LEZIONE 17
CLASS DIAGRAMS 3
Interfacce

Ingegneria del Software e Progettazione Web

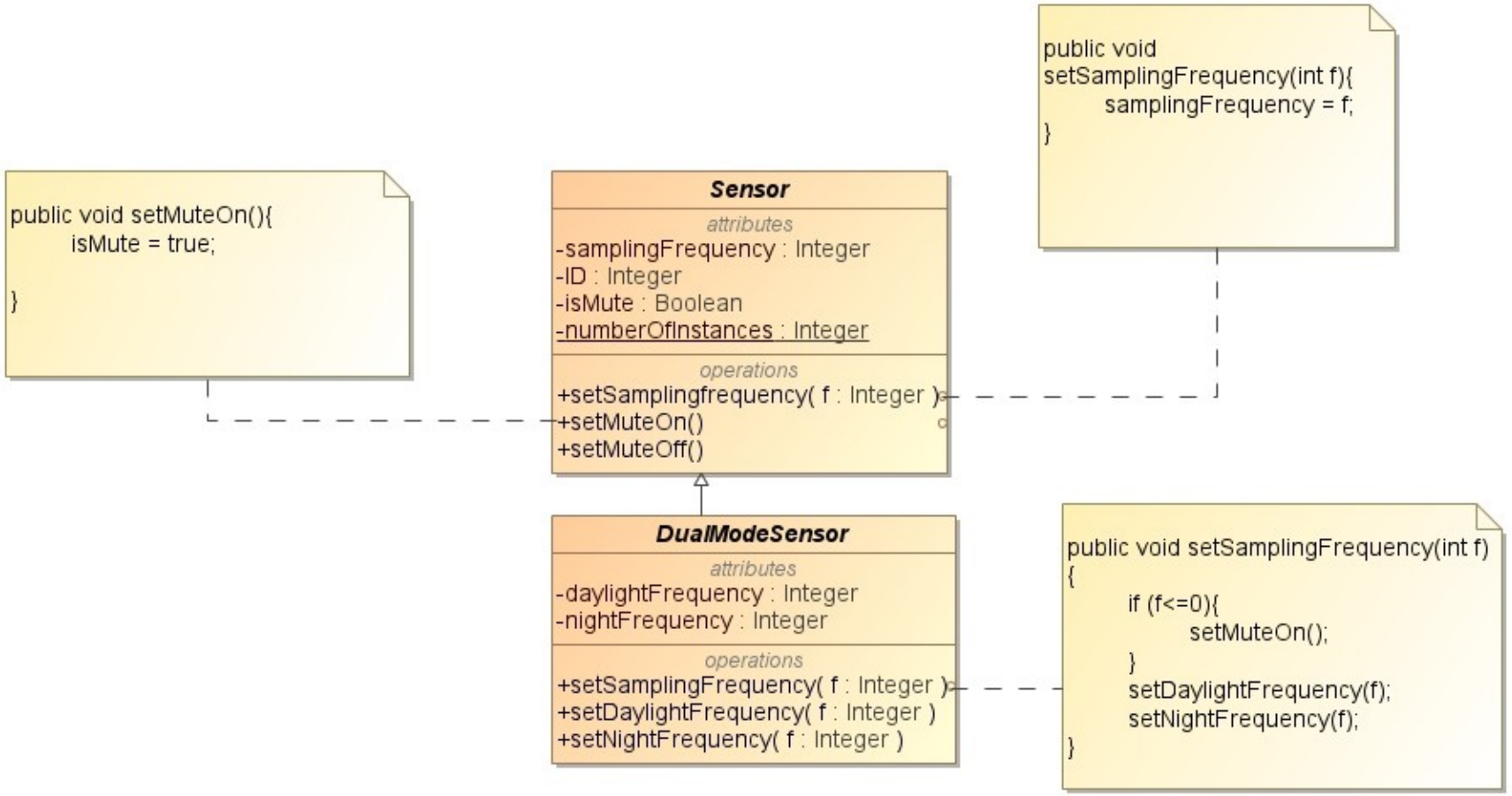
Università degli Studi di Tor Vergata - Roma

Guglielmo De Angelis
guglielmo.deangelis@isti.cnr.it

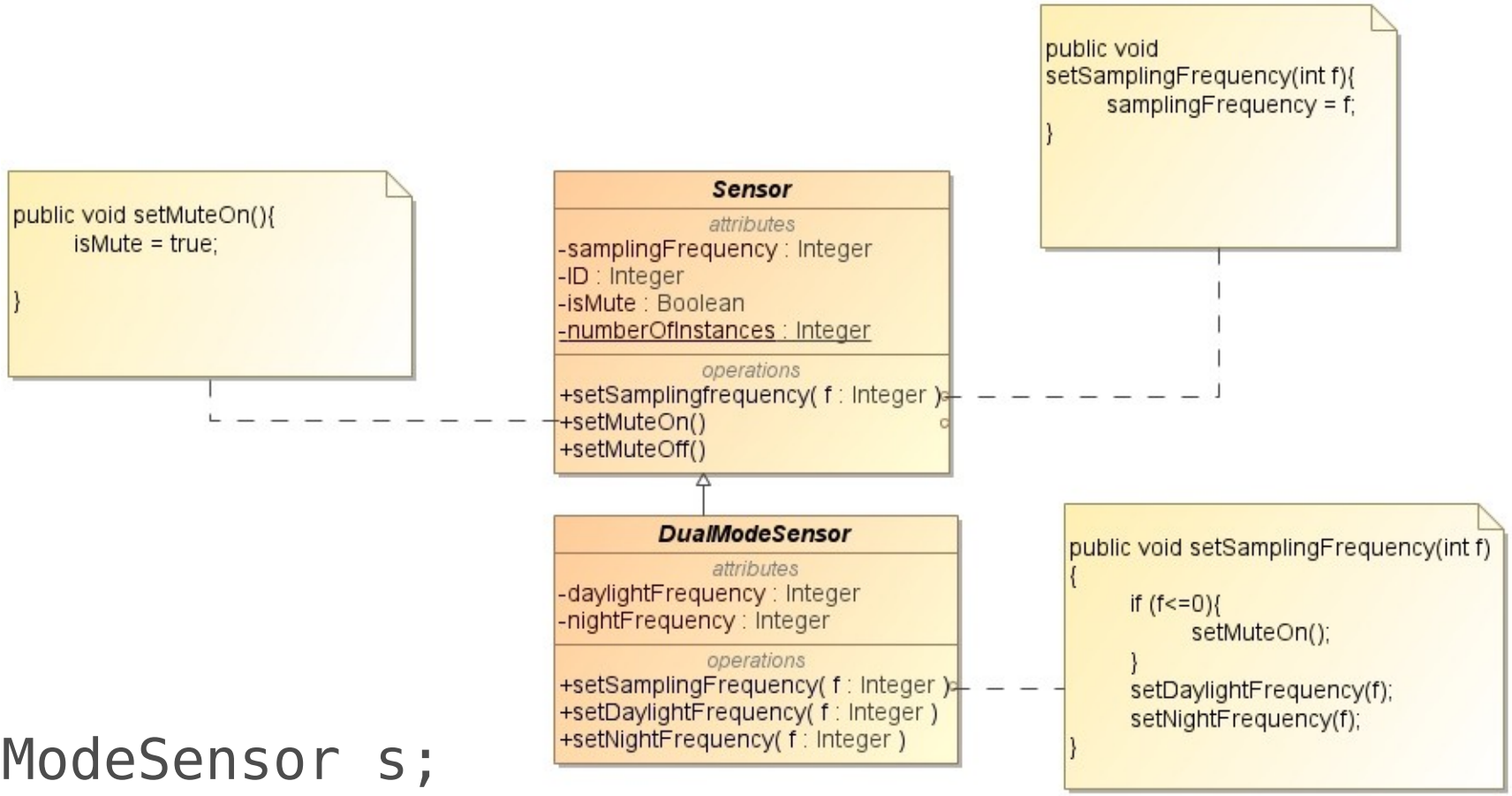
the fragile base-class problem – 1

- definizione dal Java White Paper (lezione 18-19)
  - *“Any time you add a new method or a new instance variable to a class, any and all classes that reference that class will require a recompilation, or they'll break”*
- definizione più in generale (vedi Sabané, A., Guéhéneuc, YG., Arnaoudova, V. et al. Empir Software Eng (2017) 22: 2612. https://doi.org/10.1007/s10664-016-9448-2)
  - *“Fragile Base Class Problem (FBCP) is rised when changes to either the sub-class or the base-class could cause the instances of either classes to behave unexpectedly”*
  - *“We define an Fragile Base Class Structures (FBCS) as two classes in an inheritance relationship, not necessarily a direct relationship, and with specific method declarations and definitions. An FBCS is the location where the FBCP can occur if, for example, the sub-class overrides a method of the base-class and introduces a mutual recursion.”*

the fragile base-class problem – 2

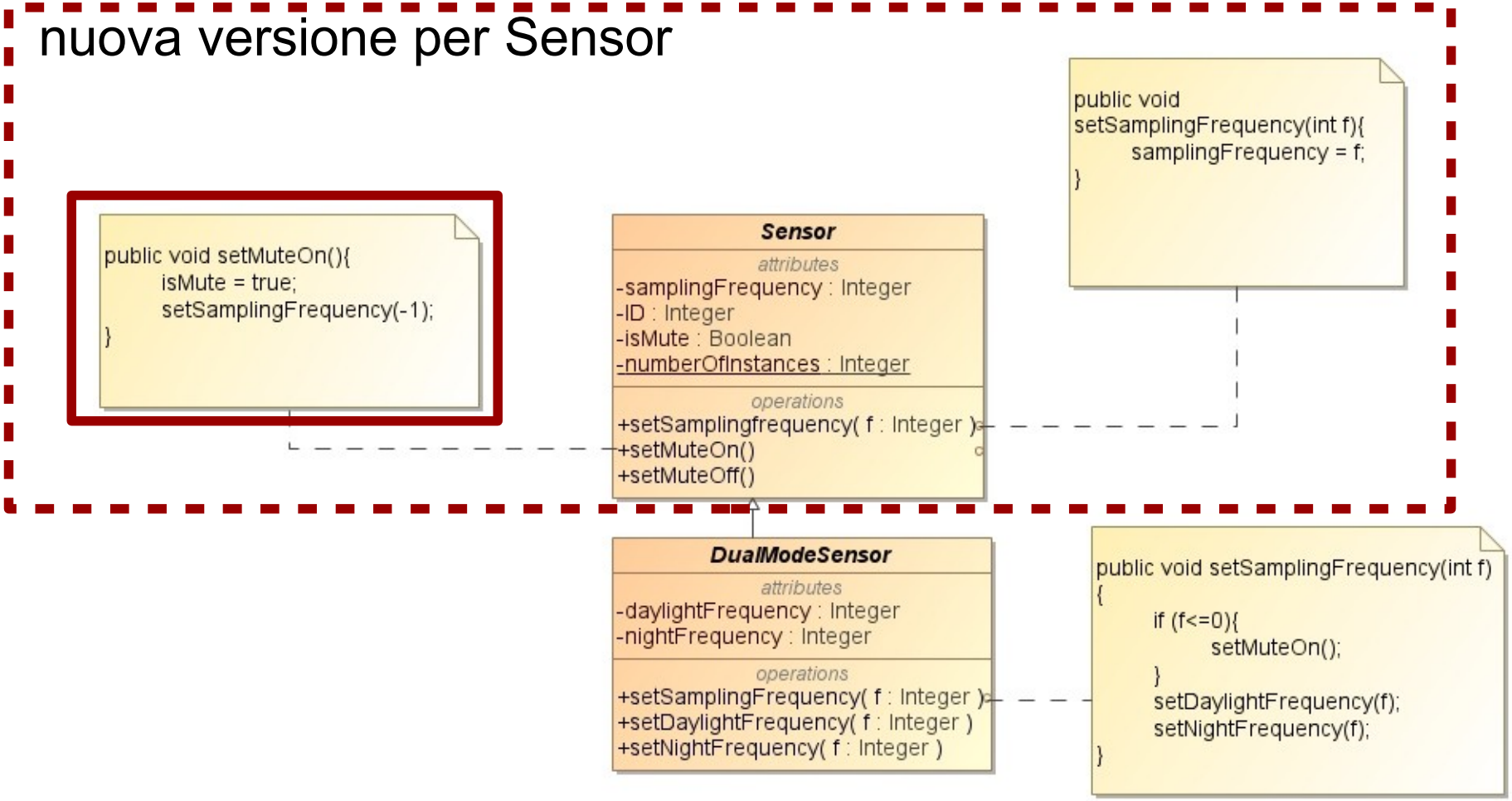


the fragile base-class problem – 2

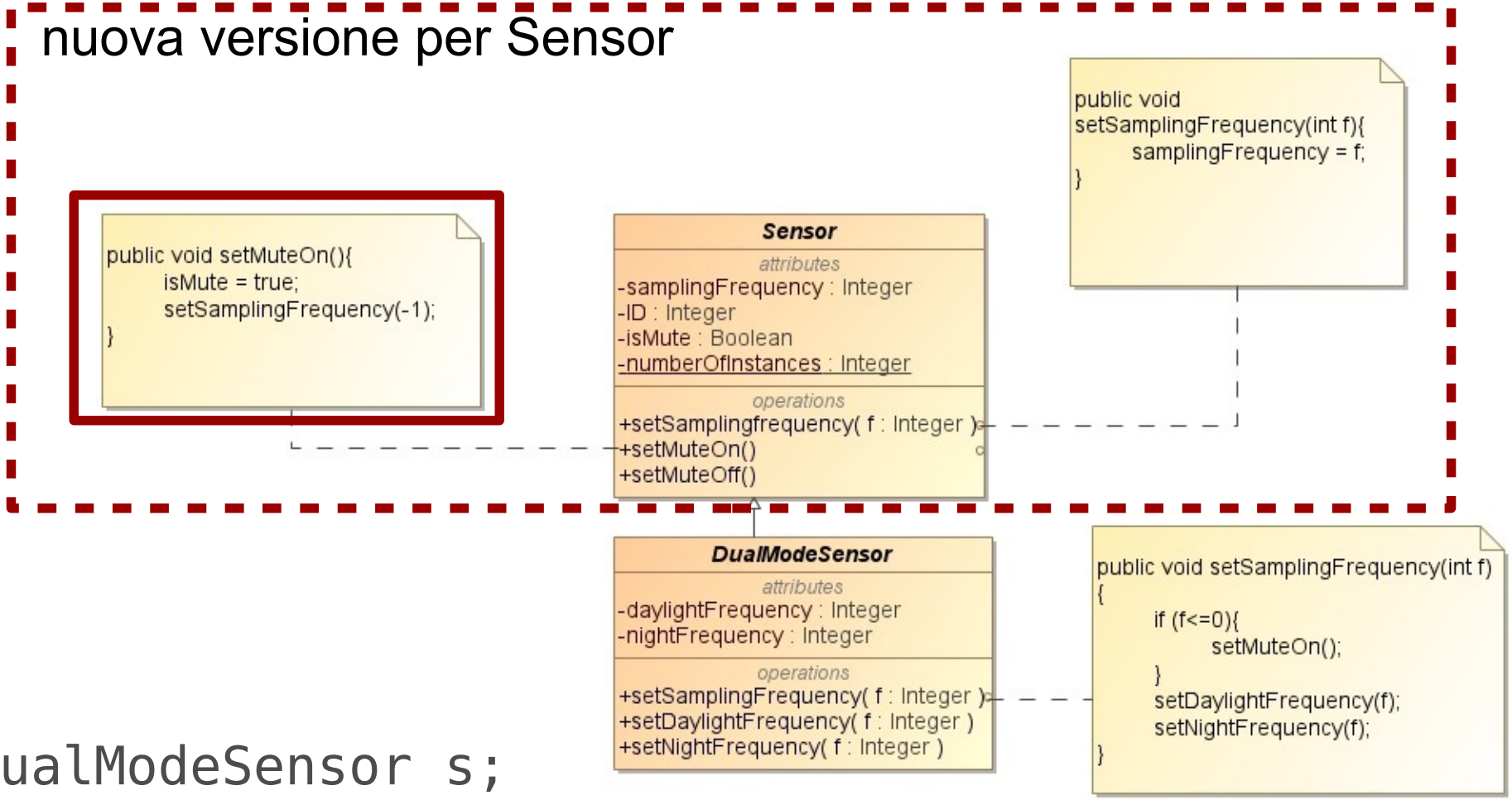



```
DualModeSensor s;
    …
s.setSamplingfrequency(-100);
```

the fragile base-class problem – 3



the fragile base-class problem – 3




```
DualModeSensor s;
    …
s.setSamplingfrequency(-100);
```

???

the fragile base-class problem – 3

nuova versione per Sensor

come gestire e progettare soluzioni che mitighino il verificarsi di questo problema?

NOTA: mitighino e NON risolvano

```
DualModeSens
    …
s.setSamplingfrequency(-100);
```

???

class diagrams

- struttura statica del sistema:
  - ha una rappresentazione logica a grafo
  - nodi + relazioni

class diagrams

- struttura statica del sistema:
  - ha una rappresentazione logica a grafo
  - nodi + relazioni
- un nodo modella una “classe” che rappresenta:
  - una entità del dominio
  - elementi che non fanno parte del dominio ma utili nell'ingegnerizzazione del sistema
- una classe è caratterizzata
  - da un nome
  - degli attributi
  - delle operazioni sugli attributi
- sono lo stesso concetto in O.O. (introdotto nelle prossime slide)
  - (semplificando) rappresentano un tipo di dato && l'insieme di operazioni che ne definiscono i modi di interazione
  - dipendentemente dalla “vista” di riferimento corrispondono ad una implementazione dell'entità

interfaccia

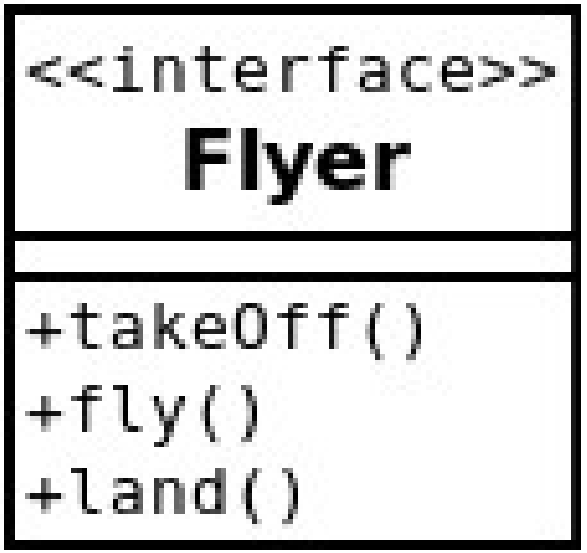
- collezioni di operazioni che sono utilizzate per specificare un servizio di una classe o di un componente
- definisce solo la segnatura delle operazioni
  - dichiara semplicemente un contratto
  - non prescrive mai una particolare implementazione
- le operazioni possono avere attributi di visibilità (come nelle classi)
- essendo una specifica parziale non può essere istanziata
- non tutti i linguaggi hanno interfacce
  - C++ NO , Java SI

interfaccia

- collezioni di operazioni che sono utilizzate per specificare un servizio di una classe o di un componente
- definisce solo la segnatura delle operazioni
  - dichiara semplicemente un contratto
  - non prescrive mai una particolare implementazione
- le operazioni posso[illegible]avere attributi di visibilità (come nelle classi)
- [illegible]ere
-

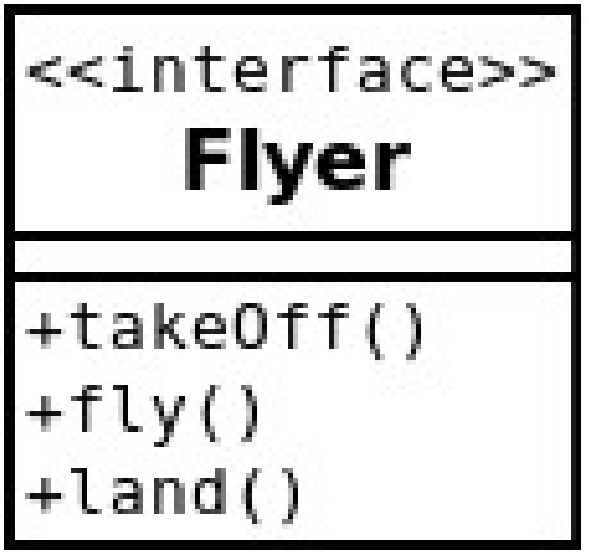
A partire dalla versione 8, JAVA introduce la possibilità di specificare comportamenti di default per le operazioni prescritte da una interfaccia
http://www.tutorialspoint.com/java8/java8_default_methods.htm

interfacce – UML



interfacce – UML && Java

```
public interface Flyer{
   /** Dichiarazione
      * dell'interfaccia
   */
   public void takeOff();
   public void fly();
   public void land();
}
```



interfacce VS classi astratte

- entrambi i concetti
  - modellano operazioni che non sono associate a nessun metodo
  - impongono alle sotto classi/interfacce l'override delle operazioni (svincolate) che definisco
  - collezioni di operazioni che sono utilizzate per specificare un servizio di una classe o di un componente
  - non consentono la creazione diretta di istanze

interfacce VS classi astratte

- entrambi i concetti
  - modellano operazioni che non sono associate a nessun metodo
  - impongono alle sotto classi/interfacce l'overriding delle operazioni (svincolate) che definisco
  - collezioni di operazioni che sono utilizzate per specificare un servizio di una classe o di un componente
  - non consentono la creazione diretta di istanze

da questo punto di vista i due concetti sembrerebbero abbastanza “SOVRAPPOSTI” ...

interfacce VS classi astratte

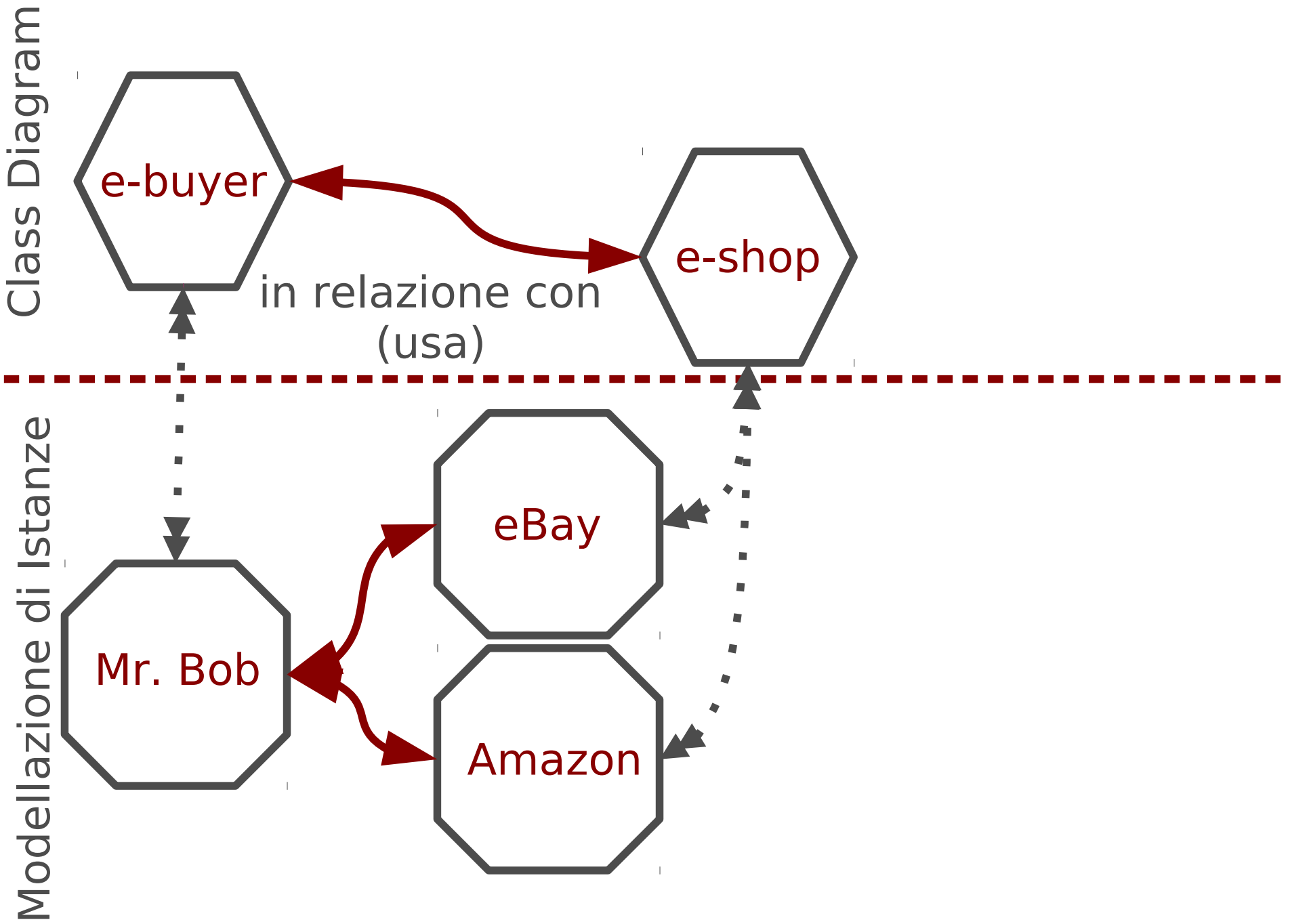
- con le interfacce
  - tutte le operazioni non hanno associato alcun metodo
  - non definiscono nessun attributo
- con le classi astratte
  - anche una sola operazione può essere astratta (i.e. non associata a nessun metodo)
  - possono definire degli attributi

interfacce VS classi astratte

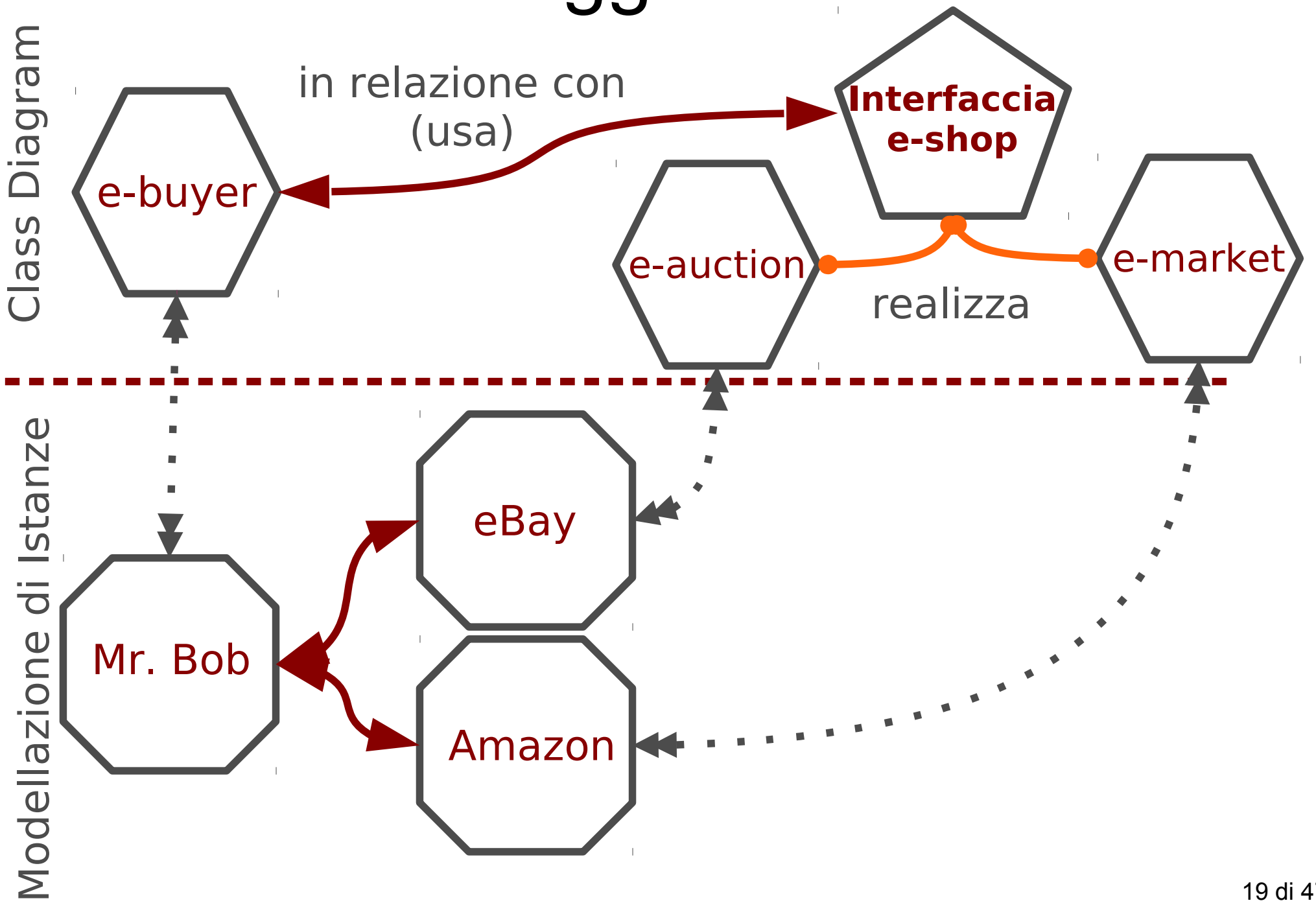
- l'idea di interfaccia
  - modella (attraverso le operazioni) un modo d'uso di (sotto-)sistema
  - generalmente rappresenta una vista del sistema piuttosto che una sua parte

- l'idea di classi astratta
  - modella una parte della struttura statica di un sistema
    - entità del dominio parzialmente definita
    - elementi parzialmente definiti utili nell'ingegnerizzazione del sistema
  - prevede la specifica un concetto di STATO (definito dagli attributi) per l'elemento modellato

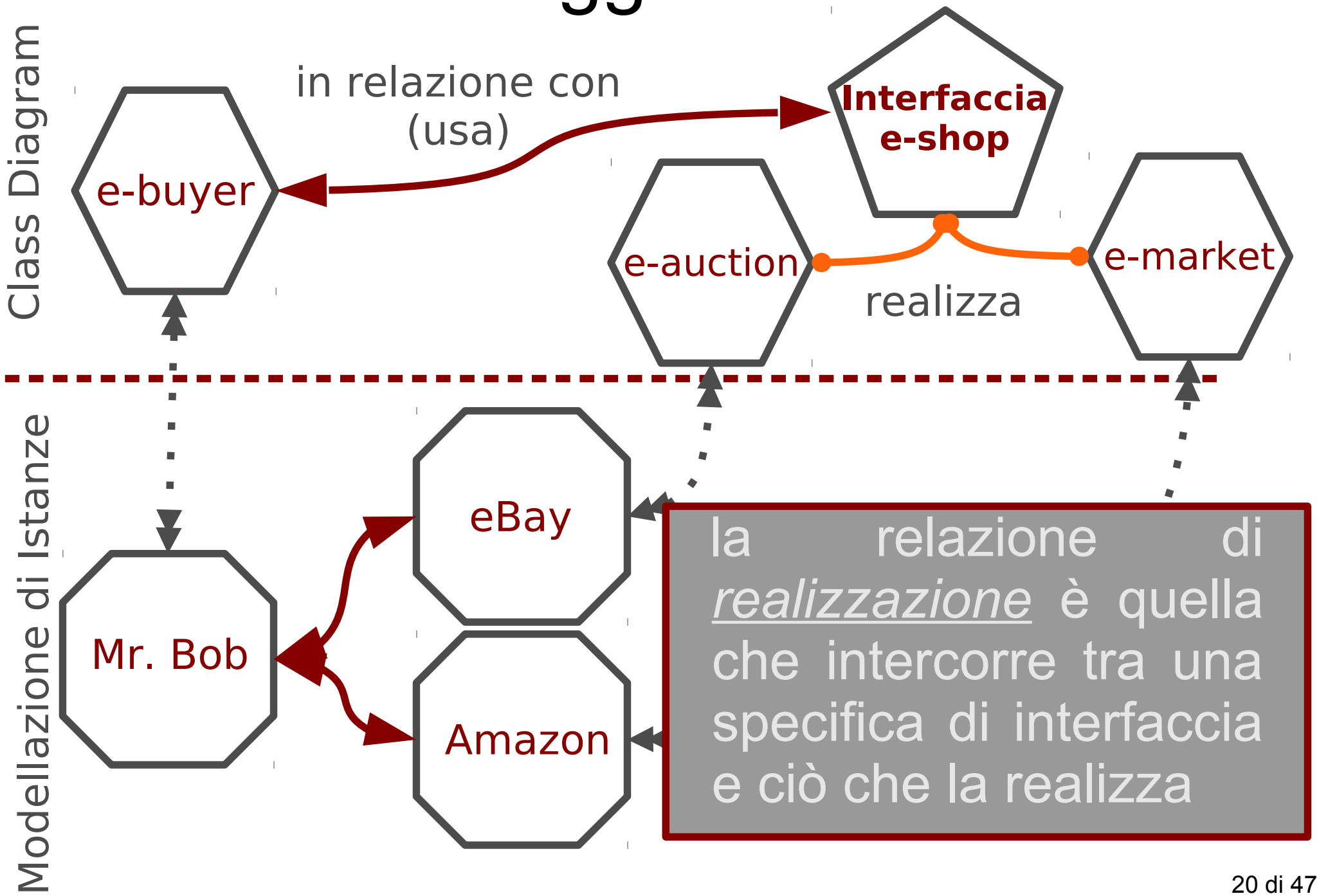
classi VS oggetti ...



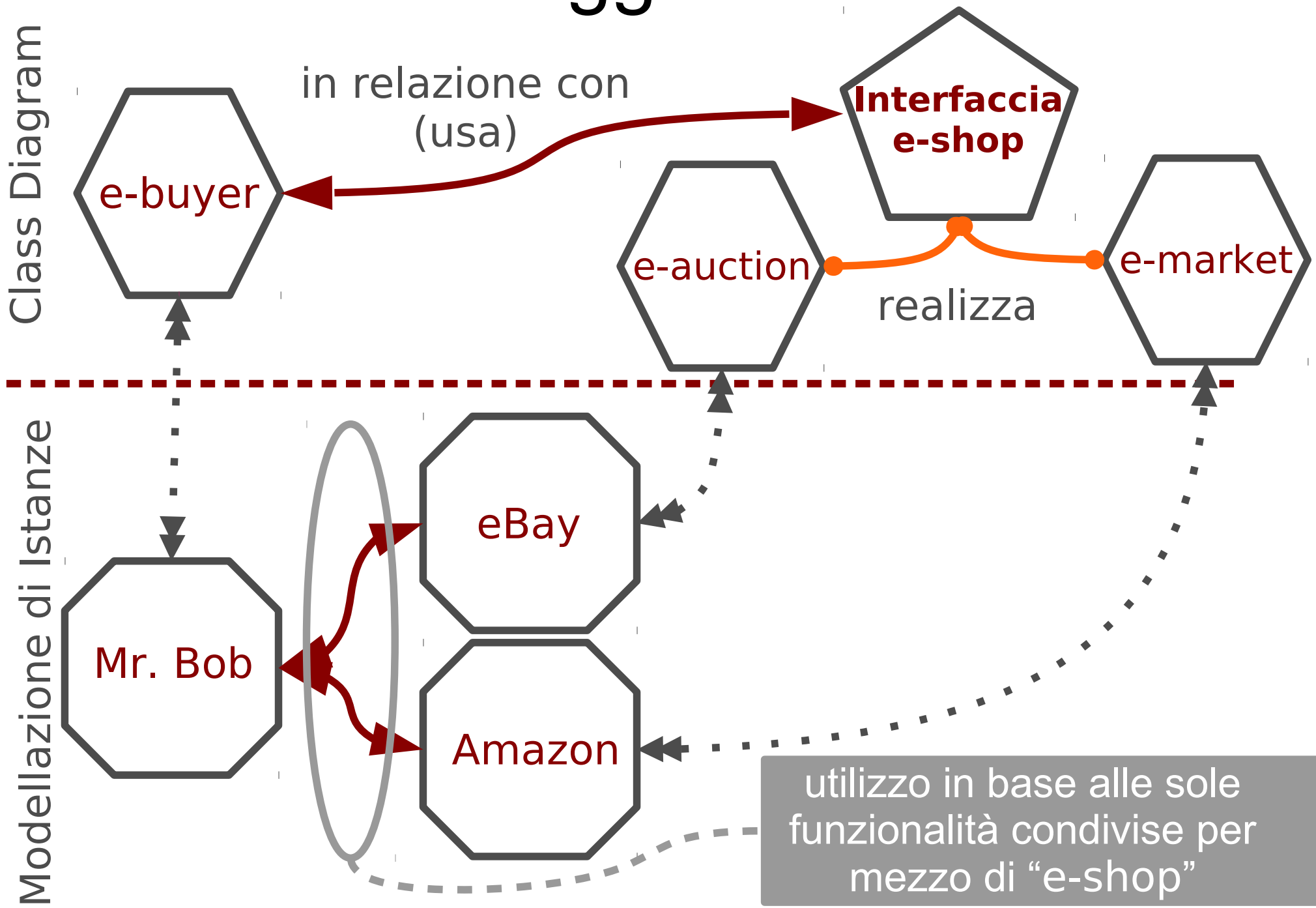
classi VS oggetti VS interfacce



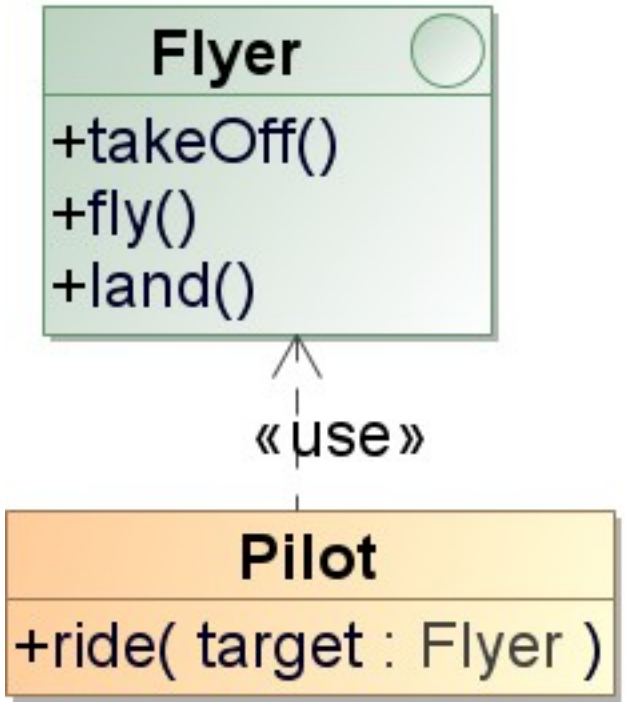
classi VS oggetti VS interfacce



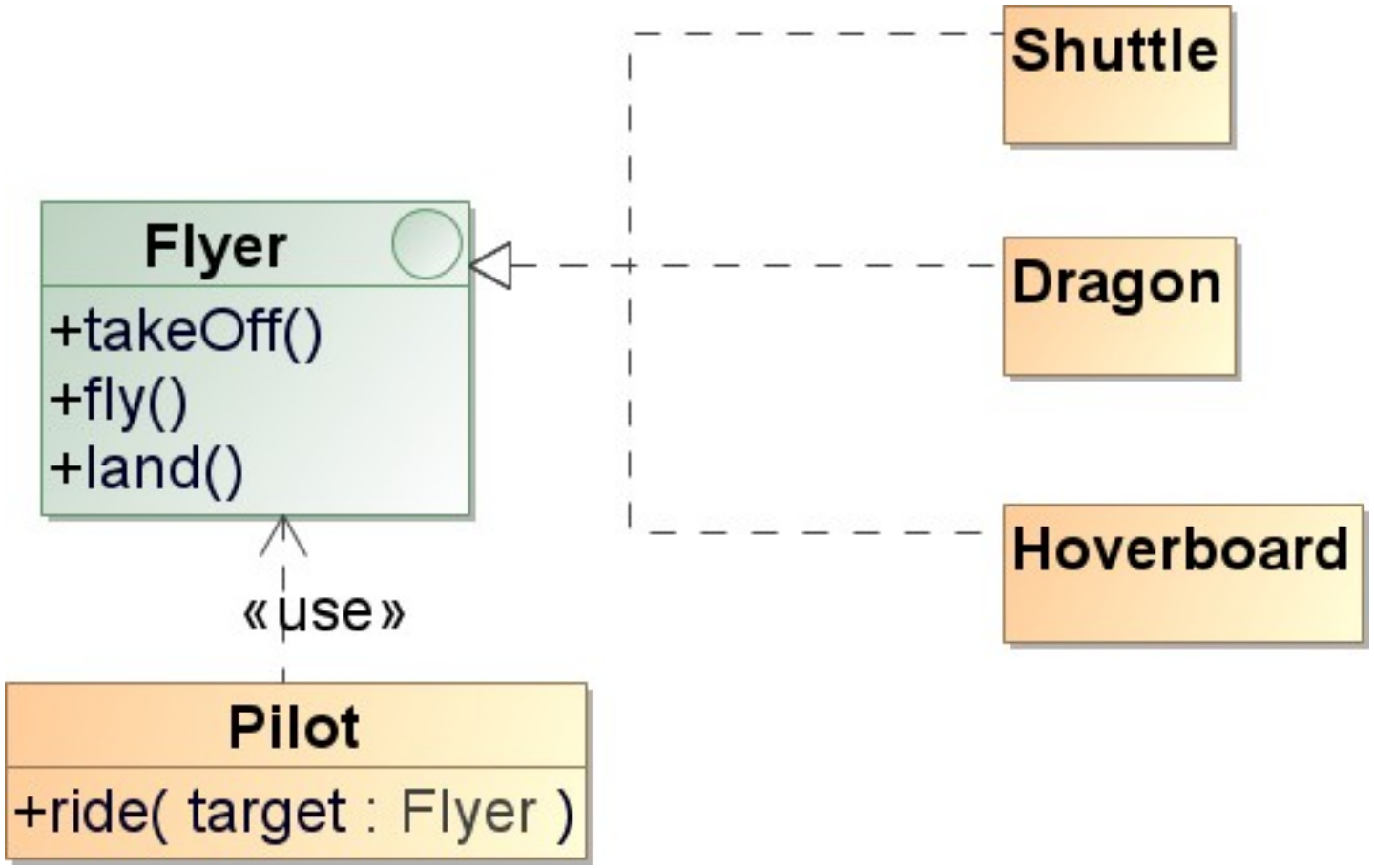
classi VS oggetti VS interfacce



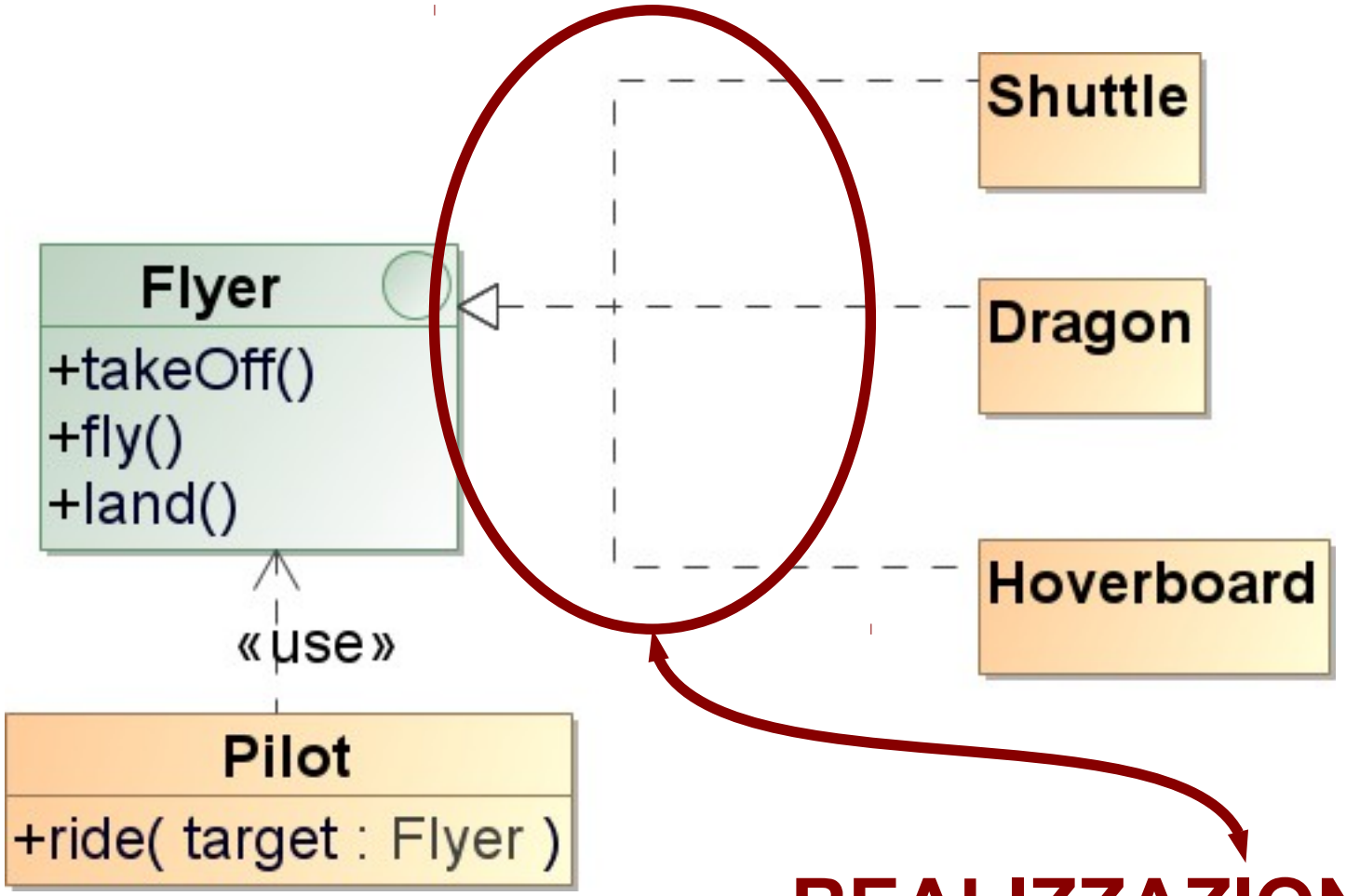
interfacce – esempio “arcade game”



interfacce – esempio “arcade game”



interfacce – esempio “arcade game”



REALIZZAZIONE in UML

interfacce – map in Java

```
public interface Flyer{
   /** Dichiarazione
      * dell'interfaccia
   */
   public void takeOff();
   public void fly();
   public void land();
}
```

interfacce – map in Java

```
public interface Flyer{
   /** Dichiarazione
      * dell'interfaccia
   */
   public void takeOff();
   public void fly();
   public void land();
}
        public class Dragon implements Flyer{
            /** Dichiarazione
             * dei metodi per l'interfaccia
             *  e corpo della classe
            */
        }
```

usare le interfacce per :

- inter-collegare sistemi diversi
  - il sistema è strutturato in base all'insieme dalle interfacce definite dai diversi sottosistemi
- definizione di architetture astratte
  - basate sulle interazioni
- aumentare la modularità di un sistema
  - gran parte delle attività di progettazione si concentrano nell'individuazione e modellazione delle principali forme di interazione per mezzo di interfacce

ereditarietà VS realizzazione

ereditarietà VS realizzazione – 1

- con l'ereditarietà si ereditano caratteristiche comuni
  - specifica: le operazioni pubbliche della classe base
  - implementazione: gli attributi, le relazioni, ed i metodi della classe
- con realizzazione si accettano specifiche di interazione
  - le operazioni pubbliche definite dall'interfaccia

ereditarietà VS realizzazione – 2

- l'ereditarietà deve essere usata esclusivamente se tra 2 classi esiste una ovvia relazione “ is a kind of ”
- caratteristiche e svantaggi:
  - è la forma più forte di interdipendenza tra classi
  - l'incapsulamento nella gerarchia è più debole
  - un cattivo uso può causare il problema della “fragilità della classe base”
    - modifiche che si ripercuotono su tutta la gerarchia

ereditarietà VS realizzazione – 3

- l'ereditarietà è necessaria se e solo se lo scopo è quello di ereditare anche dettagli implementativi della superclasse
  - nasce come la forma più basilare di riuso
  - con il tempo ha assunto forti caratteristiche semantiche (i.e. principio di sostituibilità)
- la realizzazione è utile quando si vuole definire un contratto senza accettare vincoli su dettagli implementativi
  - le interfacce non offrono nessuna possibilità di riuso
  - consente di definire un contratto e garantire che sia rispettato (almeno sintatticamente)
  - più flessibile e robusta dell'ereditarietà

ereditarietà VS realizzazione – Java

- non supporta ereditarietà multipla tra le classi
- consente che una classe implementi più interfacce contemporaneamente

- razionale :
  - una classe ha uno ed un solo tipo
    - generalizzazione → is a kind of
  - una classe può manifestarsi attraverso viste differenti
    - ogni vista definisce un modo d'uso della classe

ancora su esempio “arcade game”

```
public interface Flyer {
  public void fly();
  public void takeOff();
  public void land();
}
```

ancora su esempio “arcade game”

```
public interface Flyer {
  public void fly();
  public void takeOff();
  public void land();
}

public interface Fighter {
  public void fight();
}

public interface Swimmer {
  public void swim();
}
```

ancora su esempio “arcade game”

```
public interface Flyer {

   public void fly();

   public void takeOff();

   public void land();
}


public interface Fighter {

   public void fight();
}


public interface Swimmer {

   public void swim();
}
```

```
public class SuperHero implements
Flyer, Fighter, Swimmer {

   public void fly(){ … }

   public void takeOff(){ … }

   public void land(){ … }

   public void fight(){ … }

   public void swim(){ … }
}
```

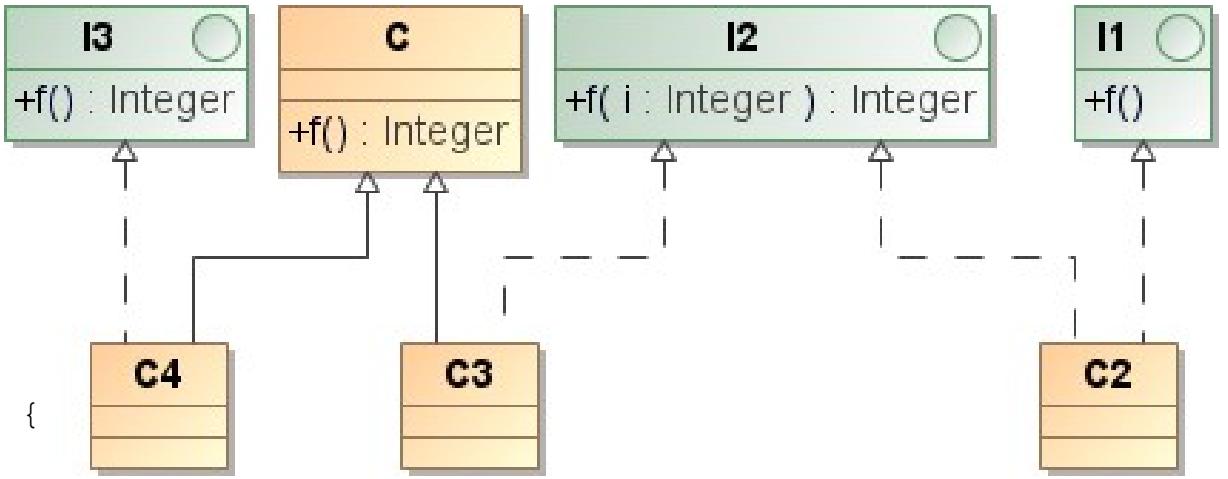
ereditarietà VS realizzazione – Java

- non supporta ereditarietà multipla tra le classi
- consente che una classe implementi più interfacce contemporaneamente
- razionale:
  - ereditarietà e realizzazione sono concetti
    - generalizzazione → is a kind of
  - una classe può manifestarsi attraverso viste differenti
    - ogni vista definisce un modo d'uso della classe

bisogna gestire opportunamente eventuali casi di collisione di nomi quando si combinano ereditarietà e realizzazione

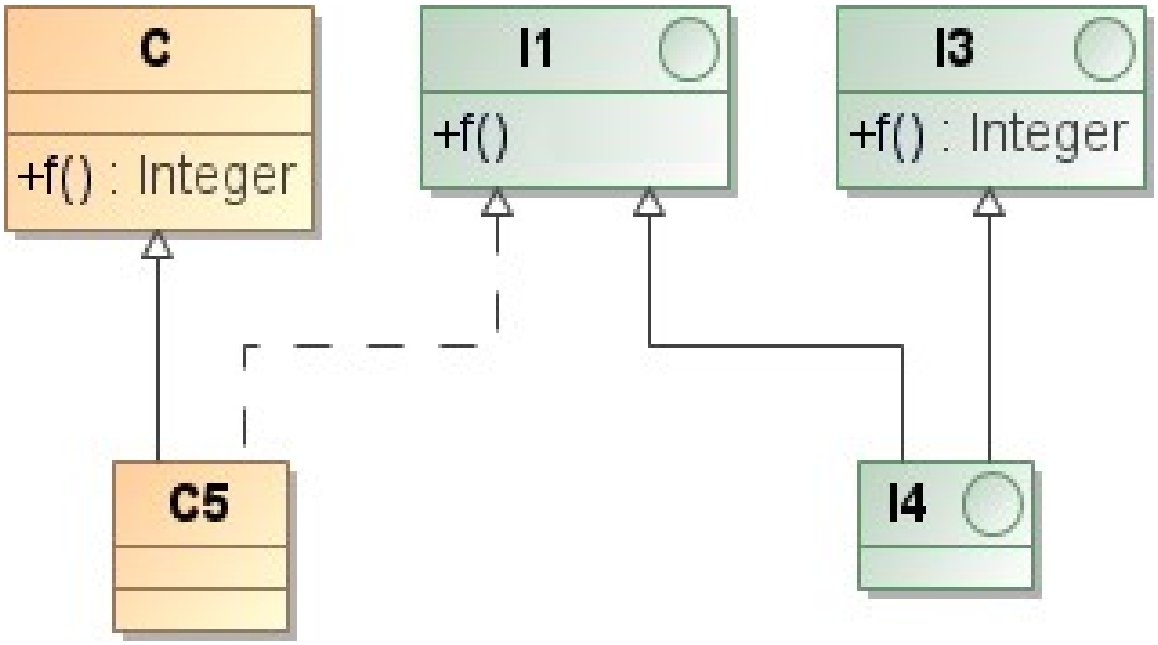
InterfaceCollision.java From “Thinking in Java”

```
public interface I1 { void f(); }
public interface I2 { int f(int i); }
public interface I3 { int f(); }
public class C {
  public int f() { return 1; }
}

public class C2 implements I1, I2 {
  public void f() {}
  // overloaded
  public int f(int i) { return 1; }
}

public class C3 extends C implements I2 {
  // overloaded
  public int f(int i) { return 1; }
}

public class C4 extends C implements I3 {
  // Identical, no problem:
  public int f() { return 1; }
}
```



InterfaceCollision.java From “Thinking in Java”

```
public interface I1 { void f(); }
public interface I2 { int f(int i); }
public interface I3 { int f(); }
public class C {
  public int f() { return 1; }
}
```



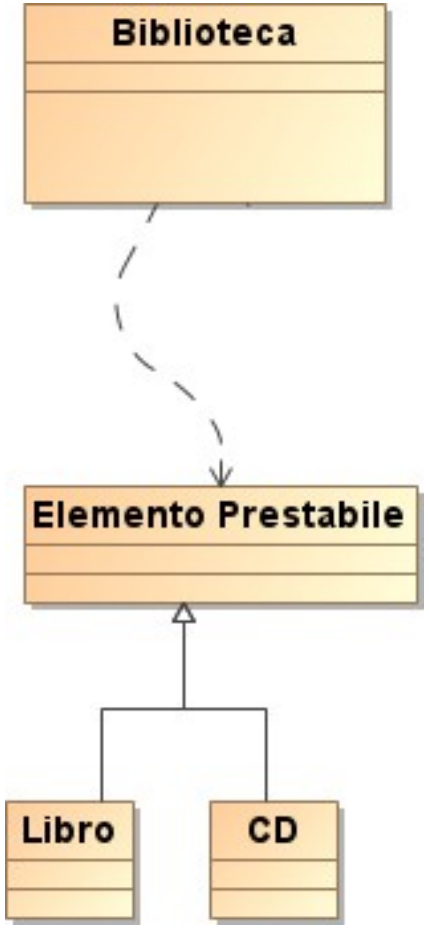

```
public class C5 extends C implements I1{
// ERROR!
    … …
}
```

```
public interface I4 extends I1, I3 {
// ERROR!
    … …
}
```

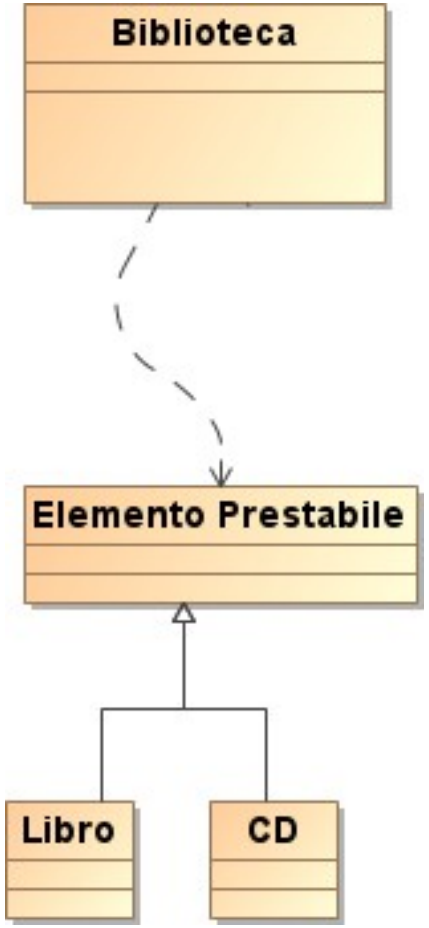
esempio – 1

modellare un semplice sistema di gestione per una biblioteca. Considerare che la biblioteca è in grado di gestire il prestito di *almeno* libri e CD

esempio – 2

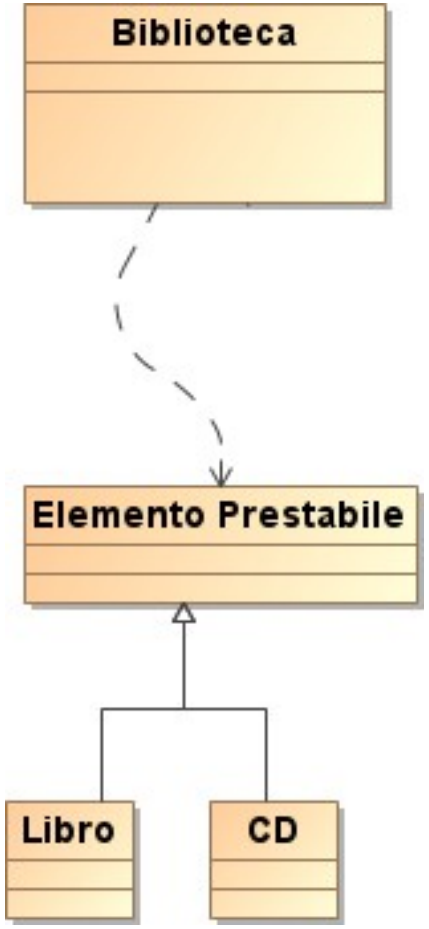


esempio – 2



il fatto che un Libro o che un CD siano *prestabili* è davvero una parte rappresentativa del *tipo* degli elementi modellati?
O forse rappresenta meglio un *ruolo* per gli elementi?

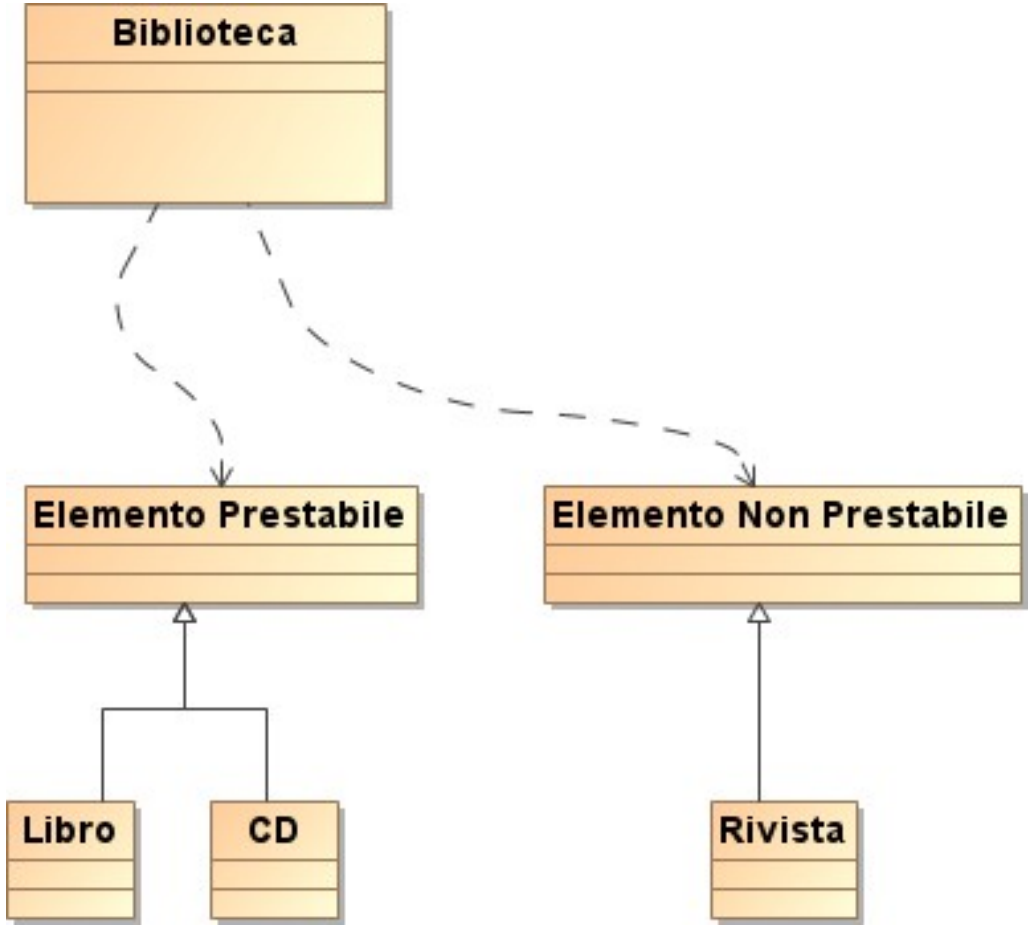
esempio – 2



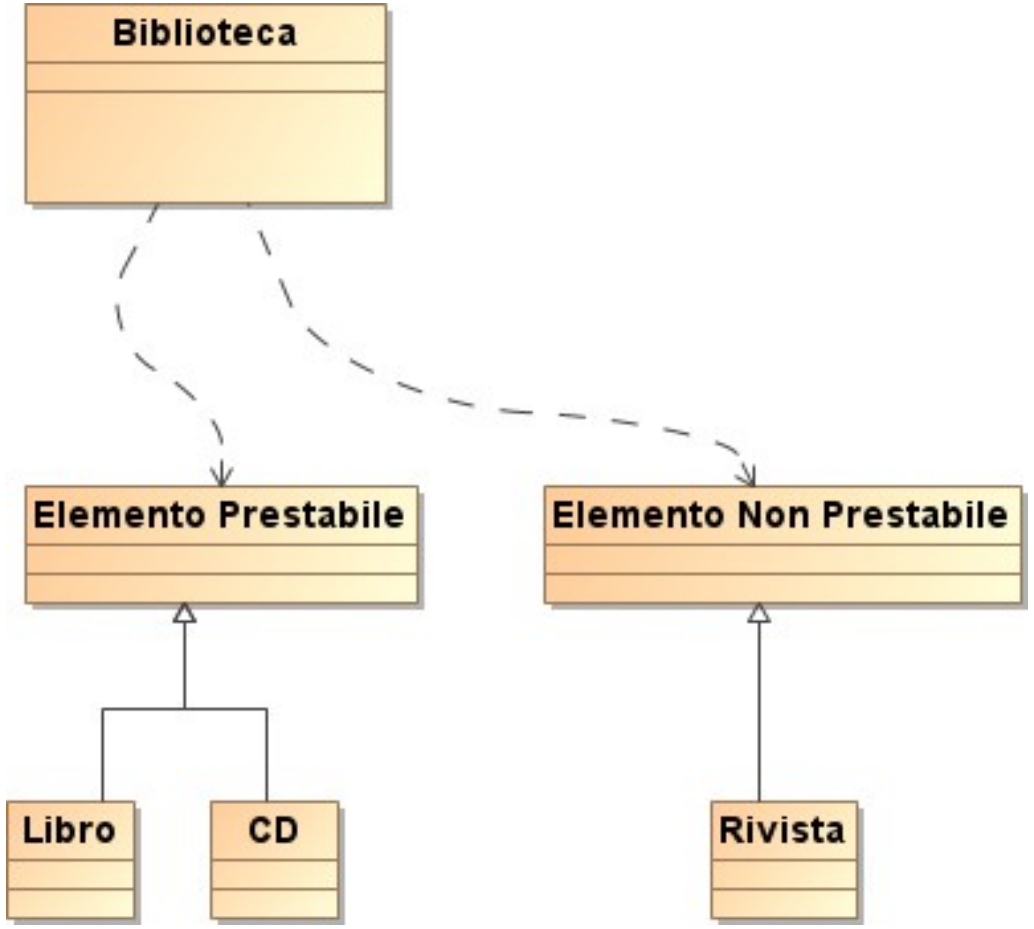
il fatto che un Libro o che un CD siano *prestabili* è davvero una parte rappresentativa del *tipo* degli elementi modellati? O forse rappresenta meglio un *ruolo* per gli elementi?

estendere il modello in modo da supportare anche la gestione elementi non prestabili come le Riviste

esempio – 3

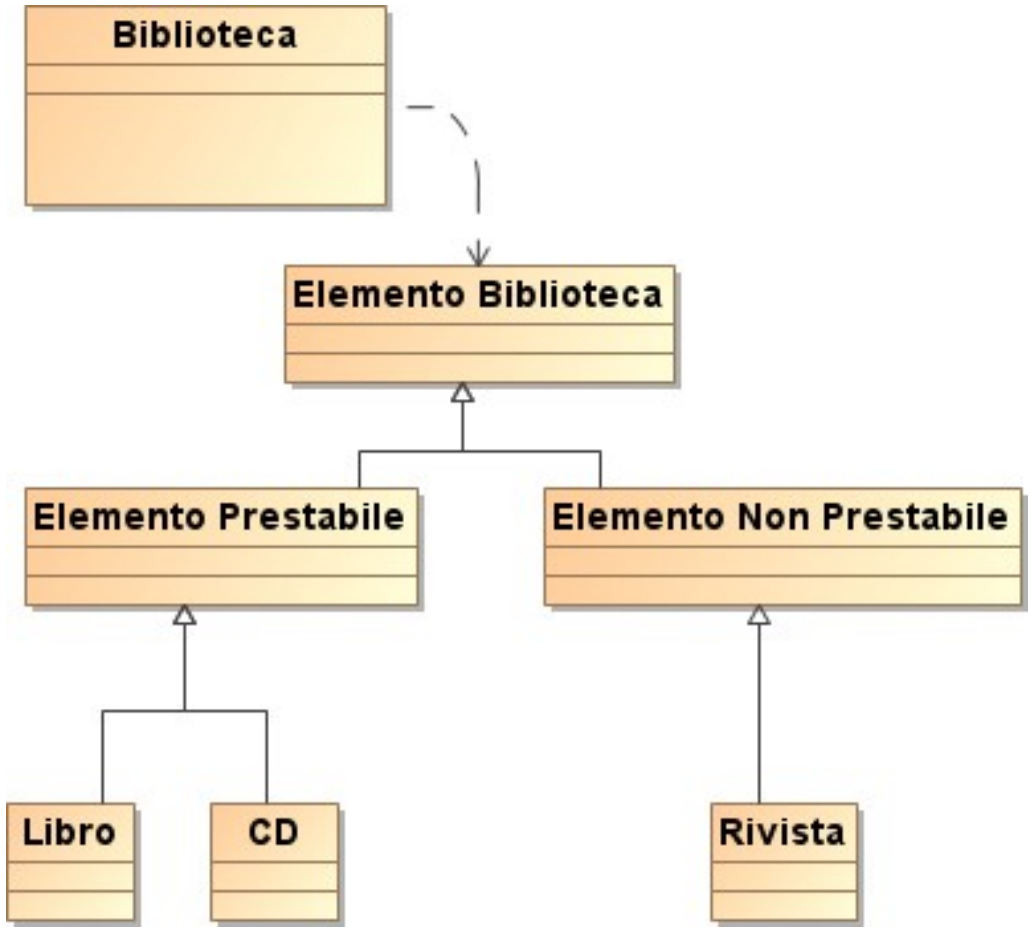

Biblioteca
Elemento Prestabile
Elemento Non Prestabile
Libro
CD
Rivista

esempio – 3

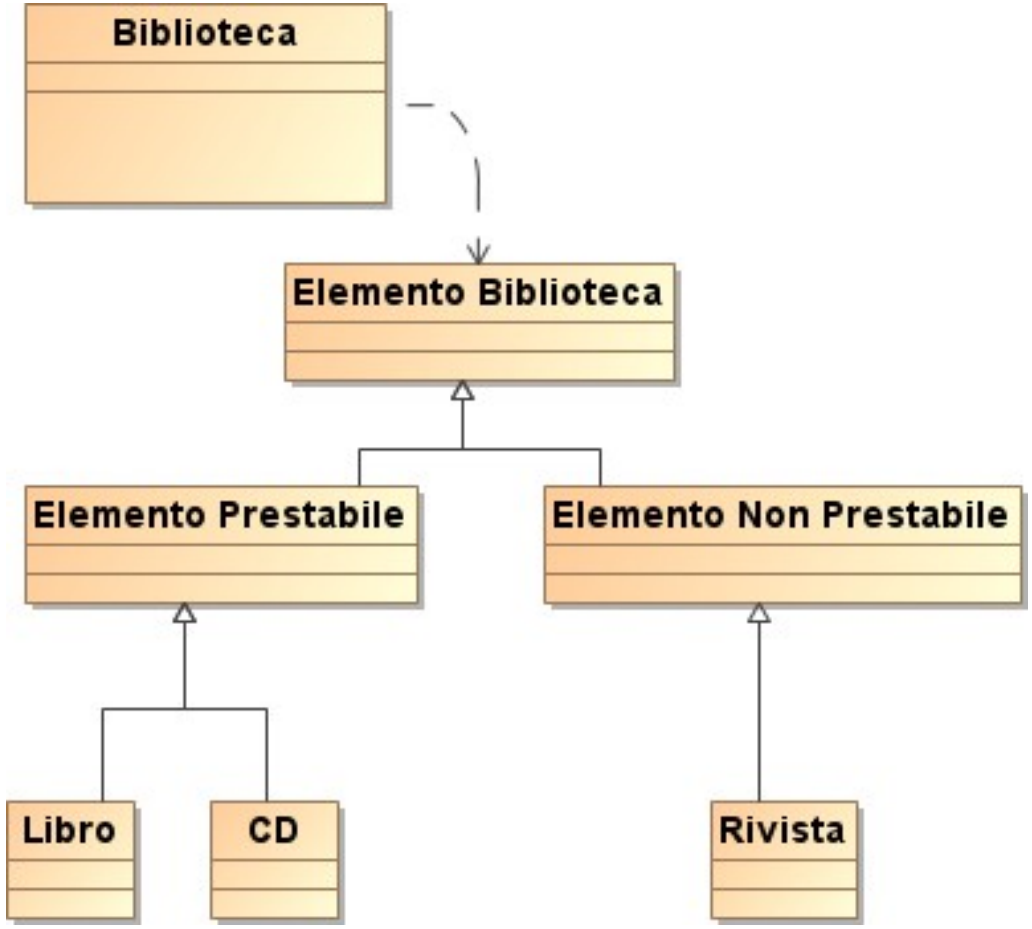


soluzione accettabile, ma non ottimale : Biblioteca gestisce Libro/CD e Rivista come tipi disgiunti … anche se differiscono solo sull'aspetto prestito

esempio – 4

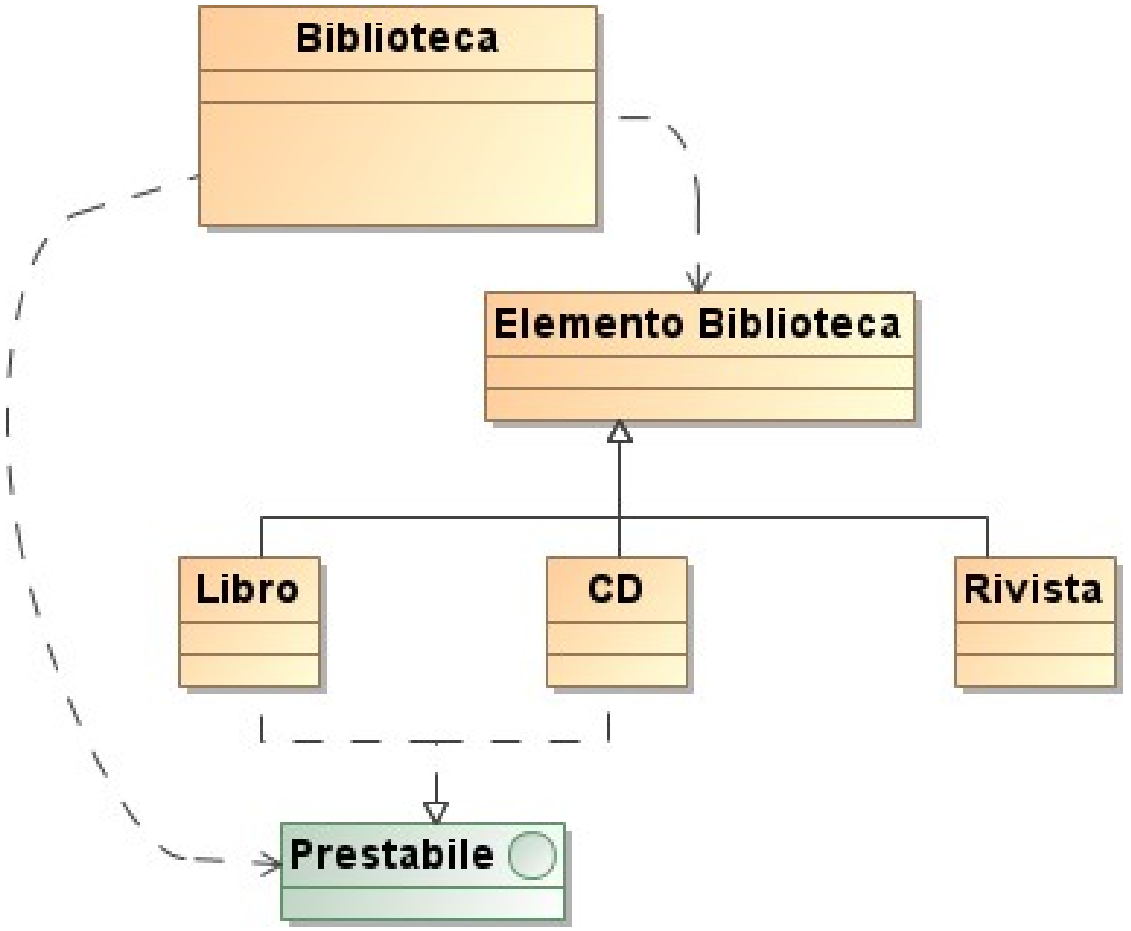

Biblioteca
Elemento Biblioteca
Elemento Prestabile
Elemento Non Prestabile
Libro
CD
Rivista

esempio – 4



meglio ma … la separazione tra Libro/CD e Rivista sembra avere origine da un "*modo d'uso*" delle classi più che da una loro effettiva tassonomia

esempio – 5


Biblioteca
Elemento Biblioteca
Libro
CD
Rivista
Prestabile