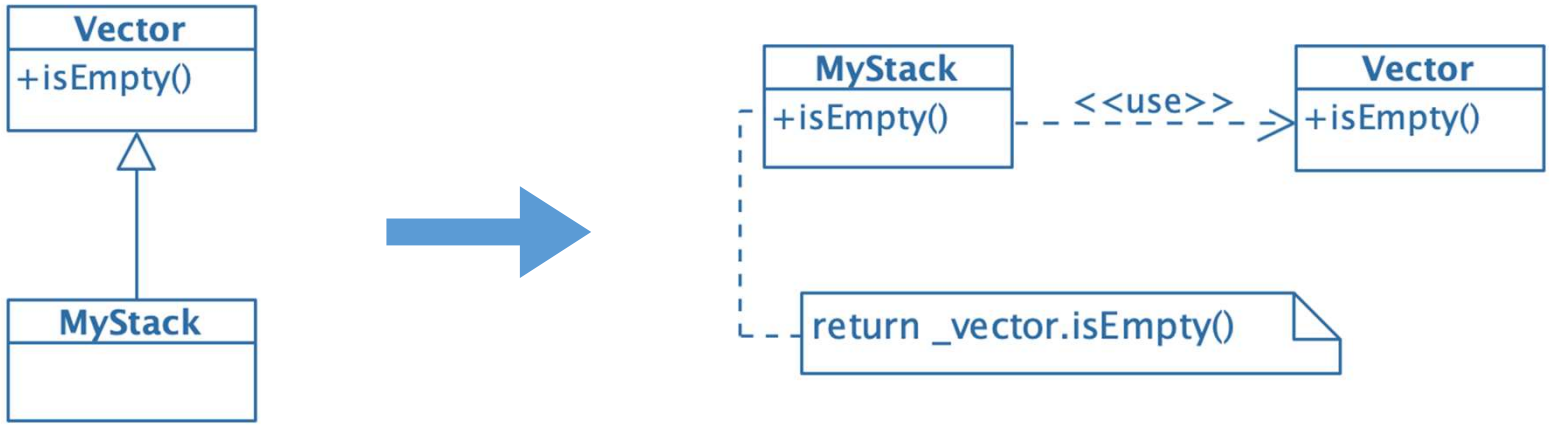
`MyStack` presenta anche problemi di **incapsulamento**

- La pila `MyStack` è strettamente legata ai dettagli di implementazione di `Vector`
- Questo può causare problemi se l'implementazione del vettore cambia in futuro

  Ad esempio, un cambiamento nell'implementazione di `Vector` potrebbe avere effetti indesiderati sulla classe `MyStack`, **violando** così il principio dell'incapsulamento

LA DELEGA

- Rende esplicito l'**utilizzo** solo **parziale** della classe delegata
- Consente di controllare **quanti e quali metodi** della classe delegata utilizzare
- Il costo è rappresentato da **metodi di delega aggiuntivi**
  - noiosi da scrivere
  - ma comunque molto semplici



ESEMPIO DI DELEGA

```
class MyStack {
  // Campo privato per contenere l'istanza di Vector
  private Vector _vector = new Vector();

  // Metodi delegati semplici per le operazioni di base
  public int size() { return _vector.size(); }
  public boolean isEmpty() { return _vector.isEmpty(); }

  // Nuovi metodi specifici della pila
  public void push(Object element) { _vector.insertElementAt(element, 0); }
  public Object pop() {
    if (_vector.isEmpty()) { throw new IllegalStateException("La pila è vuota"); }
    Object result = _vector.firstElement();
    _vector.removeElementAt(0);
    return result;
  }
}
```

QUANDO DELEGARE (INVECE DI EREDITARE)?

- L'**ereditarietà** cattura relazioni «**è un tipo di**» (piuttosto **statiche** per natura)

- Le relazioni «**è un ruolo di**» sono difficili da modellare con l'ereditarietà
  → Meglio usare la **delega**!

- Esempio: un sistema di prenotazione aerea può includere ruoli come passeggero, agente di biglietteria e personale di volo

  - Potremmo realizzare una classe `Persona` per rappresentare una persona generica e

  - Estendere `Persona` con sottoclassi corrispondenti a ciascuno dei ruoli elencati

  - **PROBLEMA: Come gestire il fatto che una persona che fa parte del personale di volo può essere anche passeggera?**

DELEGA: SVANTAGGI

- Riduce (leggermente) le **prestazioni** per l'invocazione di un'operazione di un altro oggetto (rispetto all'uso di un metodo ereditato)

- La delega non può essere utilizzata con **classi astratte**

- La delega non impone alcuna **struttura disciplinata** sul progetto

FINE DELLA DIGRESSIONE

PRINCIPI E PATTERN DI PROGETTAZIONE

Principi generali
- Information hiding (tramite controllo delle interfacce)
- Astrazione di dati e controllo
- Coesione
- Disaccoppiamento

Collezioni di principi
- SOLID
- GRASP

Design pattern
- Strategy
- State
- Factories
- Decorator
- …

Ok, ma perché?

QUALITÀ DEL SOFTWARE

Lo standard ISO/IEC 25010 definisce le caratteristiche di qualità per prodotti software

SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	PERFORMANCE EFFICIENCY	COMPATIBILITY	INTERACTION CAPABILITY	RELIABILITY	SECURITY	MAINTAINABILITY	FLEXIBILITY	SAFETY
FUNCTIONAL COMPLETENESS FUNCTIONAL CORRECTNESS FUNCTIONAL APPROPRIATENESS iso25000.com	TIME BEHAVIOUR RESOURCE UTILIZATION CAPACITY	CO-EXISTENCE INTEROPERABILITY	APPROPRIATENESS RECOGNIZABILITY LEARNABILITY OPERABILITY USER ERROR PROTECTION USER ENGAGEMENT INCLUSIVITY USER ASSISTANCE SELF-DESCRIPTIVENESS	FAULTLESSNESS AVAILABILITY FAULT TOLERANCE RECOVERABILITY	CONFIDENTIALITY INTEGRITY NON-REPUDIATION ACCOUNTABILITY AUTHENTICITY RESISTANCE	MODULARITY REUSABILITY ANALYSABILITY MODIFIABILITY TESTABILITY	ADAPTABILITY SCALABILITY INSTALLABILITY REPLACEABILITY	OPERATIONAL CONSTRAINT RISK IDENTIFICATION FAIL SAFE HAZARD WARNING SAFE INTEGRATION

(https://iso25000.com/index.php/en/iso-25000-standards/iso-25010)

FUNCTIONAL SUITABILITY

degree to which a product or system provides functions that **meet stated and implied needs** when used under specified conditions

- **Functional completeness** → quanto le funzionalità offerte coprano i task specificati e gli obiettivi degli utenti considerati
- **Functional correctness** → quanto siano accurati i risultati forniti agli utenti considerati
- **Functional appropriateness** →quanto le funzionalità offerte facilitino il raggiungimento di task e obiettivi considerati

PERFORMANCE EFFICIENCY

degree to which a system performs its functions **within specified time and throughput parameters** and is **efficient in the use of resources** under specified conditions

- **Time behaviour** → quanto vengano rispettati i requisiti in termini di *response time* e *throughput*
- **Resource utilization** → quanto vengano rispettati i requisiti in termini di quantità e tipologia di risorse computazionali utilizzate
- **Capacity** → quanto vengano rispettati i requisiti in termini di capacità (limiti massimi)

COMPATIBILITY

degree to which a system can exchange information with other systems, and/or perform its required functions while **sharing the same common environment and resources**

- **Co-existence** → quanto il sistema rimanga efficiente mentre condivide ambiente e risorse con altri sistemi (senza impattare negativamente su altri sistemi)

- **Interoperability** → quanto il sistema riesca a scambiare informazioni con altri sistemi e sfruttare mutuamente le informazioni scambiate

INTERACTION CAPABILITY

degree to which a system **can be interacted with by specified users** to exchange information in the user interface to complete specific tasks in a variety of contexts of use

- **Appropriateness recognizability** → quanto gli utenti riescano a riconoscere se il sistema sia appropriato per le loro necessità
- **Learnability** → quanto le funzionalità offerte dal sistema possano essere apprese dagli utenti considerati (in uno specifico intervallo di tempo)
- **Operability** → quanto sia facile operare e controllare il sistema
- **User error protection** → quanto il sistema prevenga/protegga gli utenti da eventuali errori operativi
- **User engagement** → quanto «engaging» sia il sistema
- **Inclusivity** → quanto il sistema possa essere utilizzato da utenti con background diversi
- **User assistance** → quanto il sistema possa essere utilizzato da utenti diversi per gli stessi task
- **Self-descriptiveness** → quanto bene siano presentate le informazioni e funzionalità del sistema, in modo rendere «ovvio» il suo utilizzo

RELIABILITY

degree to which a system **performs specified functions under specified conditions** for a specified period of time

- **Faultlessness** → quanto il sistema funzioni senza fallimenti
- **Availability** → quanto il sistema sia operativo ed accessibile quando richiesto
- **Fault tolerance** → quanto il sistema sia resiliente ad eventuali fallimenti
- **Recoverability** → quanto, in caso di fallimenti, il sistema riesca a «recuperare» ripristinando lo stato desiderato

SECURITY

degree to which a product or system **defends against attack patterns** by malicious actors and **protects information and data**

- **Confidentiality** → quanto il sistema assicuri che i dati siano accessibili solo a chi è autorizzato
- **Integrity** → quanto il sistema assicuri che lo stato del sistema e i dati siano protetti da modifiche/cancellazioni non autorizzate
- **Non-repudiation** → quanto sia dimostrabile che azioni/eventi hanno avuto luogo (per evitarnle eventuali successive ripudazioni)
- **Accountability** → quanto siano tracciabili le azioni di un'entità
- **Authenticity** → quanto sia dimostrabile che l'identità di un soggetto/risorsa sia effettivamente quella dichiarata
- **Resistance** → quanto il sistema sia resistente ad eventuali attacchi di malintenzionati

MAINTAINABILITY

degree of effectiveness and efficiency with which a system **can be modified** to improve it, correct it or adapt it to changes in environment, and in requirements

- **Modularity** → quanto (poco) i cambiamenti di un modulo del sistema impattino su altri moduli
- **Reusability** → quanto sia riusabile il sistema (o sue parti)
- **Analysability** → quanto sia facile analizzare l'impatto di un eventuale cambiamento, diagnosticare le cause di eventuali problemi, o identificare le parti da modificare
- **Modifiability** → quanto sia «modificabile» il sistema senza degradarne la qualità
- **Testability** → quanto sia facile determinare criteri di test e testare effettivamente il sistema

FLEXIBILITY

degree to which a product can be **adapted to changes** in its requirements, contexts of use or system environment

- **Adaptability** → quanto il sistema sia «adattabile» a cambiamenti hardware/software degli ambienti di esecuzione
- **Scalability** → quanto bene il sistema gestisca cambiamenti di workload (e *variability*)
- **Installability** → quanto efficientemente il sistema possa essere installato/disinstallato in ambienti specifici
- **Replaceability** → quanto il sistema possa sostituirne un altro, sviluppato per gli stessi scopi e lo stesso ambiente di esecuzion

SAFETY

degree to which a product **avoids** a state in which human life, health, property, or the environment is **endangered**

- **Operational constraint** → quanto il sistema sia vincolato a rimanere in parametri di sicurezza
- **Risk identification** → quanto gli eventuali rischi di sicurezza siano effettivamente identificati
- **Fail safe** → quanto il sistema riesca ad operare in «safe mode» in caso di fallimenti/pericoli
- **Hazard warning** → quanto il sitema fornisca «warning» su rischi inaccettabili agli operatori (o controlli interni) in modo che possano reagire tempestivamente
- **Safe integration** → quanto il sistema riesca a mantenere la *safety* se integrato con altri sistemi

STILI ARCHITETTURALI E QUALITA DEL SOFTWARE

Valutiamo le caratteristiche di alcuni stili architetturali in base a

- Disponibilità (aka. **reliability → availability**)
- Tolleranza ai guasti (aka. **reliability → fault tolerance**)
- Modificabilità (aka. **maintainability → modifiability**)
- Efficienza (aka. **performance efficiency**)
- Scalabilità (aka. **flexibility → scalability**)

scalabilità verticale vs scalabilità orizzontale

verticale (scale down/up) → aggiunta di risorse computazionali ad un singolo nodo
(es. più CPU, memoria e storage per i database)

orizzontale (scale in/out) → replicazione di nodi
(es. aggiunta di repliche di servizi web)

CLIENT-SERVER, 2-N TIER

Disponibilità	I server di ogni tier (ordine, fila) possono essere replicati, quindi anche se uno fallisse ci sarebbe solo una minor QoS.
Fault tolerance	Se un cliente sta comunicando con un server che fallisce, la maggior parte dei server reindirizza la richiesta a un server replicato in modo trasparente all'utente.
Modificabilità	Il disaccoppiamento e la coesione tipici di questa arch. favoriscono la modificabilità
Performance (efficienza)	Performance ok, ma da tenere sott'occhio: numero di threads paralleli su ogni server, velocità delle comunicazioni tra server, volume dati scambiato
Scalabilità	Basta replicare i server in ogni tier (quindi ok scale out). Unico collo di bottiglia l'eventuale base di dati che scala male orizzontalmente

PIPES AND FILTERS

Disponibilità	Avendo "pezzi" (componenti e possibilità di connetterle) sufficienti a formare una catena.
Fault tolerance	Occorre riparare una catena interrotta usando componenti replica.
Modificabilità	Si, se le modifiche interessano una o comunque poche componenti
Performance (efficienza)	Dipende dalla capacità del canale di comunicazione e dalla performance del filtro più lento.
Scalabilità	Ok anche scale out.

PUBLISH-SUBSCRIBE

Disponibilità	Si possono creare clusters di dispatcher
Fault tolerance	Si cerca un dispatcher replica
Modificabilità	Si possono aggiungere publisher e subscribers a piacere. Unica attenzione al formato dei messaggi.
Performance (efficienza)	Ok. Ma compromesso tra velocità e altri requisiti tipo affidabilità e/o sicurezza.
Scalabilità	Ok scale out: con un cluster di dispatchers si può gestire un volume molto elevato di messaggi.

P2P

Disponibilità	Dipende dal numero di nodi in rete, ma si assume si.
Fault tolerance	Gratis
Modificabilità	Si, se dell'architettura interessa solo la parte di comunicazione
Performance (efficienza)	Dipende dal numero di nodi connessi, dalla rete, dagli algoritmi. Per esempio BitTorrent ottimizza scaricando per primo il file/pezzo più raro.
Scalabilità	Gratis


Q
A

RIFERIMENTI

Contenuti

- **Capitolo 6** di "Software Engineering" (G. C. Kung, 2023)
- **Sezioni 19.1-19.3** di "Software Engineering" (G. C. Kung, 2023)

Approfondimenti

- **Principles of OOD**, Robert C. Martin (http://butunclebob.com/)