UNIVERSITÀ PARTHENOPE
DiST
Dipartimento di Scienze e Tecnologie

Programmazione 3
e
Laboratorio di Programmazione 3

Principi di OOD

Angelo Ciaramella

Introduzione

- I principi del Software Design (SD) rappresentano una linea guida
  - evitare di avere una cattiva progettazione
- I principi di SD sono stati introdotti da Robert Martin
  - *Agile Software Development: Principles, Patterns and Practices*

Sviluppo Agile

- Manifesto per lo Sviluppo Agile di Software

Stiamo scoprendo modi migliori di creare software, sviluppandolo e aiutando gli altri a fare lo stesso.
Grazie a questa attività siamo arrivati a considerare importanti:

Gli individui e le interazioni più che i processi e gli strumenti
Il software funzionante più che la documentazione esaustiva
La collaborazione col cliente più che la negoziazione dei contratti
Rispondere al cambiamento più che seguire un piano

Ovvero, fermo restando il valore delle voci a destra, consideriamo più importanti le voci a sinistra.

Caratteristiche da evitare

- Le caratteristiche di una cattiva progettazione da evitare
  - Rigidità
    - il sistema è difficile da cambiare poiché ogni cambiamento si ripercuote su molte parti del sistema
  - Fragilità
    - un cambiamento di una parte del sistema comporta la *rottura* di una parte inaspettata
  - Immobilità
    - è difficile riutilizzare qualche componente in un'altra applicazione perché non può essere estrapolata dall'applicazione corrente

Caratteristiche da evitare

- Viscosità
  - L'ambiente di sviluppo è tenuto insieme in modo inappropriato
- Complessità inutile
  - Strutture di codice non necessarie attualmente ma probabilmente in futuro
- Ripetizioni inutili
  - Inutili Cut and Paste

Caratteristiche da evitare

- Alcuni di questi problemi derivano da dipendenze mal gestite
  - Il codice è visto come un groviglio
    - "spaghetti code"

- I linguaggio OO permettono di gestire le dipendenze
  - Interfacce
    - Rompono o invertono la direzione di certe dipendenze
  - Polimorfismo
    - Invocano funzioni senza dipendere dai moduli che le contengono

Refactoring

- tecnica strutturata per modificare la struttura interna di porzioni di codice senza modificarne il comportamento esterno

- miglioramento
  - leggibilità
  - manutenibilità
  - riusabilità
  - estendibilità
  - riduzione della sua complessità

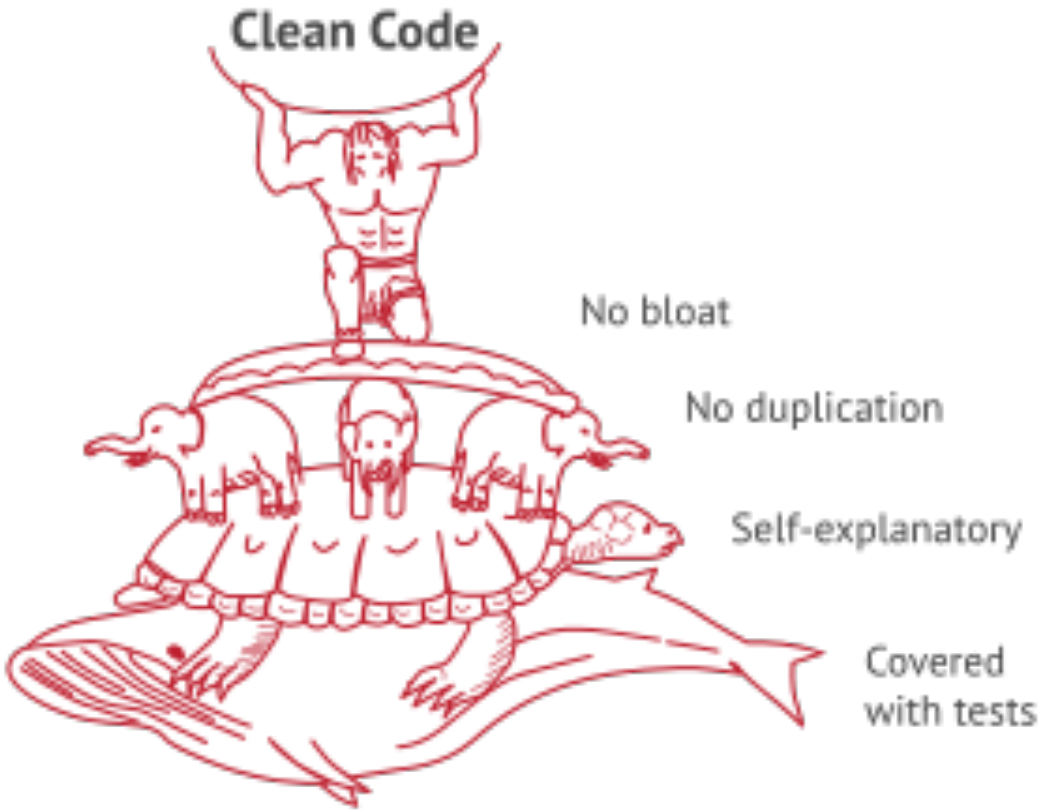
- introduzione a posteriori di design pattern

Refactoring

- elemento importante delle principali metodologie emergenti di sviluppo del software
  - metodologie agili
  - extreme programming
  - test driven development

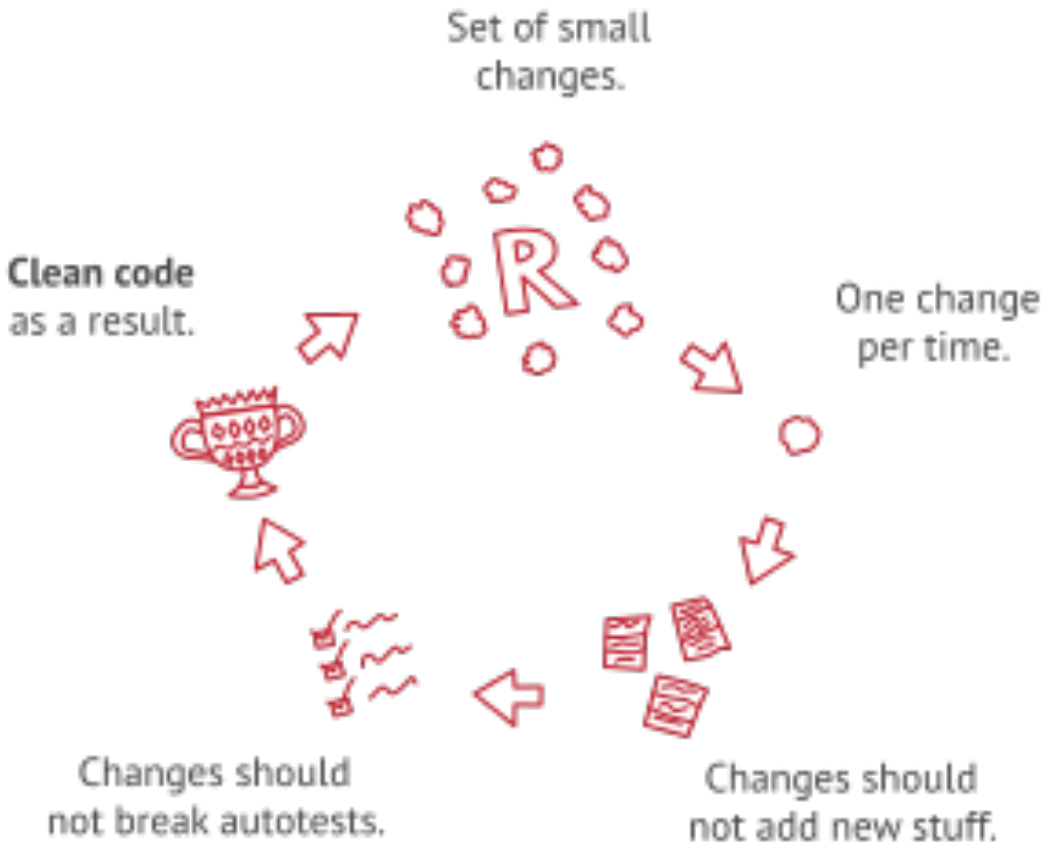
- Molti IDE forniscono supporto al refactoring del codice
  - IntelliJ IDEA
  - Eclipse
  - NetBeans

Clean Code



Clean Code

Refactoring



Fasi di Refactoring

Refactoring

- Quando
  - Rule of three
    - fai qualche cosa di simile la terza volta
  - Aggiunta di una caratteristica
  - Quando viene fissato un Bug
  - Review del codice

SOLID

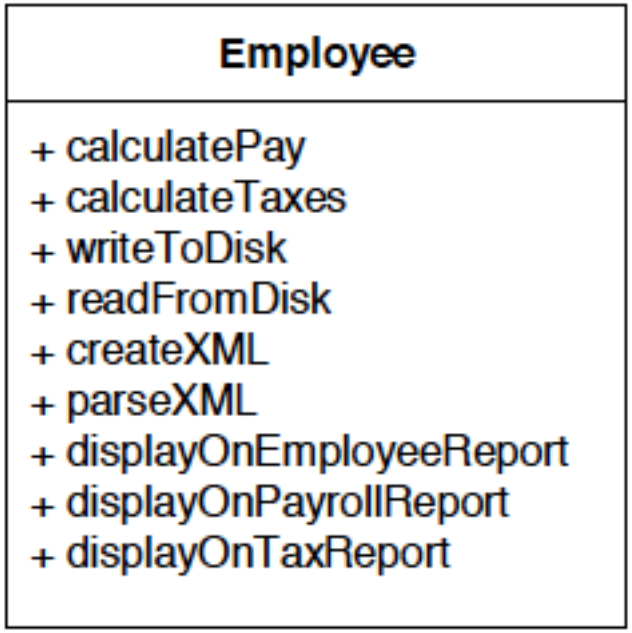
- SOLID
  - acrostico che si riferisce ai primi 5 principi dello sviluppo del software (Michael Feathers)
    - Single Responsability Principle
    - Open-Closed Principle
    - Liskov Substitution
    - Interface Segregation
    - Dependency Inversione
  - Linee guida per lo sviluppo di software estendibile e manutenibile

Single Responsability Principle (SRP)

Una classe dovrebbe avere un solo motivo per cambiare

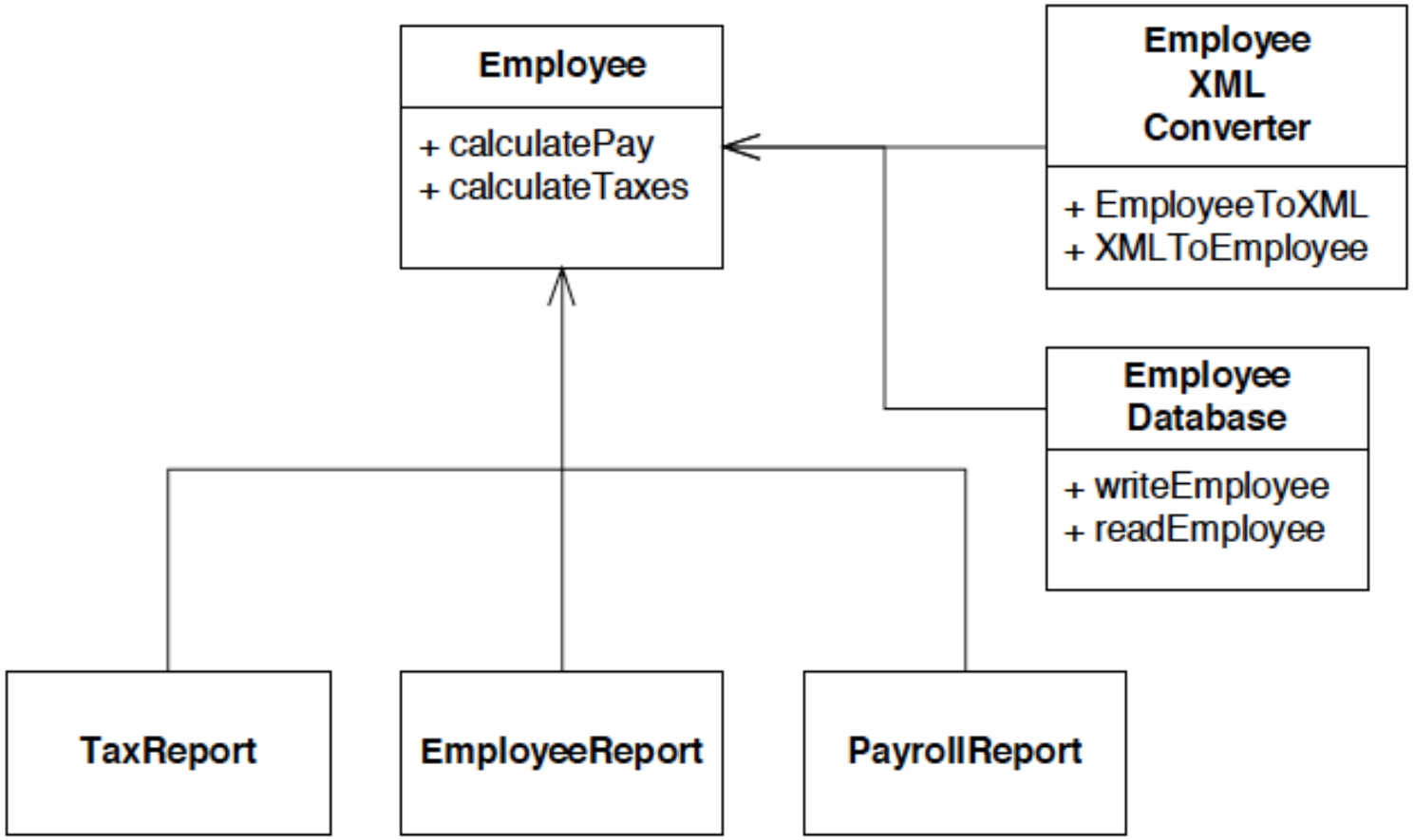
- Principio semplice e intuitivo
  - ogni elemento di un programma (classe, metodo, variabile) deve avere una sola responsabilità tale responsabilità debba essere interamente incapsulata dall'elemento stesso
  - introdotto come principio di coesione da Tom De Marco (Structured Analysis and Systems Specification, 1979)

Esempio SRP



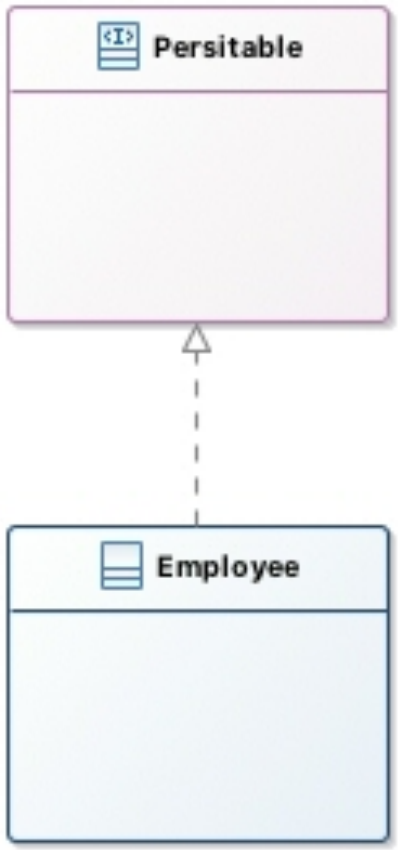
Esempio di "fragilità" – Un cambiamento da Access a Oracle prevede il cambiamento di Employee

Esempio SRP



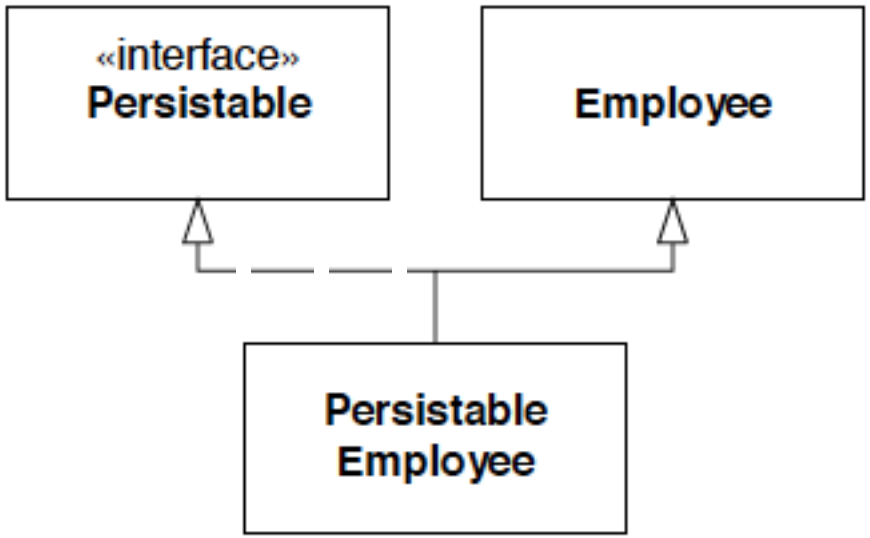
Possibile separazione di responsabilità

Esempio SRP



Esempio di "fragilità" – Un cambiamento a `Persitable` si ripercuote su tutti gli `Employee`

Esempio SRP



Possibile separazione di responsabilità

Esempio SRP

```
// single responsibility principle

interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(String content);
}

class Email implements IEmail {
        public void setSender(String sender) {
// set sender; }
        public void setReceiver(String receiver) {
// set receiver; }
        public void setContent(String content) {
// set content; }
}
```

1. L'aggiunta di un nuovo protocollo creerà la necessità di aggiungere codice per il *parsing* e la *serializzazione* del contenuto per ogni tipo di campo
2. L'aggiunta di un nuovo tipo di contenuto (come HTML) ci fa aggiungere codice per ogni protocollo implementato

Esempio SRP

```
// single responsibility principle
interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(IContent content);
}

interface IContent {
        public String getAsString();}

class Content implements IContent {
        public String getAsString() {...}
}

class Email implements IEmail {
        public void setSender(String sender) {// set sender; }
        public void setReceiver(String receiver) {// set
receiver; }
        public void setContent(IContent content) {// set
content; }
}
```

1. Cambia solo la classe `Email`
2. Cambia solo `Content`

Open Closed Principle (OCP)

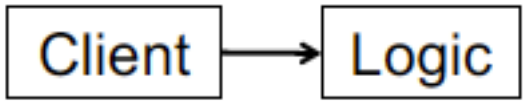
> **Le entità software come classi, moduli e funzioni dovrebbero essere aperte per l'estensione, ma chiuse per le modifiche**

- Principio molto generale
  - Quando scriviamo una classe dobbiamo garantire che per una sua estensione non c'è bisogno di cambiare la classe stessa
  - il design e la scrittura del codice dovrebbero essere fatti in modo che la nuova funzionalità dovrebbe essere aggiunta con cambiamenti minimi nel codice esistente
  - Può essere applicato a moduli, package e librerie

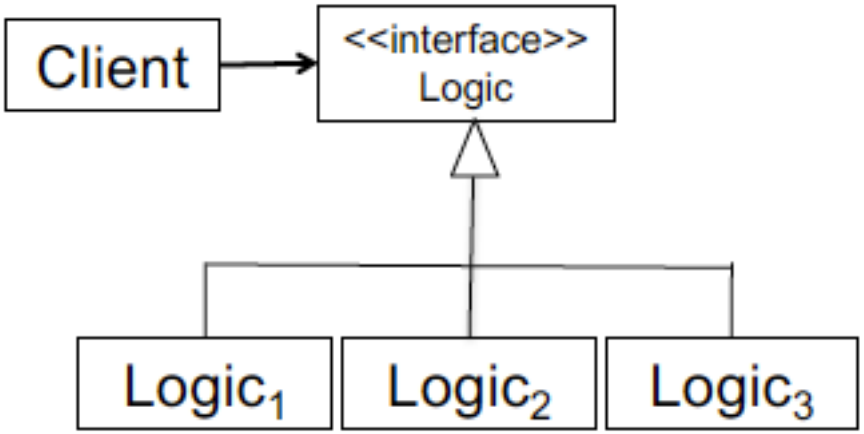
Open Closed Principle (OCP)

- OCP
  - Può essere assicurato usando classi astratte e classi concrete per implementare il loro comportamento
    - Concrete Classes che estendono Abstract Classes
- Casi particolari di OCP
  - Template pattern
  - Strategy pattern

Esempio OCP

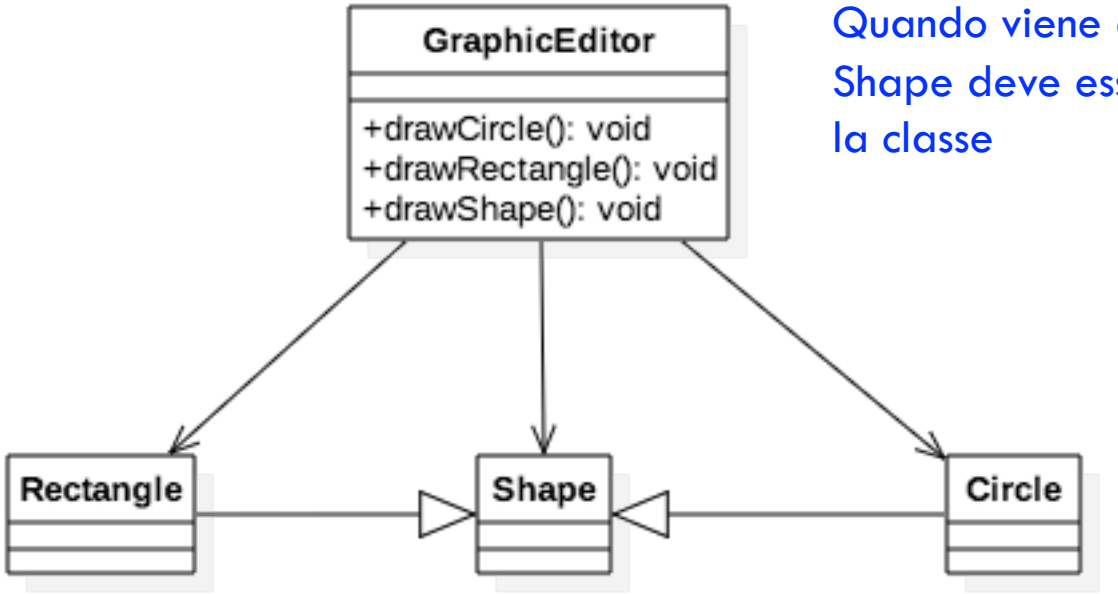


Questa relazione viola il principio Open-Closed, rende Client direttamente dipendente da Logic



Questa relazione non viola il principio Open-Closed

Esempio OCP



Quando viene aggiunto un nuovo Shape deve essere modificata la classe

Esempio di violazione dell'OCP

Esempio OCP

```
// Open-Close Principle
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) {....}
 }
```

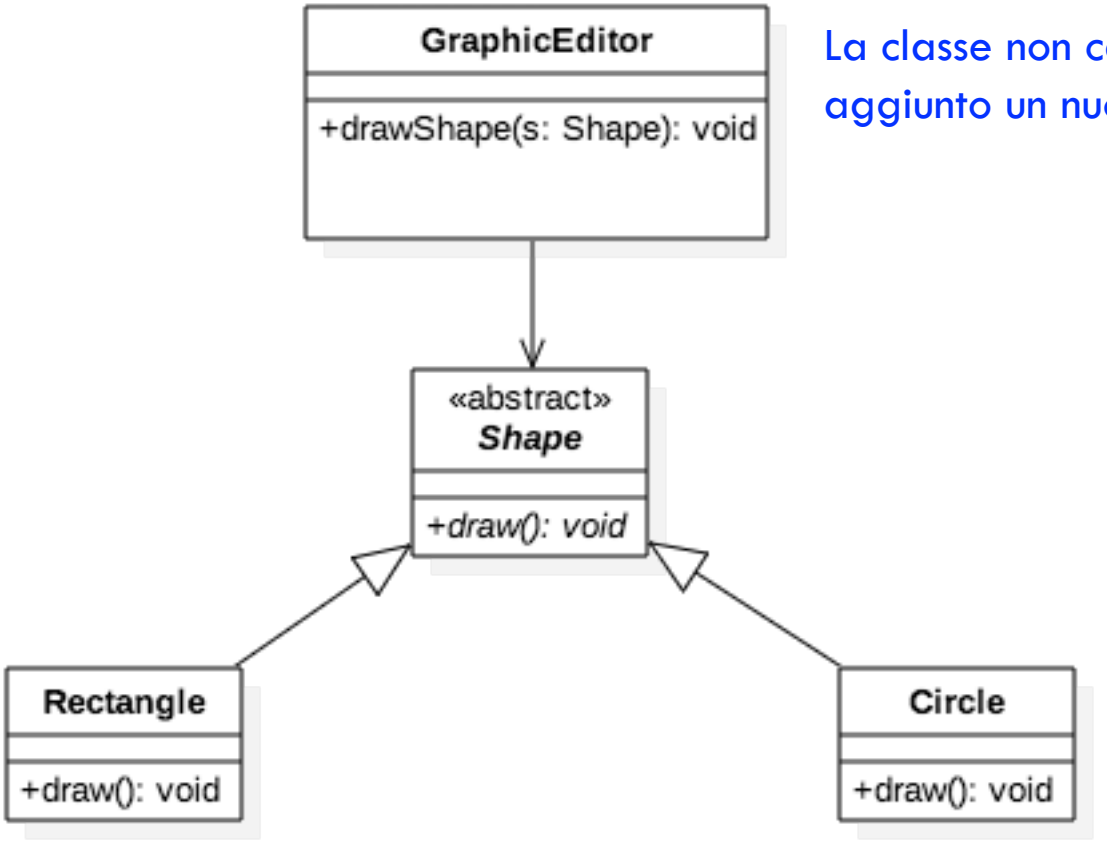
```
class Shape {
        int m_type;
 }
```


Esempio OCP

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

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


Esempio OCP



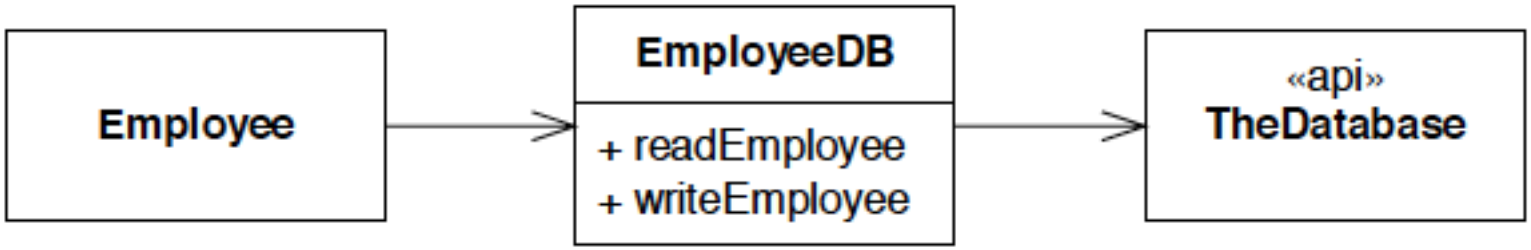
La classe non cambia quando viene aggiunto un nuovo Shape

Possibile soluzione che soddisfa l'OCP

Esempio OCP

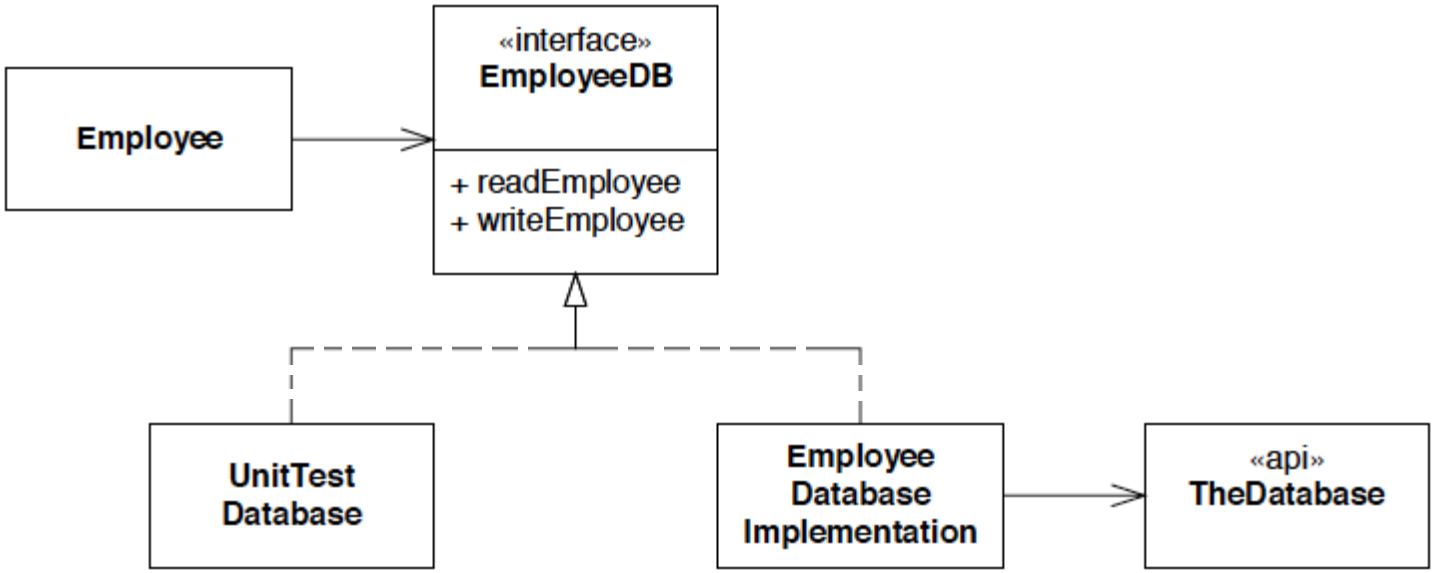
```
// Open-Close Principle
class GraphicEditor {
        public void drawShape(Shape s) {
                s.draw();
        }
 }

 class Shape {
        abstract void draw();
 }

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


Esempio OCP



Esempio di violazione dell'OCP

Esempio OCP



Possibile soluzione che soddisfa l'OCP

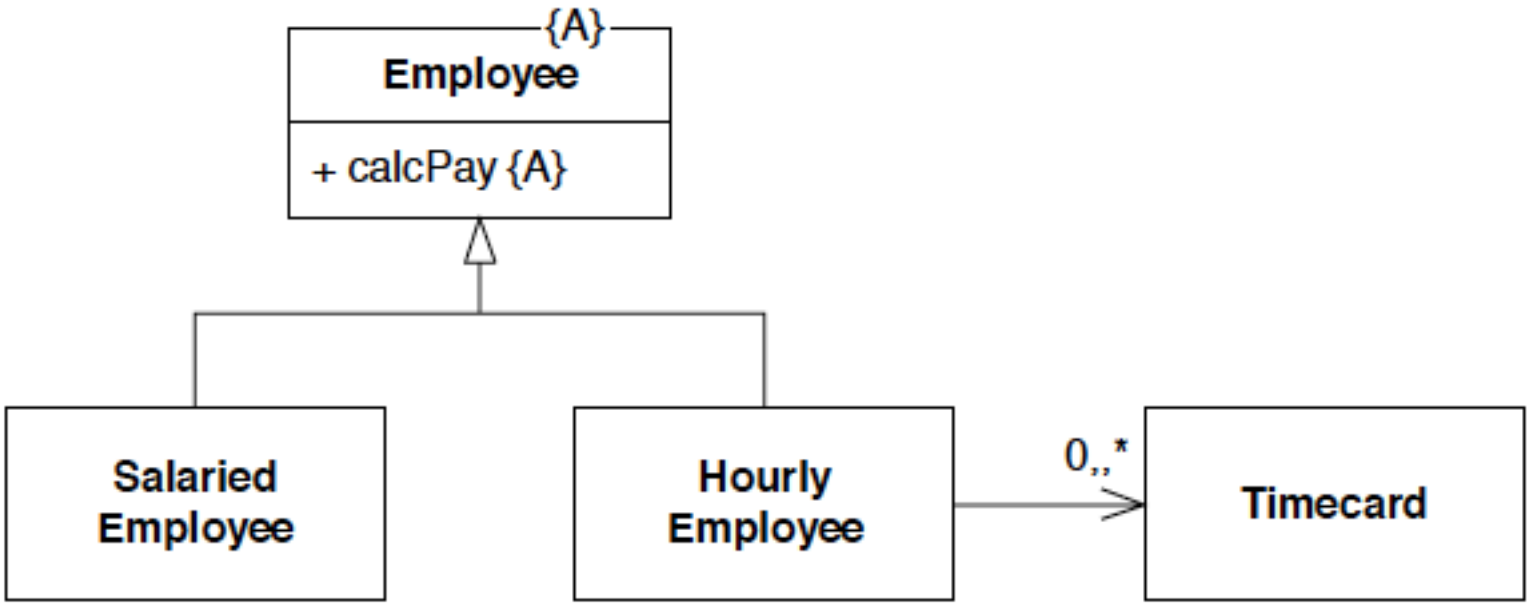
Liskov's Substitution Principle (LSP)

Se q(x) è una proprietà che si può dimostrare essere valida per oggetti x di tipo T, allora q(y) deve essere valida per oggetti y di tipo S dove S è un sottotipo di T

I tipi derivati devono essere completamente sostituibili ai loro tipi base

- Principio
  - è un'estensione dell'OCP
  - dobbiamo essere sicuri che nuove classi derivate stanno estendendo le loro classi base senza cambiare il comportamento
  - le classi derivate dovrebbero essere capaci di rimpiazzare le classi base senza cambiamenti nel codice
  - Introdotto da Barbara Liskov nel 1987

LSP



Esempio di violazione LSP

Supponiamo di aggiungere `VolunteerEmployee`, la nuova classe non rimpiazza la classe base

Indentificare la soluzione

Esempio LSP

```
// Violation of Likov's Substitution Principle
class Rectangle
{
        protected int m_width;
        protected int m_height;

        public void setWidth(int width){
                m_width = width;
        }

        public void setHeight(int height){
                m_height = height;
        }

        public int getWidth(){
                return m_width;
        }

        public int getHeight(){
                return m_height;
        }

        public int getArea(){
                return m_width * m_height;
        }
}
```


Esempio LSP

```
class Square extends Rectangle
{
        public void setWidth(int width){
                m_width = width;
                m_height = width;
        }

        public void setHeight(int height){
                m_width = height;
                m_height = height;
        }

}
```


Esempio LSP

```
class LspTest
{
        private static Rectangle getNewRectangle()
        {
                // it can be an object returned by some factory
...
                return new Square();
        }

        public static void main (String args[])
        {
                Rectangle r = LspTest.getNewRectangle();

                r.setWidth(5);
                r.setHeight(10);
// user knows that r it's a rectangle. It assumes that he's
//able to set the width and height as for the base class

                System.out.println(r.getArea());
// now he's surprised to see that the area is 100 instead of 50.
        }
}
```


Esempio LSP

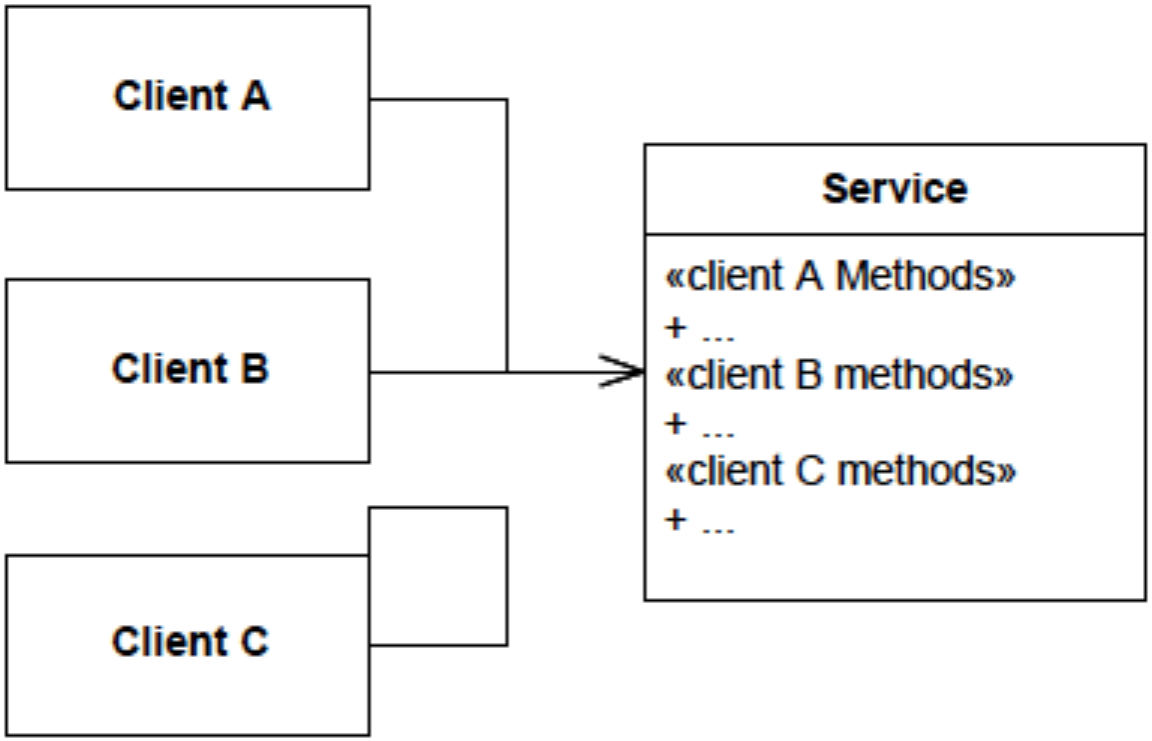
- Le entità non sono corrette
  - un quadrato è definito come un rettangolo, ma non è un rettangolo
- Unica proprietà
  - `getArea` aggiunge del codice specifico per stabilire quale formula (quali proprietà) usare in base al tipo di oggetto
  - per altre forme continua modifica di `getArea` (violazione OCP)
    - astrazione comune `Shape`, con cui esporre la proprietà `Area` da implementare nelle derivate
- L'errore concettuale è di *non legare il calcolo dell'area all'entità stessa*
  - non si tratta infatti di una violazione del SRP, perché il calcolo dell'area è legato alle caratteristiche di ogni figura
- Altro errore riguarda un altro interessante principio
  - Tell, Don't Ask
    - il codice chiede agli oggetti delle informazioni interne, mentre dovrebbe dire a tali oggetti di restituire l'area
    - la soluzione è semplice, usare il polimorfismo ed esporre semplicemente la proprietà `Area`

Interface Segregation Principle (ISP)

> un client non dovrebbe dipendere da metodi che non usa

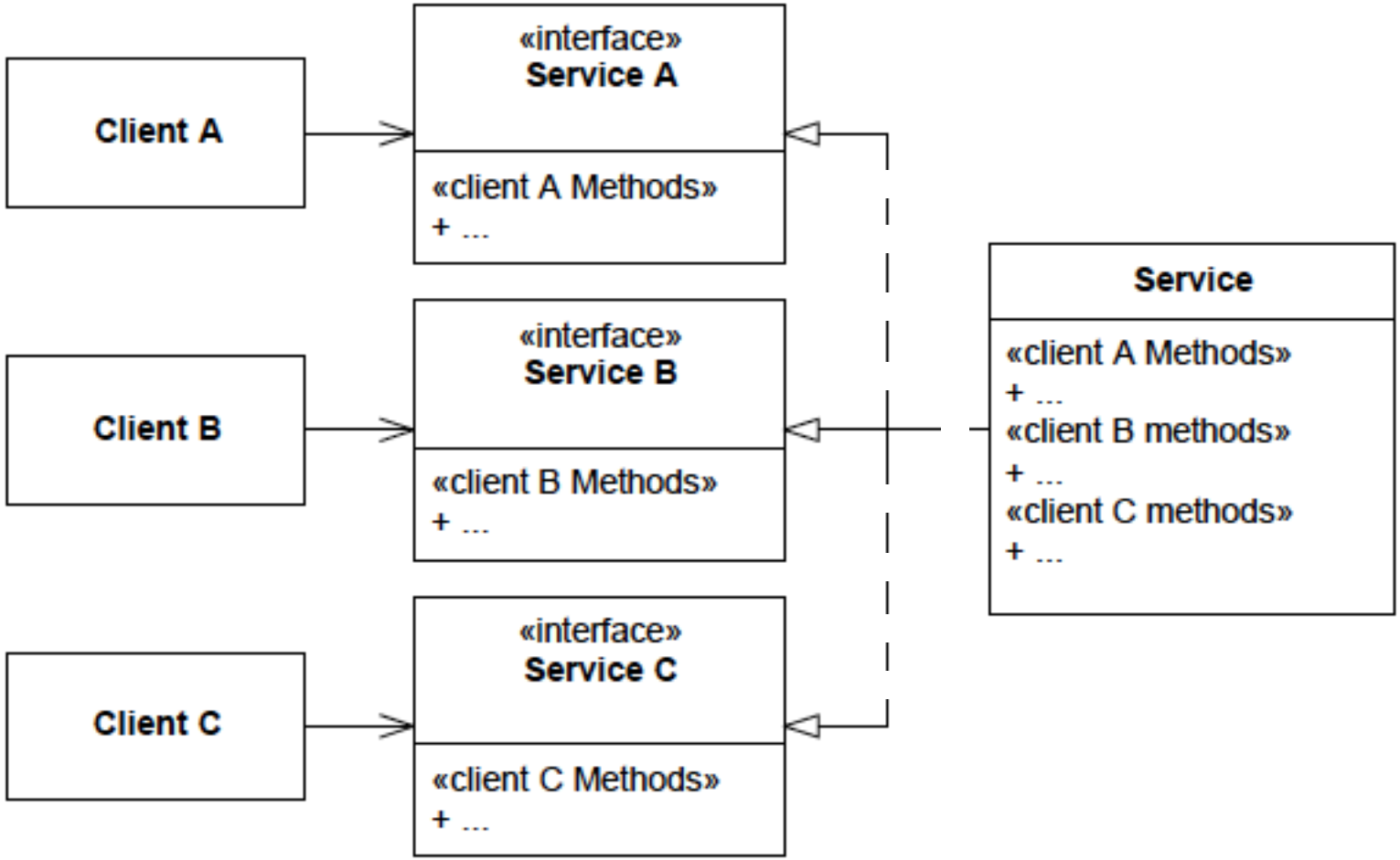
- ISP
  - è preferibile che le interfacce siano molte, specifiche e piccole (composte da pochi metodi) *piuttosto che* poche, generali e grandi

Esempio ISP



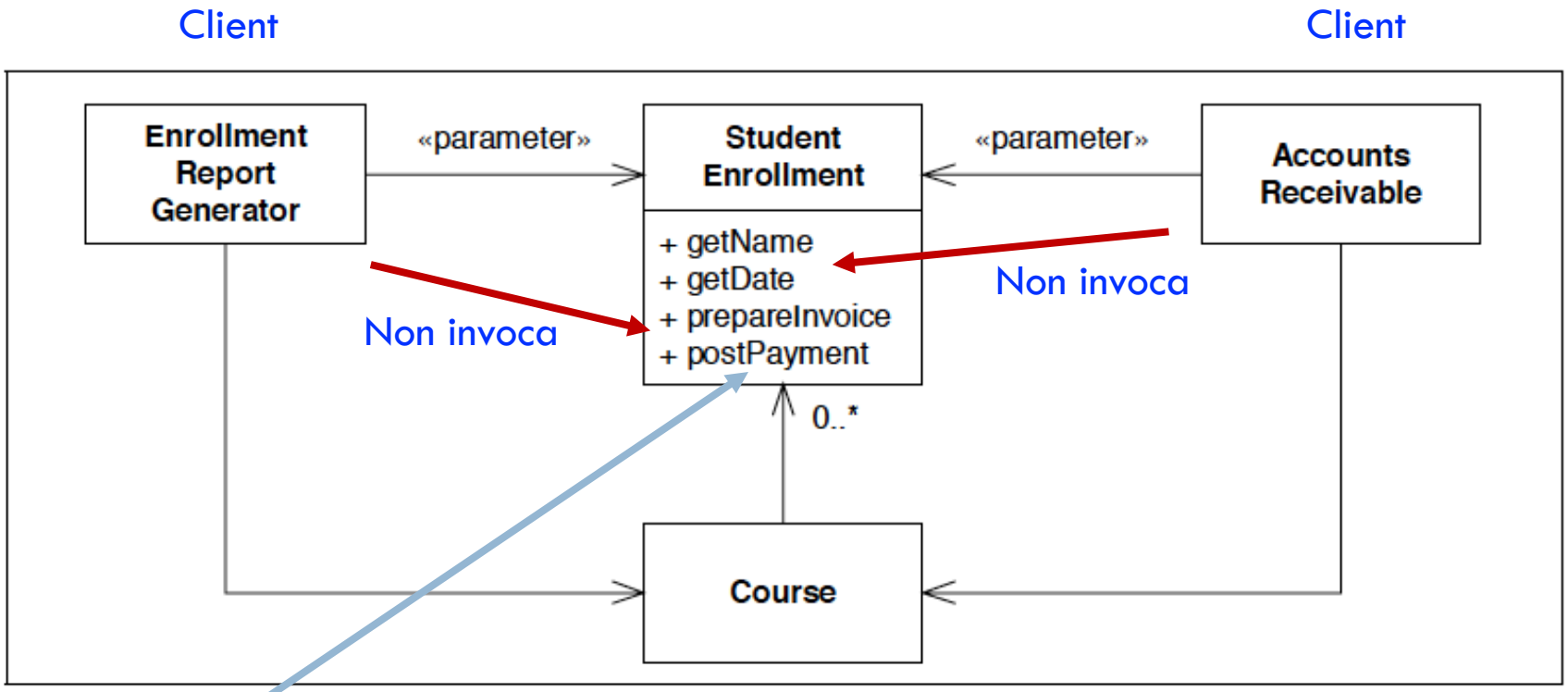
Server senza interfacce segregate: la modifica di una può impattare le altre

Esempio ISP



Server con interfacce segregate

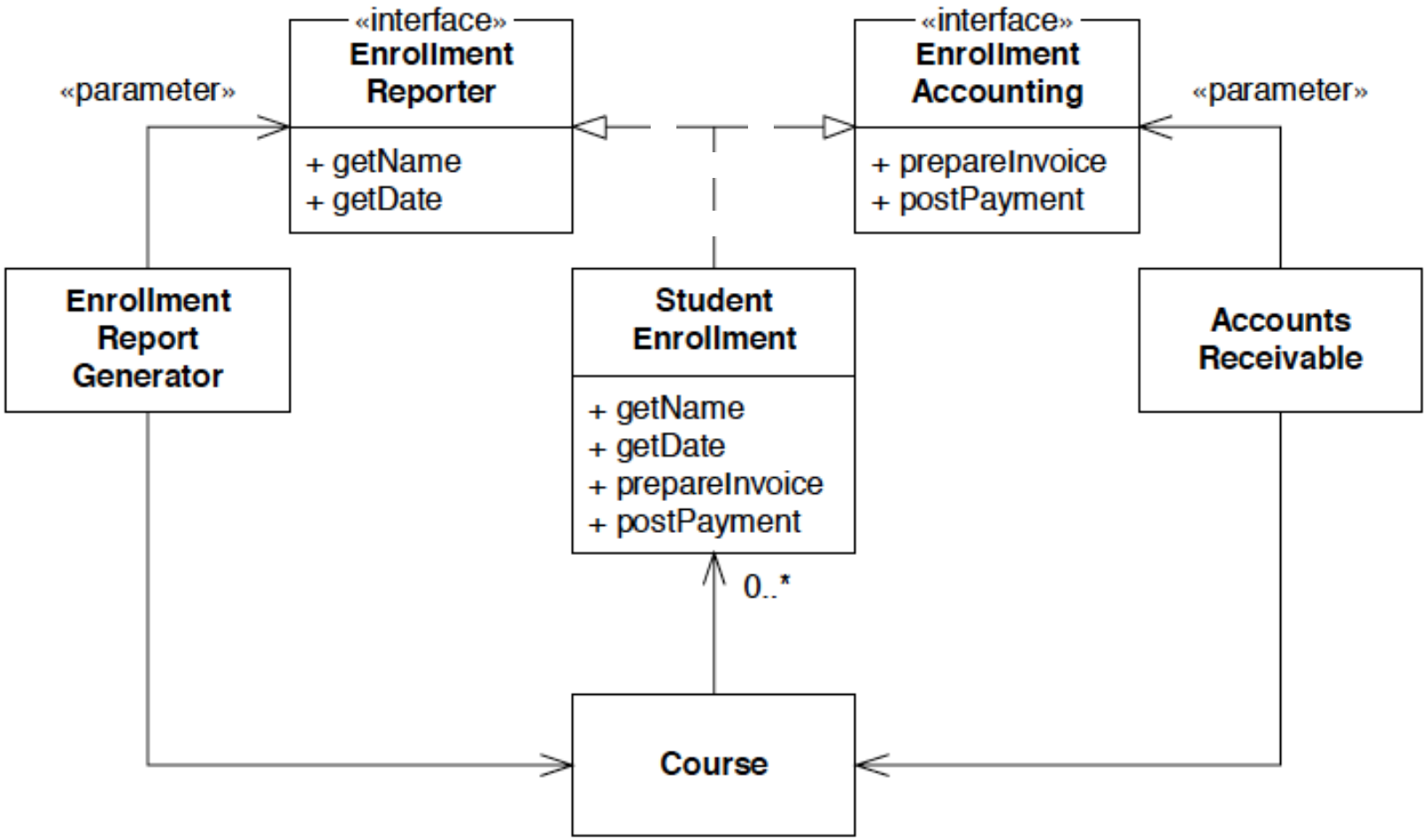
Esempio ISP



Esempio di sistema di iscrizione non segregato

aggiungiamo un nuovo argomento – è necessario ricompilare inutilmente `EnrollmentReportGenerator`

Esempio ISP



Esempio di sistema di iscrizione segregato

Esempio ISP

```
// interface segregation principle
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
        }
}
```


Esempio ISP

```
class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

Esempio di violazione dell'ISP

Supponiamo che un robot entri nellla compagnia … lavoratore ma non maniga …

Esempio ISP

```
// interface segregation principle

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
        }
}
```


Esempio ISP

```
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
        }
}
```


Esempio ISP

```
class Manager {
        Workable worker;

        public void setWorker(Workable w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

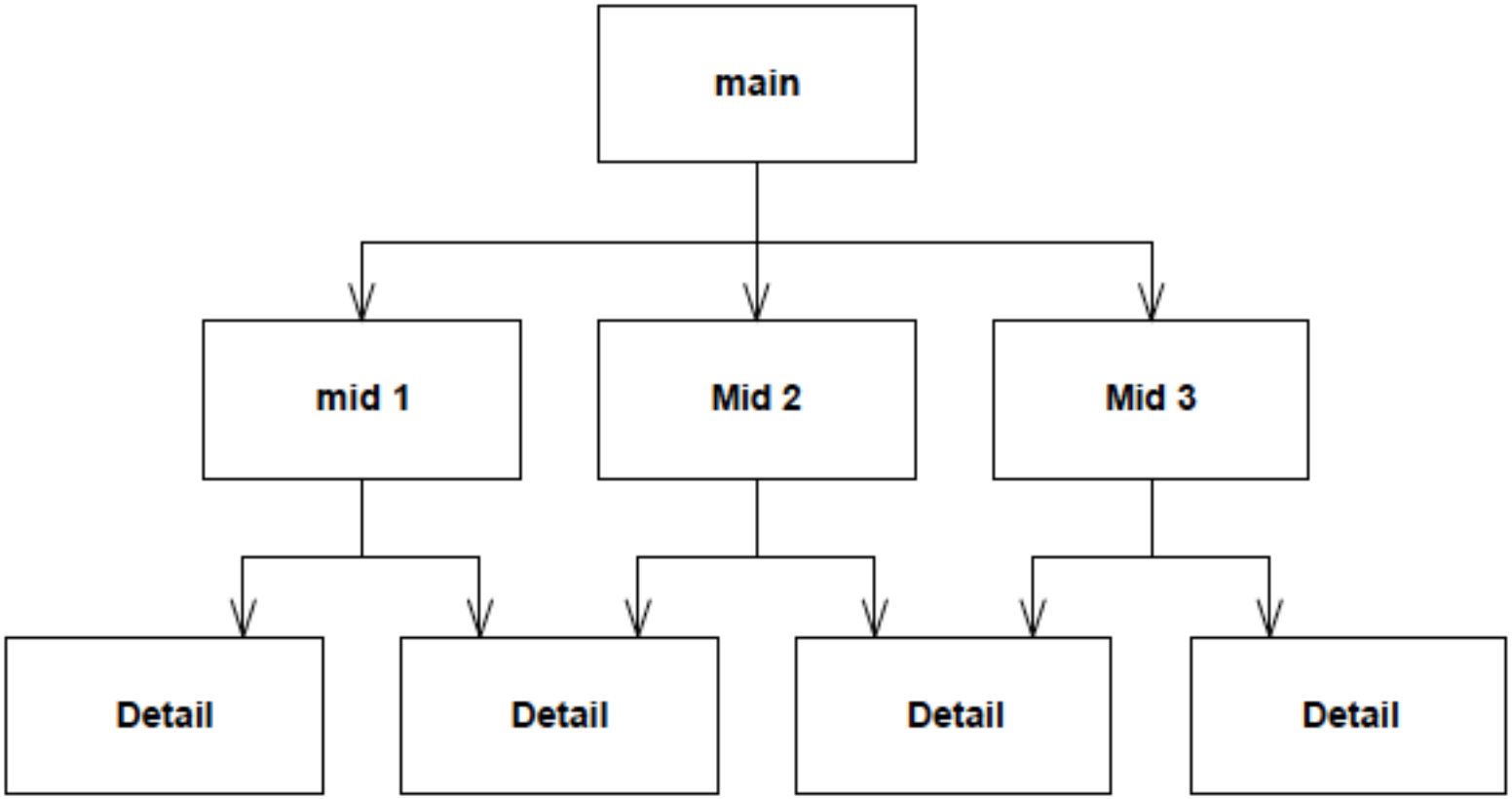
Esempio che soddisfa l'ISP

Dependency Inversione Principle (DIP)

```
I moduli di alto livello non dovrebbero dipendere
dai moduli di basso livello. Entrambi dovrebbero
dipendere dalle astrazioni.
Le astrazioni non dovrebbero dipendere dai dettagli.
I dettagli dovrebbero dipendere dalle astrazioni.
```

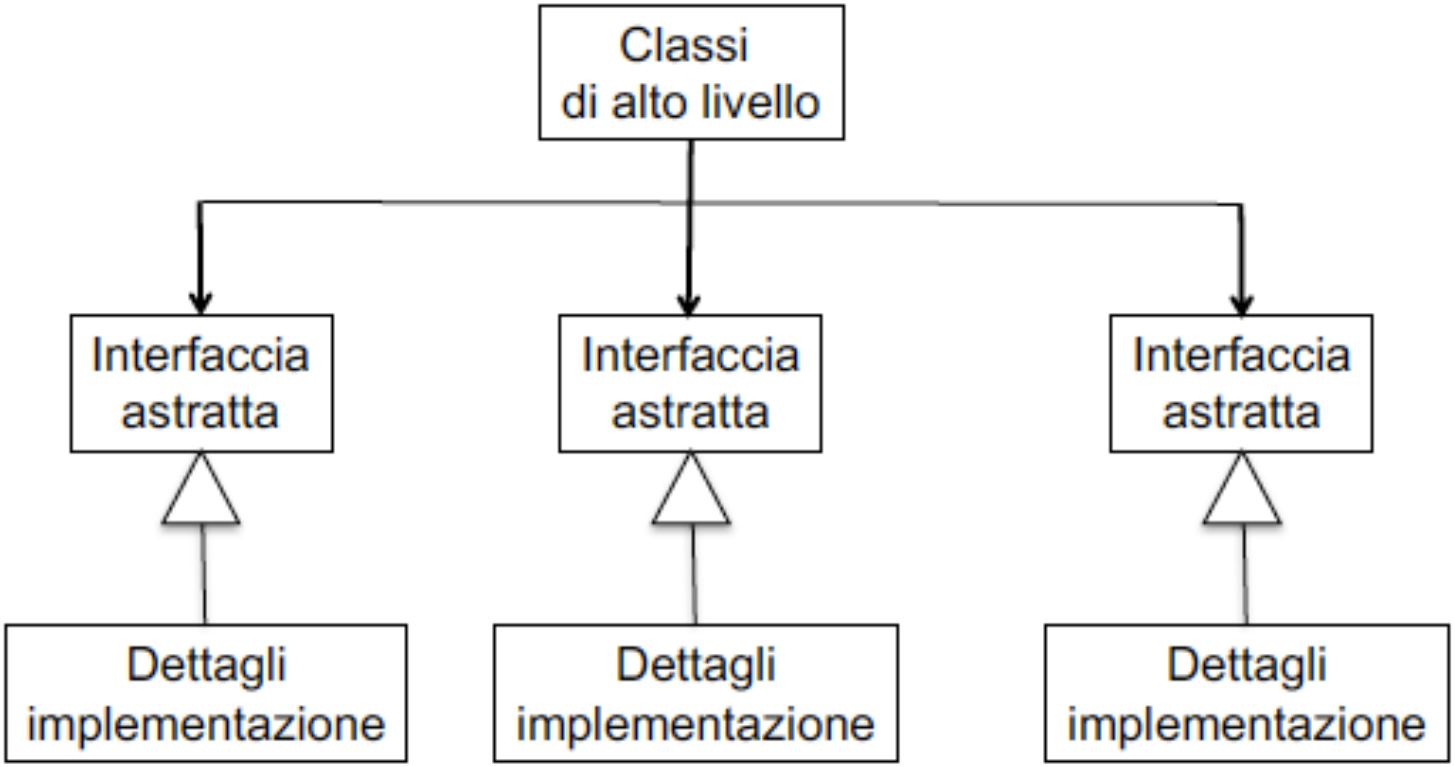
- DIP
  - per lo sviluppo di applicazioni software consideriamo
    - classi a basso livello per le operazioni primarie
    - classi ad alto livello per incapsulare la logica complessa
  - in un cattivo sviluppo del software le classi ad alto livello dipendono pesantemente da quelle a più basso livello

Esempio DIP



Architettura procedurale

Esempio DIP



Architettura OO

Esempio DIP

```
// Dependency Inversion Principle
class Worker {
        public void work() {
                // ....working
        }
}

class Manager {
        Worker worker;

        public void setWorker(Worker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}

class SuperWorker {
        public void work() {
                //.... working much more
        }
}
```

Esempio che non soddisfa il DIP

Esempio DIP

```
// Dependency Inversion Principle
interface IWorker {
        public void work();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
}
class SuperWorker  implements IWorker{
        public void work() {
                //.... working much more
        }
}
class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker = w;
        }
        public void manage() {
                worker.work();
        }
}
```

Esempio che soddisfa il DIP

Law of Demeter (LoD)

Ogni unità di programma dovrebbe conoscere solo poche altre unità di programma strettamente correlate; ogni unità di programma dovrebbe interagire solo con le unità che conosce direttamente.

- LoD
  - Motto
    - "non parlate con gli sconosciuti"
  - un oggetto non dovrebbe interagire direttamente con (usare operazioni di) oggetti a cui accede solo indirettamente (attraverso operazioni o attributi dei suoi conoscenti diretti)

LoD

- un oggetto A può richiedere un servizio (ovvero, chiamare un metodo) di un altro oggetto B ma l'oggetto A non può usare l'oggetto B per raggiungere un terzo oggetto C che possa soddisfare le sue richieste
  - implicherebbe una conoscenza dei dettagli interni di B (nello specifico, dei suoi componenti) da parte di A

- Anziché consentire ad A di interagire con un oggetto ottenuto da B (uno "sconosciuto")
  - modificare la classe B in modo da fornire direttamente nell'interfaccia di B il servizio di C che serve ad A

Esempio LoD

```
objectA.getObjectB().doSomething();

//oppure

objectA.getObjectB().getObjectC().doSomething();
```

Esempio di violazione LoD