# ALCUNI PRINCIPI DI DESIGN OBJECT-ORIENTED

“

# MINIMIZZA L’ACCESSIBILITÀ DI CLASSI E ATTRIBUTI

*Principio #1*

”

# COSA INTENDIAMO CON "ASTRAZIONE"

- Tony Hoare: *"Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences."*

- Grady Booch: *"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."*

- L'astrazione è uno dei modi per gestire la complessità

- Con l'astrazione ci focalizziamo sull'aspetto "esterno" dell'oggetto, separando il suo comportamento dalla sua implementazione

# INCAPSULAMENTO

- Grady Booch: *"Encapsulation is the process of compartmentalising the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation."*

- Craig Larman: *"Encapsulation is a mechanism used to hide the data, internal structure, and implementation details of an object. All interaction with the object is through a public interface of operations."*

- Le classi devono essere opache.

- Le classi non devono mostrare i loro dettagli implementativi interni.

# INFORMATION HIDING IN JAVA

- Usate attributi privati e i relativi metodi (accessors and mutators)

- Esempio
  ➠ Cambiate

```
public double speed;
```

  ➠ con

```
private double speed;
public double getSpeed() {
    return speed;
}
public void setSpeed(double newSpeed) {
    speed = newSpeed;
}
```

# USATE ACCESSORS E MUTATORS, NON DIRETTAMENTE GLI ATTRIBUTI

- Potete mettere dei limiti ai valori

```
public void setSpeed(double newSpeed) {
    if(newSpeed < 0) {
        sendErrorMessage(…);
        speed = Math.abs(newSpeed);
    }
    speed = newSpeed;
}
```

- Se l'attributo fosse pubblico chiunque lo usasse direttamente dovrebbe anche controllare gli eventuali limiti

# USATE ACCESSORS E MUTATORS, NON DIRETTAMENTE GLI ATTRIBUTI

- Potete cambiare il codice interno senza cambiare l'interfaccia

```
//Miglia o Kilometri
public void setSpeedInMPH(double newSpeed) {
     speedInKPH = convert(newSpeed);
}

public void setSpeedInKPH(double newSpeed) {
     speedInKPH = newSpeed;
}
```

# USATE ACCESSORS E MUTATORS, NON DIRETTAMENTE GLI ATTRIBUTI

- Possiamo usare effetti aggiuntivi

```
public void setSpeed(double newSpeed) {
    speed = newSpeed;
    notifyObserver();
}
```

- Se l'attributo fosse pubblico e qualcuno ci accedesse direttamente dovrebbe farsi carico di eseguire gli effetti aggiuntivi

# “PREFERITE LA COMPOSIZIONE ALLA EREDITARIETÀ”

*Principio #2*

# COMPOSIZIONE

- Metodo di riuso in cui le nuove funzionalità sono ottenute creando un oggetto *composto* di altri oggetti.
- La nuova funzionalità si ottiene delegandola a uno degli oggetti che vengono composti.
- Qualche volta viene anche chiamata *aggregation* o *containment*, però attenzione che autori diversi danno un significato particolare a questi termini
- Esempio
  - Aggregation - when one object owns or is responsible for another object and both objects have identical lifetimes (GoF)
  - Aggregation - when one object has a collection of objects that can exist on their own (UML)
  - Containment - a special kind of composition in which the contained object is hidden from other objects and access to the contained object is only via the container object (Coad)

# COMPOSIZIONE

- La composizione può essere
  - ➡ per referenza
  - ➡ per valore
- C++ permette entrambi i tipi di composizione
- In Java abbiamo solo referenze a oggetti.

# VANTAGGI/SVANTAGGI DELLA COMPOSIZIONE

✓ Vantaggi
- ➡ Gli oggetti contenuti sono usati dai loro contenitori solo attraverso le loro interfacce
- ➡ Possibilità di riuso “a scatola chiusa”, in quanto i dettagli interni degli oggetti contenuti non sono visibili
- ➡ Buon incapsulamento
- ➡ Minori dipendenze implementate
- ➡ Ogni classe è focalizzata su un unico compito
- ➡ La composizione si può definire dinamicamente a run-time: i contenitori possono acquisire in quel momento la referenza degli oggetti contenuti

✘ Svantaggi
- ➡ Il software risultante tende ad avere un maggior numero di oggetti
- ➡ Le interfacce devono essere definite attentamente in modo da usare molti oggetti differenti come blocchi da comporre.

# EREDITARIETÀ

- È un modo per riusare codice in cui una nuova funzionalità si ottiene estendendo l'implementazione di un oggetto esistente.
- La superclasse (l'oggetto che viene esteso) contiene esplicitamente gli attributi comuni e i metodi comuni.
- La sottoclasse (la classe specializzata) estende l'implementazione con attributi e metodi aggiuntivi.

# VANTAGGI/SVANTAGGI DELL'EREDITARIETÀ

✓ Vantaggi

- Implementare la classe derivata di solito è facile in quanto la maggior parte delle cose sono ereditate.
- È facile modificare o estendere l'implementazione che viene riusata

✘ Svantaggi

- Rompe l'incapsulamento, perché espone una classe derivata a dettagli implementativi della superclasse
- Riuso "white-box", perché i dettagli interni della superclasse spesso sono visibili dalla sottoclasse.
- Se si cambia qualcosa nella superclasse è possibile che tutte le sottoclassi debbano essere modificate.
- Le implementazioni ereditate dal superclasse non possono essere cambiate a run-time.

# EREDITARIETÀ VS COMPOSIZIONE

- Rileggete l'esempio relativo al gioco SimDuck…

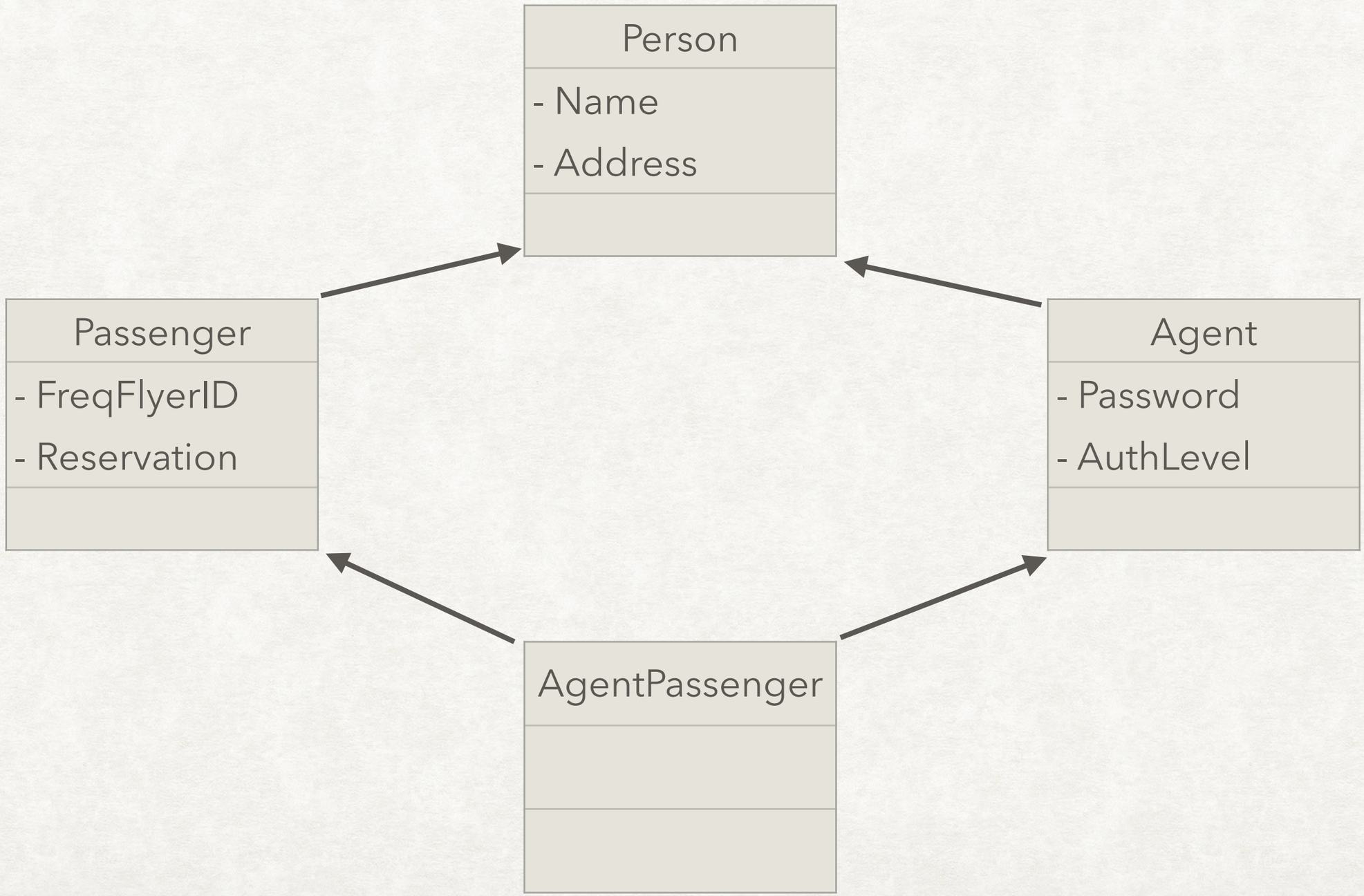
# COAD'S RULES

Usate l'ereditarietà solo se ognuno dei seguenti criteri è soddisfatto:

- Una sottoclasse esprime il concetto "è un tipo speciale di" e non "è un ruolo svolto da".
- Una istanza di una sottoclasse non ha mai bisogno di diventare un oggetto di un'altra classe.
- Una sottoclasse estende, piuttosto che sovrascrivere o rende nullo, i metodi della sua superclasse.
- Una sottoclasse non estende le funzionalità di quella che è semplicemente una utility class.
- Per una classe nell'attuale Problem Domain, la sottoclasse specializza un ruolo, una transazione o un device.

# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 1



# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 1

- "è un tipo speciale di" e non "è un ruolo svolto da"
  - ✗ NO: un passeggero è un ruolo svolto da una persona; idem per l'agente
- Non trasmuta mai
  - ✗ NO: una istanza di una sottoclasse di Person può dover cambiare da Passeggero a Agente a AgentePasseggero col tempo.
- Estende piuttosto di sovrascrivere o annullare
  - ✓ OK
- Non estende una utility class
  - ✓ OK
- Nel Problem Domain, specializza un ruolo, transazione o device
  - ✗ NO: una persona non è un ruolo, transazione o device

# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 1

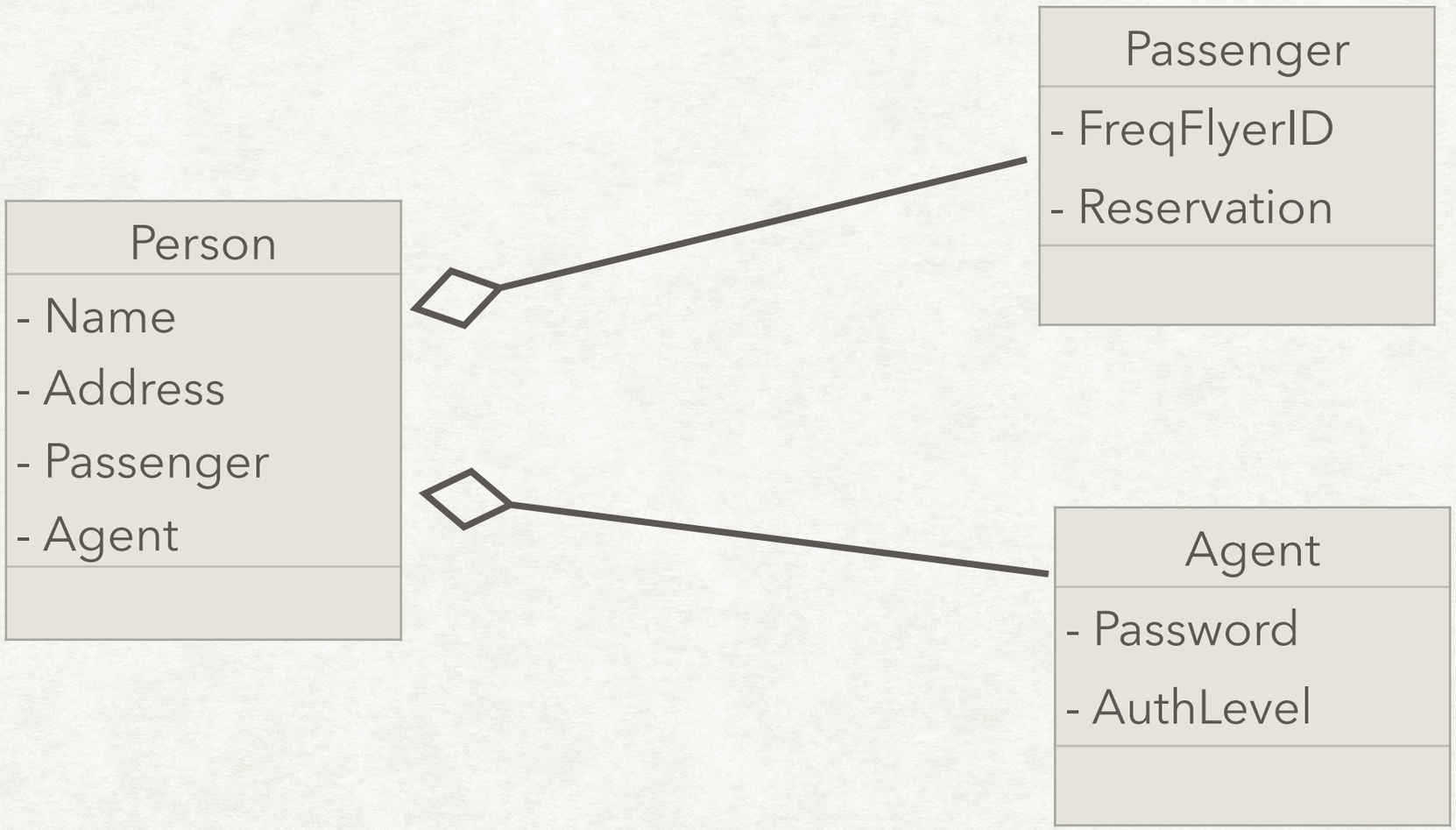
- "è un tipo speciale di" e non "è un ruolo svolto da"
  - ✗ NO: un passeggero è un ruolo svolto da una persona; idem per l'agente
- Non trasmuta mai
  - ✗ NO: una istanza di una sottoclasse di Person può dover cambiare da Passeggero a Agente a AgentePasseggero col tempo.
- Estende piuttosto di sovrascrivere o annullare
  - ✓ OK
- Non estende una utility class
  - ✓ OK
- Nel Problem Domain, specializza un ruolo, transazione o device
  - ✗ NO: una persona non è un ruolo, transazione o device

L'EREDITARIETÀ NON VA BENE!

# EREDITARIETÀ/COMPOSIZIONE

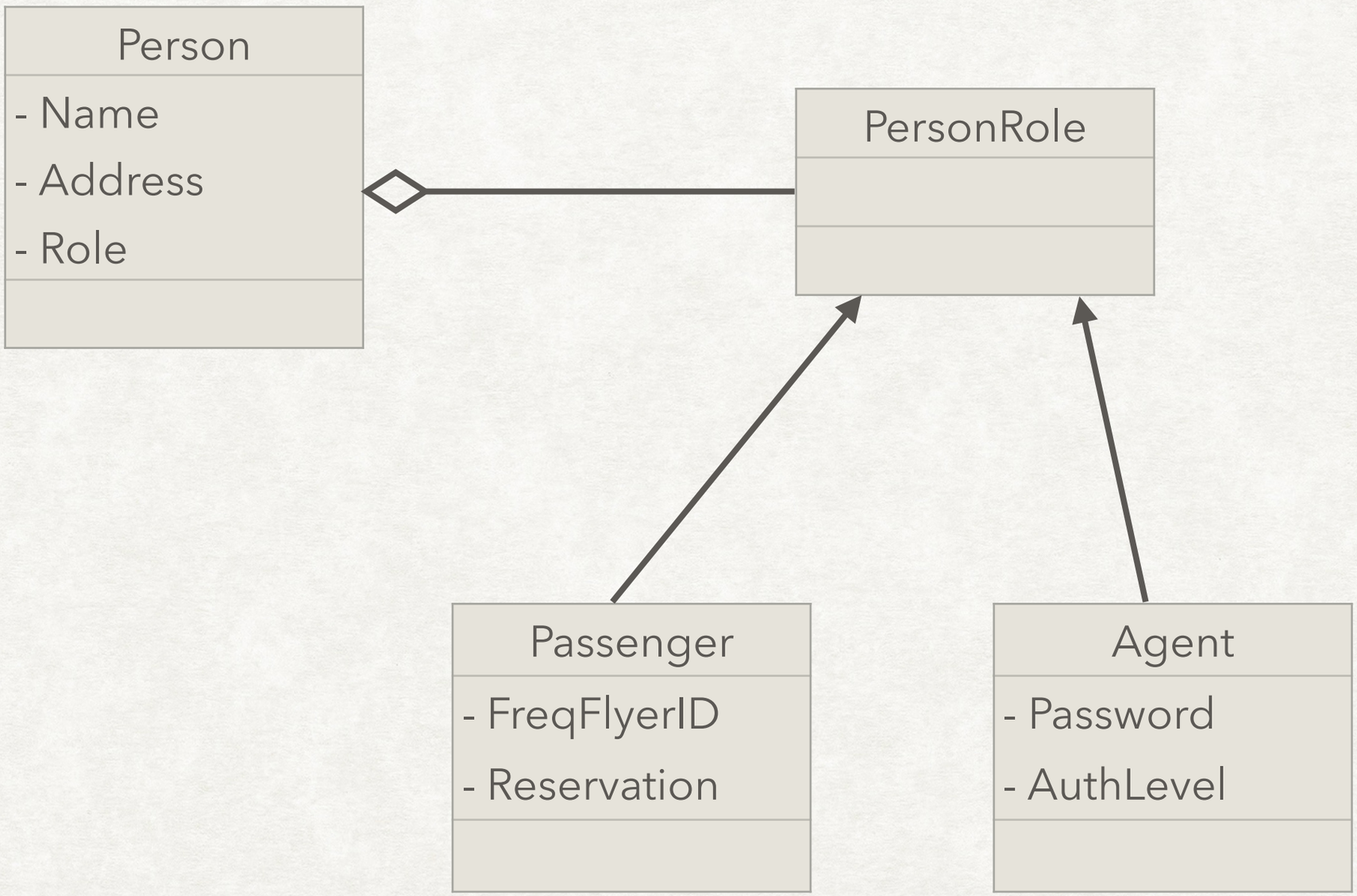
## ESEMPIO 1

Usiamo la composizione



# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 2



# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 2

- "è un tipo speciale di" e non "è un ruolo svolto da"
  - ✓ OK: Passenger e Agent sono casi particolare di ruoli di Person
- Non trasmuta mai
  - ✓ OK: un oggetto Passenger rimane un oggetto Passenger; stessa cosa per un oggetto Agent
- Estende piuttosto di sovrascrivere o annullare
  - ✓ OK
- Non estende una utility class
  - ✓ OK
- Nel Problem Domain, specializza un ruolo, transazione o device
  - ✓ OK: PersonRole è un tipo di ruolo.

# EREDITARIETÀ/COMPOSIZIONE

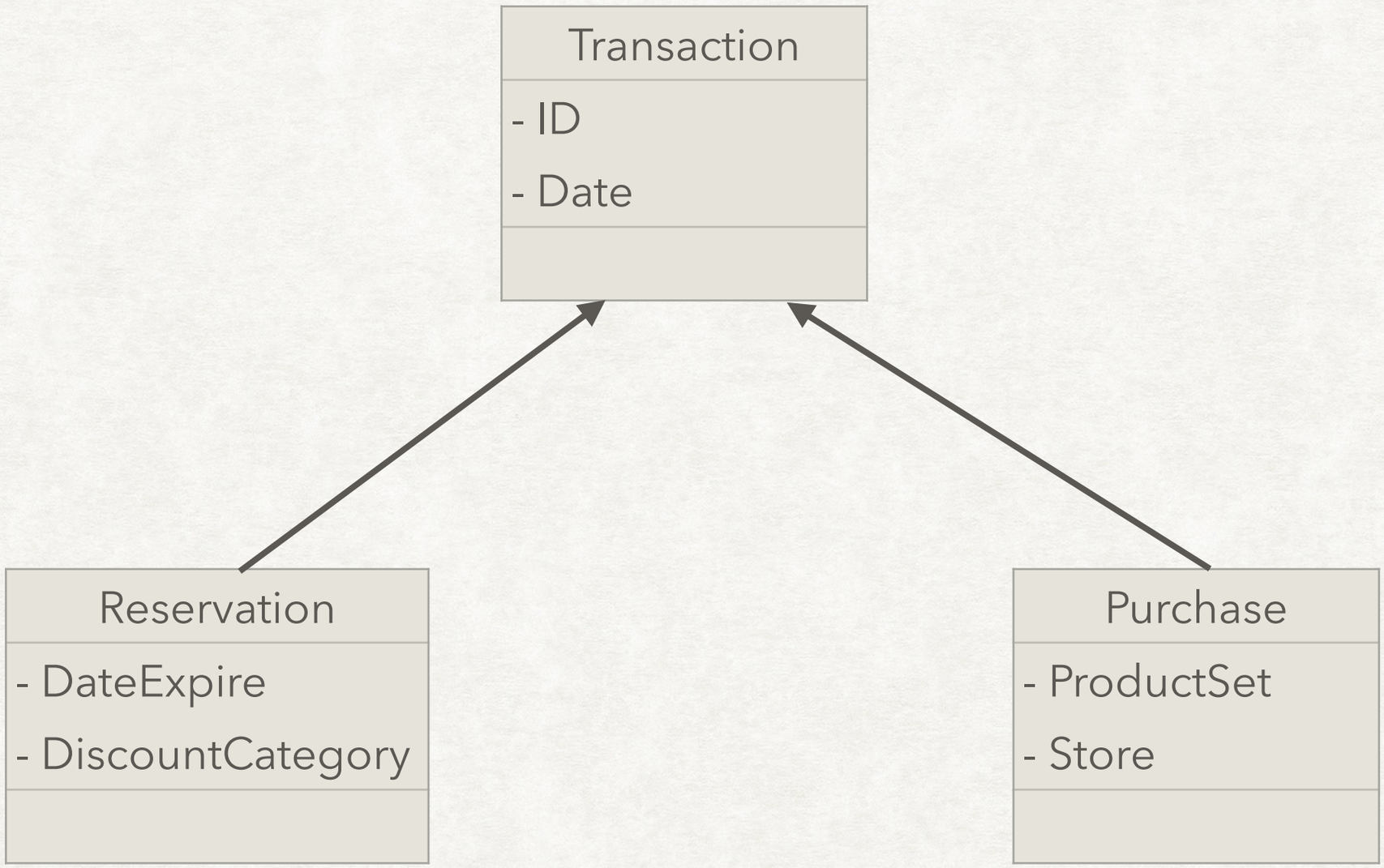
## ESEMPIO 2

- "è un tipo speciale di" e non "è un ruolo svolto da"
  - ✓ OK: Passenger e Agent sono casi particolare di ruoli di Person
- Non trasmuta mai
  - ✓ OK: un oggetto Passenger rimane un oggetto Passenger; stessa cosa per un oggetto Agent
- Estende piuttosto di sovrascrivere o annullare
  - ✓ OK
- Non estende una utility class
  - ✓ OK
- Nel Problem Domain, specializza un ruolo, transazione o device
  - ✓ OK: PersonRole è un tipo di ruolo.

L'EREDITARIETÀ VA BENE!

# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 3



# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 3

- "è un tipo speciale di" e non "è un ruolo svolto da"
  - ✓ OK: Reservation e Purchase sono un tipo particolare di Transaction
- Non trasmuta mai
  - ✓ OK: un oggetto Resevation rimane un oggetto Reservation; stessa cosa per un oggetto Purchase
- Estende piuttosto di sovrascrivere o annullare
  - ✓ OK
- Non estende una utility class
  - ✓ OK
- Nel Problem Domain, specializza un ruolo, transazione o device
  - ✓ OK: è una transazione.

# EREDITARIETÀ/COMPOSIZIONE

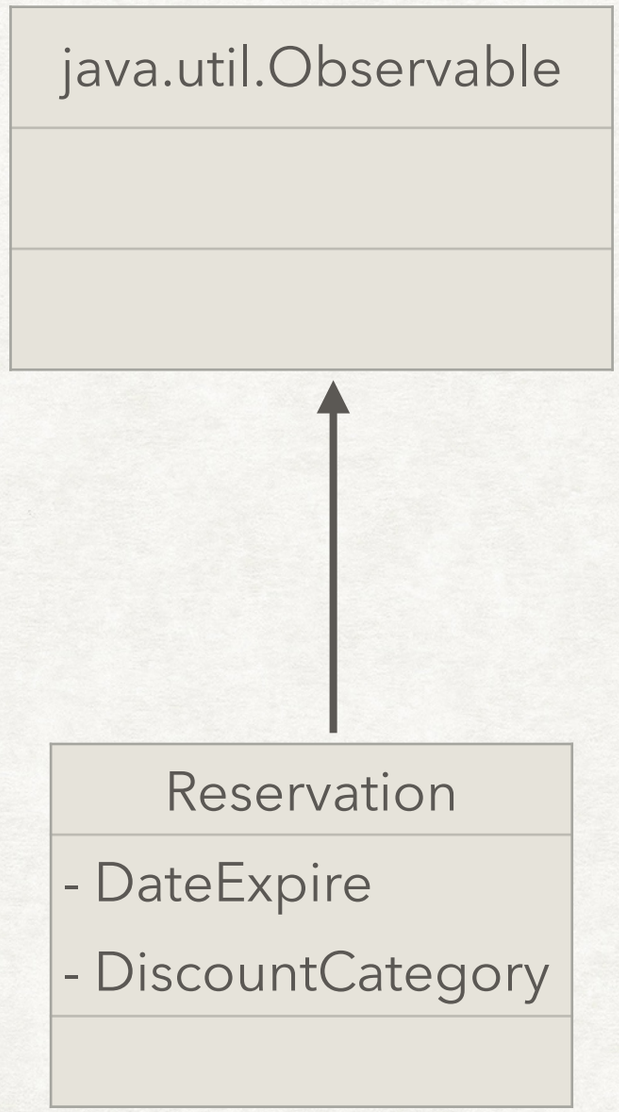
## ESEMPIO 3

- "è un tipo speciale di" e non "è un ruolo svolto da"
  - ✓ OK: Reservation e Purchase sono un tipo particolare di Transaction
- Non trasmuta mai
  - ✓ OK: un oggetto Resevation rimane un oggetto Reservation; stessa cosa per un oggetto Purchase
- Estende piuttosto di sovrascrivere o annullare
  - ✓ OK
- Non estende una utility class
  - ✓ OK
- Nel Problem Domain, specializza un ruolo, transazione o device
  - ✓ OK: è una transazione.

**L'EREDITARIETÀ VA BENE!**

# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 4



# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 4

- "è un tipo speciale di" e non "è un ruolo svolto da"
  - ✘ NO: una Reservation non è un tipo speciale di Observable.
- Non trasmuta mai
  - ✓ OK: una Reservation rimane tale.
- Estende piuttosto di sovrascrivere o annullare
  - ✓ OK
- Non estende una utility class
  - ✘ NO: Observation è proprio una utility class.
- Nel Problem Domain, specializza un ruolo, transazione o device
  - ✘ Non Applicabile: Observation è una utility class, non un Problem Domain

# EREDITARIETÀ/COMPOSIZIONE

## ESEMPIO 4

- "è un tipo speciale di" e non "è un ruolo svolto da"
  - ✘ NO: una Reservation non è un tipo speciale di Observable.

- Non trasmuta mai
  - ✓ OK: una Reservation rimane tale.

- Estende piuttosto di sovrascrivere o annullare
  - ✓ OK

- Non estende una utility class
  - ✘ NO: Observation è proprio una utility class.

- Nel Problem Domain, specializza un ruolo, transazione o device
  - ✘ Non Applicabile: Observation è una utility class, non un Problem Domain

L'EREDITARIETÀ NON VA BENE!

# EREDITARIETÀ/COMPOSIZIONE

## PER CONCLUDERE

- Sia Ereditarietà che Composizione sono importanti per il riuso del codice.
- L'Ereditarietà è stato (ab)usata agli inizi della programmazione OO.
- Col tempo si è capito che il design può essere più ri-usabile e più semplice usando invece la Composizione.
- Ovviamente l'insieme delle classi da comporre può essere aumentato usando l'Ereditarietà (ricordate SimDuck…)
- Per cui Ereditarietà e Composizione lavora insieme

**PREFERITE LA COMPOSIZIONE ALL'EREDITARIETÀ**

“

# PROGRAM TO AN INTERFACE, NOT AN IMPLEMENTATION

*Principio #3*

”

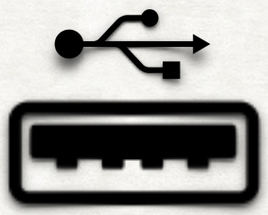
# INTERFACCE

- Una Interface è un insieme di metodi di un oggetto, che un qualsiasi altro oggetto sa di poter invocare.
- Un oggetto può avere più di una Interface (in pratica, una interface è un sottoinsieme di tutti i metodi che un oggetto implementa)
- Un Type è una specifica interface di un oggetto
- Oggetti differenti possono avere lo stesso Type, e lo stesso oggetto può avere Type differenti
- Un oggetto è noto agli altri oggetti solo attraverso la sua interface.
- In un certo senso, le Interface esprimono una forma molto limitata di "è un tipo di": "è un tipo che supporta questa Interface"
- Interface è fondamentale per poter cambiare oggetti senza dover cambiare codice ("pluggability")

# INTERFACCE

Interfaccia vs implementazione

USb bus



Hard disk

CDROM

Printer

Digital Camera

ecc...

USB è una astrazione dell'hardware.
Un PC è costruito usando l'interfaccia USB, non i singoli device

# IMPLEMENTATION INHERITANCE VS INTERFACE INHERITANCE

- Implementation Inheritance (class inheritance) - l'implementazione di un oggetto è definita attraverso l'implementazione di un altro oggetto.
- Interface Inheritance (subtyping) - descrive quando un oggetto può essere usato al posto di un altro oggetto.
- C++
  - l'ereditarietà comporta entrambe le cose
  - l'interface inheritance avviene se si eredita da una classe completamente astratta
- Java
  - java dispone di un costrutto separato per la interface inheritance: java interface

# VANTAGGI DELLE INTERFACE

- Vantaggi
  - i client non devono conoscere la classe specifica dell'interfaccia che stanno usando
  - un oggetto può essere facilmente rimpiazzato da un altro oggetto
  - le connessioni tra oggetti non sono più hardwired a un oggetto di una specifica classe (maggiore flessibilità)
  - abbassa l'accoppiamento
  - aumenta la possibilità di riuso
  - aumenta le possibilità di usare composizione in quanto l'oggetto contenuto può essere di una qualsiasi classe che implementa quella interfaccia

- Svantaggi
  - un leggero aumento della complessità di design

# ESEMPIO INTERFACCIA

```
// Interface IManeuverable provides the specification
// for a maneuverable vehicle.

public interface IManeuverable {
public void left();
public void right();
public void forward();
public void reverse();
public void climb();
public void dive();
public void setSpeed(double speed);
public double getSpeed();
}

public class Car
implements IManeuverable { // Code here. }

public class Boat
implements IManeuverable { // Code here. }

public class Submarine
implements IManeuverable { // Code here. }
```

Così un'altra classe può muovere un veicolo senza sapere il tipo di veicolo (auto, nave, …)

```
public void
travel(IManeuverable vehicle) {
vehicle.setSpeed(35.0);
vehicle.forward();
vehicle.left();
vehicle.climb();
}
```

“

# OPEN-CLOSED PRINCIPLE: SOFTWARE DEVE ESSERE APERTO A ESTENSIONI MA CHIUSO A MODIFICHE

*Principio #4*

”

# IL PRINCIPIO OPEN-CLOSE (OCP)

- In pratica: dovremmo progettare moduli che non debbano mai essere cambiati: "il vecchio codice non si cambia"…
- Se dobbiamo estendere il comportamento del sistema, scriviamo nuovo codice, non modifichiamo l'esistente.
- Moduli che soddisfano il criterio OCP soddisfano i due criteri
  - ✓ aperti a estensioni: il comportamento dei moduli può essere esteso per soddisfare nuovi requisiti
  - ✓ chiusi a modifiche: il sorgente dei  moduli non può cambiare
- Come si ottiene?
  - ✓ Astrazione
  - ✓ Polimorfismo
  - ✓ Ereditarietà
  - ✓ Interfacce

# IL PRINCIPIO OPEN-CLOSE (OCP)

- Di solito non è possibile avere tutti i moduli che soddisfano il principio: cercare di minimizzare il numero di moduli che non lo soddisfano.
- Il principio OCP è uno dei capisaldi alla base della programmazione OO.
- Soddisfare questo principio normalmente porta a un elevato livello di riusabilità e mantenibilità

# IL PRINCIPIO OPEN-CLOSE (OCP)

## ESEMPIO

- Consideriamo il seguente metodo di una classe

```
public double totalPrice(Part[] parts) {
    double total = 0.0;
    for(int i=0; i<parts.length; i++) {
        total += parts[i].getPrice();
    }
    return total;
}
```

- Lo scopo del metodo è chiaro
- Se Part è una base class o una interface e usiamo polimorfismo ecco che questa classe può accomodare nuovi tipi di **`parts`** *senza* dover essere modificata.
- È conforme a OCP.

# IL PRINCIPIO OPEN-CLOSE (OCP)

## ESEMPIO

- Supponiamo però alcuni tipi di parti (motherboard e memorie) debbano avere il prezzo incrementato del 30%.
- Che ne dite di questo codice?

```
public double totalPrice(Part[] parts) {
    double total = 0.0;
    for(int i=0; i<parts.length; i++) {
        if(parts[i] instanceof Motherboard)
            total += (1.3 * parts[i].getPrice());
        else if (parts[i] instanceof Memory)
            total += (1.3 * parts[i].getPrice());
        else
            total += parts[i].getPrice();
    }
    return total;
}
```

# IL PRINCIPIO OPEN-CLOSE (OCP)

## ESEMPIO

- Soddisfa il principio? Manco per sogno! 😝
- Ogni volta che cambiamo quali parti hanno un sovrapprezzo dobbiamo modificare il metodo! Quindi non siamo "chiusi per modifiche".
- In origine non c'erano sovrapprezzi e il codice è stato scritto in un modo; il cambio di requirement comporta che dobbiamo aggiustare il codice per aderire al principio.
- Se incorporassimo il sovrapprezzo dentro al metodo getPrice di Part?

# IL PRINCIPIO OPEN-CLOSE (OCP)

## ESEMPIO

```
// Class Part is the superclass for all parts.
public class Part {
    private double price;
    public Part(double price) (this.price = price;}
    public void setPrice(double price) {this.price = price;}
    public double getPrice() {return price;}
}

// Class ConcretePart implements a part for sale.
// Pricing policy explicit here!
public class ConcretePart extends Part {
    public double getPrice() {
        // return (1.45 * price); //Premium
        return (0.90 * price); //Labor Day Sale
    }
}
```

- Così però dobbiamo modificare tutte le classi derivate da **`Part`**…Idee migliori?

# IL PRINCIPIO OPEN-CLOSE (OCP)

## ESEMPIO

```
// The Part class now has a contained PricePolicy object.
public class Part {
    private double price;
    private PricePolicy pricePolicy;
    public void setPricePolicy(PricePolicy pricePolicy) {
        this.pricePolicy = pricePolicy;
    }
    public void setPrice(double price) {this.price = price;}
    public double getPrice() {return pricePolicy.getPrice(price);}
}

// Class PricePolicy implements a given price policy.
public class PricePolicy {
    private double factor;
    public PricePolicy (double factor) {
        this.factor = factor;
    }
    public double getPrice(double price) {return price * factor;}
}
```

# IL PRINCIPIO OPEN-CLOSE (OCP)

## ESEMPIO

- Abbiamo “estratto” la parte che cambia e l’abbiamo poi incapsulata (SimDuck, ricordate?….)
- Così possiamo cambiare a run time **`PricePolicy`**, che è contenuto in **`Part`**, senza dover cambiare **`Part`**
- Questo modo di fare è OCP compliant!

“

# FUNZIONI CHE USANO REFERENZA A SUPERCLASSI, DEVONO POTER USARE CLASSI DERIVATE SENZA ACCORGERSENE

*Principio #5*

”

# IL PRINCIPIO DI SOSTITUZIONE DI LYSKOV

- Sembra ovvio visto come funziona il polimorfismo.
- Esempio

```
public void drawShape(Shape s) {
    //codice
}
```

- La funzione dovrebbe funzionare con qualunque delle sottoclassi derivate dalla classe Shape.
- Ma dobbiamo fare attenzione mentre scriviamo le sottoclassi e non violare inintenzionalmente il principio.
- Se una funzione non rispetta il principio, probabilmente contiene referenze esplicite a qualche delle sottoclassi o superclassi. Spesso queste funzioni violano anche OCP perché devono essere modificate ogni volte che si crea una sottoclasse.

# IL PRINCIPIO DI SOSTITUZIONE DI LYSKOV

## ESEMPIO

- Considerate la seguente classe Rectangle

```
// A very nice Rectangle class.
public class Rectangle {
     private double width;
     private double height;
     public Rectangle(double w, double h) {
          width = w;
          height = h;
     }
     public double getWidth() {return width;}
     public double getHeight() {return height;}
     public void setWidth(double w) {width = w;}
     public void setHeight(double h) {height = h;}
     public double area() {return (width * height);
}
```

# IL PRINCIPIO DI SOSTITUZIONE DI LYSKOV

## ESEMPIO

- Come facciamo per una classe Square? Un quadrato è un rettangolo, quindi la classe Square dovrebbe essere derivata dalla classe Rectangle, giusto? Vediamo…

```
//A square class
public class Square extends Rectangle {
    public Square(doubl s) { super (s, s);}

    public void setWidth(double w){
        setWidth(w);
        setHeight(w);
    }
    public void setHeight(double h){
        setHeight(h);
        setWidth(h);
    }
}
```

override perché Height e Width sono uguali in un quadrato

# IL PRINCIPIO DI SOSTITUZIONE DI LYSKOV

## ESEMPIO

- Cosa potrebbe succedere…

```
public class TestRectangle{
    public static void testLSP(Rectangle r){
        r.setWidth(4.0);
        r.setHeight(5.0);
        System.out.println("width 4, height 5, Area " +
                           r.Area());
    }

    public static void main(String args[] ){
        Rectangle r = new Rectangle(1.0,1.0);
        Square s = new Square(1.0);
        testLSP(r);
        testLSP(s);
    }
}
```

# IL PRINCIPIO DI SOSTITUZIONE DI LYSKOV

## ESEMPIO

- Matematicamente, un quadrato è un rettangolo, ma un "oggetto" quadrato non è un "oggetto" rettangolo perché (in questo caso) il comportamento dei due oggetti non coincide.
- Se il comportamento non coincide, non possiamo usare il polimorfismo.
- **Suggerimento**: se dei semplici metodi come setWidth e setHeight necessitano di override, ricontrollate l'ereditarietà..

# IL PRINCIPIO DI SOSTITUZIONE DI LYSKOV

## CONCLUSIONE

- LSP chiarisce che la relazione IS-A è incentrata sul comportamento di un oggetto
- Perché LSP sia valido, tutte le sottoclassi devono avere lo stesso comportamento che ci si aspetta dalla base class.
- Un subtype non deve avere più constraints del base type, perché deve essere usato al posto del base type.
- Ciò che LSP ci garantisce è che una subclass possa sempre essere usata al posto della sua base class!

“

# UNA CLASSE DOVREBBE DIPENDERE DALLE ASTRAZIONI, NON DA CLASSI CONCRETE

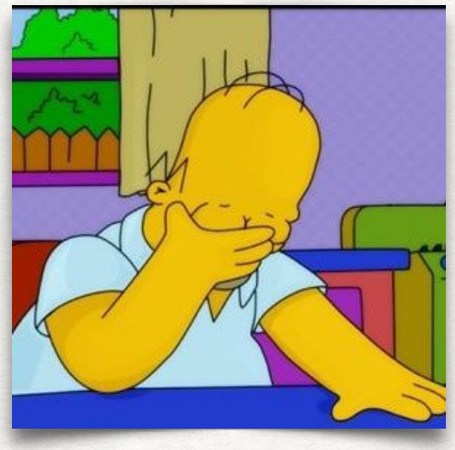
# DEPENDENCY INVERSION PRINCIPLE (DIP)

*Principio #6*

”

# DIP

- “High level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions”



- Detto così non è molto chiaro…

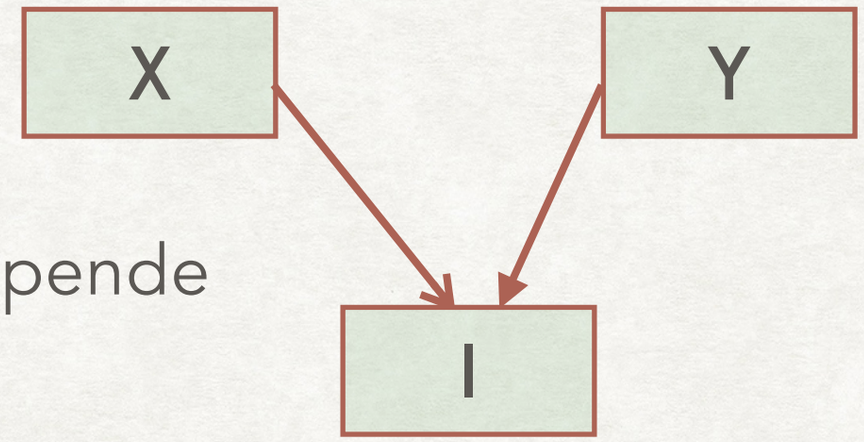
- In modo meno rigoroso:
  - nessuna variabile può avere un puntatore o una referenza ad una classe concreta
  - nessuna classe dovrebbe derivare da una classe concreta
  - nessun metodo dovrebbe fare override di un metodo implementato da una delle sue base class

# DIP

## ESEMPIO

- un oggetto x di classe X chiama un metodo di un oggetto y di classe Y: la classe X dipende da Y


- La dipendenza si può invertire introducendo una terza classe, una interfaccia I che contiene i metodi che X può chiamare di Y
- Y va cambiata perché implementi I
- X e Y dipendono da I, mentre I non dipende da nulla




- questo modo di fare viene detto "inversion of control" oppure "dependency inversion"

SE UNA CLASSE/MODULO È CONCRETO MA ESTREMAMENTE STABILE, POSSIAMO IGNORARE DIP...