# Fase di Progettazione: Principi di progettazione e qualità di un progetto

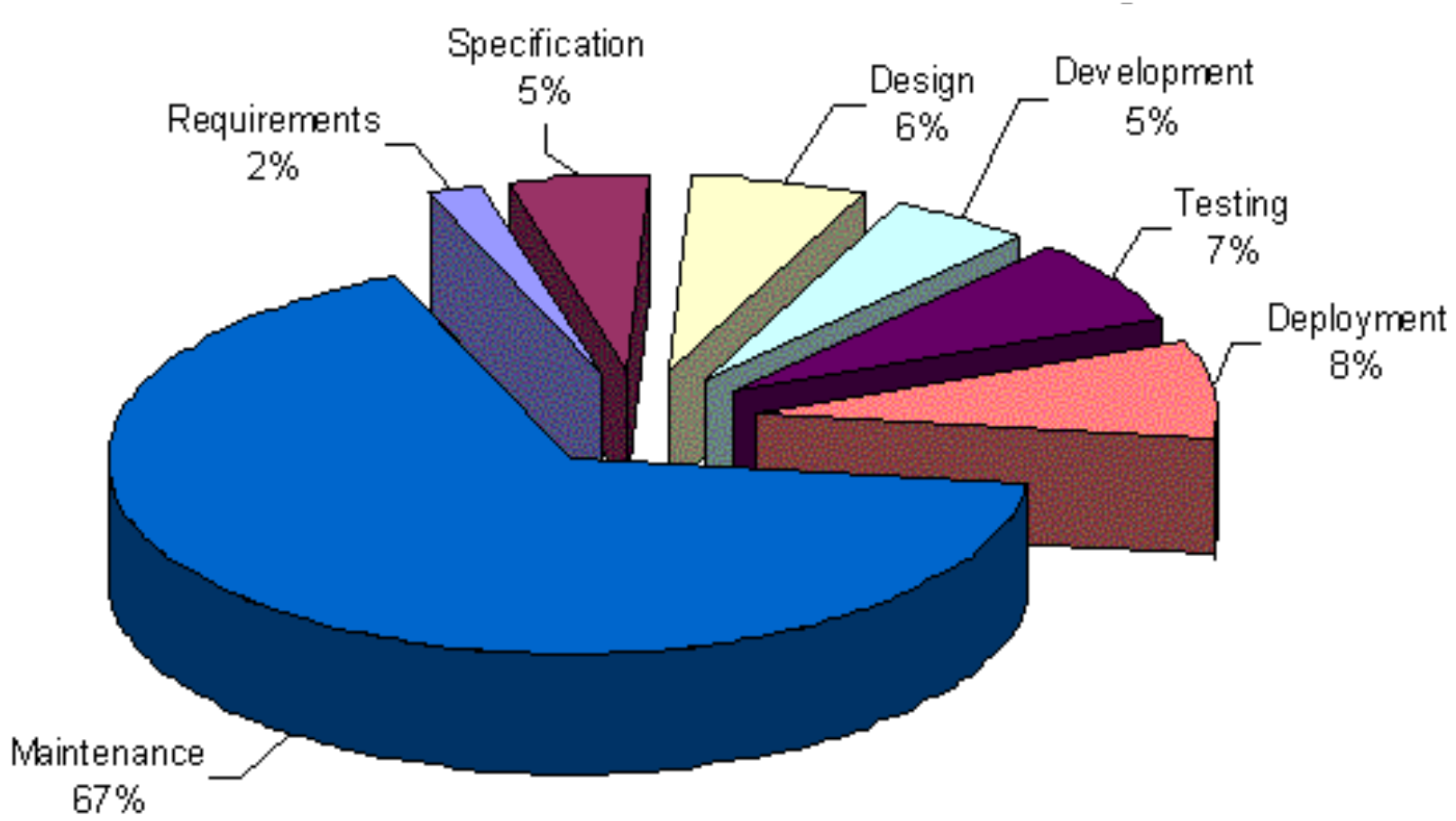
Roberta Gori, Laura Semini
Ingegneria del Software
Dipartimento di Informatica
Università di Pisa

# Progettare Software

- Obiettivo della fase di progettazione non è solo la pianificazione del lavoro
- Ma anche pensare alla qualità:
  - manutenzione e
  - riuso

# I costi del software (dalla lezione 1)

- Il costo di un prodotto software durante le fasi del suo ciclo di vita



# La manutenzione (dalla lezione 1)

- La manutenzione include tutti i cambiamenti al prodotto software, anche dopo che è stato consegnato al cliente
- Si divide in :
  - manutenzione correttiva(20%), rimuove gli errori lasciando invariata la specifica
  - manutenzione migliorativa, consiste in cambiamenti alla specifica e nell'implementazione degli stessi, può essere:
    - Perfettiva (60%): modifiche per migliorare le qualità del software, introduzione di nuove funzionalita', miglioramento delle funzionalità esistenti.
    - Adattativa (20%): modifiche a seguito di cambiamenti nell'ambiente legislativo, cambiamenti nell'Hardware, nel Sistema operativo, ecc.
      - Esempio: IVA dal 22% al 20% float aliquota=22; ...; prezzotot =prezzo+(prezzo*aliquota)/100

# Progettare pensando alla manutenzione e al riuso

- Le buone pratiche e le tecniche di progettazione mirano a produrre un sistema, non solo che realizza i requisiti funzionali e di qualità, ma anche:
  - Facilmente mantenibile: su cui è facile (ed economico) fare manutenzione
  - Riusabile: è facile riusare parti del sistema in altri sistemi futuri

# Principi e pattern di progettazione (OO)

## Principi generali

1. Information Hiding
   - Controllo delle interfacce
2. Astrazione
   - Dati e controllo
3. Coesione
4. Disaccoppiamento

## Collezioni di principi noti in letteratura

SOLID

GRASP

# Unità di progettazione: componenti e moduli

- **Componente**
  - elemento a "run time"
- Un componente di un sistema ha un'interfaccia ben definita verso gli altri componenti
- La progettazione dovrebbe facilitare composizione, riuso e manutenzione dei componenti

- **Modulo**
  - elemento a "design time"
- I moduli sono anche
  - unità di incapsulamento: permettono di separare l'interfaccia dal corpo
  - unità di compilazione quando possono essere compilati separatamente

# 1. Information hiding (vs incapsulamento)

- L'incapsulamento indica la proprietà degli oggetti di mantenere al loro interno (incapsulare) sia gli attributi che i metodi, cioè stato e comportamento dell'oggetto.

- Alcuni attributi e metodi possono anche essere nascosti all'interno dell'oggetto, cioè resi invisibili agli altri oggetti; si parla in questo caso di **information hiding**

- Attributi e metodi si possono «nascondere» dichiarandoli privati.

- L'incapsulamento permette l'information hiding, non lo garantisce: una variabile pubblica è incapsulata, non nascosta

# 1. Information hiding

- Separazione tra interfaccia (visibile) e implementazione (invisibile/privata/nascosta)
- Componenti o moduli come scatole nere
  - Fornitori e clienti di funzionalità
  - È nota solo l'interfaccia
- Sono mantenuti nascosti
  - Algoritmi usati
  - Strutture dati interne (tra cui variabili)

# 1. Information hiding: interfaccia e corpo

- L'interfaccia di un'unità di progettazione esprime ciò che l'unità offre o richiede:
  - gli elementi dell'interfaccia sono visibili alle altre unità.
- Il corpo contiene l'implementazione degli elementi dell'interfaccia e realizza la semantica dell'unità
  - Il corpo è tenuto nascosto alle altre unità

# 1. Information hiding: vantaggi

- Comprensibilità
  - nessuna necessità di comprendere i dettagli implementativi di un'unità per usarla.
- Manutenibilità
  - si può cambiare l'implementazione di una unità senza dover modificare le altre.
- Lavoro in team
  - La separazione corpo-iterfaccia facilita lo sviluppo da parte di persone che lavorano in modo indipendente, il riuso, le riparazioni e le riconfigurazioni
- Sicurezza
  - I dati di una unità possono essere modificati solo da funzioni interne alla stessa, e non dall' esterno.

# 1. Information hiding: tecniche note

- Tecniche a voi già note da corsi precedenti per realizzare information hiding:
  - Variabili private cui si accede solo con setters e getters

# Accessors & Mutators (aka getters & setters) per Information Hiding

- Modalità standard per accedere agli attributi di una classe
  - set()
  - get()
- Si nasconde la rappresentazione dei dati

# Accessors (la get)

- La get( ) non deve avere side effects, i.e. non deve modificare lo stato dell'oggetto.
- Inoltre è buona pratica restituire un dato come valore e non come riferimento in modo che se viene modificato non abbia effetto sull'oggetto originario
  - A meno che non ci sia una reale necessità, restituire un clone

# Mutators (la set)

- La set( ) permette una modifica **controllata** delle proprietà di un oggetto
  - cambia lo stato dell'oggetto, ma è possibile fare dei controlli prima che la modifica sia effettuata

# Discussion: accessors and mutators

- Molti editori permettono di generare automaticamente set e get per ogni attributo che viene aggiunto a una classe.
- Dobbiamo lasciarli fare? ;-)
- Ci sono almeno due svantaggi
  - In primis, potrebbero non essere necessari. Una volta introdotti, altri moduli (componenti) potrebbero usarle e a quel punto devono essere mantenuti.
  - In secondo luogo, banalmente, si potrebbe non voler permettere un accesso.

# 2. Astrazione sul controllo: librerie

- Nei linguaggi di programmazione tradizionali per modulo si intende una prima forma di astrazione effettuata sul flusso di controllo e il concetto di modulo è identificato con il concetto di **subroutine o procedura**

- Una procedura può effettivamente nascondere una scelta di progetto riguardante l'algoritmo utilizzato.
  - Esempio: algoritmi di ordinamento

- Le procedure in quanto astrazioni sul controllo sono utilizzate come parti di alcune classi di moduli, che prendono il nome di librerie (ad esempio, librerie di funzioni matematiche e grafiche)

# 2. Astrazione sui dati: ADS

- Un'astrazione di dato (o **Struttura Dati Astratta**, ADS) è un modo di incapsulare un dato in una rappresentazione tale da regolamentarne l'accesso e la modifica
- Interfaccia rimane stabile anche in presenza di modifiche alla struttura dati
- Non puramente funzionali (come le librerie)
  - Risultati dell'attivazione di un'operazione comportano modifiche al dato (cambia lo stato)

# 3. Coesione

- Proprietà di una unità di realizzazione (o sottosistema)
  - grado in cui una unità realizza "uno e un solo concetto"
  - funzionalità "vicine" devono stare nella stessa unità
    - vicinanza per tipo, algoritmi, dati in ingresso e in uscita

- **L'obbiettivo del progettista è creare sistemi coesi**, in cui tutti gli elementi di ogni unità di progettazione siano strettamente collegati tra loro.

# Classe per nulla coesa

```
public class Activities
{
        public void PrintDocument(Document doc) {
                ……
        }

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

        public void CalculateDist(int x1, int y1, int x2, int y2){
                ……
        }
}
```

# Tipi (e gradi) di coesione

- **Coesione funzionale**: raggruppa parti che collaborano per realizzare una funzionalità
  - È la situazione ideale.
- **Coesione comunicativa**: tra elementi che operano sugli stessi dati di input o contribuiscono agli stessi dati di output.
  - Esempio: aggiornare il record nel database e inviarlo alla stampante.
  - Non è un buon modo di raggruppare, non supporta il riuso
- **Coesione procedurale**: tra elementi che realizzano i passi di una procedura.
  - Esempio: calcola la media di uno studente, stampa la media di uno studente
  - Le azioni sono debolmente connesse e difficilmente riutilizzabili.

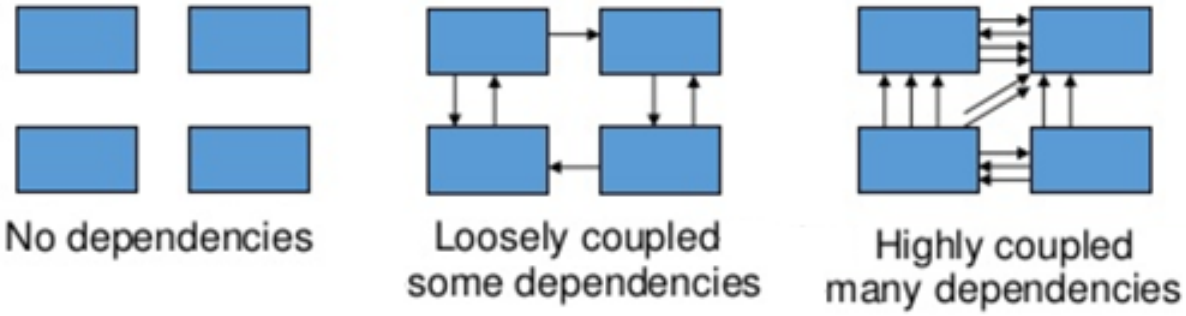
# Tipi (e gradi) di coesione (cont'd)

- **Coesione temporale**: tra azioni che devono essere fatte in uno stesso arco di tempo
  - Esempio: azioni da fare all'apertura dell'anno accademico
  - Le azioni sono debolmente connesse e difficilmente riutilizzabili.
  - Soluzione preferibile: dividere le azioni in diverse unità, avere una routine che manda a tutte loro un evento di avvio
- **Coesione logica**: tra elementi che sono logicamente correlati e non funzionalmente.
  - Esempio: un componente legge input da nastro, disco e rete. Le operazioni sono correlate, ma le funzioni sono significativamente diverse.
  - Le operazioni sono debolmente connesse e difficilmente riutilizzabili.
- **Coesione accidentale**: tra elementi non correlati ma piazzati assieme
  - Esempio: Classe Activities vista 2 lucidi fa
  - È la peggiore forma di coesione.

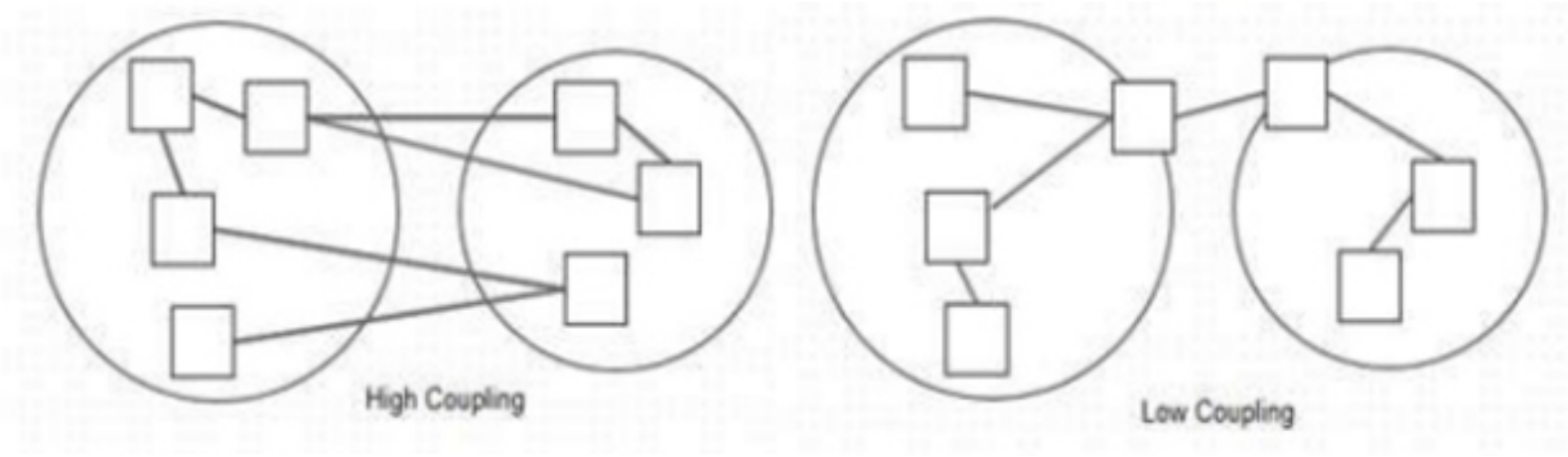
# 4. Disaccoppiamento (aka uncoupling)

- Proprietà di un insieme di unità di progettazione (in generale di una architettura)
  - grado in cui un'unità di progettazione è "legata" ad altre unità di progettazione
  - Dipendenze, scambio di messaggi…
- **L'obbiettivo del progettista è creare sistemi disaccoppiati**, in cui le unità di progettazione non sono strettamente legate (coupled) tra loro.

# 4. Disaccoppiamento




Il grado di disaccoppiamento si valuta tra diverse unità, non all'interno delle unità



# Coesione vs disaccoppiamento

- Si hanno vantaggi con sistemi che esibiscono
  - un alto grado di coesione
  - un basso accoppiamento
- Maggior riuso e migliore mantenibilità
- Ridotta interazione fra (sotto)sistemi
- Miglior comprensione del sistema
- Garantire un alto grado di coesione normalmente riduce il grado di accoppiamento.

# SOLID

Cinque principi di base di progettazione e programmazione object-oriented.

Si applicano principalmente in fase di progettazione di dettaglio

Autore: Robert C. Martin (Aka uncle Bob)

# SOLID

- Single Responsibility Principle
  - Una classe (o metodo) dovrebbe avere solo un motivo per cambiare.
- Open Closed Principle
  - Estendere una classe non dovrebbe comportare modifiche alla stessa.
- Liskov Substitution Principle
  - Istanze di classi derivate possono essere usate al posto di istanze della classe base.
- Interface Segregation Principle
  - Fate interfacce a grana fine e specifiche per ogni cliente.
- Dependency Inversion Principle
  - Programma guardando le interface non l'implementazione.


# SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

# SOLID 1: Single Responsibility Principle

- Una classe (un modulo, un metodo) dovrebbe avere solo un motivo per cambiare
  - La responsabilità è identificata come un motivo per cambiare.
  - Questo principio afferma che se abbiamo 2 motivi per cambiare una classe, dobbiamo dividere la funzionalità in due classi.
    - In futuro, se dobbiamo fare un cambiamento, lo facciamo nella classe che realizza la funzionalità.
    - Se dovessimo fare un cambiamento in una classe che ha più responsabilità, le modifiche potrebbero influenzare altre funzionalità della classe e a cascata tutti i moduli che le usano

# SOLID 1: Single Responsibility Principle

- Il Single Responsibility Principle fu introdotto da Tom DeMarco nel suo libro Structured Analysis and Systems Specification, 1979.
- Robert Martin ha reinterpretato il concetto e definito la responsabilità come "un motivo per cambiare".
- E' un altro modo per dire che una classe deve essere **funzionalmente coesa** (e realizzare una sola funzionalità)

# Una classe dovrebbe avere un solo motivo per cambiare

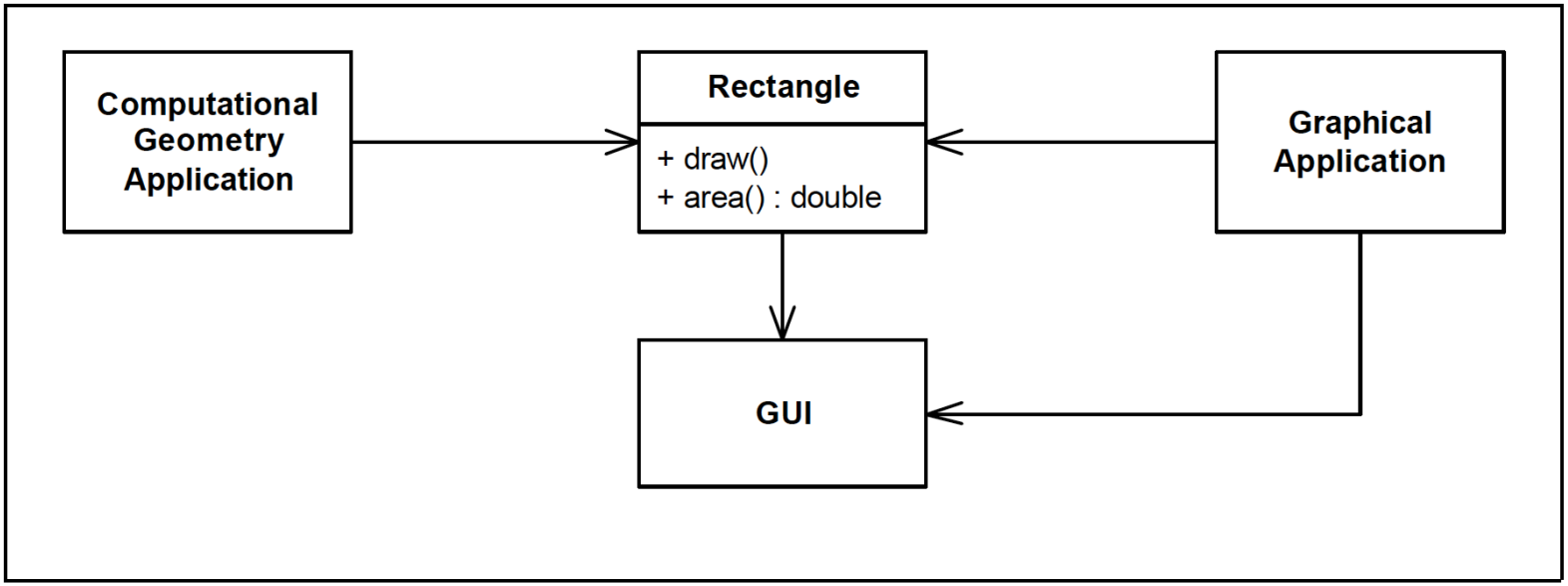



**Figure 8-1**
More than one responsibility

# Separare le responsabilita'

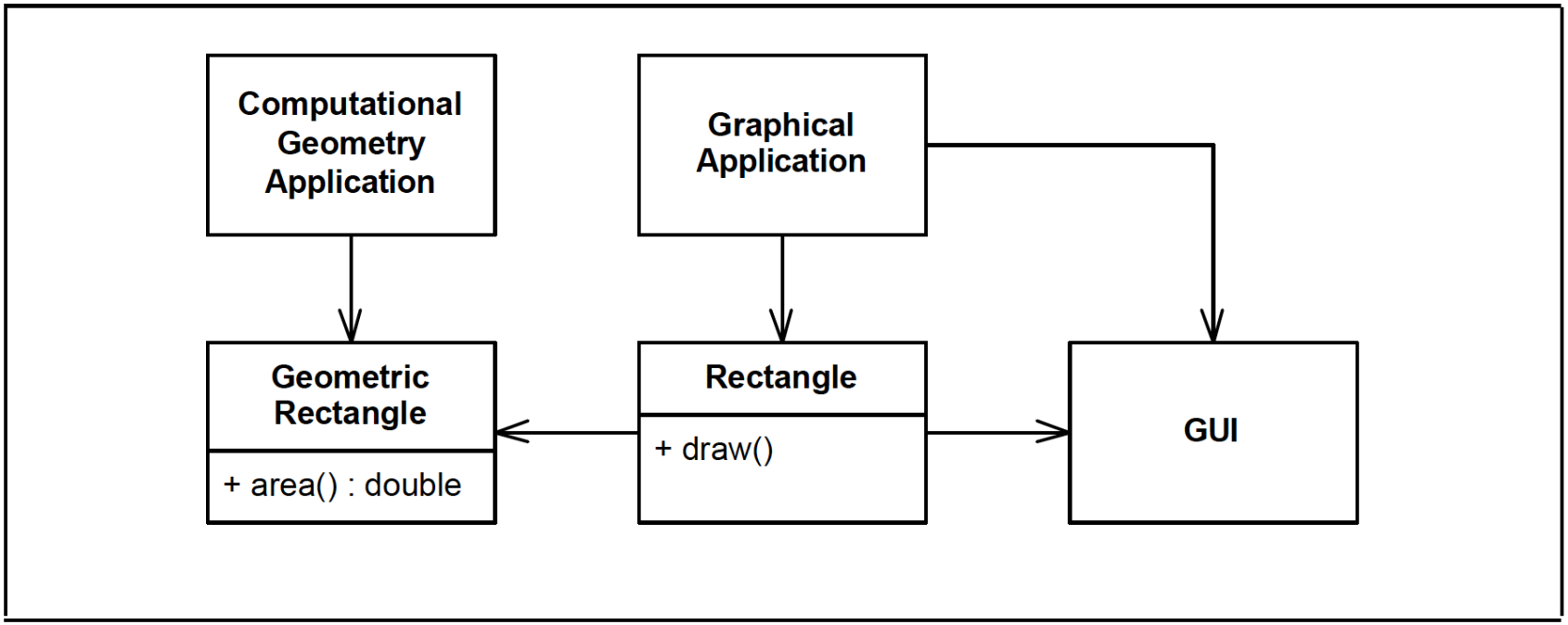



**Figure 8-2**
Separated Responsibilities

# Come decidere che due responsanbilità vanno divise

**Listing 8-1 (Continued)**

Modem.cs -- SRP Violation

```
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

- Due responsabilità:
  - metodi per la connessione
  - metodi per il trasferimento dati

Queste due responsabilità dovrebbero essere separate?

Discussione nei prox. 2 lucidi

# Dipende da come può cambiare l'applicazione

- se l'applicazione può cambiare richiedendo il cambiamento della segnatura dei metodi per la connessione allora la precedente soluzione risulta rigida perché le classi che chiamano send() e read() devono essere ricompilate e ricontrollate più spesso di quanto non vorremmo

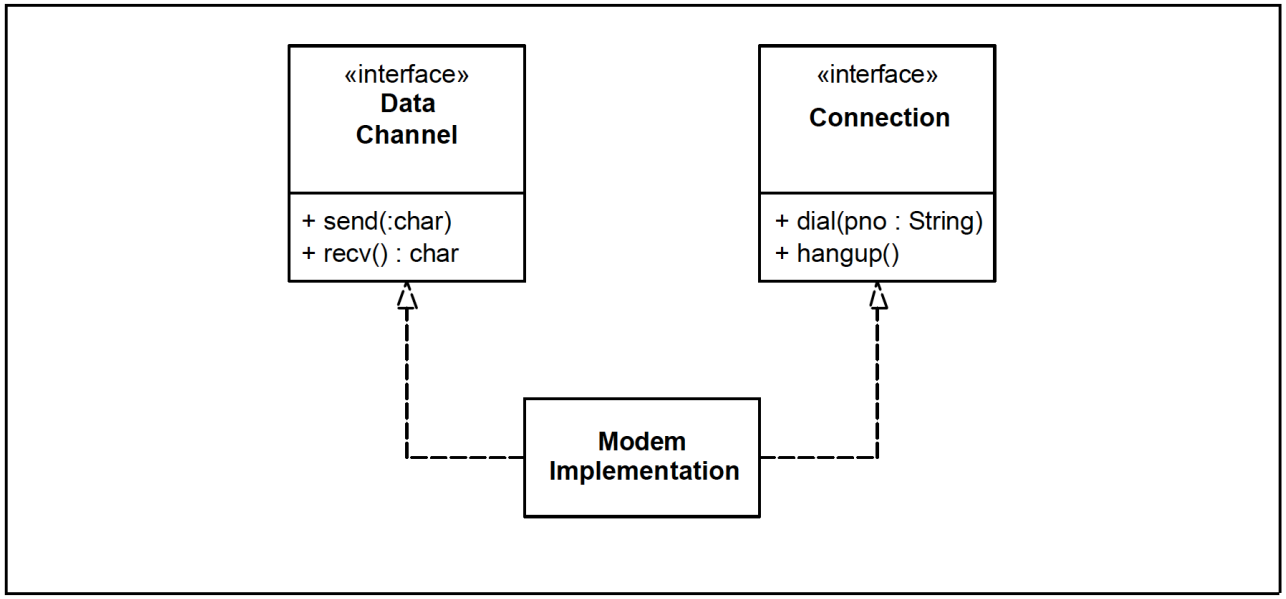



**Figure 8-3**
Separated Modem Interface

# Eccezione alla regola

- Se d'altra parte l'applicazione non può cambiare in modo da richiedere il cambiamento delle due diverse responsabilità in momenti diversi, allora separarle introdurrebbe una complessità non necessaria.

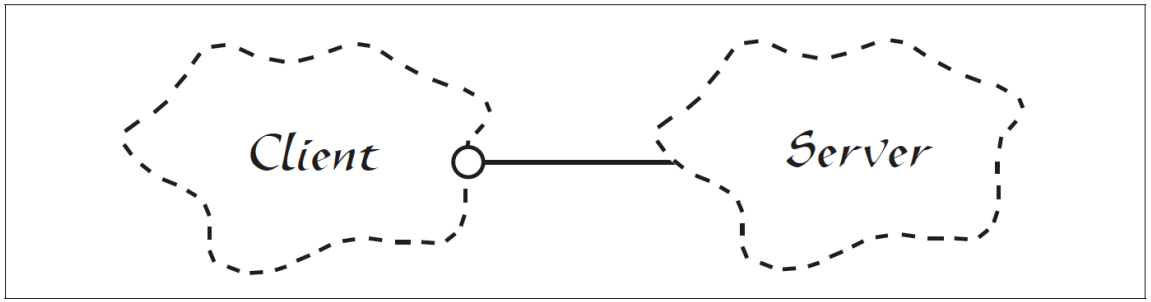
Riassumendo:

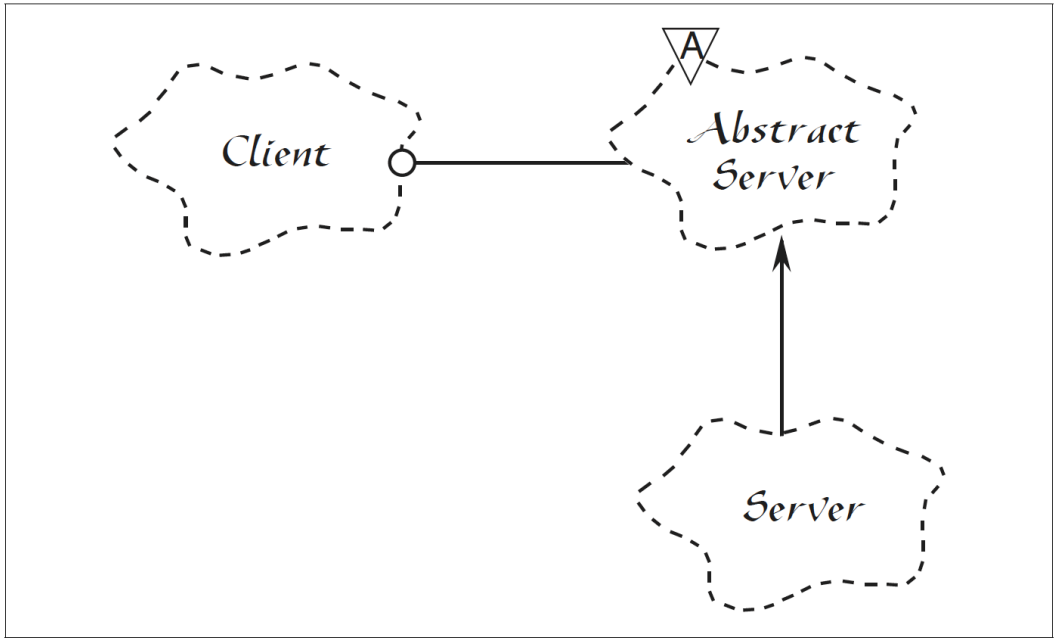
Un motivo di cambiamento è tale solo se è una reale possibilità di cambiamento del sistema

# SOLID 2: Open Closed Principle

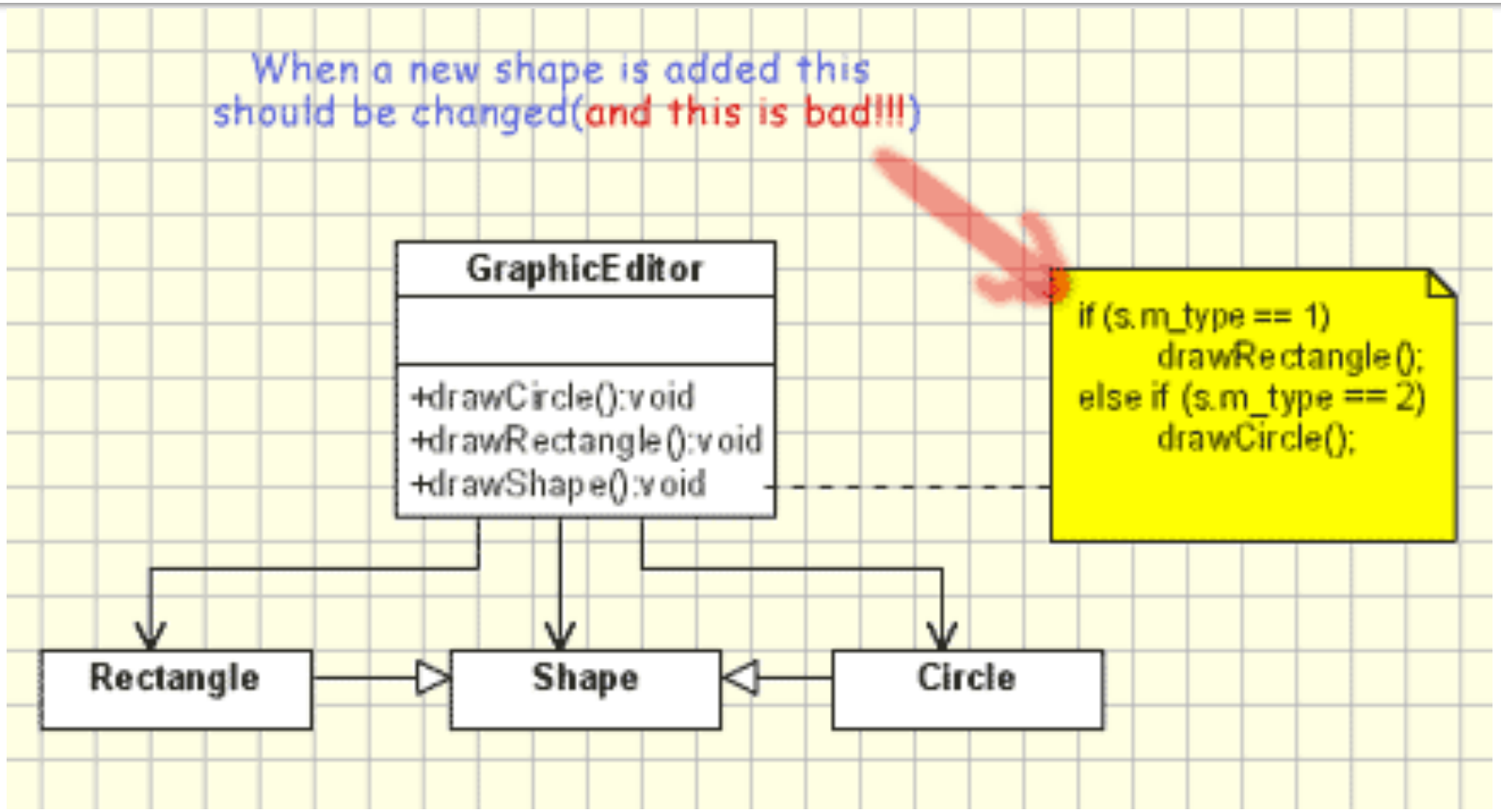
- L'entità software  (classi, moduli e funzioni) devono essere aperte per estensione, ma chiuse per modifiche.
  - Disegnare moduli che non cambiano. Quando i requisiti cambiano, estendere il comportamento del modulo aggiungendo nuovo codice ma non cambiando quello vecchio che già funziona
  - Disegnare classi in modo che sia possibile  estenderle ma senza cambiarle.  Questo è possibile mediante  l'uso di classi astratte e classi concrete che le implementano.

# Uso della classe concreta vs classe astratta per il server



# SOLID 2: Open Closed Principle: ad ex.



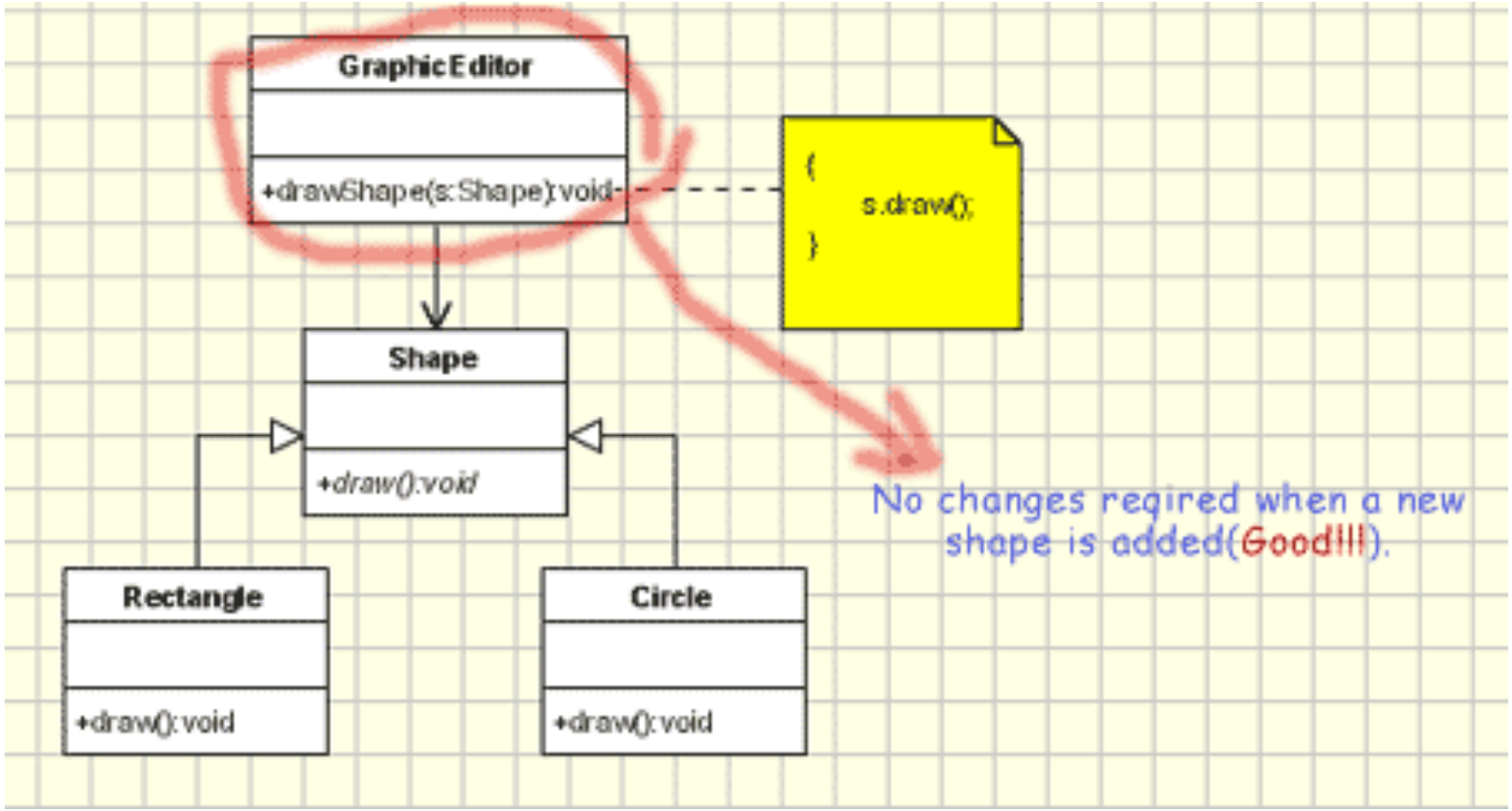
# SOLID 2: Open Closed Principle.
## Esempio di non applicazione del principio

```
class GraphicEditor {
        public void drawShape(Shape s) {
                if (s.m_type==1)  drawRectangle(s);
                else if (s.m_type==2) drawCircle(s);
        }
        public void drawCircle(Circle r) {….}
        public void drawRectangle(Rectangle r) {….}
}

class Shape {int m_type; }

class Rectangle extends Shape {
        Rectangle() {super.m_type=1;}
}

 class Circle extends Shape {
        Circle() {super.m_type=2; }
}
```

# SOLID 2: Open Closed Principe

- Aperti alle estensioni: usate astrazioni
  - Interfacce
  - classi astratte
- Chiusi al cambiamento: usate la delega + astrazioni
  - Esempi nei prossimi lucidi

# SOLID 2: Open Closed Principle applicato

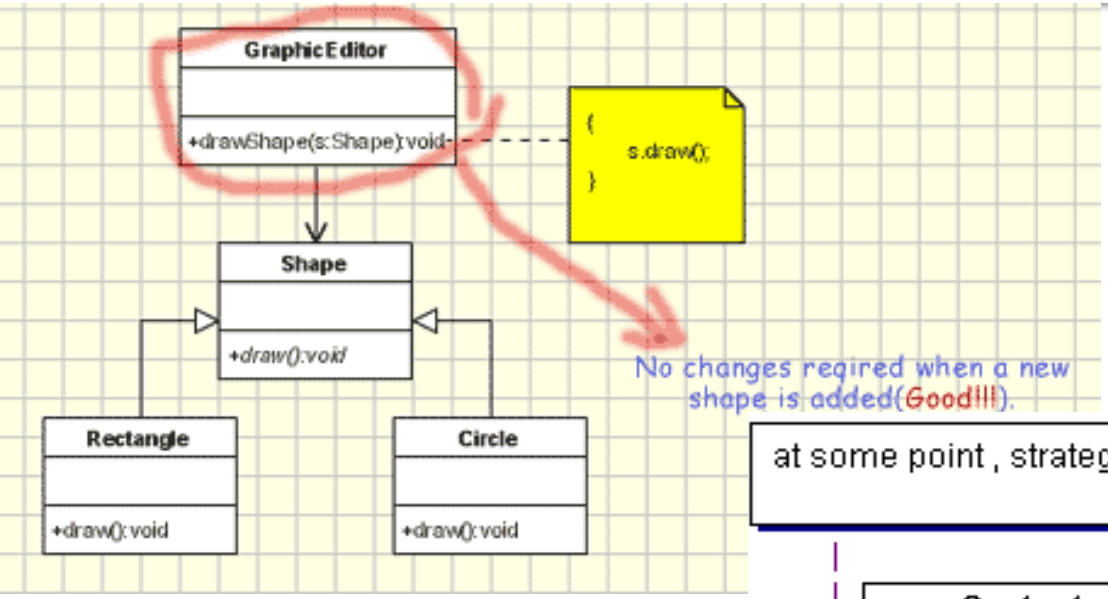


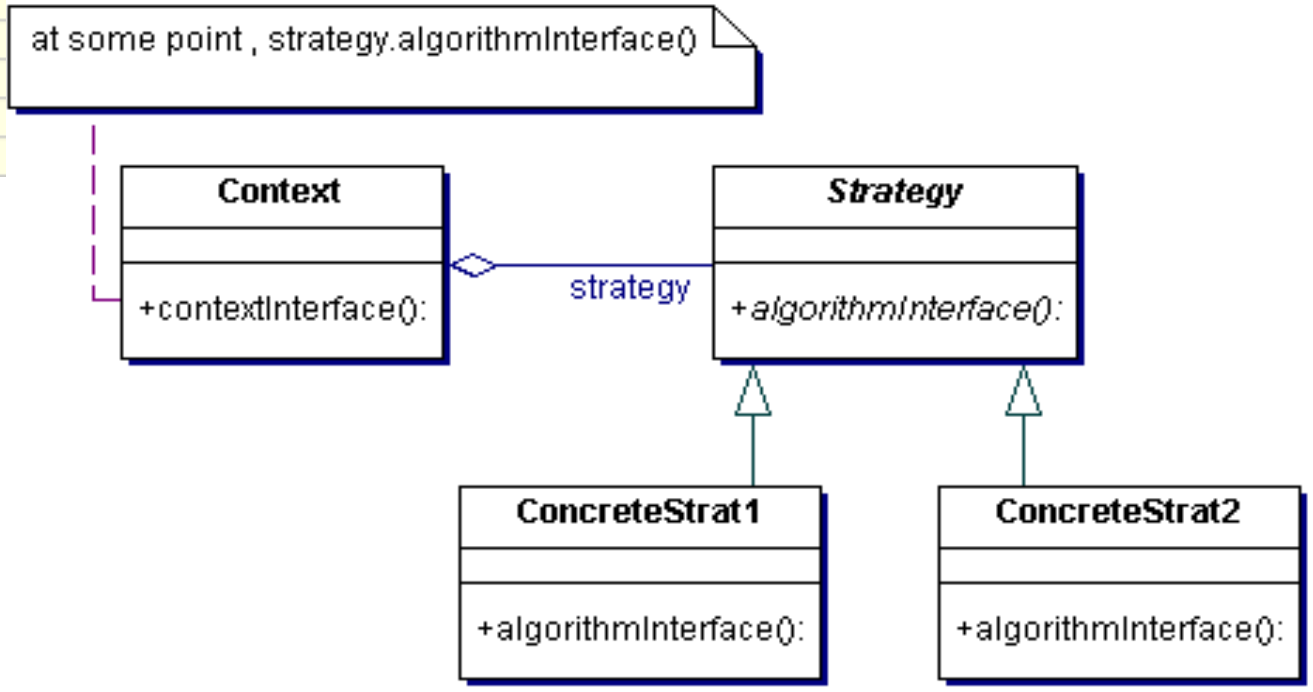
# SOLID 2: Open Closed Principle applicato

```
class GraphicEditor {
        public void drawShape(Shape s) {
                s.draw();
        }
}

class Shape {
        abstract void draw();
}

class Rectangle extends Shape  {
        public void draw() {
                // draw the rectangle
        }
}
```

- Si sposta il codice che dipende dalle classi concrete nelle classi concrete stesse
- Si opera per «delega», applicando di fatto il design pattern Strategy

# Design pattern Strategy: struttura



# Design pattern Strategy: codice

```
public class Context() {
   private Strategy strategy;
            // we set the strategy in the constructor…
   public Context(Strategy strategy) {  this.strategy = strategy;}
            //…. or dinamically
   public void setStrategy(Strategy strategy) { this.strategy = strategy; }
   public void executeTheStrategy() { strategy.doSomething();}
}
public interface Strategy {
   public void doSomething();
}
public class Strategy1 implements Strategy {
   public void doSomething() { System.out.println("Execute strategy 1");  }
}
public class Strategy2 implements Strategy {
   public void doSomething() { System.out.println("Execute strategy 2");  }
}
```

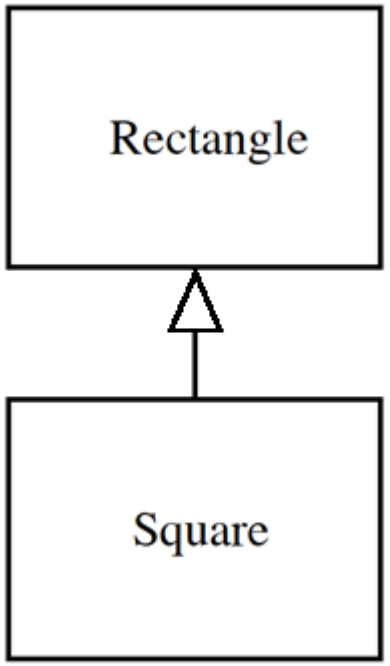
# SOLID 3: Liskov Substitution Principle

- Il Principio di Sostituzione, definito da Barbara Liskov, fondamentalmente dice:
  - Se S è sottotipo di T, allora per ogni oggetto o1 di tipo S esiste un oggetto o2 di tipo T, tale che, dato un qualsiasi programma P definite in termini di T, il comportamento di P è immutato quando o1 è usato al posto di o2

- Le classi derivate devono potersi sostituire alle classi base.

# Example: un problema sottile

```
class Rectangle
{
  public:
    void   SetWidth(double w)  {itsWidth=w;}
    void   SetHeight(double h) {itsHeight=w;}
    double GetHeight() const   {return itsHeight;}
    double GetWidth() const    {return itsWidth;}
  private:
    double itsWidth;
    double itsHeight;
};
```




Cosa succede per il quadrato?

- Spreco di memoria
- Inconsistenza con l'uso di GetHeight and GetWidth

# Override Set Width e SetHeight

```
void Square::SetWidth(double w)
{
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}
```

```
Square s;
s.SetWidth(1); // Fortunately sets the height to 1 too.
s,SetHeight(2); // sets width and heigt to 2, good thing.
```

# SOLID 4: Interface Segregation Principle

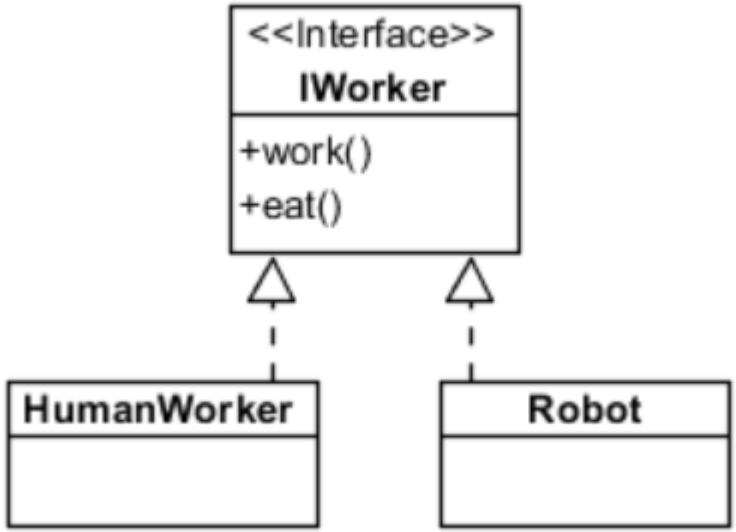
- Fate interfacce a grana fine e specifiche per ogni cliente.
  - I client non devono dipendere da interface che non usano.
- Questo principio dice che occorre prestare attenzione al modo in cui si scrivono le interfacce.
  - Mettere solo i metodi necessari.
  - Ogni metodo che si aggiunge, anche se inutile, deve essere implementato.

# SOLID 4: Interface Segregation Principle

- Il metodo eat ( ),

deve essere implementato
da tutti gli Workers

  - ...compresi i Robot
  - Succede spesso di creare sottoclassi o implemenentazioni di una interfaccia per cui non tutti i metodi della superclasse/interfaccia hanno senso



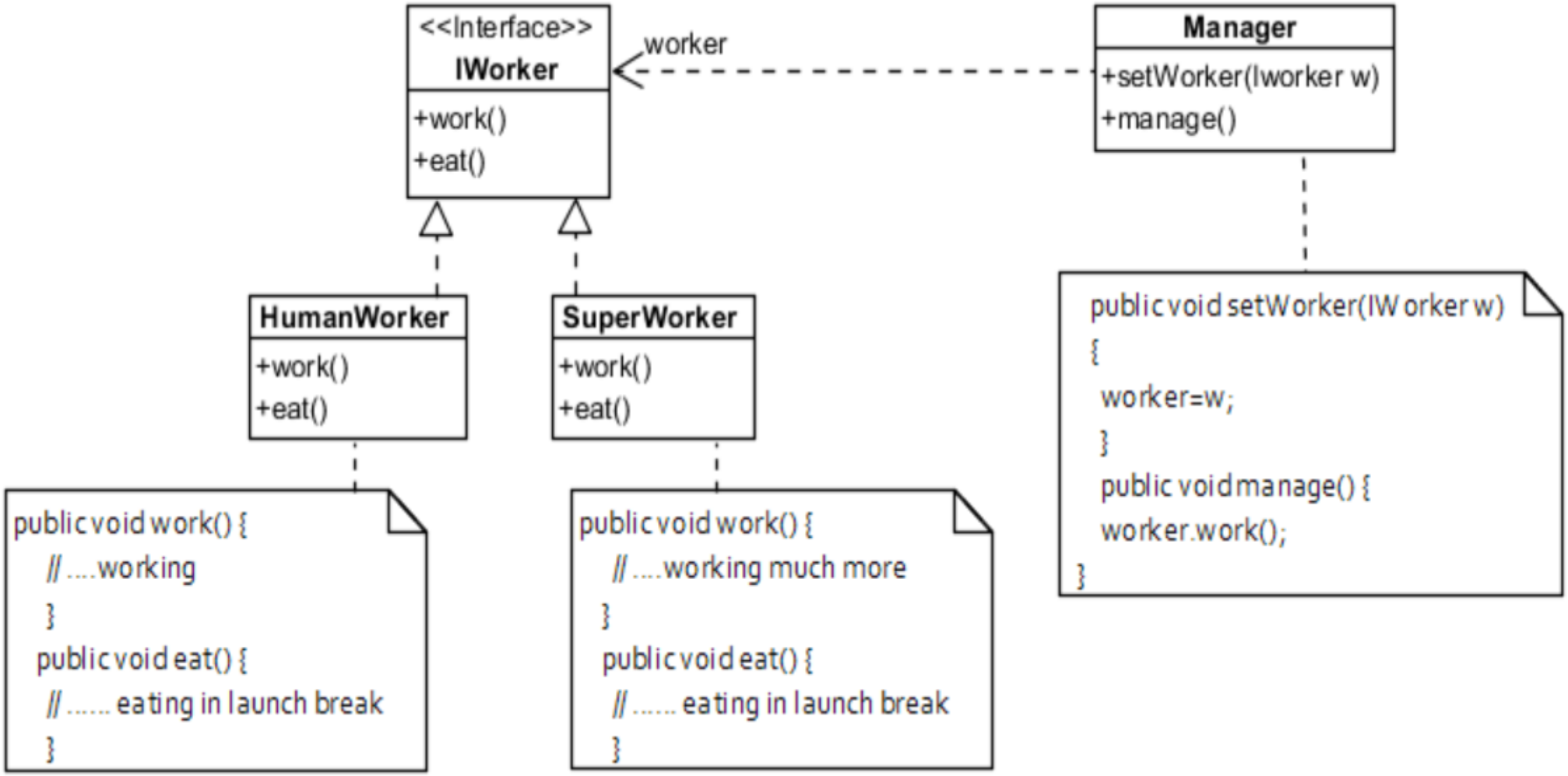

- Evitate le interface che contengono metodi non specifici (polluted or fat interfaces).

# Cominciamo esempio senza robot....

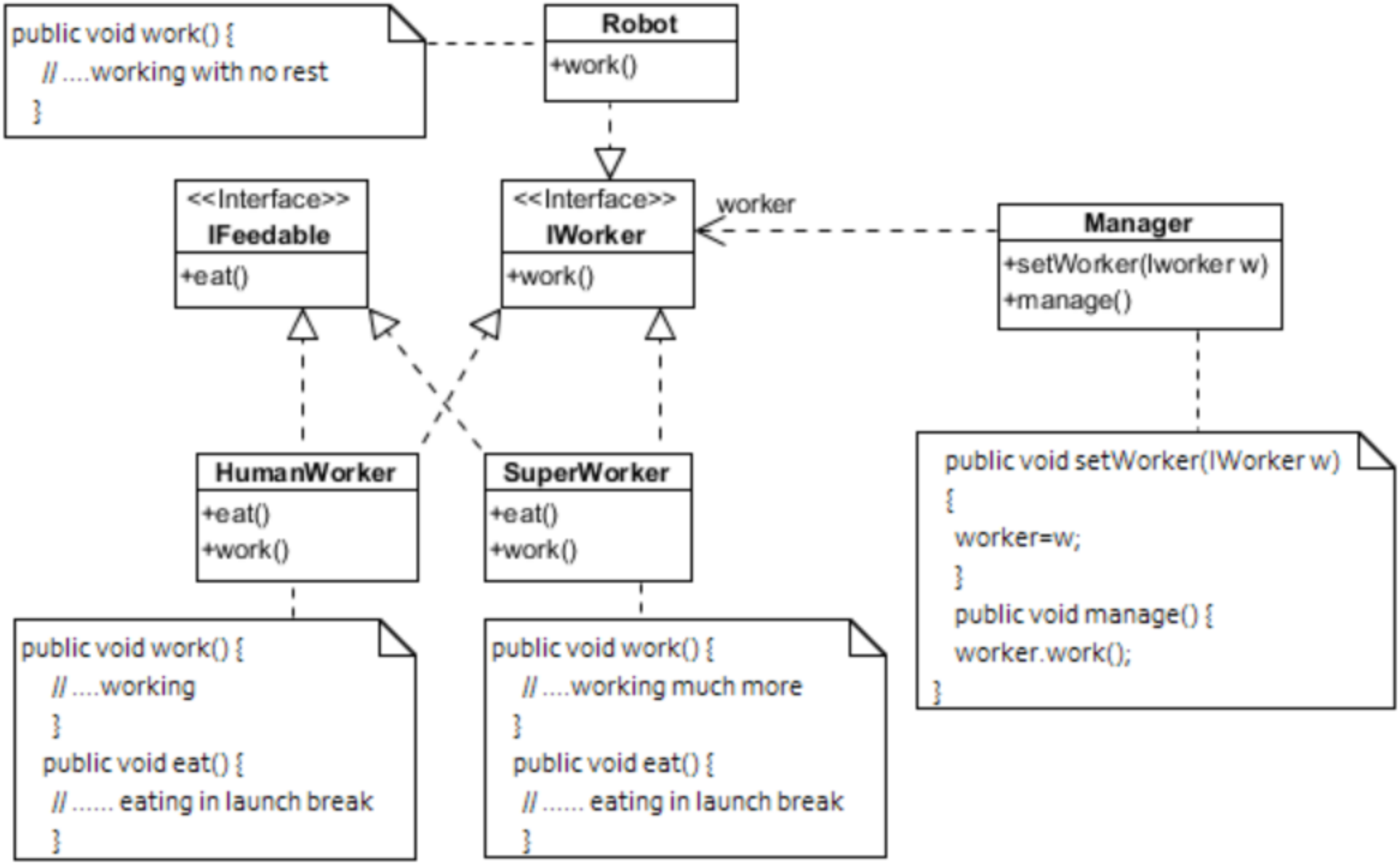
```
interface IWorker {
        public void work();
        public void eat();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
        public void eat() {
                // ..... eating in launch break
        }
}
```

```
class SuperWorker implements IWorker{
        public void work() {
                //.... working much more
        }
        public void eat() {
                //.... eating in launch break
        }
}
class Manager {
        IWorker worker;
        public void setWorker(IWorker w) {
                worker=w;
        }
        public void manage() {
                worker.work();
        }
}
```

# Esempio: diagramma delle classi



# interface segregation principle – esempio rivisitato



# interface segregation principle – esempio rivisitato, il codice

```
interface IWorkable {
        public void work();
}
interface IFeedable{
        public void eat();
}
class Worker implements IWorkable, IFeedable{
        public void work() { // ....working
        }
        public void eat() { //.... eating in launch break
        }
}
class Robot implements IWorkable{
        public void work() { // ....working
        }
}
```

```
class SuperWorker implements IWorkable,
IFeedable{
    public void work() { //.... working much more
        }

    public void eat() {//.... eating in launch break
        }
}
class Manager {
  IWorker worker;
  public void setWorker(IWorker w) {
        worker=w;
        }
  public void manage() {
        worker.work();
        }
}
```

# SOLID 5: Dependency Inversion Principle

- Principio di inversion della dipendenza

- Programma 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.

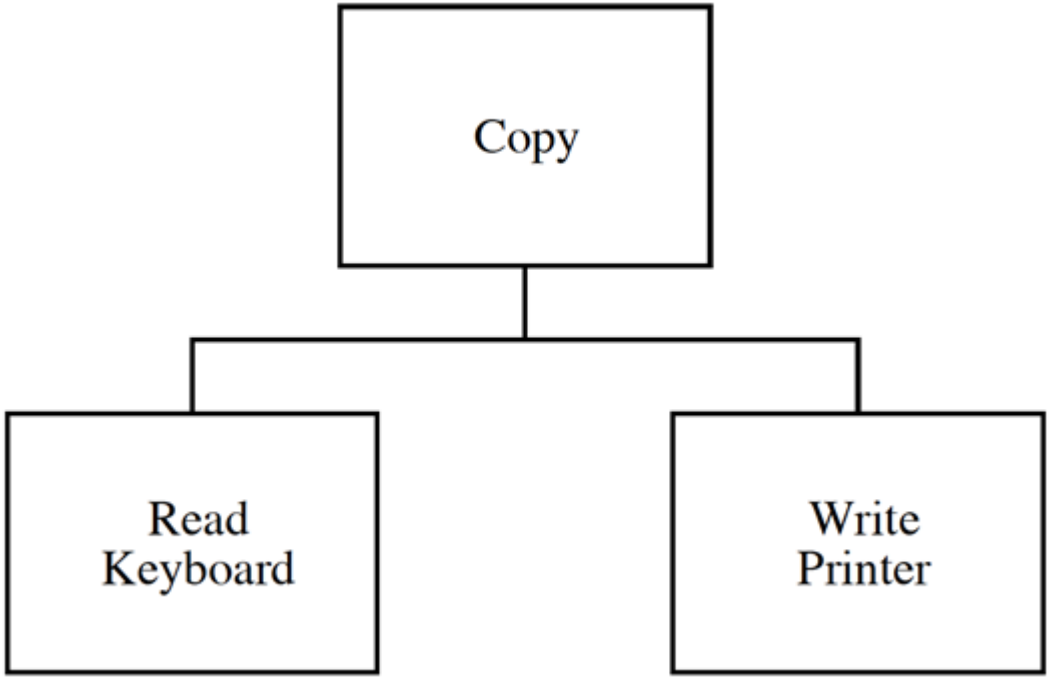

# DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

# SOLID 5: Dependency Inversion Principle

- In sostanza, non ci si deve mai basare su implementazioni concrete di alcuna classe ma solo su astrazioni.

# Un esempio: dipendenza da implementazione



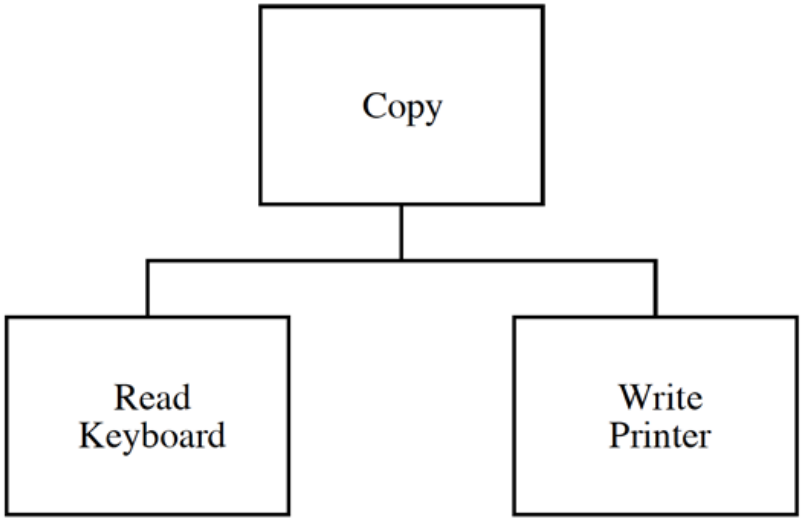

**Listing 1. The Copy Program**

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

3 moduli per leggere un carattere della tastiera e stamparlo sulla stampante. Il modulo copy ne chiama altri 2.

# Dipendenze da implementazione

Mentre i moduli read Keyboard  e Write Printer possono essere riutilizzati il metodo copy no!

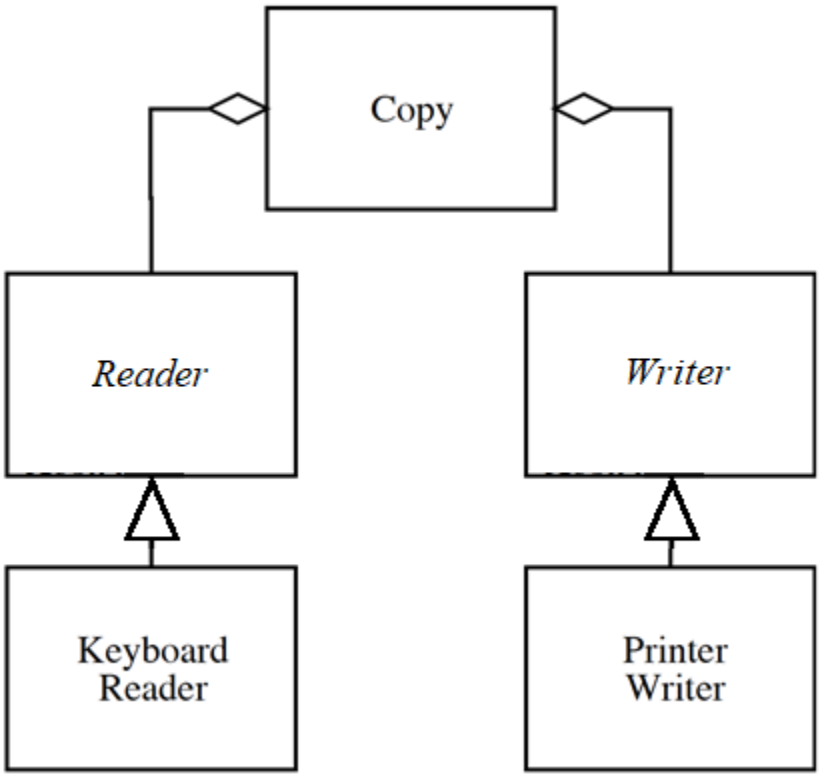



**Listing 1. The Copy Program**

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

Ad esempio se vogliamo copiare dalla tastiera e scrivere sul disco dobbiamo aggiungere un if

# Applichiamo il principio: dipendenza da astrazioni




**Listing 3: The OO Copy Program**

```
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);
}
```

# Ancora un altro esempio: in caso di errore vogliamo scrivere un messaggio

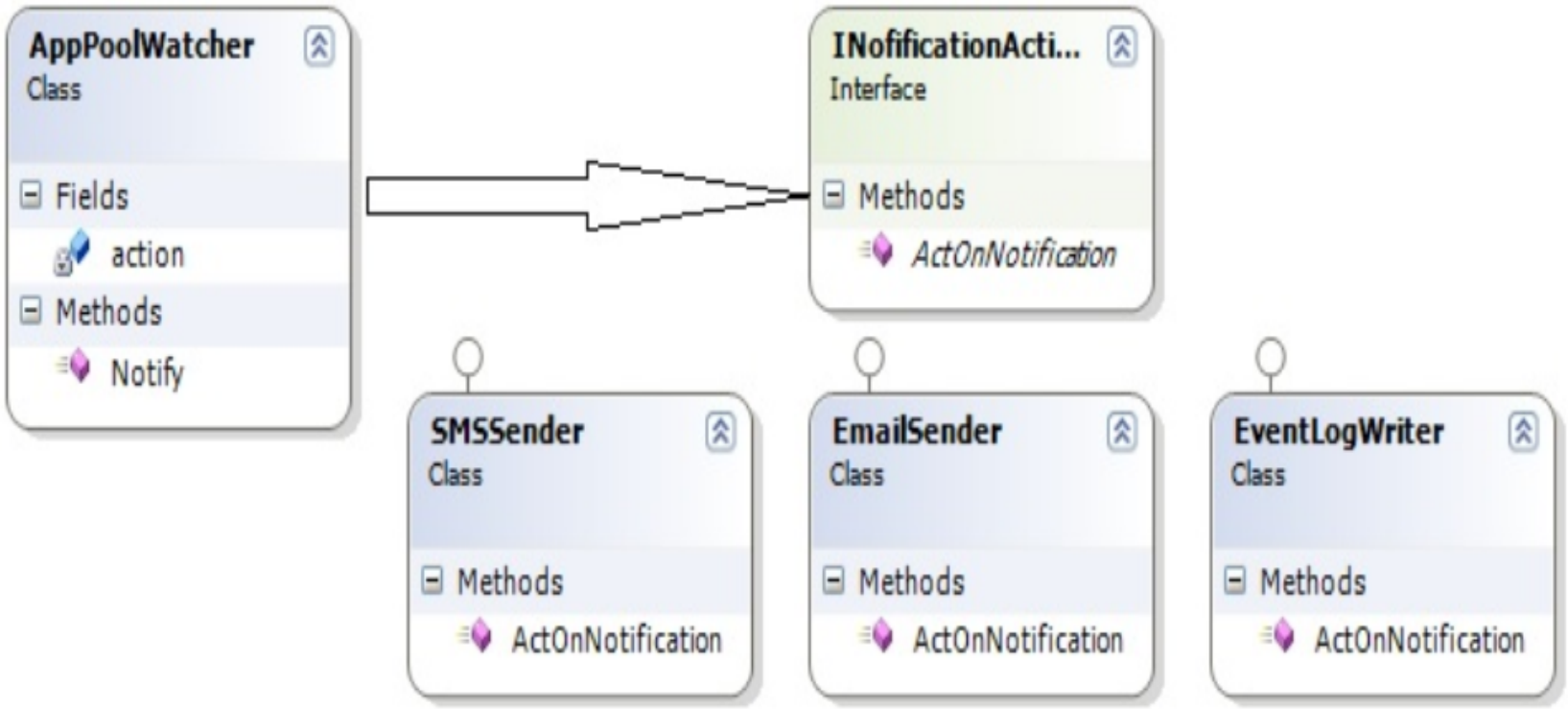
```
class EventLogWriter
{
   public void write(string message)
   {
      //Write to event log here
   }
}
```

```
class AppPoolWatcher
{ // Handle to EventLog writer to write to the logs
   EventLogWriter writer = null;

   // This function will be called when the app pool
has problem
   public void Notify(string message)
   {
      if (writer == null)
      {writer = new EventLogWriter(); }
      writer.write(message);
   }
}
```

# in caso di errore vogliamo scrivere un messaggio o mandare una mail

- ..poi viene aggiunto un nuovo requisito: mandare una mail all'amministratore di rete per qualche specifico errore
  - Come si fa?
  - Un' idea è creare una classe per inviare mail e mantenerne un riferimento in AppPoolWatcher, anche se in ogni istante verrà usato solo un oggetto, o EventLogWriter o EmailSender.
- … e poi magari viene richiesto di mandare SMS….
  - Occorre creare un'altra classe, le cui istanze sono riferite da AppPoolWatcher
- Il principio di inversione delle dipendenze dice di disaccoppiare il sistema in modo che il modulo di alto livello (AppPoolWatcher) dipenda da una astrazione. Tale astrazione sarà concretizzata dalle classi che definiscono le operazioni concrete.

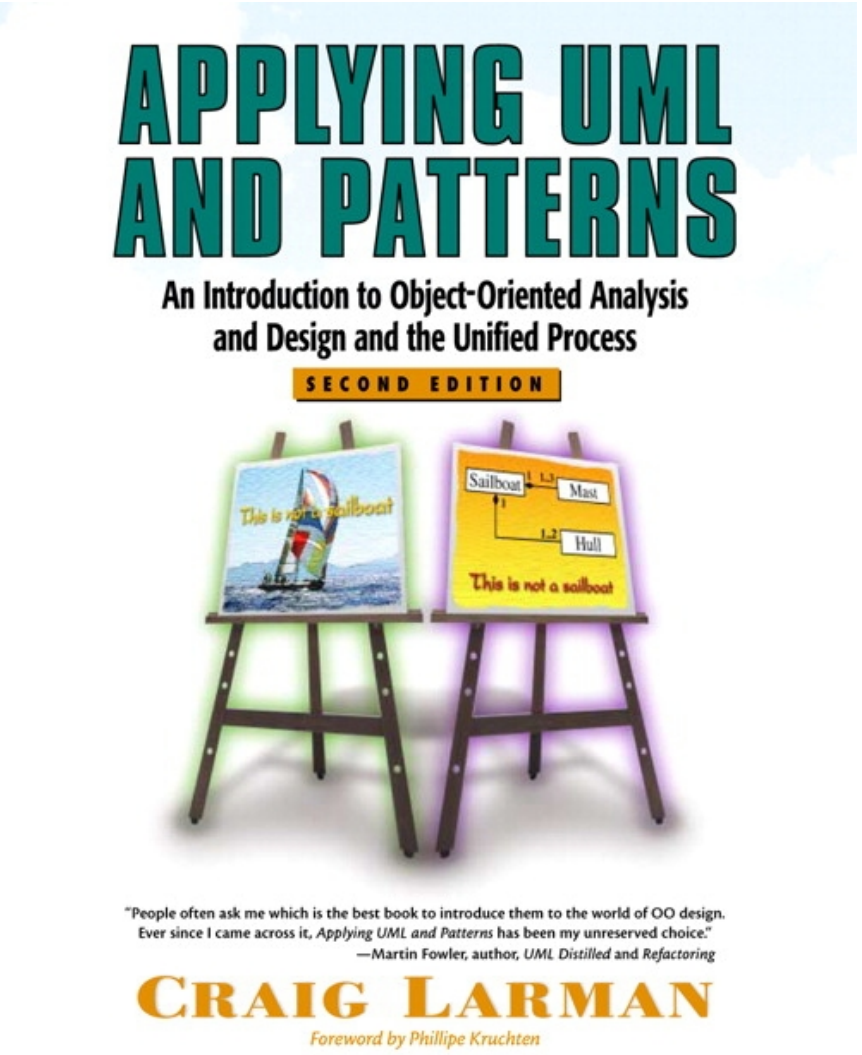
# SOLID 5, dependency Inversion Principle: solution



# GRASP

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

- Un'altra famiglia di principi di progettazione



# OO design

- 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 dalla realizzazione dei casi d'uso

- Con realizzazione del caso d'uso si intende:
  - Descrivere come un particolare caso d'uso è realizzato nel progetto, in termini di oggetti collaborativi.
  - Il lavoro di realizzazione del caso d'uso è un'attività di progettazione, il progetto cresce con ogni nuova realizzazione del caso d'uso.
  - Per realizzare i casi d'uso si usano diagrammi di interazione e pattern
  - Per realizzare i casi d'uso si **assegnano responsabilità alle classi**

# GRASP: Assegnare responsabilità

- Le responsabilità sono legate al dominio del problema
- Nel modello di progettazione, le responsabilità sono gli obblighi che un oggetto ha, definiti in termini di comportamento.
- Esistono due tipi 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.

# Responsibilità 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, ma 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

# I 9 patterns di GRASP

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

# Stili dell’architettura e qualità del software

# Qualità analizzate

Valutiamo le caratteristiche di alcuni stili architetturali in base alle seguenti caratteristiche di qualità:

- Disponibilità
  - Capacità di offrire un servizio in modo il più possibile costante
- Fault tolerance
- Modificabilità
- Performance (efficienza)
- Scalabilità

# Scalabilità

La scalabilità di un'applicazione può essere definita come la capacità di aumentare il throughput dell'applicazione in proporzione all'aumento dell'hardware utilizzato per ospitare l'applicazione.

In altre parole, se un'applicazione è in grado di gestire 100 utenti con su un singolo nodo hardware, l'applicazione dovrebbe essere in grado di gestire 200 utenti quando il numero di nodi hw è raddoppiato.

# Scale up, scalabilità verticale

La scalabilità verticale si riferisce all'aggiunta di memoria e CPU a una singolo nodo.

La scalabilità verticale è particolarmente utile con i database. I database hanno le seguenti necessità:

- Grande spazio di memoria condivisa,
- Molti thread dipendenti,
- Alto grato di interconnessione interna

# Scale out, scalabilità orizzontale

Con scalabilità orizzontale si intende l'aggiunta di più nodi hw. La scalabilità orizzontale è ideale per applicazioni Web, che presentano alcune delle seguenti caratteristiche:

- Poco spazio di memoria condiviso
- Molti thread indipendenti
- Bassa interconnessione

Possibile anche con sistemi operativi diversi

# Client-server, 2 o N-tier

| | |
|---|---|
| Disponibilità | I server di ogni tier(ordine, fila) possono essere replicati, quindi se uno fallisce c'è 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 di ricambio" (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 |

# Coordinatore di processi

| | |
|---|---|
| Disponibilità | Il coordinatore è un punto critico: deve essere replicato se si vuole garantire disponibilità. |
| Fault tolerance | Occorre specificare compensazione: cosa fare se un server fallisce  Se fallisce il coordinatore: occorre ridirigere su un coordinatore replica. |
| Modificabilità | Posso modificare liberamente i servers purché non cambino le funzionalità esportate. |
| Performance (efficienza) | Il coordinatore deve essere in grado di servire più richieste concorrenti. La performance del processo è limitata dal server più lento. Se non tutti i server sono necessari, si usa un time-out. |
| Scalabilità | Replicando il coordinatore. Scala sia up che out. |

# Architetture basate su comunicazione asincrona

| | |
|---|---|
| Disponibilità | Basta replicare le code |
| Fault tolerance | Se una coda fallisce, si cerca una replica |
| Modificabilità | Unico vincolo il formato dei messaggi |
| Performance (efficienza) | Si possono spedire migliaia di messaggi al secondo. Compromesso tra affidabilità/sicurezza e performance |
| Scalabilità | Le code possono essere ospitate presso origine e destinazione della comunicazione, ma anche da clusters grandi a piacere di servers. |

# Riferimenti e approfondimenti

- Principles Of OOD, Robert C. Martin
  - http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- Applying UML and Patterns, Craig Larman