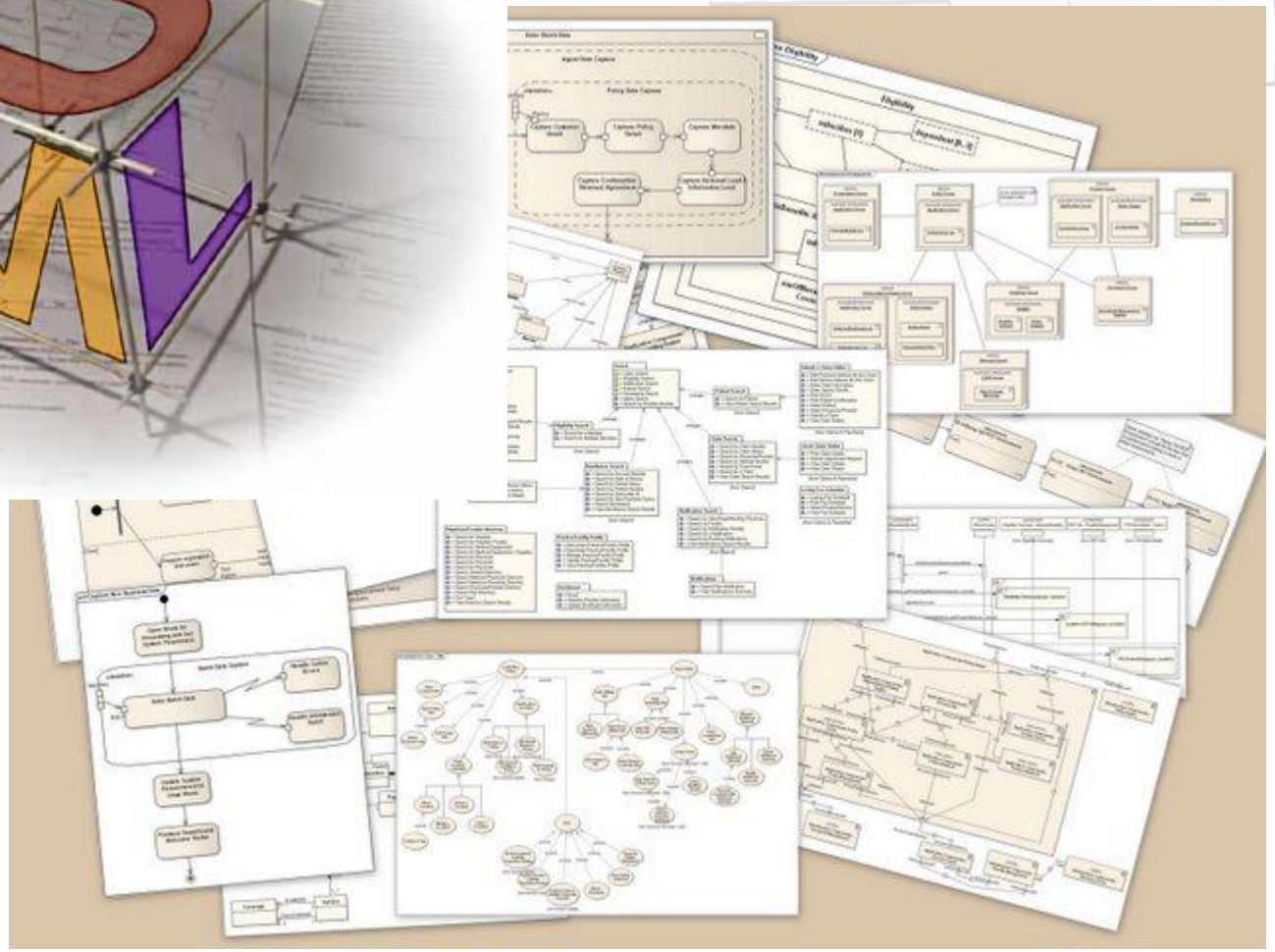
UML e i diagrammi di classe

Sintassi e Linee Guida

Dr. Andrea Baruzzo

andrea.baruzzo@dimi.uniud.it

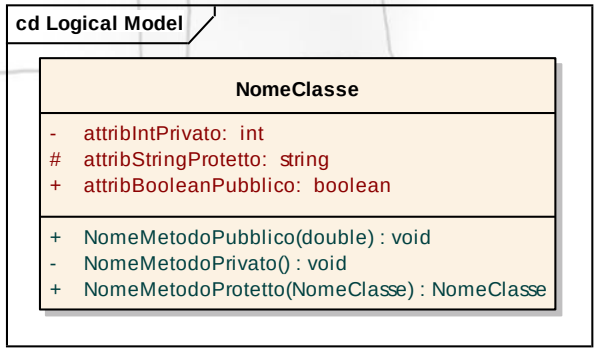


Agenda

1. Classi, attributi, metodi
2. Relazioni fra classi
3. Ereditarietà
4. Classi astratte e interfacce
5. Contenimento
6. Associazione e dipendenze
7. Package, vincoli e note

Classi, attributi, metodi

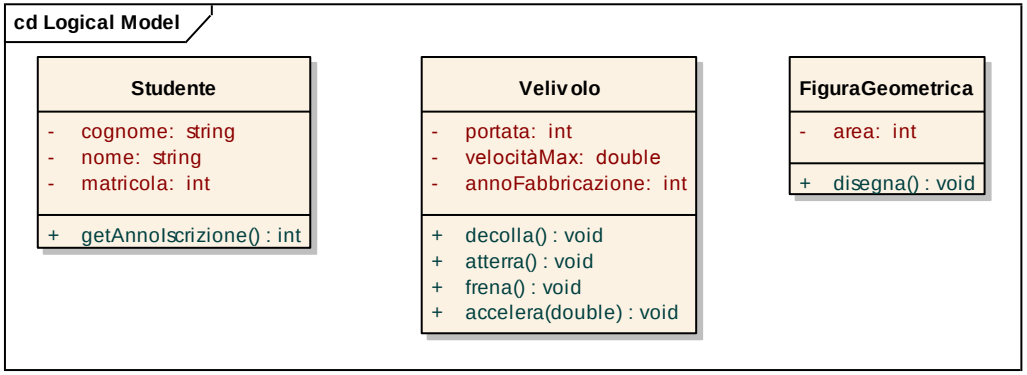
- Una classe in UML si rappresenta con un rettangolo avente tre **compartimenti**:
  - **Nome** della classe
  - **Attributi**
  - **Metodi**
- In realtà ci possono essere ulteriori compartimenti
  - Vincoli, note, ecc.



- **Visibilità**:
  - Pubblica (il default per i metodi)
  - Privata (tipica per gli attributi)
  - Protetta (ereditarietà protetta, visibilità legata al package in Java)
- Non è obbligatorio mostrare tutti i compartimenti
- Le classi modellano le **entità** del sistema

Alcuni esempi

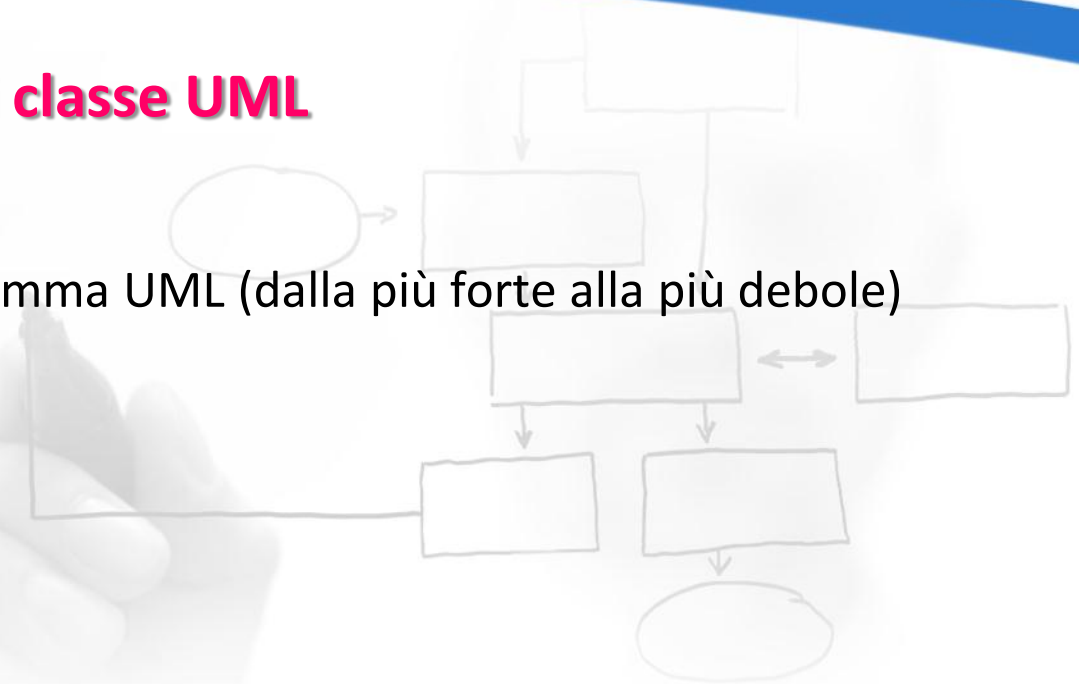
- Convenzioni sui nomi
  - Classe: lettera maiuscola iniziale per ogni parola
  - Attributi: lettera minuscola iniziale, maiuscola iniziale per ogni successiva parola
- Metodi:
  - (C++) maiuscola iniziale per ogni parola
  - (Java) stessa degli attributi



Agenda

1. Classi, attributi, metodi
2. Relazioni fra classi
3. Ereditarietà
4. Classi astratte e interfacce
5. Contenimento
6. Associazione e dipendenze
7. Package, vincoli e note

Relazioni nei diagrammi di classe UML

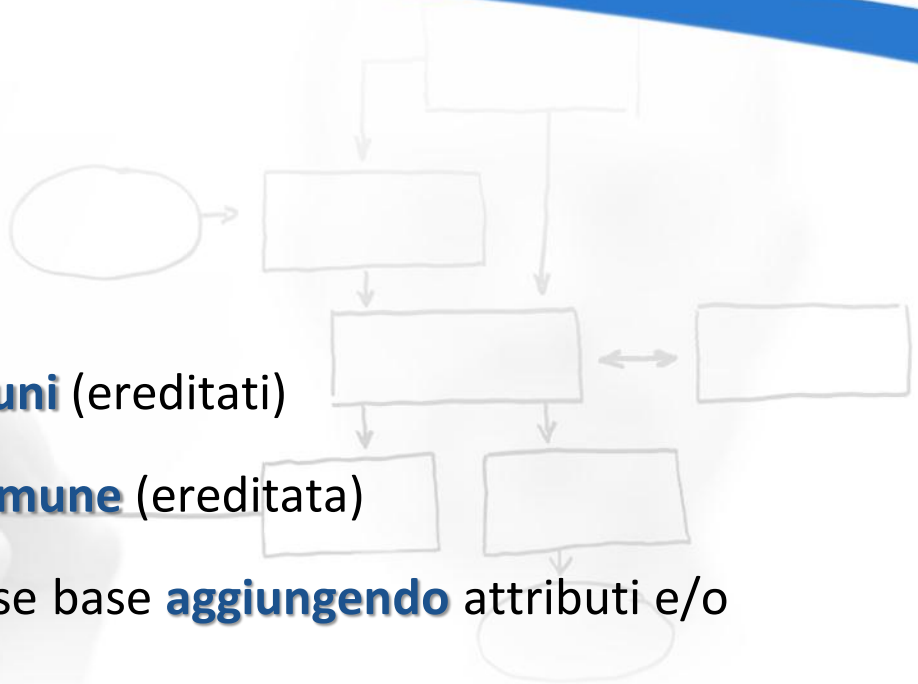


- Relazioni tra classi in un diagramma UML (dalla più forte alla più debole)
- Ereditarietà
- Contenimento
  - Aggregazione (debole)
  - Composizione (forte)
- Associazione
- Dipendenze

Agenda

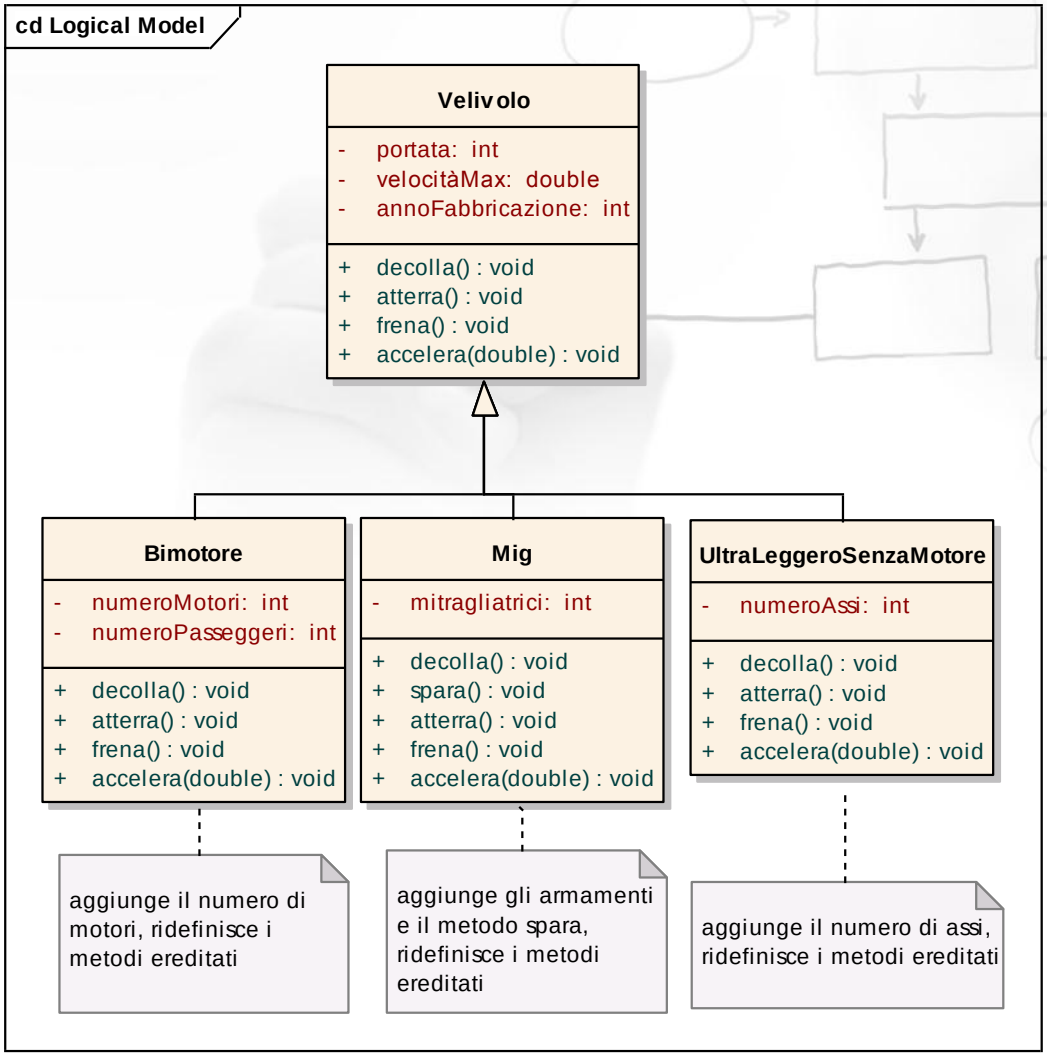
1. Classi, attributi, metodi
2. Relazioni fra classi
3. Ereditarietà
4. Classi astratte e interfacce
5. Contenimento
6. Associazione e dipendenze
7. Package, vincoli e note

Ereditarietà



- Classe base, classi derivate
- La classe base definisce **attributi comuni** (ereditati)
- La classe base definisce **interfaccia comune** (ereditata)
- Le classi derivate **specializzano** la classe base **aggiungendo** attributi e/o comportamenti (metodi)
- Le classi derivate possono anche **ridefinire** metodi ereditati (**overriding**)
- Le classi derivate possono **invocare** i metodi delle classi base, purché definiti pubblici o protetti
- In UML la relazione che esprime l'ereditarietà è la **generalizzazione**

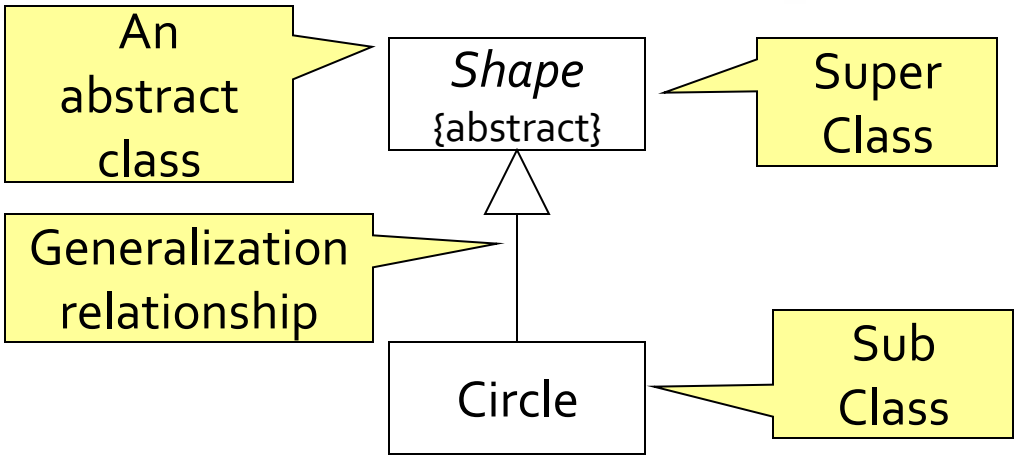
Esempio di gerarchie di classi



Ereditarietà e Principio di Sostituibilità di Liskov

- L'ereditarietà è spesso abusata perchè vista solo come meccanismo di riuso del codice
- Solitamente vale il **Principio di Sostituibilità di Liskov**
  - *Oggetti di una classe derivata (subclass) sono sostituibili ad oggetti della classe più generale (base class or super-class)*
- Relazione "is kind of"

{abstract} is a tagged value that indicates that the class is abstract.
The name of an abstract class should be italicized



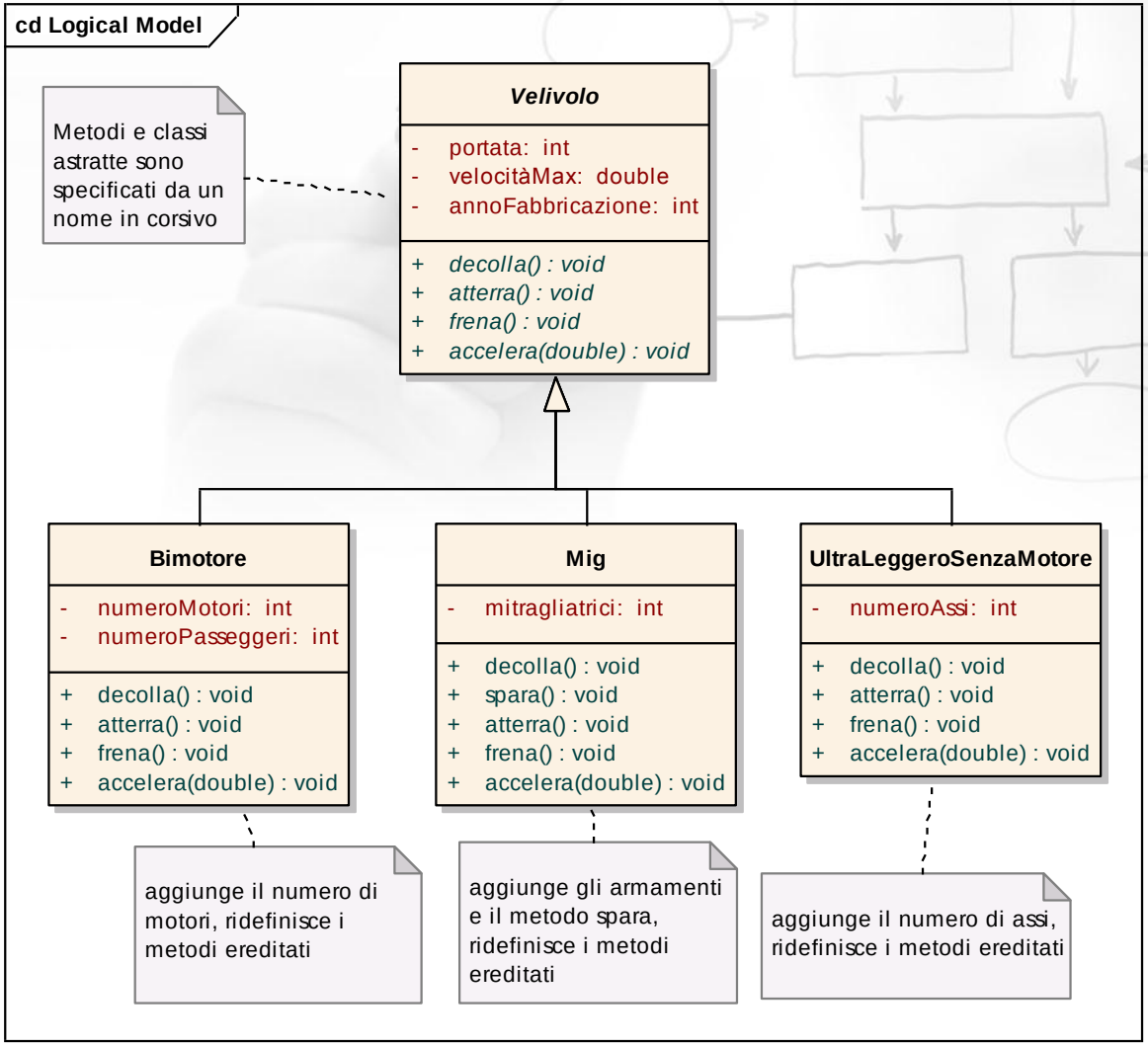
Agenda

1. Classi, attributi, metodi
2. Relazioni fra classi
3. Ereditarietà
4. Classi astratte e interfacce
5. Contenimento
6. Associazione e dipendenze
7. Package, vincoli e note

Classi astratte e interfacce

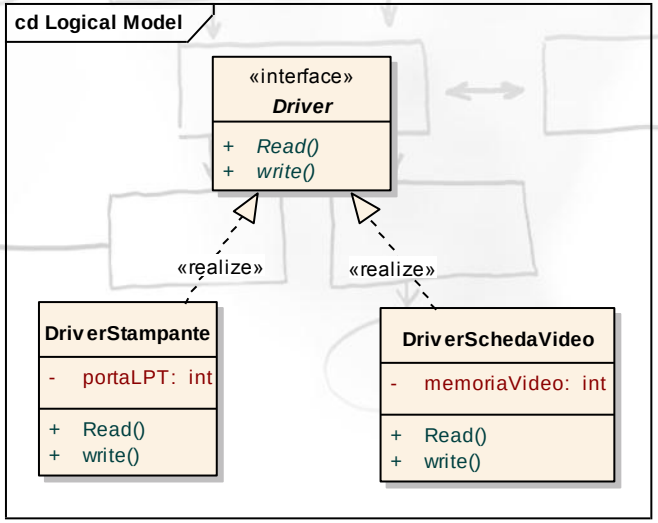
- Una **classe astratta** ha almeno un metodo virtuale puro (metodo astratto puro)
- Un'**interfaccia** non ha attributi e ha solo metodi virtuali puri (metodi astratti puri)
- Eccezione Java: le interfacce possono dichiarare come attributi delle costanti
  - Tecnica spesso abusata che maschera un progetto raffazzonato
  - Le interfacce dovrebbero definire un ***protocollo di comunicazione***
- Le classi che estendono (derivano da) una classe base astratta devono fornire un'implementazione per tutti i metodi astratti
- Lo stesso vale nel caso delle interfacce

Esempio di classe astratta



Esempio di interfaccia

- La freccia tratteggiata indica esattamente la “**realizzazione**”,ossia l’implementazione di un’interfaccia
- Da non confondere con la freccia continua dell’ereditarietà
- Lo stereotipo chiarisce la semantica, ma è opzionale



Relazioni tra classi – Realizzazione

- In UML la relazione che esprime l'implementazione di un'interfaccia si chiama **realizzazione**
  - Una realizzazione quindi indica che una classe implementa il comportamento specificato da una qualche interfaccia (intesa come protocollo)
- Una classe interfaccia può essere "realizzata" (implementata) da più classi concrete
- Una classe può realizzare (implementare) più interfacce
- Due notazioni UML diverse:

Agenda

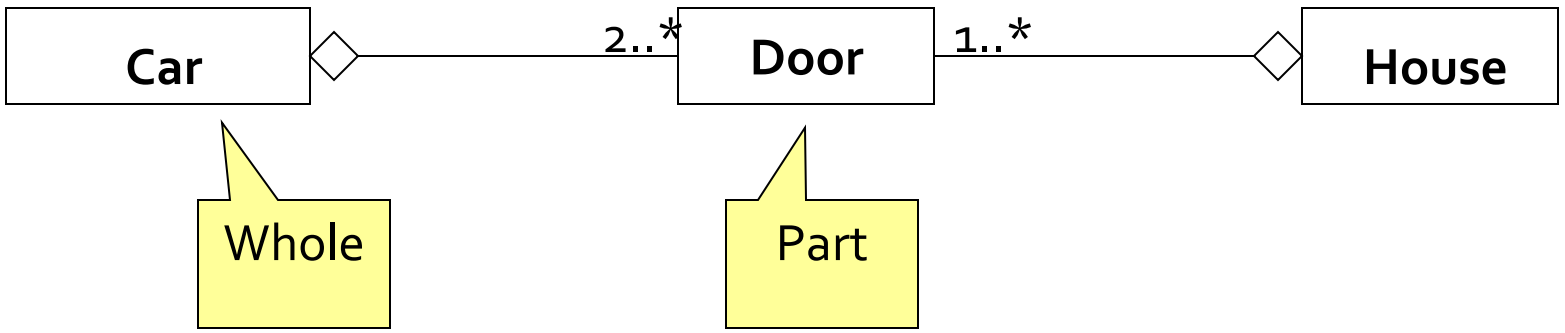
1. Classi, attributi, metodi
2. Relazioni fra classi
3. Ereditarietà
4. Classi astratte e interfacce
5. Contenimento
6. Associazione e dipendenze
7. Package, vincoli e note

Contenimento

- Due forme:
  - Forte, detta **composizione**
  - Debole, detta **aggregazione**
- La composizione associa composto e componente per tutta la vita dei due oggetti
- La composizione è esclusiva: uno specifico oggetto componente non può appartenere a due composti contemporaneamente
- Se ciò non è vero, si usa l'aggregazione

Relazioni tra classi – Aggregazione

- Forma speciale di associazione (vedremo tra poco …)
  - Modella la relazione “tutto- parte” (whole-part) tra un oggetto aggregato, il “tutto” (whole), e uno o più oggetti “parte” (part).
- Rappresenta fisicamente due relazioni leggermente diverse:
  - “is a part-part of”
  - “holds/contains”
- Esempi: car-door; house-door; hangar-airplane

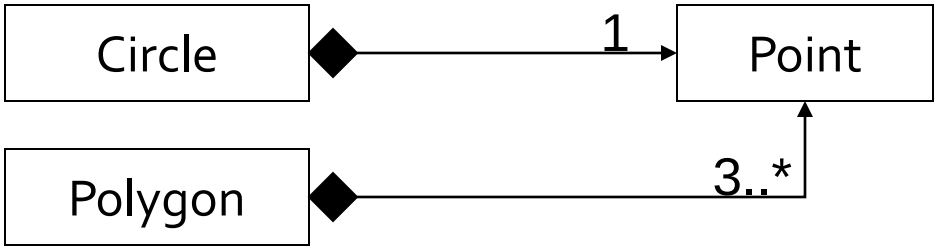


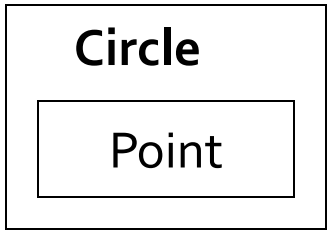
Relazioni tra classi – Aggregazione (continua)

- Non sempre è semplice decidere …
- Aggregation tests:
  - L'espressione "è-parte- di" ("part of") viene usata per descrivere questa relazione?
    - A door is "part of" a car
  - Esistono alcune operazioni del "tutto" che possono essere applicate alle singole "parti"?
    - Move the car, move the door.
  - Esistono particolari valori di attributi che possono essere propagati dal "tutto" alle "parti"?
    - The car is blue, therefore the door is blue.
  - Esiste un'intrinseca asimmetria nella relazione per cui una classe è subordinata ad un'altra?
    - A door is part of a car. A car is not part of a door.

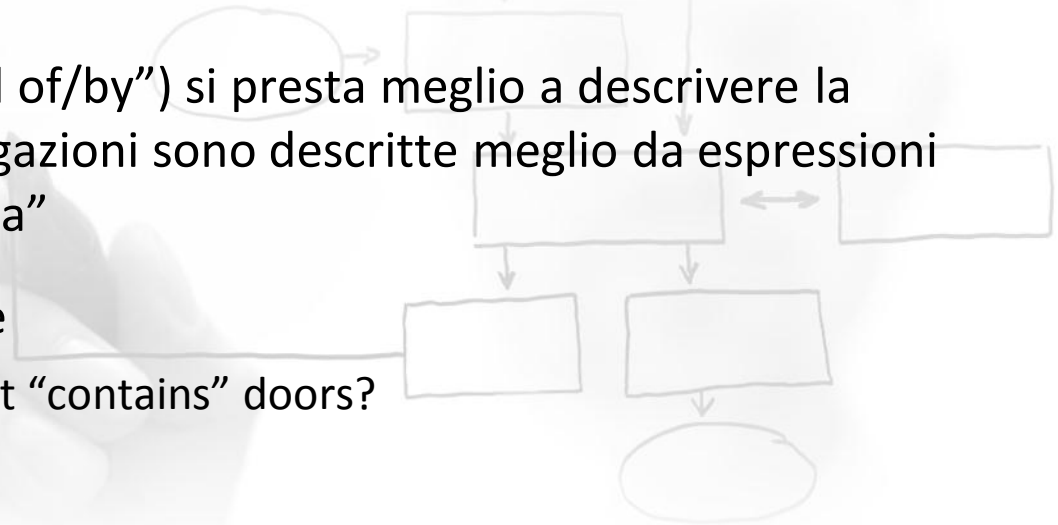
Relazioni tra classi – Composizione

- Una forma forte di aggregazione
- Il “tutto”, detto “composto”, è il solo proprietario (owner) delle proprie parti, dette “componenti”
  - L’oggetto “parte” appartiene al più ad un unico “tutto”
  - La molteplicità dalla parte del “tutto” può essere solo 0 oppure 1
- Il tempo di vita (life-time) della “parte” è subordinato (minore o uguale) a quello del “tutto”
  - L’oggetto “composto” gestisce la creazione e la distruzione delle sue parti (ownership completa)





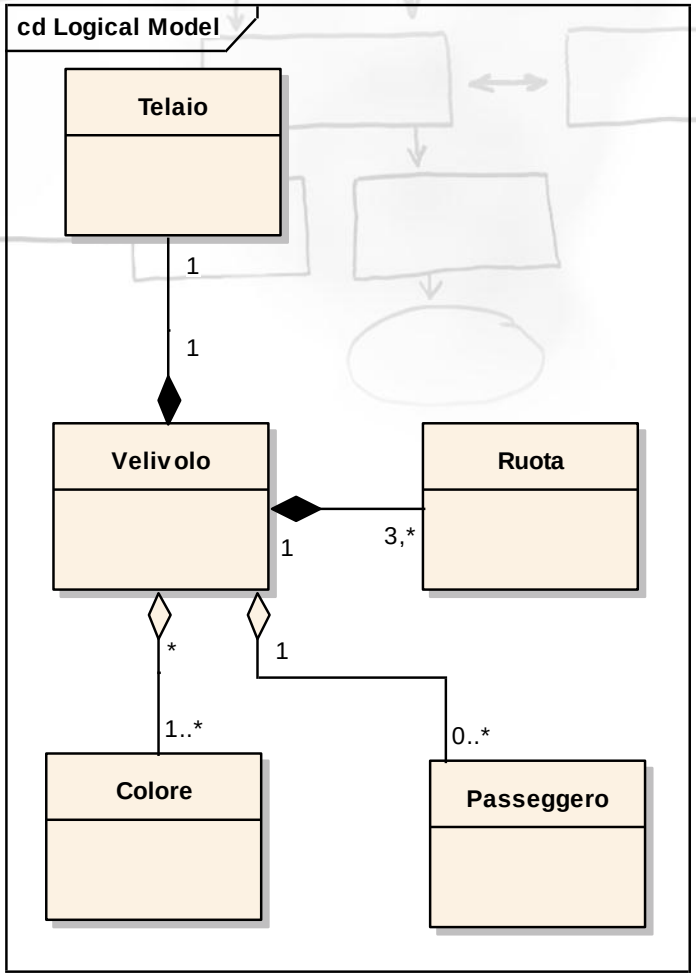
Relazioni tra classi – Composizione vs. Aggregazione



- Spesso, “part of” (“is composed of/by”) si presta meglio a descrivere la composizione, mentre le aggregazioni sono descritte meglio da espressioni come “contains”, “holds”, “has a”
- Talvolta difficile da discriminare
  - A car “is composed” of doors or it “contains” doors?
- Non perderci troppo tempo …
  - Sono dettagli che il più delle volte portano a scrivere comunque lo stesso codice!
  - Paralisi dell’analisi!
- Aggregazione e composizione descrivono forme di contenimento
  - L’importante è riconoscere questa relazione più astratta e distinguerla da altre relazioni
    - Dipendenze e generalizzazione in primis!

Esempi di contenimento

- Composizione e aggregazione
- Le molteplicità sono importanti!
- L'aggregazione è anche sinonimo di (possibile) condivisione; semantica del contenimento "by reference" (puntatori)



Agenda

1. Classi, attributi, metodi
2. Relazioni fra classi
3. Ereditarietà
4. Classi astratte e interfacce
5. Contenimento
6. Associazione e dipendenze
7. Package, vincoli e note

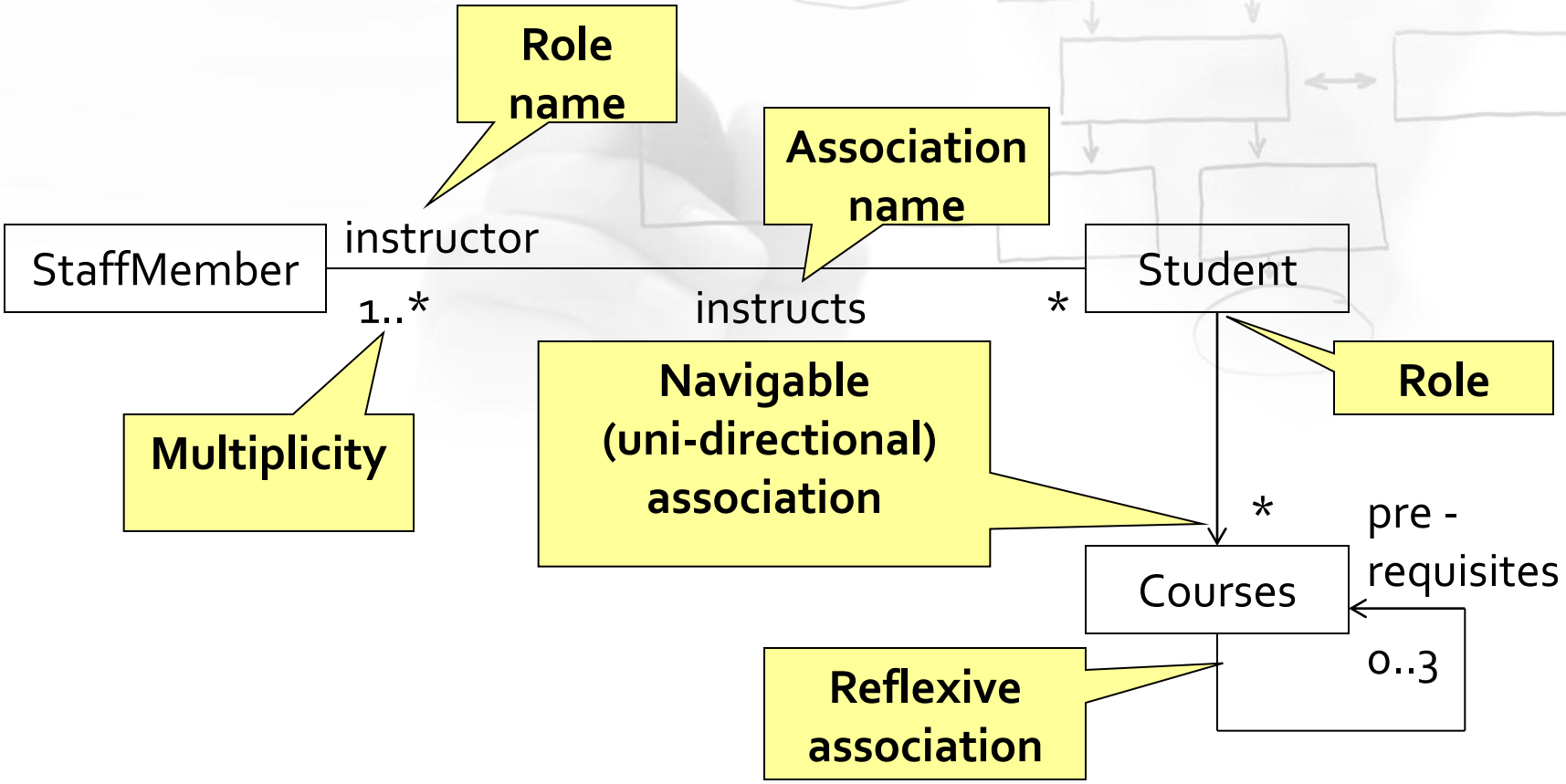
Relazioni tra classi - Associazione

- **Relazione semantica** tra due o più classi che specifica la presenza di connessioni tra le corrispondenti istanze (oggetti)
  - L'associazione è rappresentata da una linea continua che può essere direzionale
    - Monodirezionale
    - Bidirezionale (composta da due associazioni monodirezionali)
  - In un'associazione monodirezionale, la classe di partenza è detta *origine*, mentre quella di arrivo è detta *destinazione*
- **Relazione strutturale** che specifica che gli oggetti della classe origine sono connessi agli oggetti della classe destinazione
  - Il numero di istanze legate dall'associazione è specificato dalle *molteplicità* agli estremi dell'associazione
- Esempio: "An Employee works in a department of a Company"

Relazioni tra classi – Associazione (continua)

- Un'associazione tra due classi indica che gli oggetti ad un estremo dell'associazione "***riconoscono***" gli oggetti all'altro estremo e possono mandare loro dei messaggi
- Proprietà utile per scoprire associazioni non triviali usando dei diagrammi di interazione (ad es. i diagrammi di sequenza)
- Il **nome** dell'associazione è rappresentato da un'etichetta posta al centro dell'associazione stessa
  - Il nome è solitamente un verbo
- Un **ruolo** è un'etichetta posta ad (almeno) uno degli estremi di un'associazione, nel punto di congiunzione con la classe
  - Può indicare il ruolo svolto dalla classe a cui si riferisce nell'ambito dell'associazione
  - Solitamente è un nome (talvolta visto come attributo stesso della classe collegata)
  - Obbligatorio solo per relazioni di associazione riflessive

Relazioni tra classi – Associazione (continua)



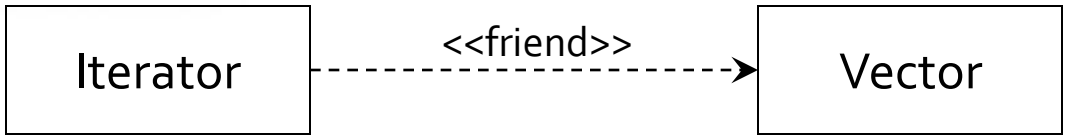
Relazioni tra classi – Associazione (continua)

- **Molteplicità**
  - Indica il *numero di istanze* della classe più vicina che possono esistere a run-time in una *configurazione valida* del sistema …
  - … che sono riferite da una *singola istanza* della classe che sta all'estremo opposto della relazione
  - Specifica se un'associazione è obbligatoria o meno
  - Fornisce un *intervallo di validità* (estremo inferiore ed estremo superiore) relativamente al numero di istanze contemporaneamente presenti e legate a ciascun ruolo dell'associazione in un dato momento
  - Indicatori di molteplicità:

Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8

Relazioni tra classi – Dipendenza

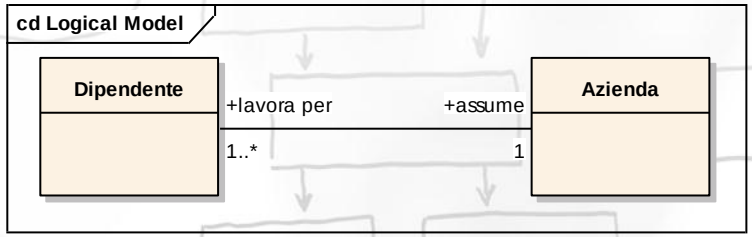
- Una dipendenza rappresenta una relazione semantica tra due classi, nonostante non ci sia un'esplicita associazione tra esse
- Una classe dipende da un'altra quando esiste un riferimento della seconda nella prima (non è possibile compilare la prima senza la seconda)
  - Ad esempio, una classe "nomina" al suo interno una classe esterna
    - Passaggio di parametri all'interno di un metodo
    - Tipo di ritorno di un metodo
    - Creazione o distruzione
    - Eccezioni

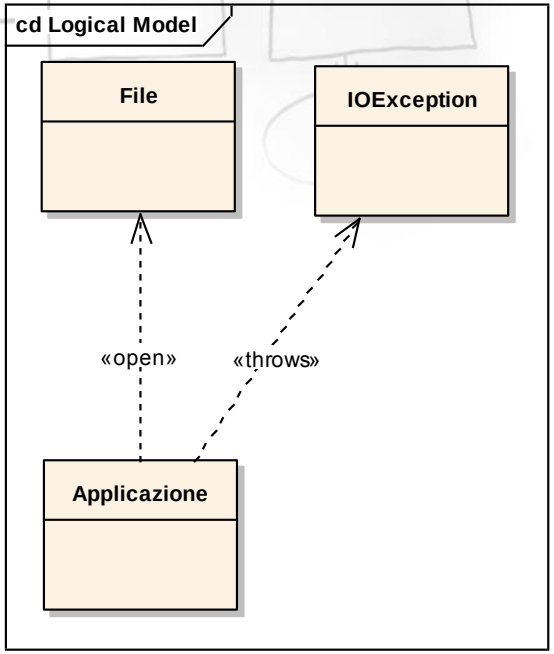


- Uno **stereotipo** può essere usato per denotare il tipo di dipendenza
  - Chiarisce la semantica della dipendenza
  - Attenzione a non esagerare: stereotipi non standard rendono meno comprensibile il design (opacità del software)

Associazione e dipendenze

- L'**associazione** esprime un legame di **ruoli**, tipicamente di natura strutturale (quindi non transiente)
  - Il dipendente, nel suo ruolo, è perennemente legato all'azienda per la quale lavora.
- La **dipendenza** esprime tipicamente un generico legame di natura spesso transiente, senza ulteriore semantica più forte
  - L'eccezione viene collegata alla classe Applicazione solo qualora "scatta".
  - Lo «stereotipo» nella dipendenza definisce la semantica intuitiva della dipendenza



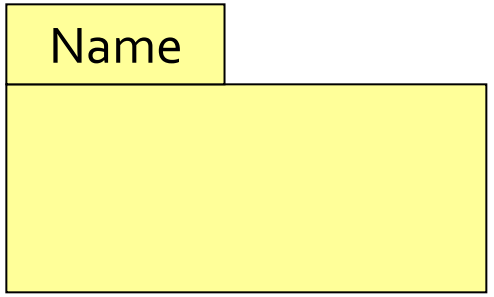


Agenda

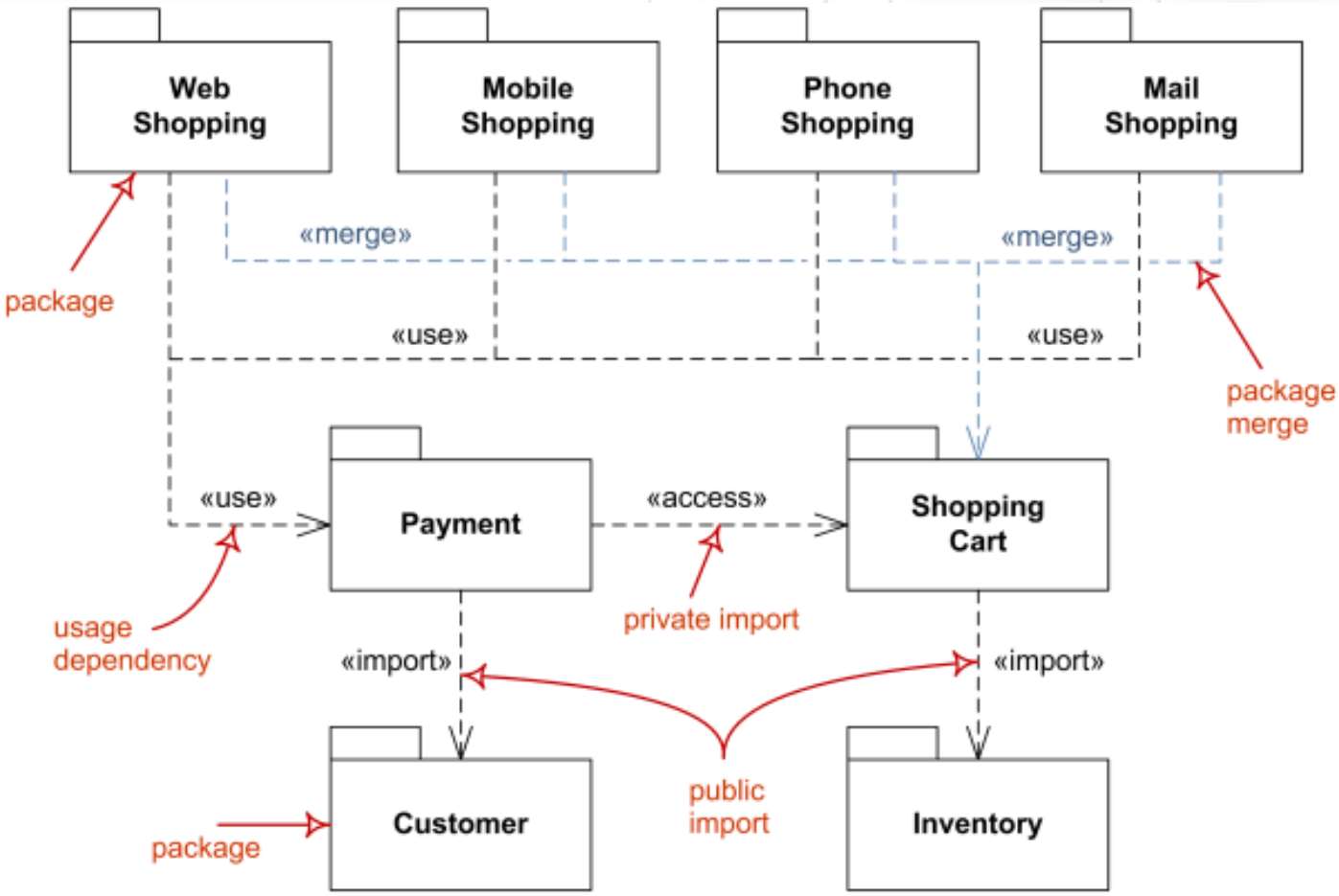
1. Classi, attributi, metodi
2. Relazioni fra classi
3. Ereditarietà
4. Classi astratte e interfacce
5. Contenimento
6. Associazione e dipendenze
7. Package, vincoli e note

Package

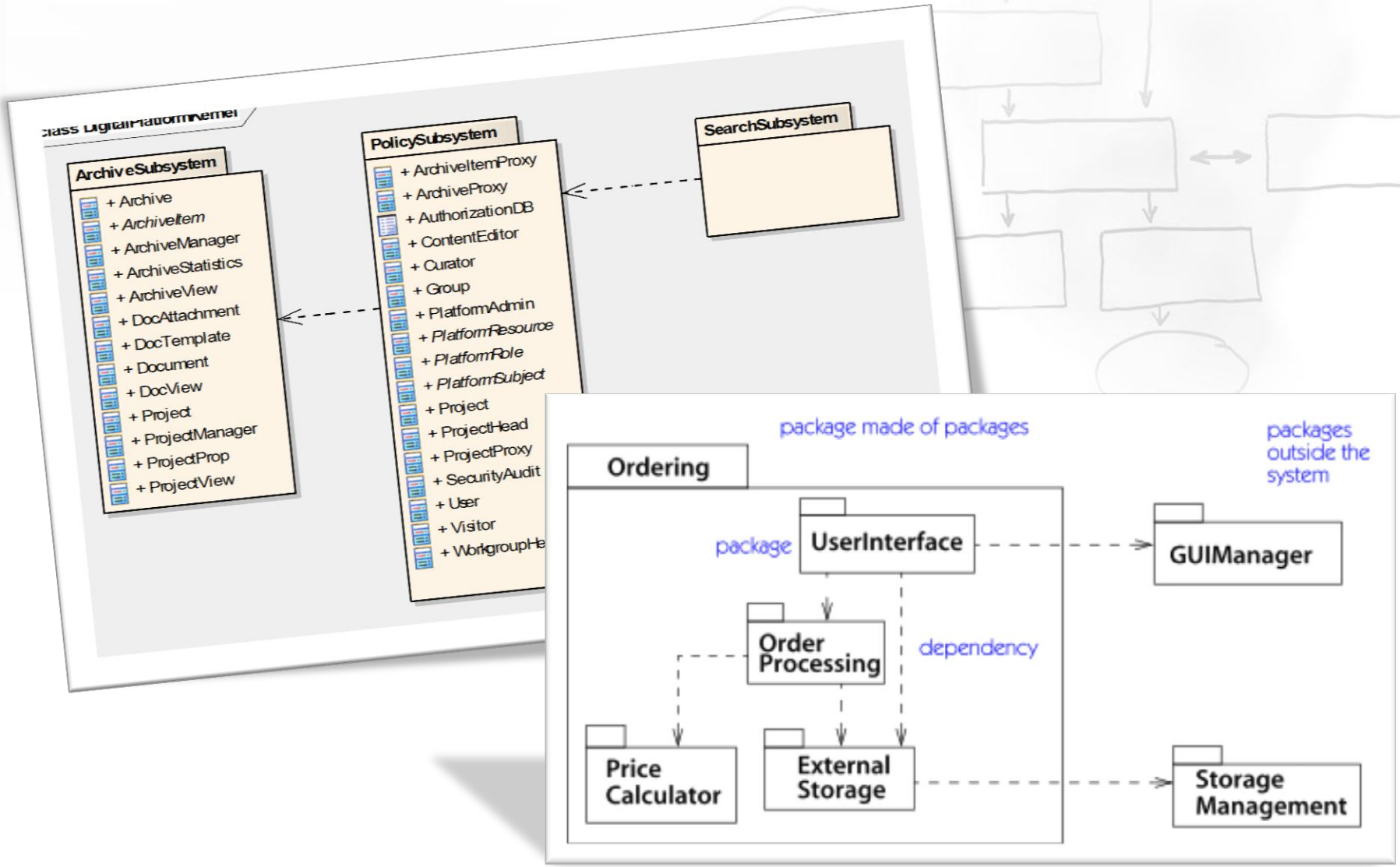
- Un package è un **elemento di raggruppamento**
- Serve per **organizzare** elementi UML correlati tra loro
  - Criterio della coesione
- Si applica sia ai diagrammi, sia alle classi e ai casi d'uso
- Un package non deve essere necessariamente tradotto un una componente fisica del sistema (sottosistema)
  - Es. I package per raggruppare requisiti funzionali



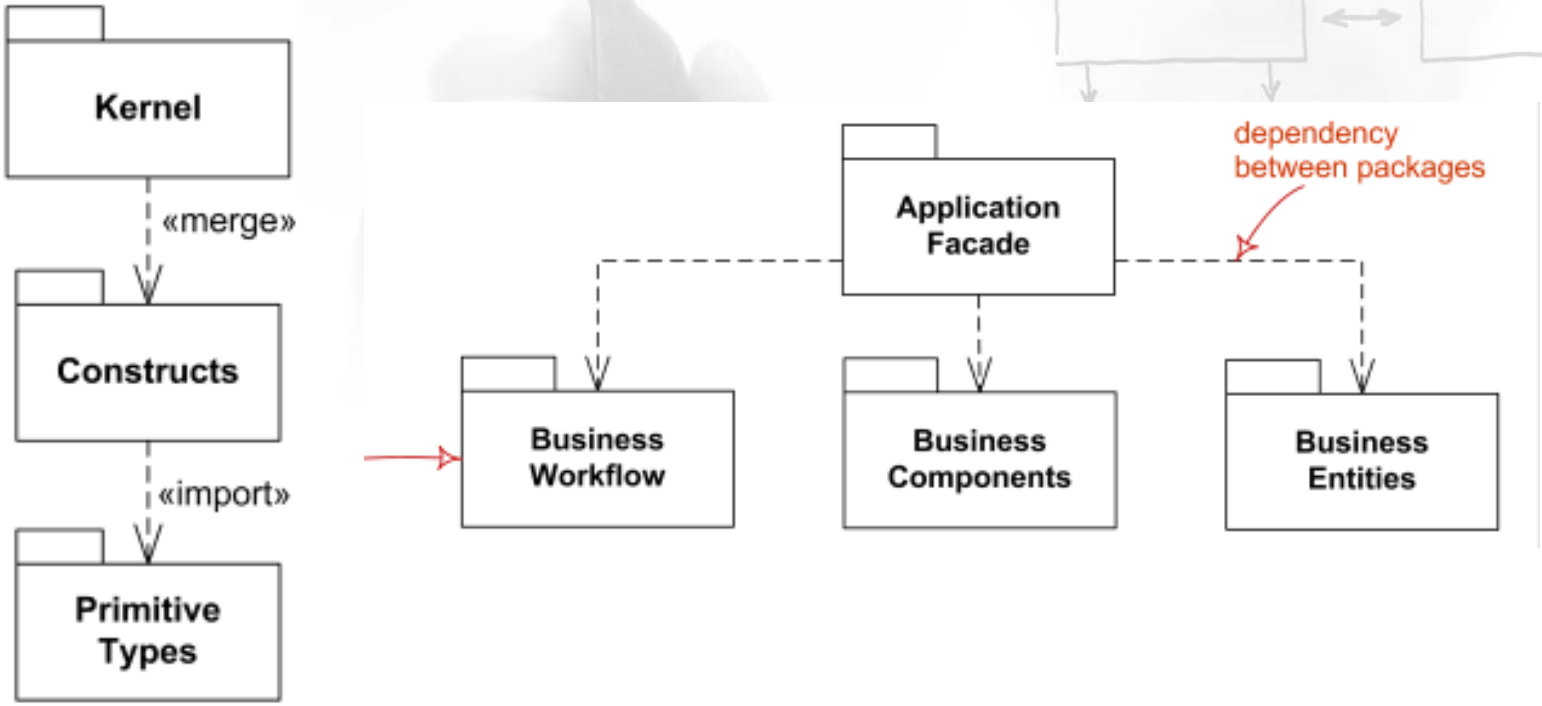
Diagrammi dei package



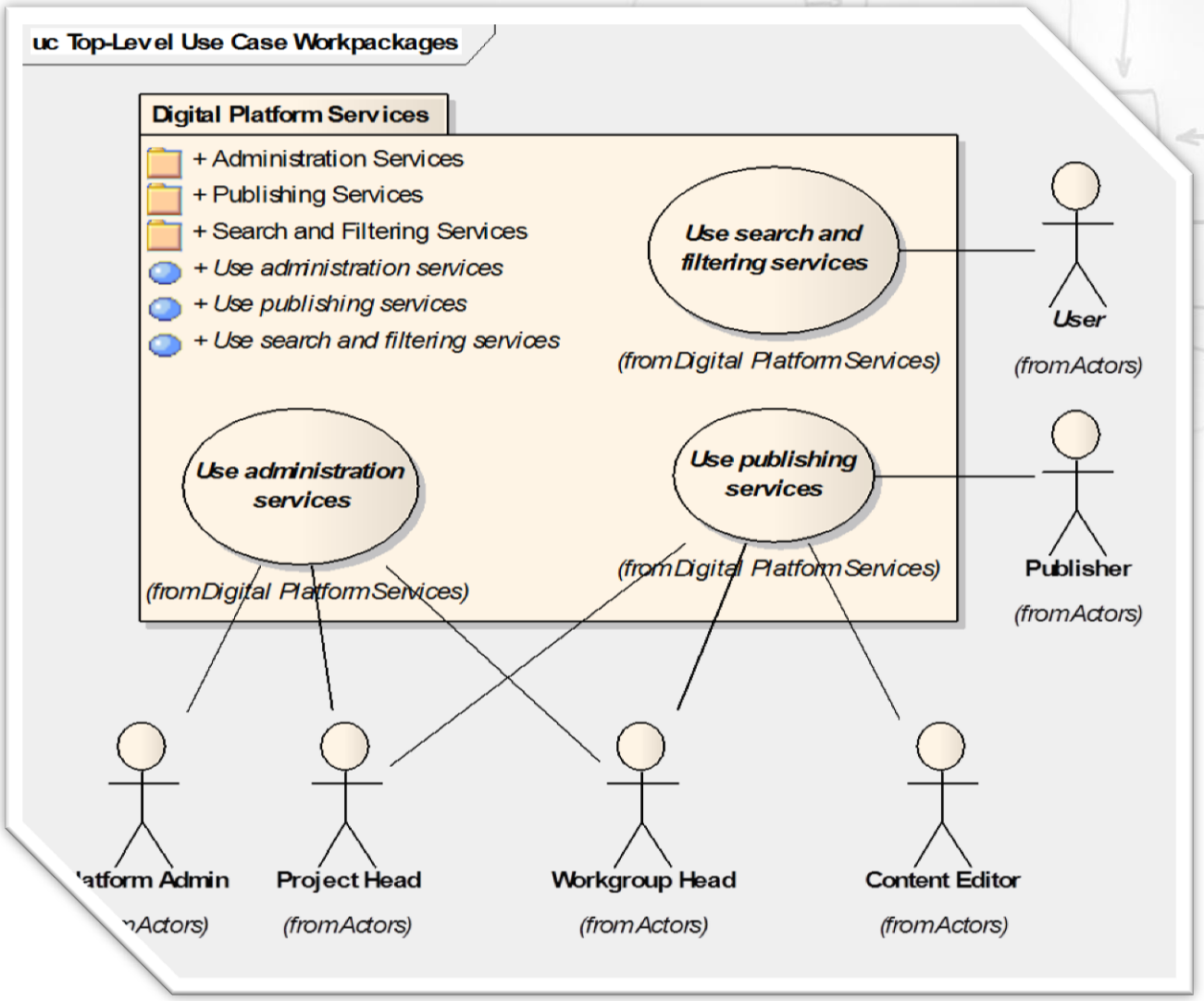
Diagrammi dei package



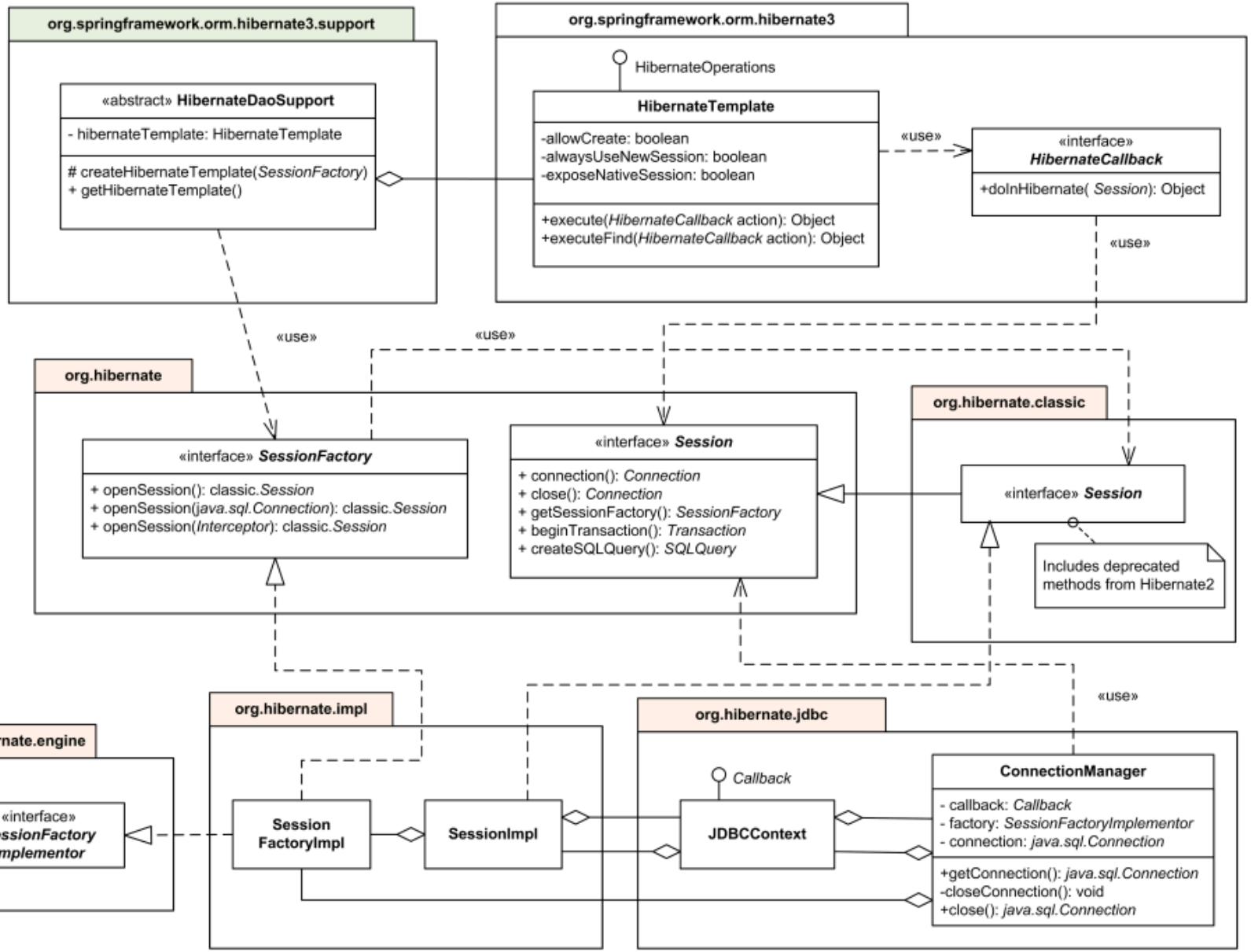
Diagrammi dei package



Package nei diagrammi dei casi d'uso



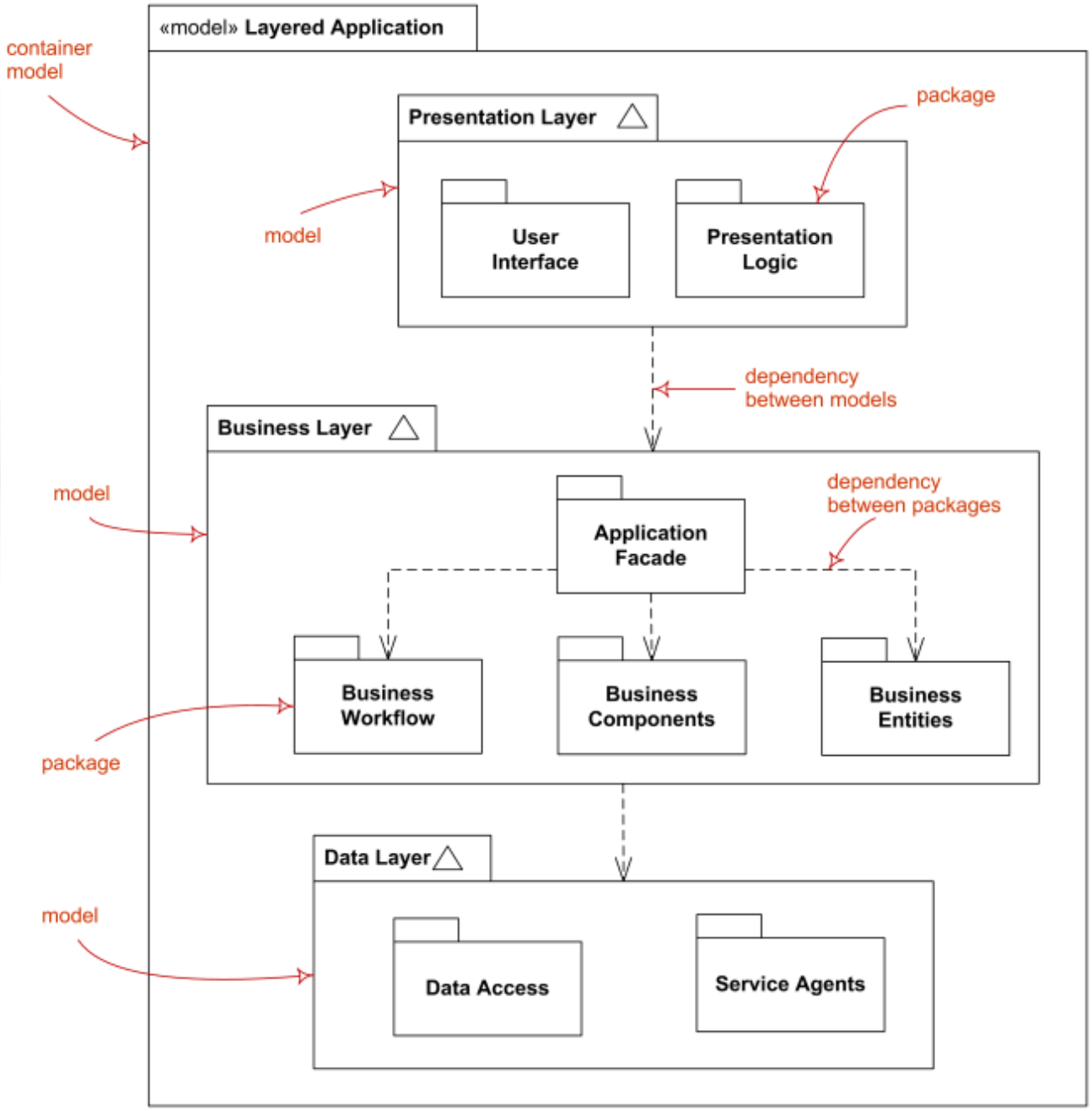
Package nei diagrammi di classe



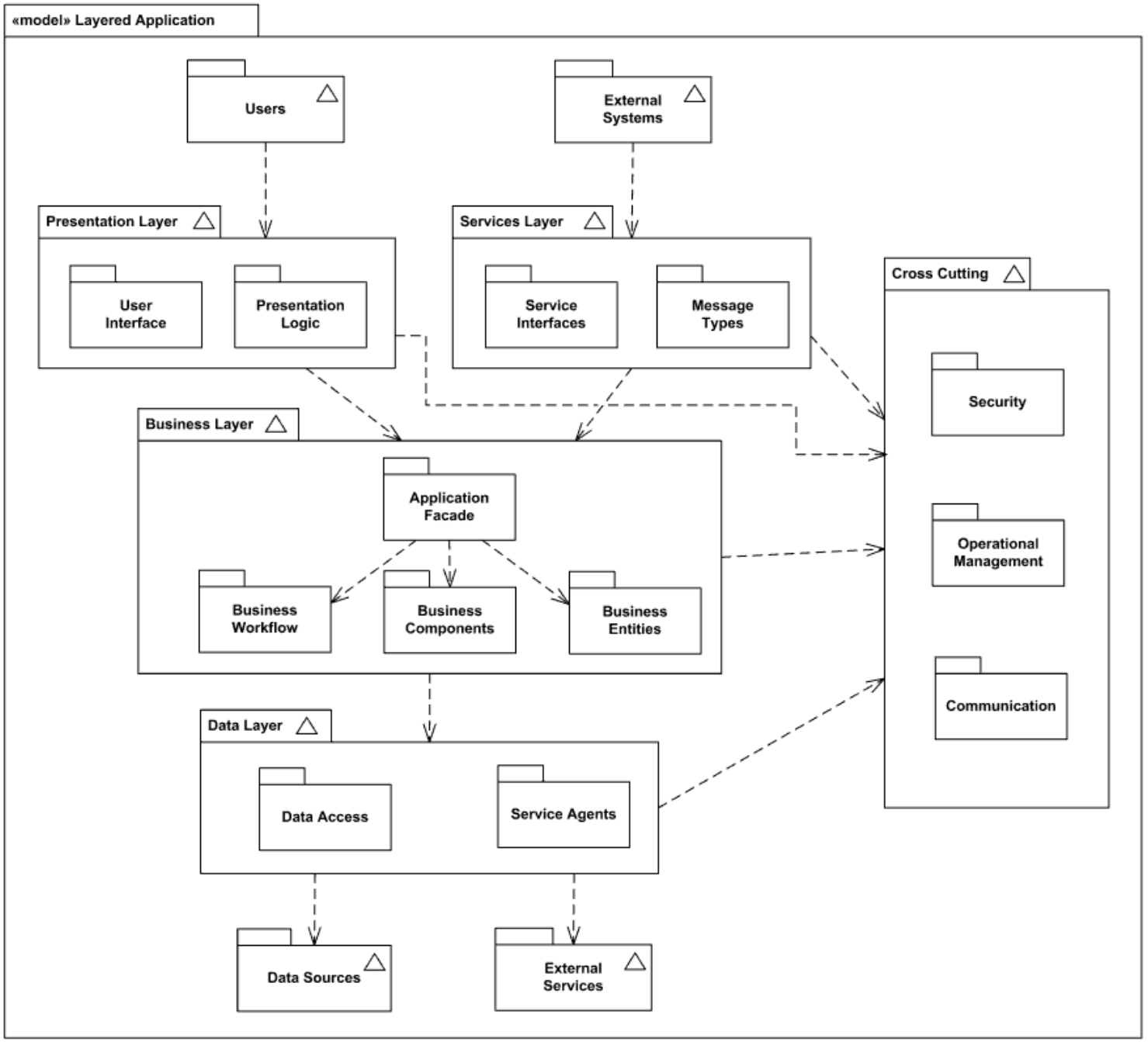
Package nei Model Diagram

Model diagram is UML auxiliary structure diagram which shows some abstraction or specific view of a system

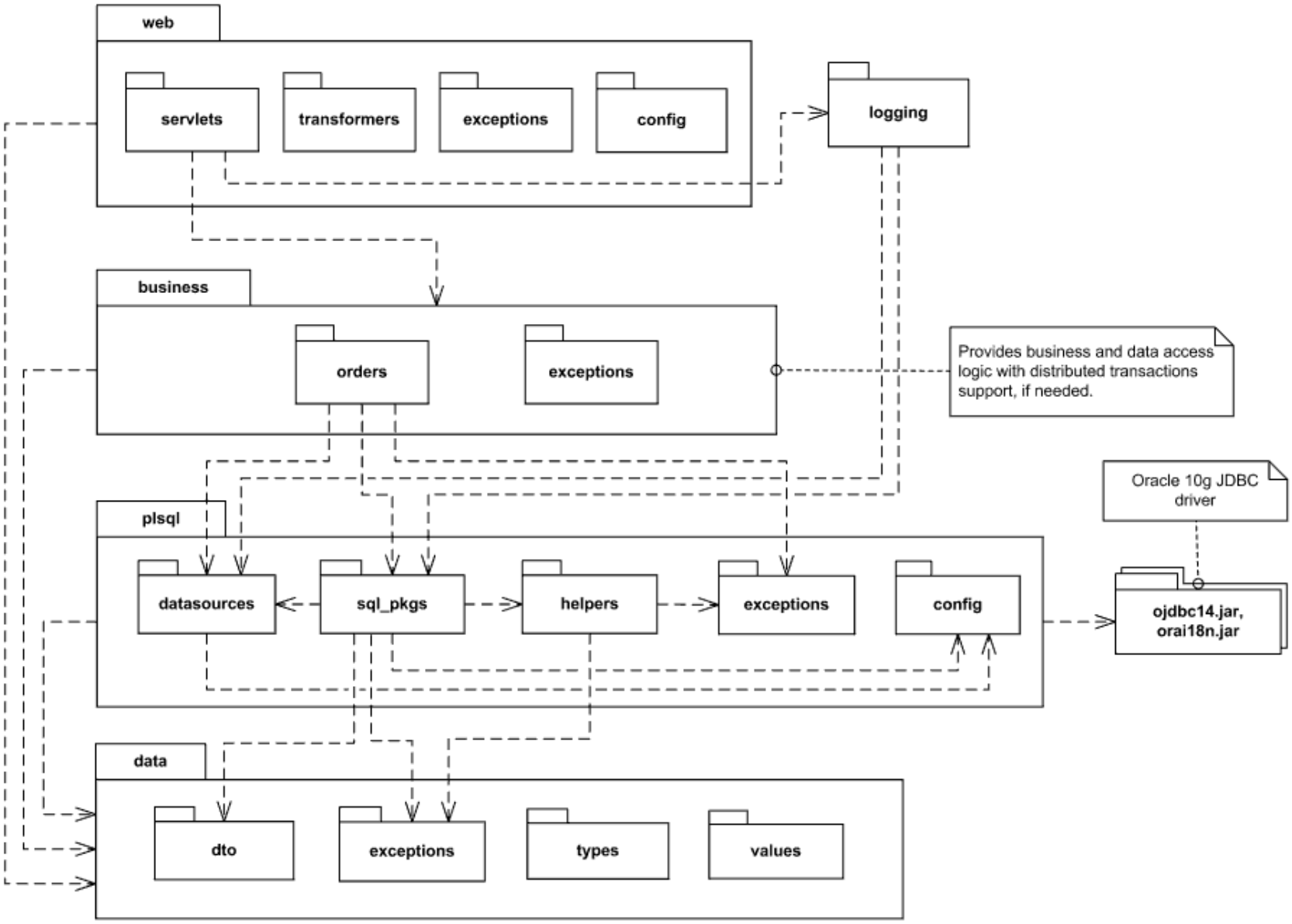
Model diagram describe some architectural, logical or behavioral aspects of the system



Layered Application Architecture

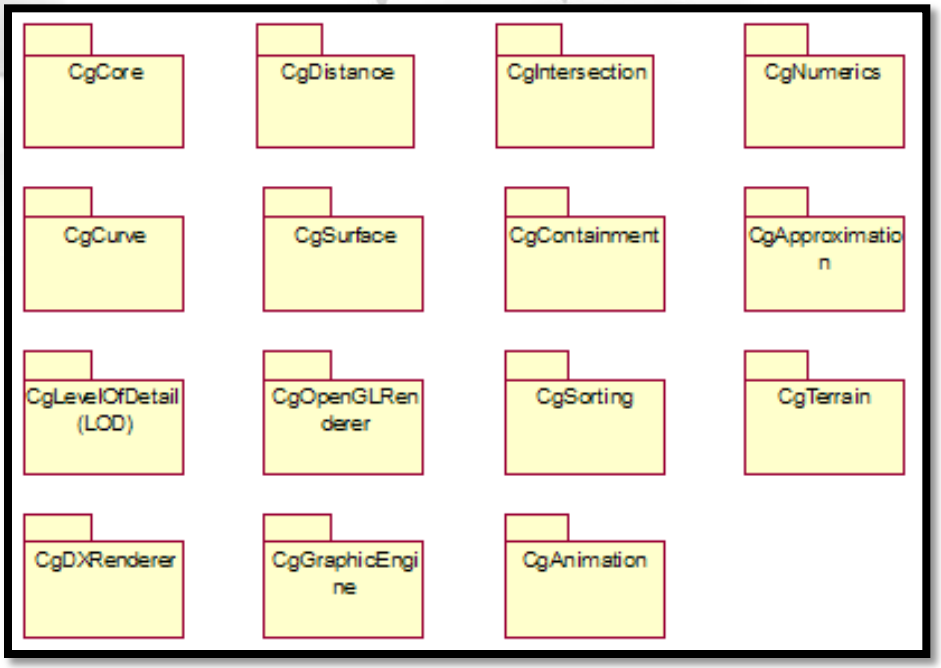


Web Application Architecture

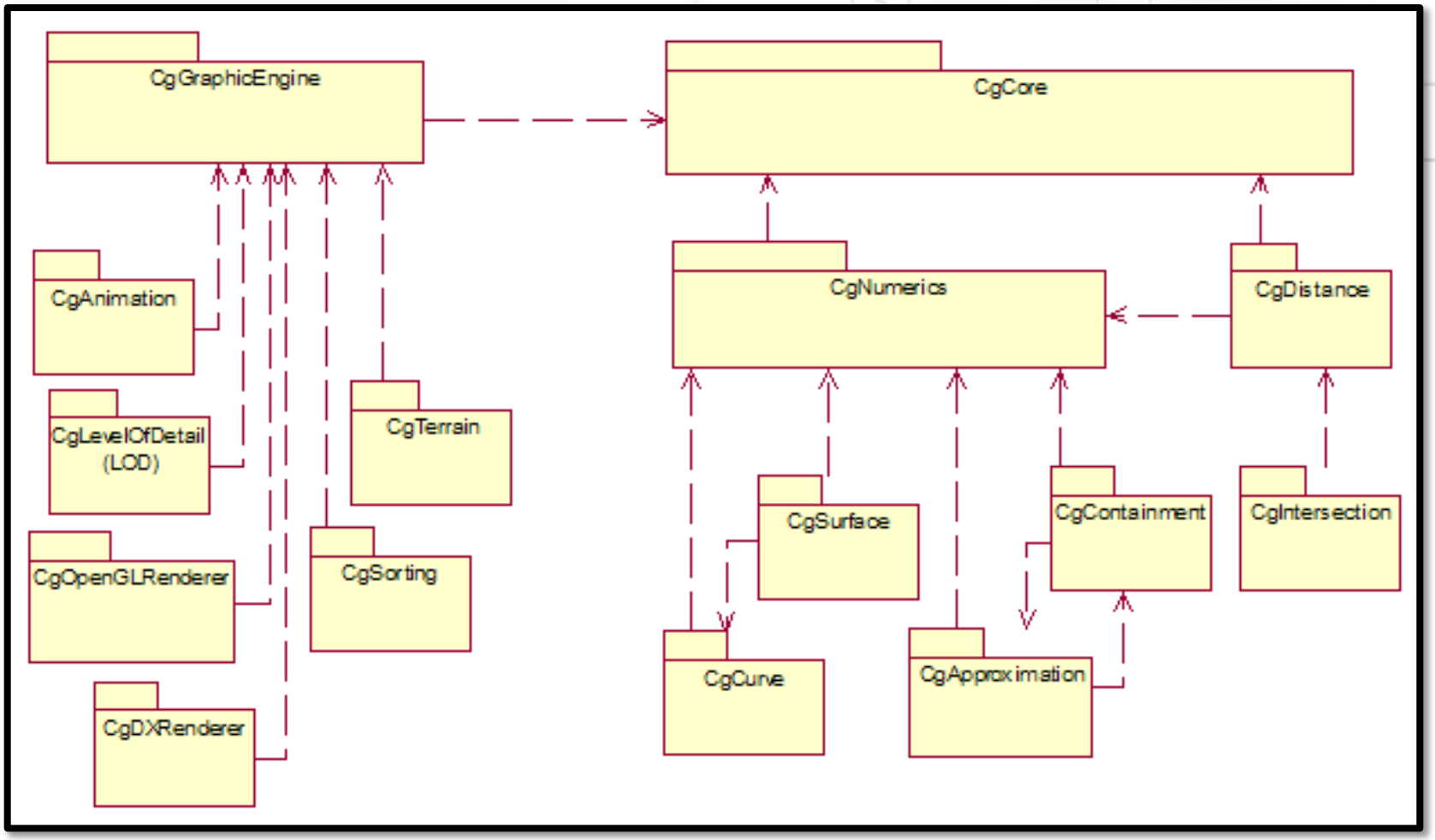


Un diagramma dei package poco informativo (relazioni soppresse)

- Diagramma di package poco informativo.
- Elenca una serie di package, ma non esplicita relazioni tra i package
- È utile esplicitare le relazioni tra i package, in particolare dipendenze e i livelli d'astrazione

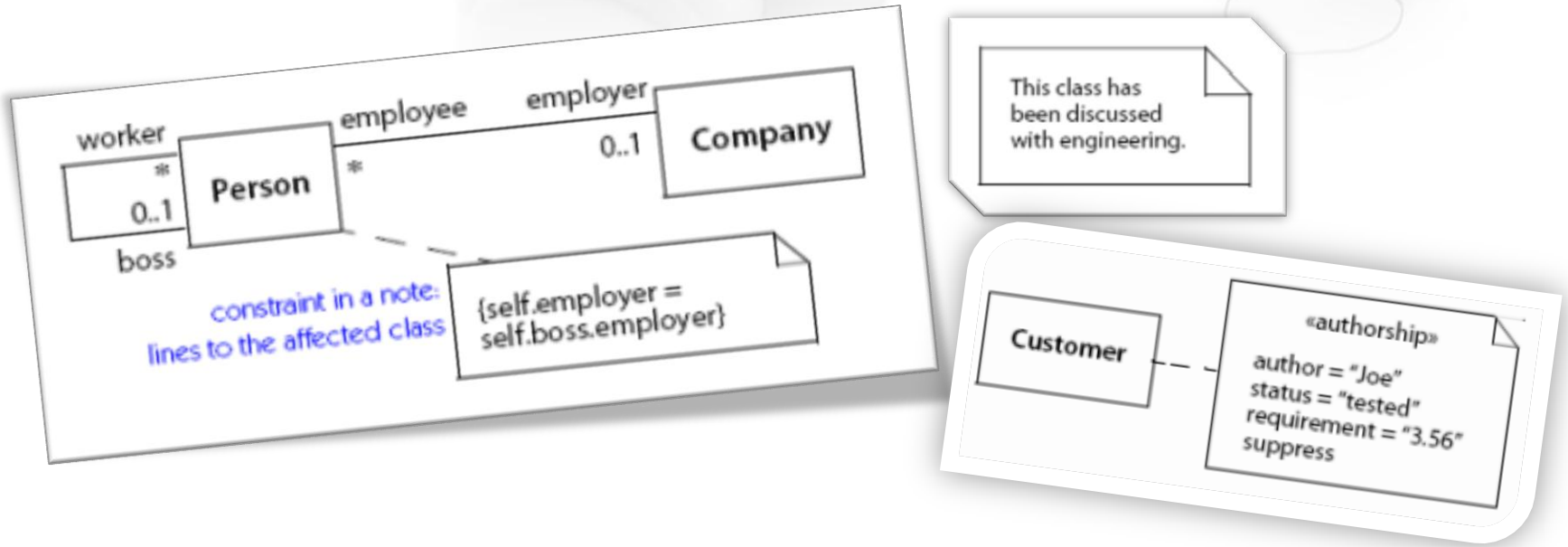


Un diagramma dei package con le dipendenze



Vincoli (constraint), regole e note

- Vincoli e note servono per annotare, tra le altre cose,
  - Associazioni, attributi, operazioni e classi.
- Un vincolo è una restrizione semantica espressa mediante un'espressione booleana



I vincoli sono utili per …

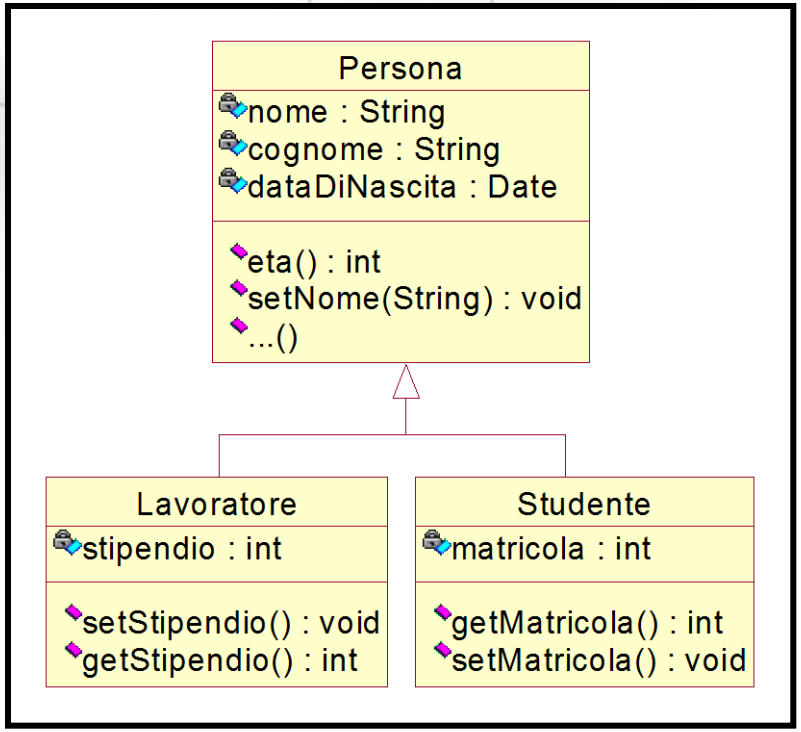
- Documentare assunzioni (relative ad aspetti di analisi, design e/o implementazione)
- Descrivono invarianti
- Design by contract :
  - Invarianti : sempre true per ogni object, in qualsiasi stato
  - Pre-condizioni : devono essere true al momento della chiamata del metodo
  - Post-condizioni : devono essere true al termine (uscita) del metodo

Ragionare sui diagrammi di classe

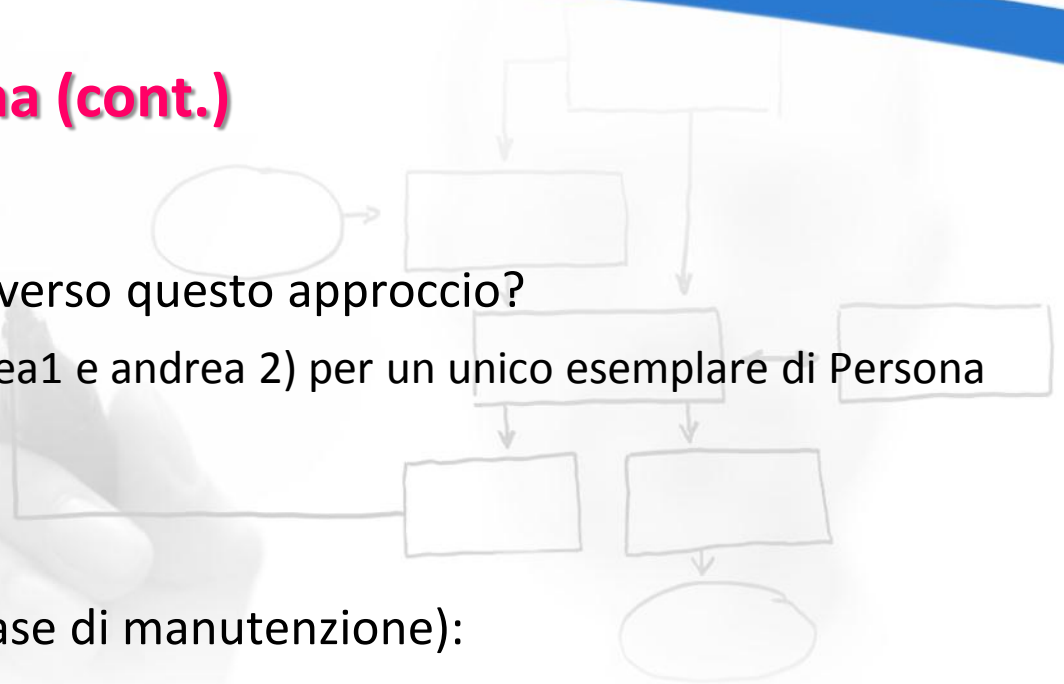
- I diagrammi di classe permettono di **visualizzare il design** ad un livello di astrazione superiore rispetto al codice.
- Ciò è utile per **ragionare sulle scelte di progetto** che facciamo.
- I diagrammi di classe, corredati di vincoli (e.g. invarianti, precondizioni, postcondizioni, asserzioni sugli stati, ecc.) sono utili per capire se il nostro design è “sound”
- Ad esempio, le seguenti sono delle buone gerarchie di classi? In caso negativo, quali sono i possibili problemi?
  - Persona <– Studente; Persona <– Lavoratore
  - Figura <– Rettangolo <– Cerchio

Esempi: la gerarchia Persona

- Gerarchia Persona-Studente e Persona-Lavoratore
- L'ereditarietà è una relazione statica!
  - Cosa succede quando la stessa persona smette di essere studente ed inizia ad essere lavoratore? (Si spera che prima o poi succeda :)
  - … devo creare una nuova istanza di Lavoratore, copiare lo stato dell'istanza di Studente, "valorizzare" la parte di stato specifica di Lavoratore (e.g. stipendio), cancellare la vecchia istanza di Studente …
  - … ma adesso ho un'istanza diversa dalla precedente, però la persona è sempre la stessa!



Esempi: la gerarchia Persona (cont.)

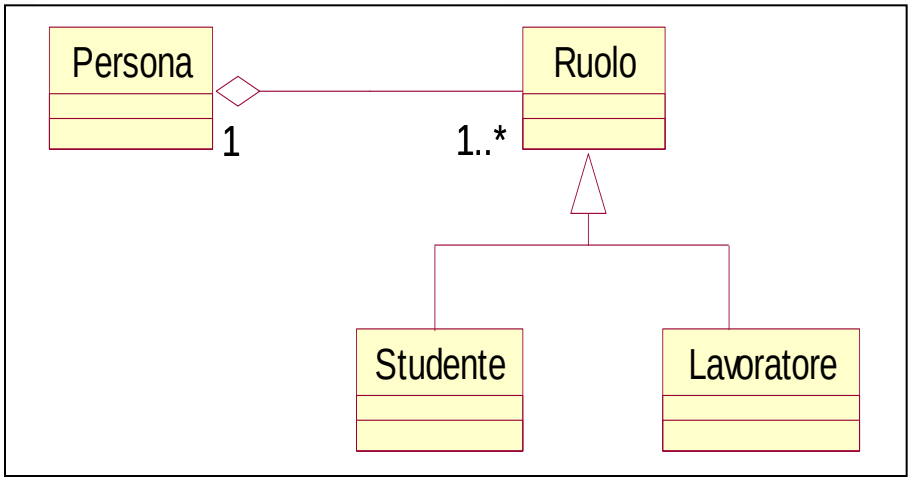


- Quali problemi emergono attraverso questo approccio?
  - Due oggetti diversi e “vivi” (andrea1 e andrea 2) per un unico esemplare di Persona (identità): andrea!

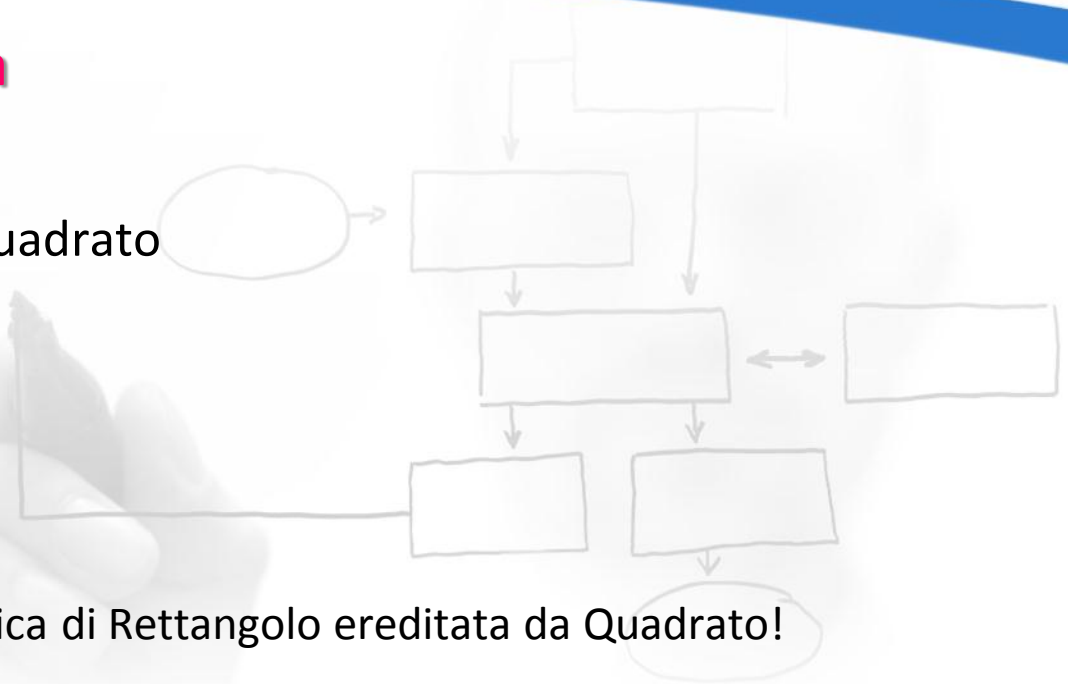
- Possibili incoerenze (anche in fase di manutenzione):
  - Ho copiato correttamente lo stato di andrea1 nello stato di andrea2?
  - Ho inizializzato completamente andrea2?
  - Una volta diventato Lavoratore, il numero di matricola di andrea1 non dovrebbe essere più valido. Chi lo garantisce? Ossia chi mi garantisce che il programmatore si ricordi sempre *anche* di cancellare la vecchia istanza?

Esempi: la gerarchia Persona (cont.)

- Come risolvere il problema? Capendo che ciò che sto modellando non è l'identità di una Persona, bensì il suo Ruolo!
- Linea guida: preferire sempre il contenimento all'ereditarietà. Proviamo a sfruttarla.
- Vantaggi:
  - Una persona può svolgere più ruoli contemporaneamente (studente e lavoratore)
  - Una persona può cambiare ruolo dinamicamente



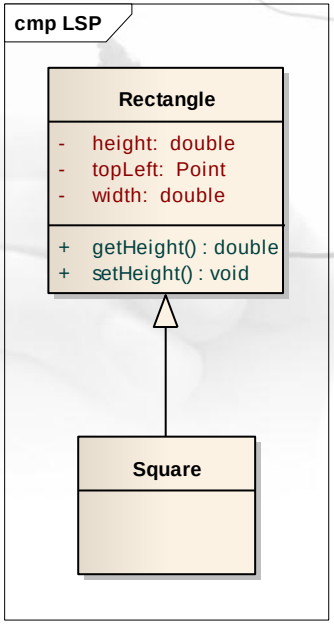
Esempi: La gerarchia Figura



- Gerarchia Figura-Rettangolo-Quadrato
- Il Rettangolo è una Figura
- Il Quadrato è una Figura
- Il Quadrato è un Rettangolo?
  - Dipende … dall'interfaccia pubblica di Rettangolo ereditata da Quadrato!
- **Considerazione:** per essere una gerarchia robusta, dobbiamo esaminare contemporaneamente sia l'implementazione di Rettangolo, sia quella di Quadrato
  - Vogliamo evitare che si verifichino situazioni incoerenti
    - Ricordiamo che un qualsiasi Quadrato può sostituire un qualsiasi riferimento a Rettangolo, per ogni programma P, senza che venga per questo alterato il Comportamento di P in modo inatteso
    - Ossia senza generare uno stato di P non valido

Una definizione problematica di Rettangolo …

```
-- The Rectangle class.
public class Rectangle
{
   private Point topLeft;
   private double width;
   private double height;

   public double Width
   {
      get { return width; }
      set { width = value; }
   }

   public double Height
   {
      get { return height; }
      set { height = value; }
   }
}
```




```
-- Redefining the properties of
-- Rectangle in Square
public class Square : Rectangle …
   public new double Width
   {
      set
      {
         base.Width = value;
         base.Height= value;
      }
   }
   public new double Height
   {
      set
      {
         base.Height = value;
         base.Width = value;
      }
   }
}
```

```
-- A simple test driver
…
Square s = new Square();
s.SetWidth(1);    // Fortunately sets the height to 1 too
s.SetHeight(2);  // sets width and height to 2: the Square's invariant holds!
```

```
-- what happens now?
void f(Rectangle r)
{
   r.SetWidth(32);
}
```

calls the property *Rectangle.SetWidth*!

Rettangoli e Quadrati

- Aspetto tecnico: nell'esempio precedente la classe base non dichiara le proprietà come **virtual**
  - la gerarchia non è conforme al Principio di Sostituibilità di Liskov:
- Aspetto tecnico: Inoltre il **polimorfismo non è abilitato** anche per un altro motivo:
  - anziché fare l'*override* (sovrascrittura) delle proprietà, si è preferito nasconderle mediante la parola chiave *new* (ridefinire un metodo che oscura eventuali sue versioni presenti sulla catena di ereditarietà, senza ereditare alcuna versione)
- Basta mettere la parola chiave virtual (che abilita il polimorfismo) affinché la gerarchia rispetti LSP?
  - No! servono condizioni più forti ed esplicite
  - Vediamo di capire meglio perché …

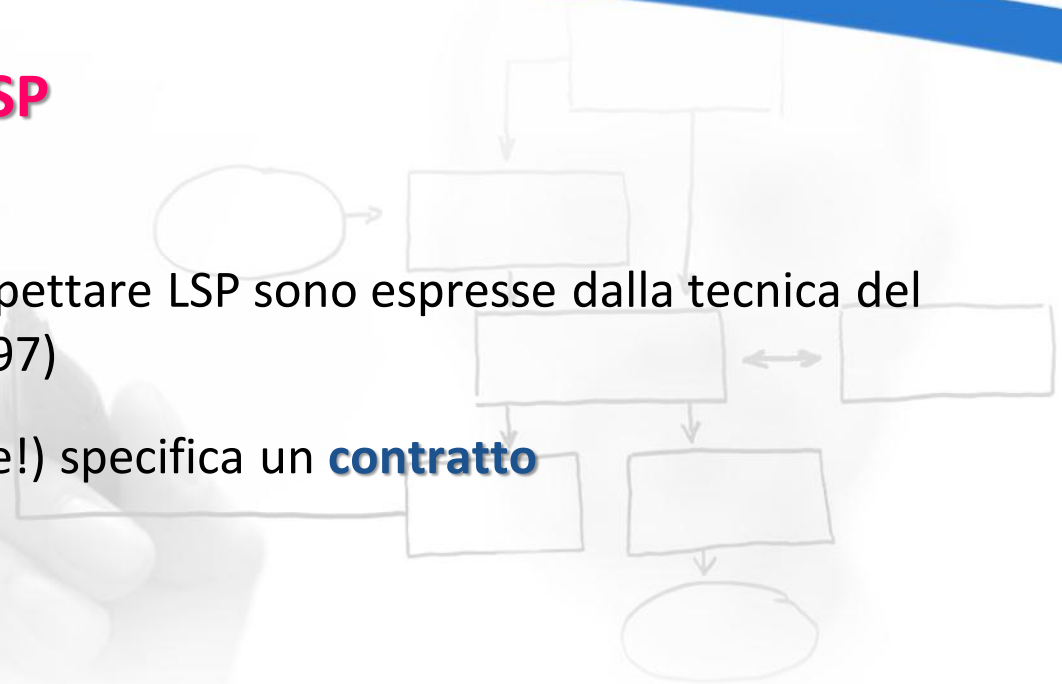
Violazione subdola di LSP

```
-- Violation of LSP.

public class Rectangle
{
   ...
   public virtual double Width {
      get { return width; }
      set { width = value; }
   }
   public virtual double Height {
      get { return height; }
      set { height = value; }
   }
}

public class Square : Rectangle {
   public override double Width  {
      set {
         base.Width = value; base.Height= value;
      }
   }
   public override double Height {
       set {
          base.Height = value; base.Width = value;
       }
   }
}
```

```
-- A problematic polimorfic function
...
void checkMaxBB(Rectangle maxBoundingBox) {
   maxBoundingBox.Width = 4;
   maxBoundingBox.Height = 5; // minimal bounding box of 20? Not always!
   ...
   if (maxBoundingBox.Area() >20)  // what if maxBoundingBox is a Square????
   {
      throw new Exception("Overhead detected: area too large!");
   }
}
```

- Problema reale (aspetto "semantico"):
  - Il codice cliente (**checkMaxBB**()) assume che cambiare la larghezza di un *qualsiasi* rettangolo lasci inalterata la base!
  - Ciò non vale per i rettangoli che sono anche quadrati!
  - La gerarchia **Rectangle-Square** non rispetta LSP
  - NB. L'invariante di classe di **Square** continuano a valere. Fallisce l'invariante di **Rectangle**!

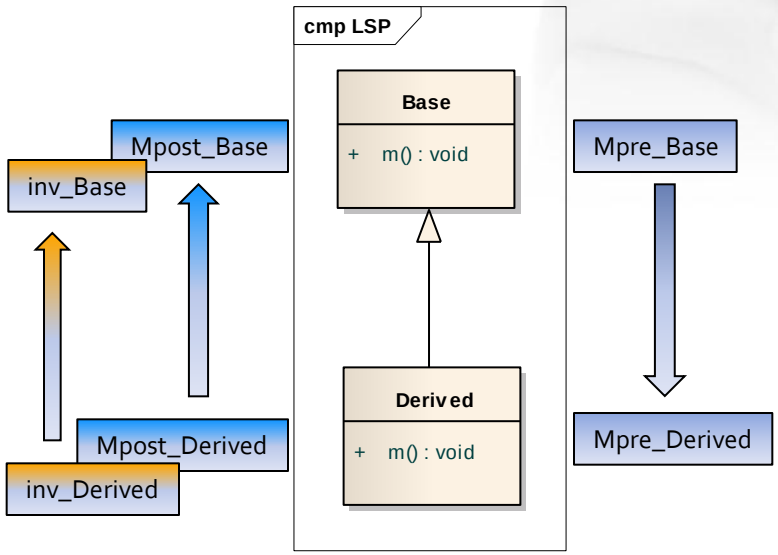
Cosa serve per rispettare LSP



- Le condizioni necessarie per rispettare LSP sono espresse dalla tecnica del **Design by Contract** (Meyer, 1997)
- Ogni classe (anche le classi base!) specifica un **contratto**
  - *invarianti di classe*
  - *precondizioni*
  - *postcondizioni*
- I contratti in presenza dell'ereditarietà si ereditano!
  - Nelle classi derivate:
    - le precondizioni devono essere uguali o più deboli
    - le postcondizioni devono essere uguali o più forti
    - gli invarianti devono essere uguali o più forti
  - In altre parole, **DERIVANDO** devo **CHIEDERE DI MENO** (o uguale, ma mai di più) e **GARANTIRE DI PIU'** (o uguale, ma mai di meno)

Il contratto (parziale) di Rectangle

```
-- a postcondition of Rectangle.Width

assert((width == w) && (height == old.height));
```



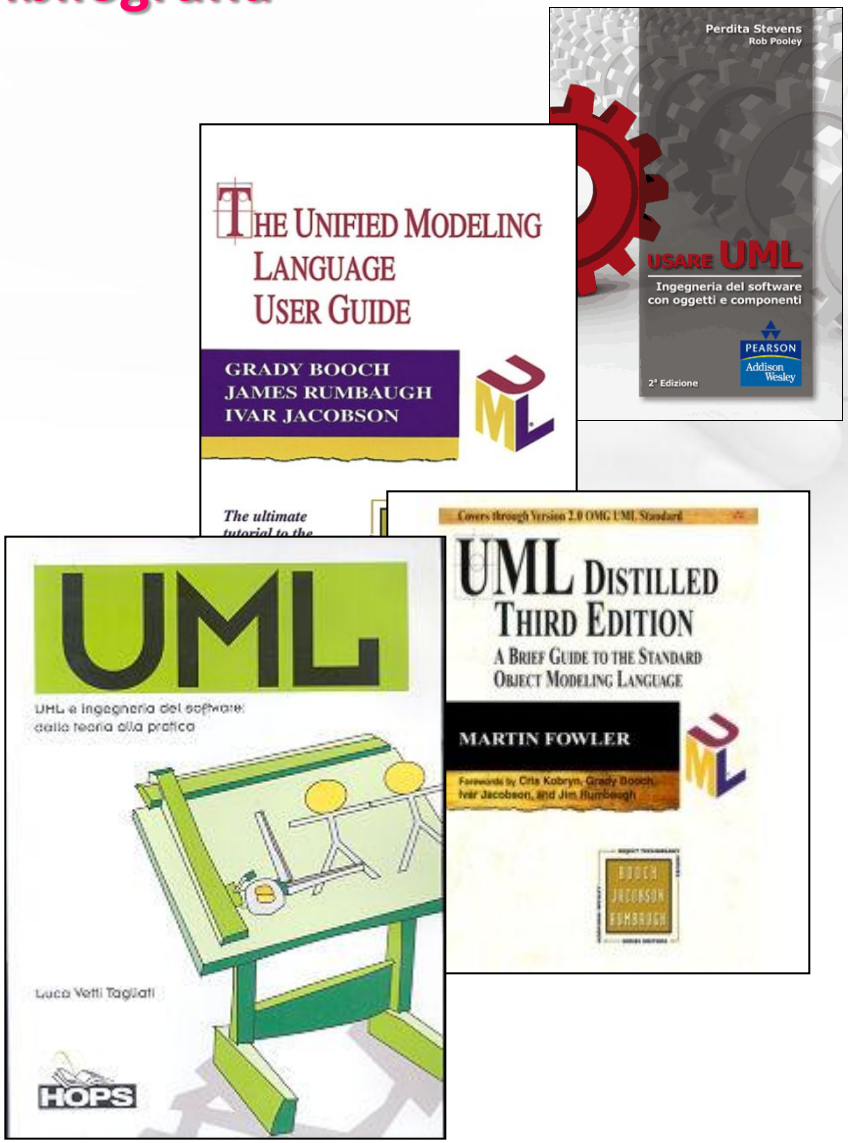
- Se equipaggiamo **Rectangle.Width** con questa asserzione, è facile verificare che:
  - la proprietà set di **Square.Width** (versione di **Width** nella classe derivata) ha una postcondizione più debole!
    - Infatti non richiede il vincolo:

      (height == old.height)
  - la classe **Square** viola in contratto della classe base!
    - in particolare, la proprietà **Square.Width** viola il contratto di **Rectangle**

LSP: considerazioni finali

- Rispettare il principio LSP è una linea guida fortemente raccomandata
  - È un trade-off: rispettare completamente LSP può essere costoso; talvolta "singoli casi degeneri" possono essere tollerati ma va valutato caso per caso
    - Ci deve essere una buona ragione economica per NON soddisfare LSP
    - Deve valere per casi estremamente isolati!
- Indizi che fanno capire una violazione di LSP
  - Pre-condizione nelle classi derivate più forte
  - Post-condizione nelle classi derivate più debole
  - Invariante di classe derivata più debole
  - Un metodo polimorfo degenere (implementazione nulla) solo nella classe derivata
    - Se c'è l'esigenza di "cancellare" (rimuovere) un metodo polimorfo a livello di classe derivata, allora è evidente che la classe derivata in questione non è un sostituto della classe base

Bibliografia



- P. Stevens, R. Pooley- “Usare UML: Ingegneria del Software con Oggetti e Componenti”, Addison-Wesley, 2008
- G. Booch, J. Rumbaugh, I. Jacobson- “The Unified Modeling Language User Guide 2/E”, Addison-Wesley, 2005
- M. Fowler – “UML Distilled: A Brief Guide to the Standard Object Modeling Language 3/E”, Addison Wesley, 2003
- L.V. Tagliati – “UML e Ingegneria del Software: dalla Teoria alla Pratica”, Tecniche Nuove, 2003

Domande?
Commenti?
Dubbi?