# Progettazione del sw

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

# Progettare prima di produrre

- "The architect's two most important tools are: the eraser in the drafting room and the wrecking bar on the site" [Frank Lloyd Wright]
  - Anche nel software: nonostante l'apparente omogeneità dei materiali tra progetto e realizzazione: il codice è più "duro" dei modelli.
- Tipico della produzione industriale
- Alcune motivazioni
  - Complessità del prodotto
  - Organizzazione e riduzione delle responsabilità
  - Controllo preventivo della qualità

# Obiettivi della progettazione

- Trasformare la complessità globale del sistema
  - Ridurre le difficoltà di comprensione e realizzazione
  - Organizzare il sistema in sottosistemi di minore complessità
  - Identificare i sottosistemi riusabili
- Soddisfare i requisiti di qualità del produttore
  - il modello progettuale deve essere una guida leggibile e comprensibile per chi genera il codice, per chi lo verifica e per chi dovrà mantenerlo e fornire supporto agli utenti, una volta che il software sarà operativo
  - il modello progettuale deve fornire un'immagine completa del software, che copra il dominio dei dati, delle funzioni e del comportamento da un punto di vista realizzativo

# Obiettivi della progettazione (cont'd)

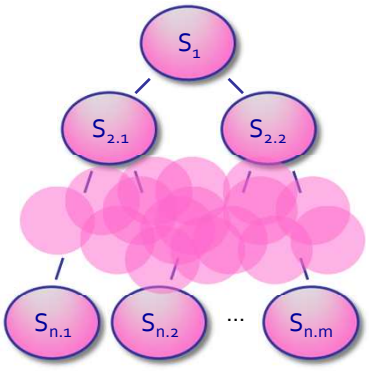
- Produrre la documentazione necessaria
  - Per far procedere la codifica senza ulteriori informazioni
  - Per tracciare i requisiti nelle unità
  - Per definire le configurazioni del sistema
- Associare componenti fisici alle unità
  - Per organizzare il lavoro di codifica
- Definire gli strumenti per le prove delle unità
  - Casi di prova e componenti fittizi per le prove e l'integrazione

## Progettazione di alto livello e di dettaglio

- Progettazione di alto livello (o architetturale )
  - identificazione e specifica delle parti del sistema e delle loro inter-connessioni
- Progettazione di dettaglio
  - decisione su come la specifica di ogni parte sarà realizzata

## Progettazione di alto livello e di dettaglio, approcci

- Top-down
  - decomposizione di problemi
- Bottom-up
  - composizione di soluzioni
- Sandwich
  - approccio "naturale"



## Architettura software, def.

- L'architettura di un sistema software (in breve architettura software) è la struttura del sistema, costituita dalle parti del sistema, dalle relazioni tra le parti e dalle loro proprietà visibili.

## Progetto di dettaglio

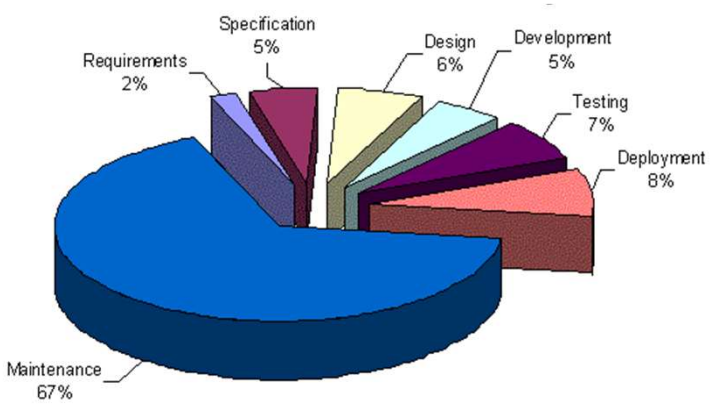
- Definizione di unità di realizzazione (sottosistema terminale)
  - Un sistema che non deve essere ulteriormente trasformato
  - Quando un'altra trasformazione avrebbe un costo ingiustificato o porterebbe a un'inutile esposizione di dettagli
  - Un sottosistema definito (ad esempio, un componente)
  - Un insieme di funzionalità affini (ad esempio, un package)
- Descrizione delle unità di realizzazione
  - Da zero o per specializzazione di sistemi esistenti
  - Definizione delle caratteristiche "interessanti"
    - stato dei/interazione tra sistemi

## Progettare Software

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

## I costi del software (dalla lezione 1)

Il prodotto software durante la sua vita diverse fasi:
analisi, specifica, progettazione, codifica, testing, manutenzione e ritiro



# 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

## Progettazione e riprogettazione

- Gli interventi di manutenzione (in particolare migliorativa) spesso richiedono più di una patch, a volte il sistema ha bisogno di una «rinnovata», si parla allora di:
  - Refactoring
  - Riprogettazione
  - Reingegnerizzazione

## Refactoring (ristrutturazione)

- Refactoring è il processo di cambiare un sistema software senza alterarne il comportamento esterno ma migliorandone la struttura interna" [Fowler]
- Si cercano
  - ridondanze
  - elementi inutilizzati
  - algoritmi inefficienti
  - strutture dati inappropriate o deboli

## Reingegnerizzazione e riprogettazione

- Riprogettazione
  - In caso di interventi, anche radicali, sia sulla struttura del sistema sia sul codice
  - Quando i requisiti sono definiti e sostanzialmente stabili e la «rinnovata» parte dalla (ri)progettazione
- Reingegnerizzazione
  - In seguito a successivi e massicci interventi di manutenzione
  - In seguito a un cambio di piattaforma o di tecnologia
  - Quando devono essere aggiunte o modificate funzionalità e ciò richiede non solo interventi sul codice ma anche sui requisiti

## Principi e pattern di progettazione (OO)

Information Hiding
  Controllo delle interfacce
Astrazione
  Dati e controllo
Coesione
Disaccoppiamento

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

# Principi generali di buona progettazione

Principi generali da seguire, sia in fase di progettazione architetturale che in fase di progettazione di dettaglio

Information hiding
Astrazione sul controllo e sui dati
Coesione
Disaccoppiamento

## Information hiding

- Separazione tra interfaccia e implementazione
- Componenti o moduli come scatole nere
  - Fornitori e clienti di funzionalità (relazione d'uso)
  - È nota solo l'interfaccia (dichiarazione dei servizi)
- Sono mantenuti nascosti
  - Algoritmi usati
  - Strutture dati interne

## 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 atre unità

# 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.

# Information hiding: tecniche note

- Tecniche a voi già note da corsi precedenti per information hiding:
  - Setters e getters
  - Librerie
  - Strutture Dati Astratte

## 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, si potrebbe non voler permettere un accesso.

## 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)

## Astrazione sui dati

- 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)

## Coesione

- Proprietà di un (sotto)sistema
  - grado in cui un sottosistema realizza "uno e un solo concetto"
  - funzionalità "vicine" devono stare nello stesso sottosistema
  - vicinanza per tipo, algoritmi, dati in ingresso e in uscita
- L'obbiettivo del progettista è creare sottosistemi coesi, in cui tutti gli elementi di un sottosistema siano strettamente collegati tra loro.

## (Dis)accoppiamento

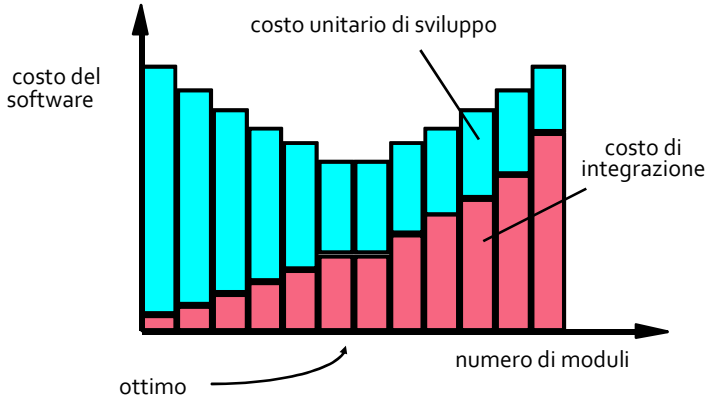
- Proprietà di un insieme di (sotto)sistemi
  - grado in cui un sottosistema è "legato" ad altri sottosistemi
  - Dipendenze, scambio di messaggi…
- L'obbiettivo del progettista è creare sottosistemi non strettamente legati tra loro.
- Aka uncoupling, aka coupling

## Coesione vs disaccoppiamento

- Si hanno vantaggi con (sotto)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.

## Coesione vs disaccoppiamento e dimensione dei sottosistemi

- Qual è il giusto numero di moduli?



# 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.


## SOLID 1: Single Responsibility Principle

- Una classe (un modulo, un metodo) dovrebbe avere solo un motivo per cambiare
  - In questo contesto, 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. Ogni classe gestirà solo una responsabilità e in futuro, se dobbiamo fare un cambiamento, lo facciamo nella classe che realizza la funzionalità. Quando dobbiamo fare un cambiamento in una classe che ha più responsabilità, le modifiche potrebbero influenzare altre funzionalità della classe e tutti i moduli che le usano devono essere ridestati
  - 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".

# Una classe dovrebbe avere un solo motivo per cambiare

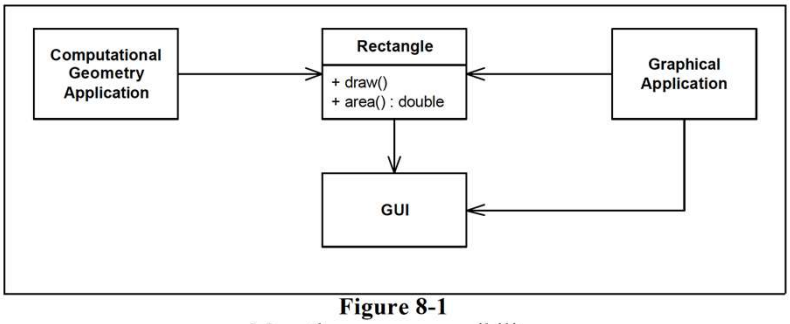



**Figure 8-1**
More than one responsibility

# Separare le responsabilita'

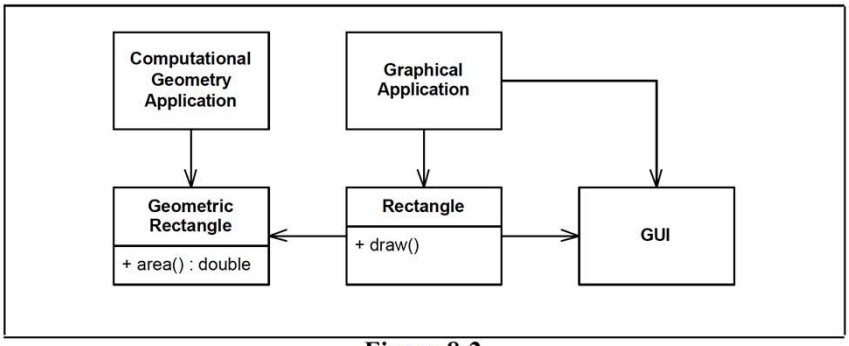



**Figure 8-2**
Separated Responsibilities

# Attenzione a trovare la giusta misura

**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 responsabilita' :
  - metodi per la connessione
  - metodi per il trasferimento dati

Queste due responsabilita' dovrebbero essere separate?

# Dipende da come puo' cambiare l'applicazione

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

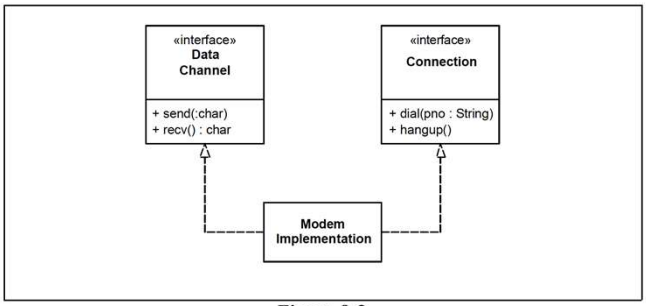



**Figure 8-3**
Separated Modem Interface

- se d'altra parte l'applicazione non puo' cambiare in modo da richiedere il cambiamento delle due diverse responsabilita' in momenti diversi allora separarle introdurrebbe una complessita' non necessaria.

- Riassumendo:

Un motivo di cambiamento e' tale solo se e' una reale possibilita' 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 gia' funziona
  - Disegnare classi in modo che sia possibile estenderle ma senza cambiarle. Questo e' possibile mediante l'uso di classi astratte e classi concrete che le implementano.

# Uso della classe concreta server

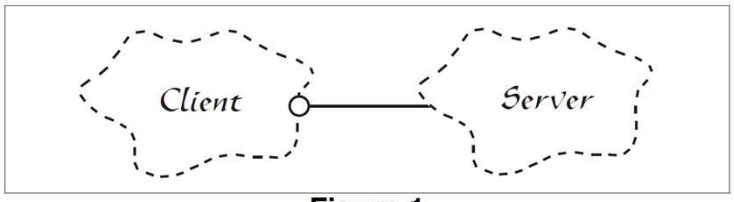



**Figure 1**
Closed Client

# Uso della classe astratta server

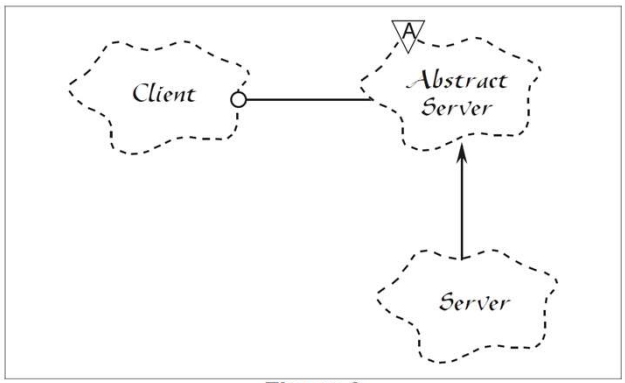
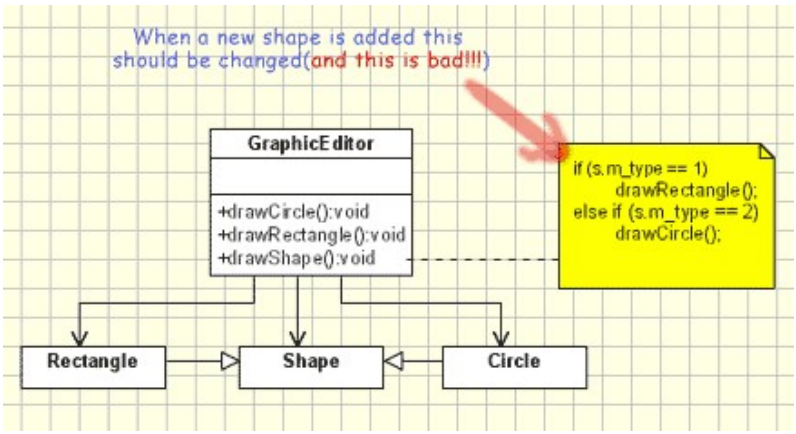



**Figure 2**
Open Client

# SOLID 2: Open Closed Principle: ad ex.



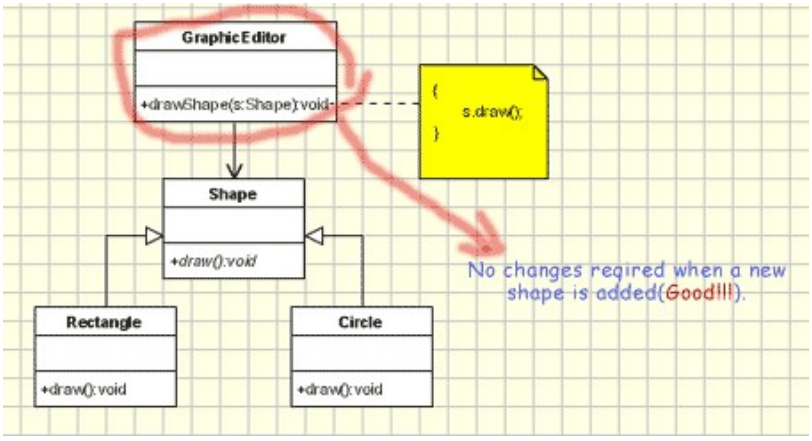

# SOLID 2: Open Closed Principle.
## Esempio di non applicazione del principio

```
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

- Usate astrazioni
  - Interfacce
  - classi astratte

# 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
        }
}
```

## L'importanza dell'esperienza

- In genere la chiusura non puo' essere completa ma deve essere strategica.
- Il progettista deve scegliere i tipi di cambiamenti rispetto ai quali "chiudere" il suo progetto
- Questa scelta richiede esperienza nel capire quali siano i tipi di cambiamenti piu' probabili per quell'applicazione e quell'industria

## Alcune "dritte"

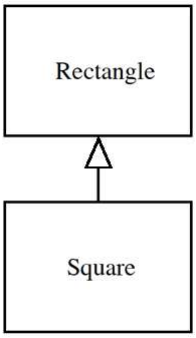
- nelle classi usare variabili private
- non utilizzare mai variabili globali

## 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

```
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;
};
```

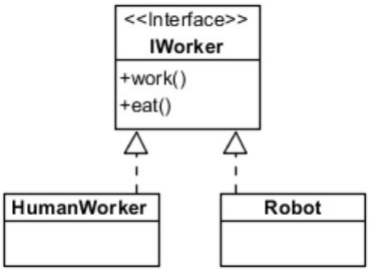



# SOLID 4: Interface Segregation Principle

- 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



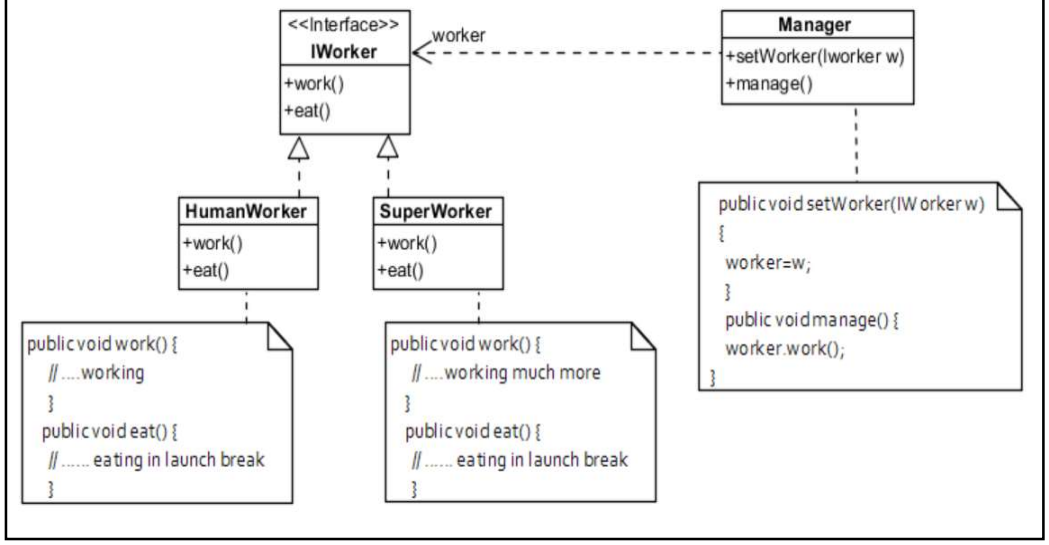

- Evitate le interface che contengono metodi non specifici (polluted or fat interfaces).

# interface segregation principle – anche senza i Robot...

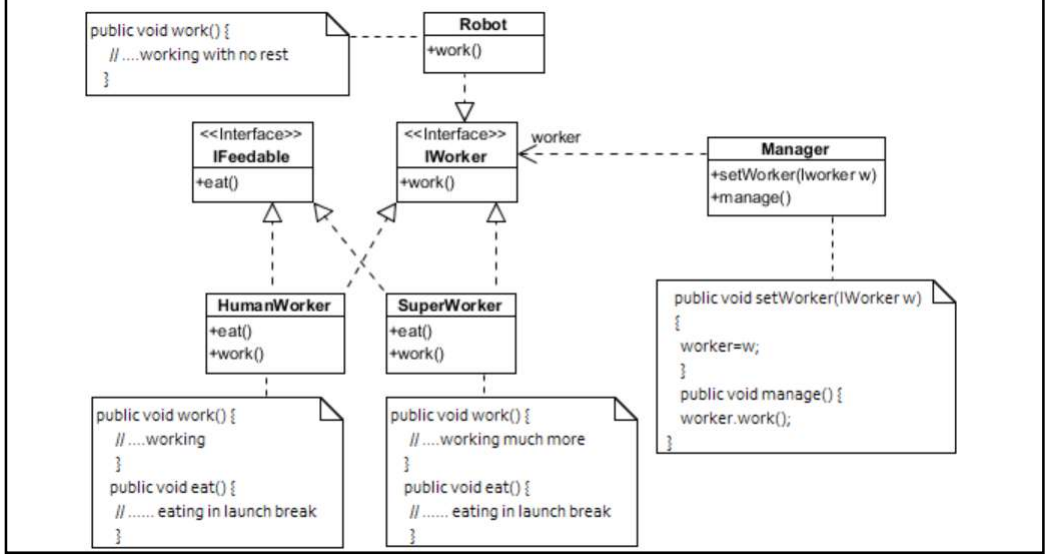
```
interface IWorker {
        public void work();
        public void eat();
}

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



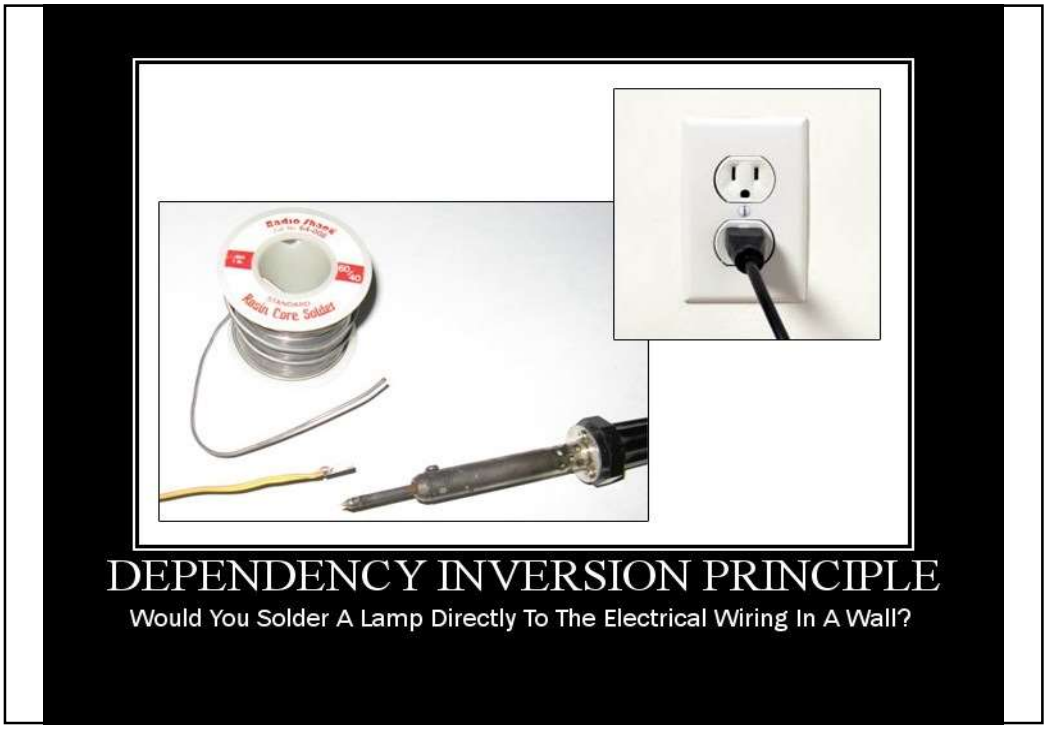
# interface segregation principle – esempio rivisitato, il codice

```
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

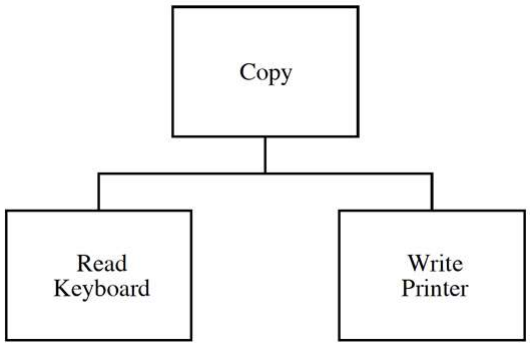
- Principio di inversion della dipendenza
- Programma per l'interfaccia, non per l'implementazione.
  - I moduli di alto livello non devono dipendere da quelli di basso livello.
    - Entrambi devono dipendere da astrazioni;
  - Le astrazioni non devono dipendere dai dettagli;
    - sono i dettagli che dipendono dalle astrazioni.




## SOLID 5: Dependency Inversion Principle

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

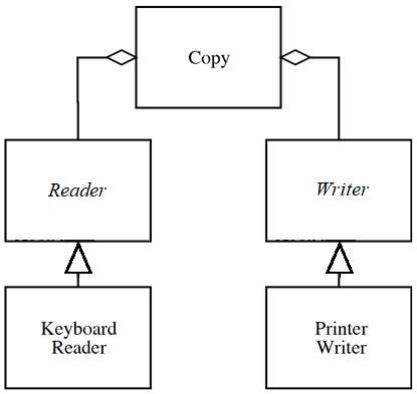
# Un esempio: dipendenza da implementazione




**Listing 1. The Copy Program**

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

# Un esempio: 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);
}
```

# SOLID 5: Dependency Inversion Principle: esempio negativo

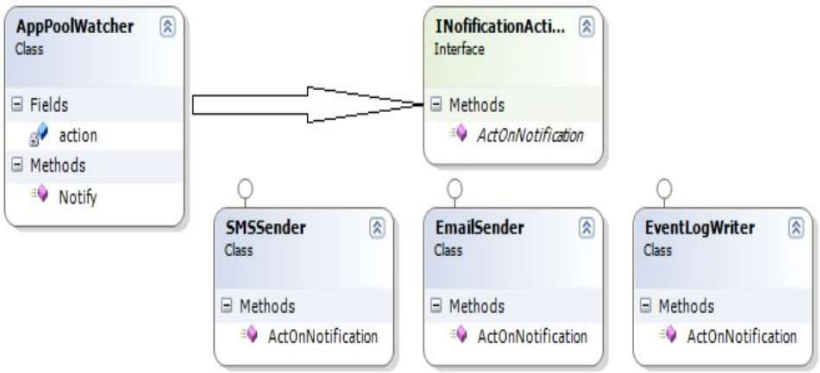
```
class EventLogWriter
{
   public void write(string message)
   {
      //Write to event log here
   }
}
```

```
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);
   }
}
```

# SOLID 5, dependency Inversion Principle: problems with the example

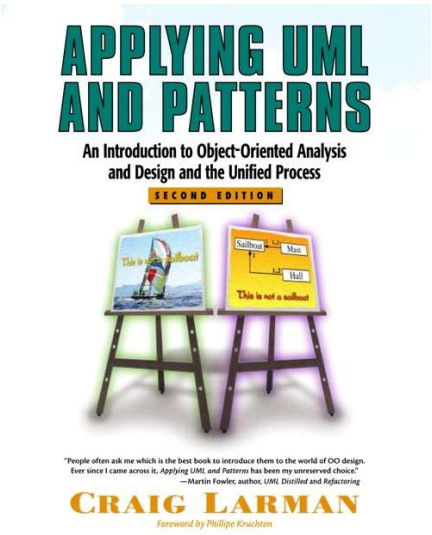
- ..poi viene aggiunto un nuovo requisito: mandare una mail all'amministratore di rete per qualche specifico errore
  - Come si fa?
  - Un' idea è creare una classe per inviare mail e mantenerne un riferimento in AppPoolWatcher, anche se in ogni istante verrà usato solo un oggetto, o EventLogWriter o 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

# Riferimenti e approfondimenti

- Sezione 3.4, pp 63-65 Fuggetta &C.
- Principles Of OOD, Robert C. Martin
  - http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- Applying UML and Patterns, Craig Larman