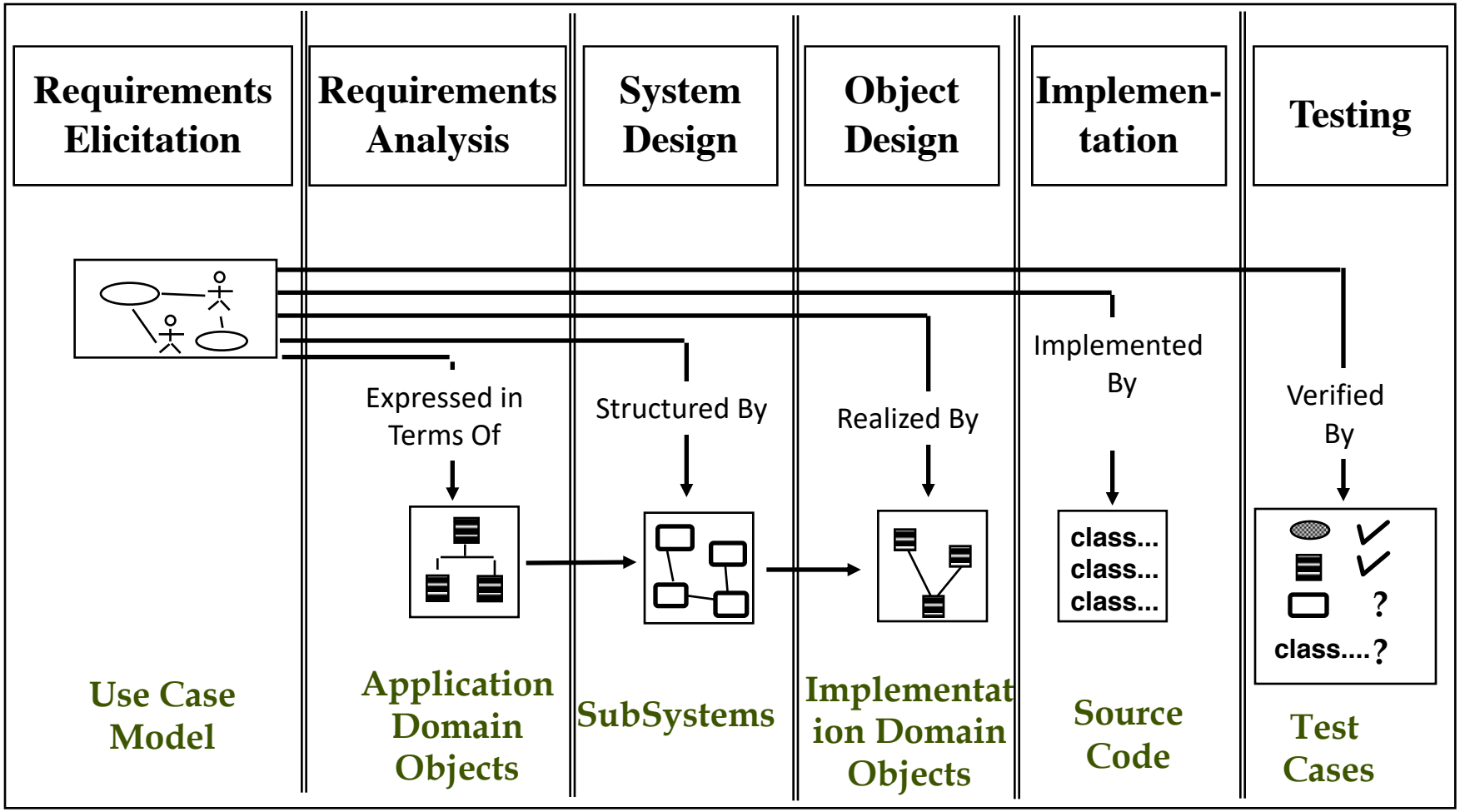
Object-Oriented Software Engineering
Conquering Complex and Changing Systems

Chapter 8, Object Design

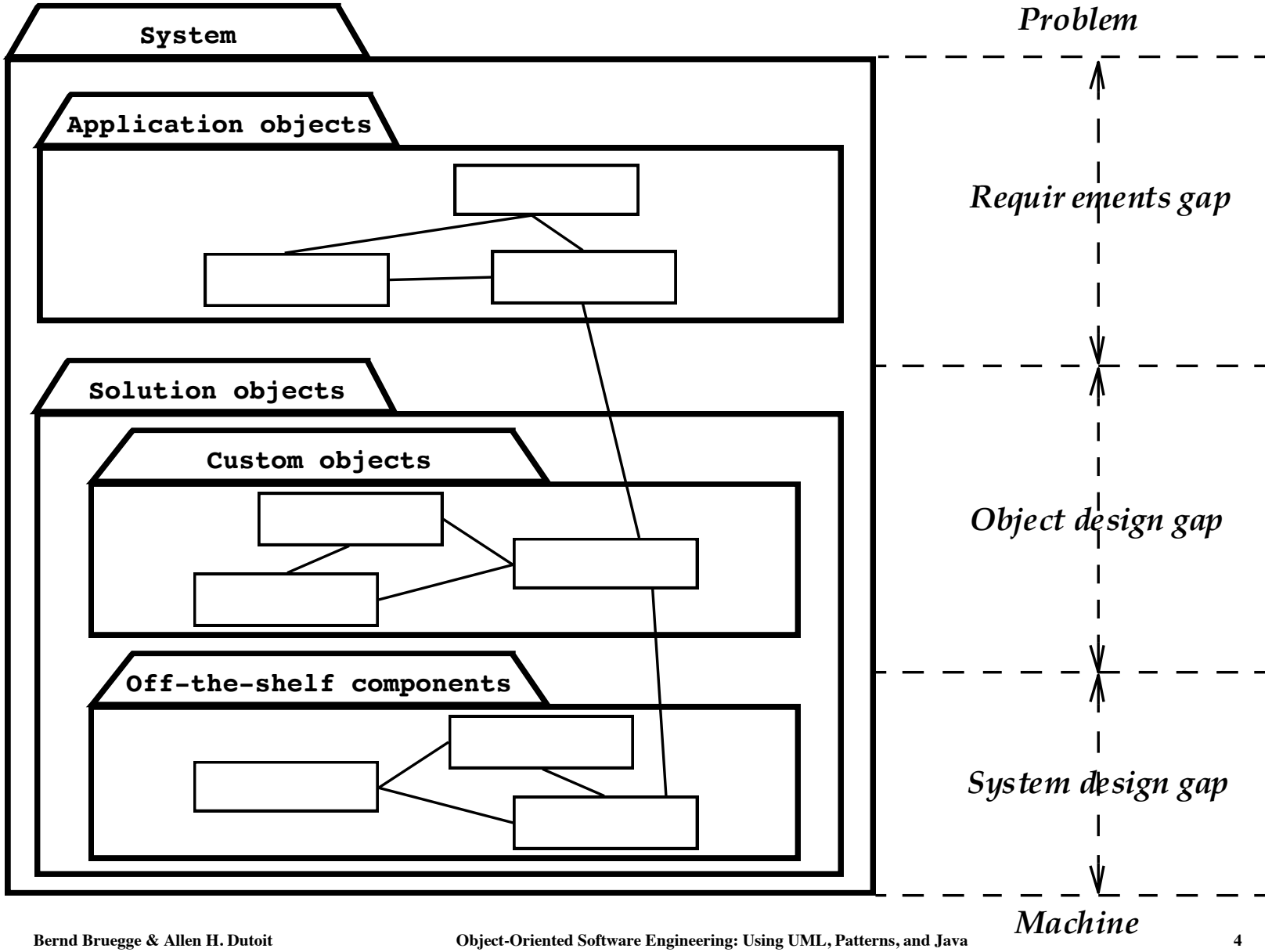
Software Lifecycle Activities



Object Design

- Durante l'analisi si descrive lo scopo del sistema, e si identificano gli oggetti di applicazione
- Durante il system design viene descritta l'architettura del sistema, la piattaforma HW/SW che permette di selezionare le componenti off-the-shelf, etc.
- Durante l'object design chiudiamo il gap tra oggetti di applicazione e componenti off-the-shelf identificando oggetti di soluzione e raffinando gli oggetti esistenti.
- L'Object Design include:
  - **Riuso**
  - **Specifica dei servizi**
  - **Ristrutturazione del modello ad oggetti**
  - **Ottimizzazione del modello ad oggetti**

Object Design: Chiudere il Gap



Object Design

- L'Object design è il processo che si occupa di
  - **aggiungere dettagli all'analisi dei requisiti e**
  - **prendere decisioni di implementazione**
- L'object designer deve scegliere fra diversi modi di implementare il modello di analisi con l'obiettivo di minimizzare il tempo di esecuzione, la memoria ed altri costi.
- Analisi dei Requisiti: Modello funzionale e modello dinamico definiscono le operazioni per il modello ad oggetti
- Object Design: Iteriamo nel processo di assegnazione delle operazioni al modello ad oggetti
- Object Design serve come base dell'implementazione

Attività Object Design - Riuso

- Le componenti off-the-shelf identificate durante il system design sono utilizzate nella realizzazione di ogni sottosistema
- Vengono selezionate librerie di classi ed altre componenti utili per strutture dati e servizi di base
- Vengono selezionati dei Design Pattern per risolvere problemi comuni e per proteggere classi da futuri cambiamenti
- Molte volte le componenti devono essere adattate prima di poterle utilizzare. Può essere fatto
  - **Attraverso oggetti wrapper**
  - **Raffinandoli utilizzando l'ereditarietà**
- Durante tutte queste attività gli sviluppatori devono decidere tra *buy-versus-build* trade-off

Attività Object Design – Specifica delle interfacce

- I servizi forniti dai sottosistemi (identificati durante il system design) sono specificati in termini di interfacce di classi, incluso operazioni, argomenti, tipi per le firme, ed eccezioni.
- Sono identificati anche ulteriori operazioni ed oggetti necessari per trasferire dati tra i sottosistemi.
- Il risultato di questa attività è una specifica completa delle interfacce per ogni sottosistema.
- La specifica delle interfacce dei sottosistemi è spesso chiamata "API (Application Programmer Interface) del sottosistema"

Attività Object Design – Ristrutturazione

- Il modello di sistema viene modificato per aumentare il riuso del codice o per soddisfare altri design goal .
- Attività tipiche sono:
  - **Trasformare associazioni N-arie in binarie**
  - **Implementare associazioni binarie attraverso riferimenti**
  - **Fondere classi simili in differenti sottosistemi in un'unica classe**
  - **Trasformare classi con nessun comportamento in attributi**
  - **Decomporre classi complesse in classi più semplici**
  - **Aumentare l'ereditarietà ed il packaging modificando classi ed operazioni**
- Durante la ristrutturazione ci si occupa anche di come soddisfare design goal come mantenimento, leggibilità, e comprensione del modello di sistema.

Attività Object Design – Ottimizzazione

- Durante l'ottimizzazione ci si occupa di soddisfare i requisiti di performance del modello di sistema.
- Questa attività include:
  - **Cambiare gli algoritmi per rispondere ai requisiti di memoria e velocità**
  - **Ridurre le molteplicità nelle associazioni per velocizzare le query**
  - **Aggiungere associazioni ridondanti per aumentare l'efficienza**
  - **Modificare l'ordine di esecuzione**
  - **Aggiungere attributi derivati per migliorare il tempo di accesso agli oggetti**
  - **Aprire l'architettura (aggiungere la possibilità di accedere a strati di basso livello)**

Attività di Object Design

- L'attività di object design non è sequenziale, di solito viene realizzata in maniera concorrente
- Ci potrebbero però essere delle dipendenze
  - **Una componente off-the-shelf può vincolare il tipo di eccezioni di un' operazione e quindi può influenzare l'interfaccia di un sottosistema**
  - **La ristrutturazione e l'ottimizzazione possono ridurre il numero di oggetti da implementare e quindi aumentare il riuso**
- Di solito vengono realizzate **prima** le attività di **riuso** e di **specifica delle interfacce**, ottenendo un modello ad oggetti di design che viene verificato rispetto ai corrispondenti casi d'uso
- Una volta che il modello si è stabilizzato vengono svolte le attività di **ristrutturazione** ed **ottimizzazione**.

Object-Oriented Software Engineering
Using UML, Patterns, and Java

Capitolo 9, Object Design: Specificare le Interfacce

Object Design: Specificare le interfacce

- Durante l'object design identifichiamo e raffiniamo gli oggetti "solution" per realizzare i sottosistemi definiti durante il system design
- La comprensione di ogni oggetto diventa più approfondita
- System design: il focus è l'identificazione di grandi parti di lavoro da assegnare ai vari team o sviluppatori
- Object design: il focus è la specifica dei confini tra i vari oggetti
- **Specifica delle interfacce**: il focus è
  - **comunicare chiaramente e precisamente i dettagli di basso livello degli oggetti del sistema**
  - **e descrivere precisamente l'interfaccia di ogni oggetto così che non ci sia necessità di lavoro di integrazione per oggetti realizzati da diversi sviluppatori**

(mentre aumenta il pressing per le consegne…)

Object Design: Specificare le interfacce

- Specifica delle interfacce, attività:
  - **Identificare attributi e operazioni mancanti**
  - **Specificare le signature e la visibilità di ogni operazione**
  - **Specificare le precondizioni (sotto le quali un'operazione può essere invocata e quelle che determinano un'eccezione)**
  - **Specificare le postcondizioni**
  - **Specificare le invarianti**
- *Object Constraint Language* (OCL) consente di specificare precondizioni, postcondizioni e invarianti
- Euristiche e linee guida ci consentono di scrivere vincoli leggibili

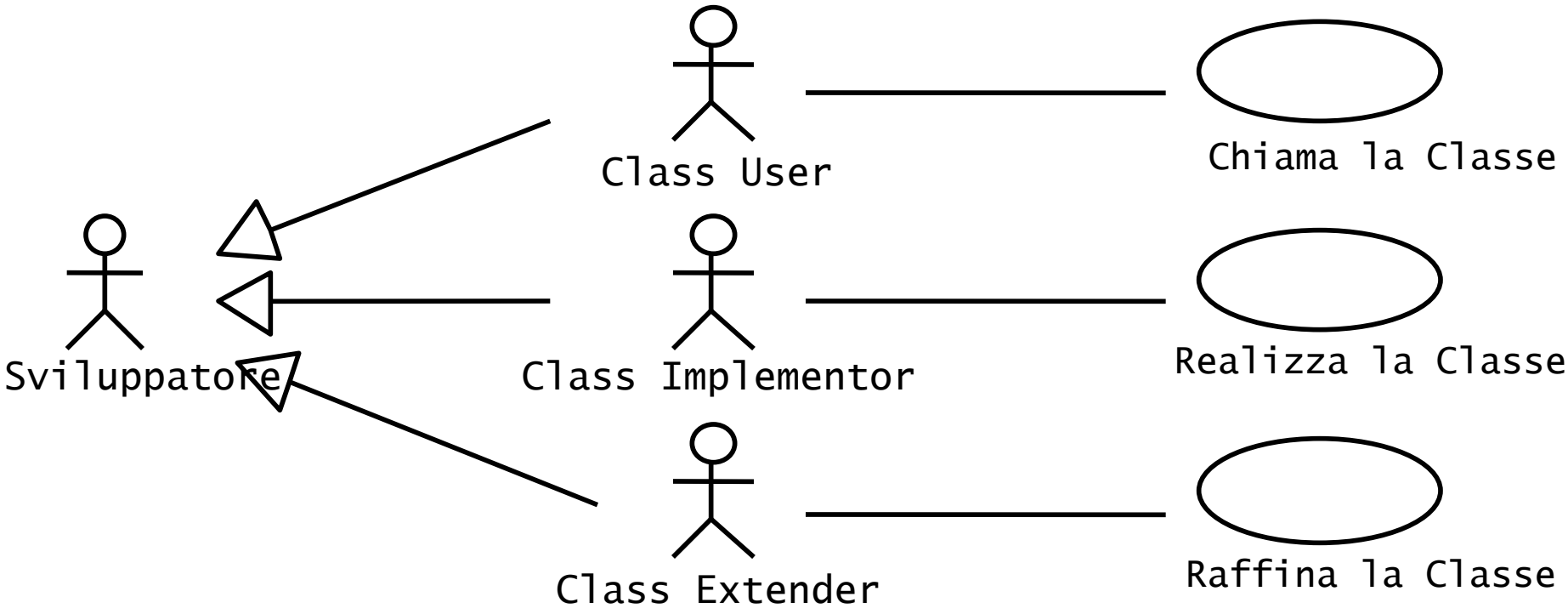
Overview della specifica delle interfacce

- Tutti i modelli sin qui costruiti forniscono **una visione parziale del sistema**, molti pezzi mancano e altri sono da raffinare:
  - Il modello a oggetti di analisi: descrive gli oggetti entity, boundary e control che sono visibili all'utente
  - La decomposizione in sottosistema: descrive come questi oggetti sono partizionati in pezzi coesi realizzati da diversi team. Ogni sottosistema fornisce un insieme di servizi (ad alto livello) ad altri sottosistemi
  - Il mapping Hardware/software: identifica le componenti che costituiscono la macchina virtuale su cui costruiamo gli oggetti soluzione (es. classi e API definite da componenti esistenti)
  - Use case Boundary: descrivono dal punto di vista dell'utente, casi amministrativi e eccezionali gestiti dal sistema
  - Design pattern (selezionati durante l'object design reuse): descrivono object design parziali che risolvono questioni specifiche

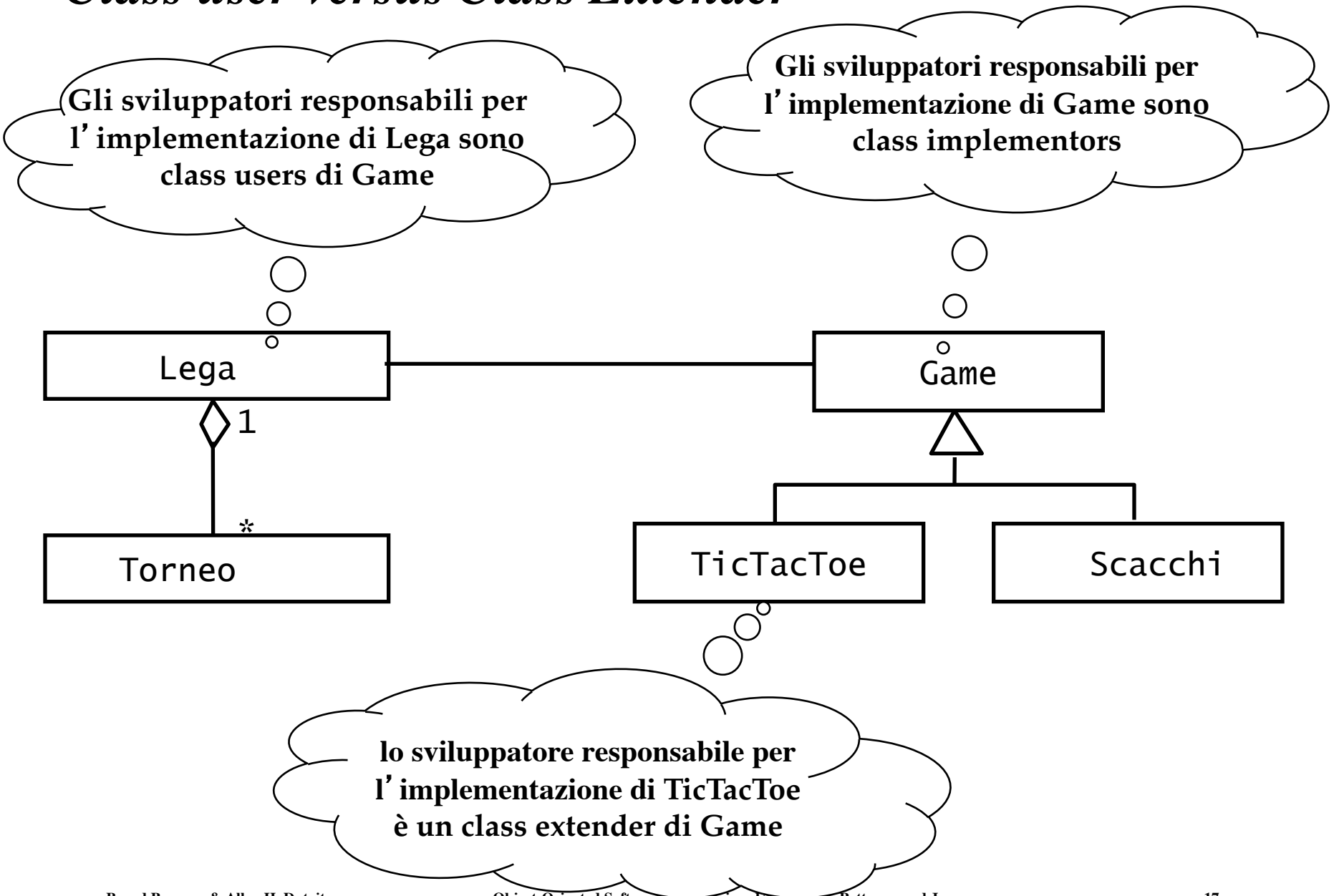
Overview della specifica delle interfacce

- L'obiettivo dell'object design è
  - **produrre un modello che integri tutte le informazioni in modo coerente e preciso**
- L'Object Design Document (**ODD**) contiene la specifica di ogni classe per supportare lo scambio di informazioni consentendo di prendere decisioni consistenti sia tra i vari sviluppatori che con gli obiettivi di design

Gli sviluppatori giocano ruoli diversi durante l'Object Design



Class user versus Class Extender



Specificare Interfacce

- Attività di analisi dei requisiti
  - **Identificare attributi e operazioni senza specificare il loro tipo e i loro parametri.**
- Object design: tre attività
  1. **Aggiungere informazione relativa alla visibilità**
     1. **Diversi sviluppatori hanno diverse necessità e non tutti accedono alle operazioni e agli attributi di una classe**
  2. **Aggiungere informazione sui tipi e sulle signature**
     1. **Il tipo di un attributo fornisce informazione sul range dei valori consentiti e le possibili operazioni**
     2. **La signature fornisce informazioni similari sui parametri delle operazioni e eventuale valori di ritorno**
  3. **Aggiungere contratti**
     1. **Consentono ai vari sviluppatori di condividere le stesse informazioni sulle classi**

Tipi e signature

- Il **tipo** di un attributo specifica il range dei valori che può avere quell'attributo e le operazioni che possono essere applicate
  - **ES. maxNumPlayers della classe Torneo rappresenta il max numero di giocatori che possono essere accettati in un certo Torneo.**
    - **Il suo tipo è int.**
    - **Le operazioni sono confronto, somma, sottrazione, o moltiplicazione di altri interi a maxNumPlayers**
- I parametri delle operazioni e i valori di ritorno hanno una specifica di tipo che vincola il range dei valori dei parametri e del valore restituito
- Signature dell'operazione: tupla che fornisce informazioni sui tipi dei parametri e del valore di ritorno
  - **ES: acceptPlayer() prende un parametro di tipo Player e non ha un valore di ritorno**
    - **Signature: acceptPlayer(Player): void**
  - **ES: getMaxNumPlayers() non prende parametri e restituisce un intero**
    - **Signature: getMaxNumPlayers() : int**

1. Aggiungere informazione sulla visibilità

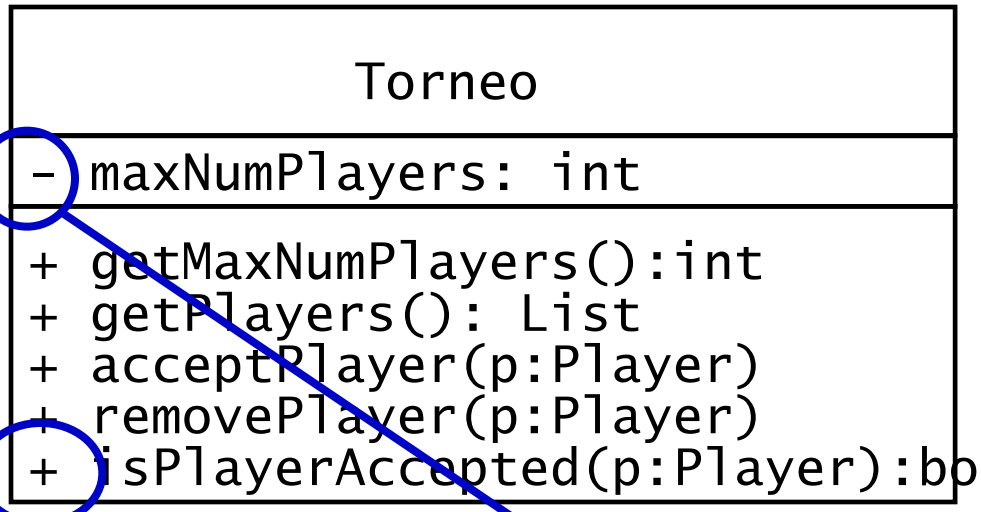
UML definisce 3 livelli di visibilità:

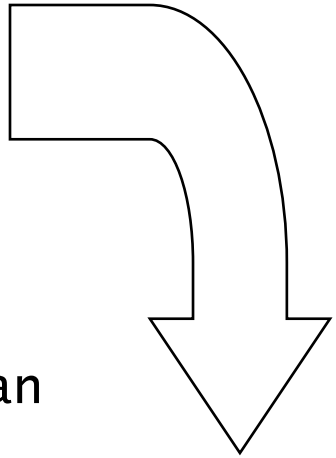
- : Privato (Solo per Class implementor):
  - **A un attributo privato può accedere solo la classe in cui è definito.**
  - **Un'operazione privata può essere invocata solo dalla classe in cui è definita.**
  - **Ad attributi e operazioni private non possono accedere sottoclassi o altre classi.**

#: Protetto (Class extender):
  - **A un attributo o operazione protetto può accedere solo la classe in cui è definito e ogni discendente della classe.**

+: Pubblico (Class user):
  - **A un attributo o operazione pubblica possono accedere tutte le classi (interfaccia pubblica).**

Implementazione della visibilità UML in Java





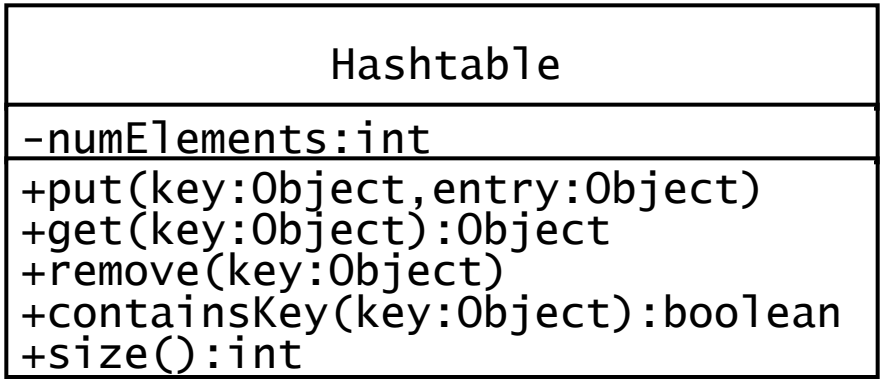
```
public class Torneo {
        private int maxNumPlayers;

        public Tournament(League l, int
maxNumPlayers)
        public int getMaxNumPlayers() {…};
        public List getPlayers() {…};
        public void acceptPlayer(Player p) {…};
        public void removePlayer(Player p) {…};
        public boolean isPlayerAccepted(Player p)
{…};
```

Euristiche per Information Hiding

- Definisci attentamente l'interfaccia pubblica per le classi così come per i sottosistemi
- Applica sempre il principio "Need to know".
  - **Solo se qualcuno necessita di accedere all'informazione, rendilo possibile , ma solo attraverso ben definiti canali, in modo che conosci sempre l'accesso.**
- Meno una operazione sa
  - **Più bassa sarà la probabilità che sarà influenzata da qualche cambiamento**
  - **Più facilmente la classe potrà essere cambiata**
- Trade-off: Information hiding vs efficienza
  - **Accedere a un attributo privato potrebbe essere troppo lento (per esempio per sistemi in real-time o giochi)**

2. Aggiungere Informazione di Tipo alle Signature



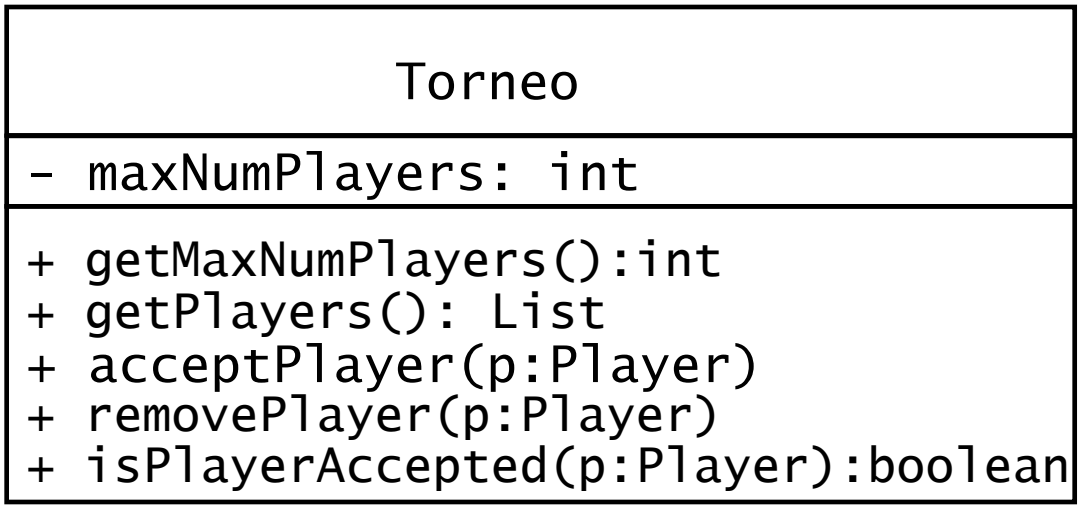
3. Aggiungere Contratti

- Spesso l'informazione di tipo non è sufficiente a specificare il range dei valori consentiti di un attributo
  - **Es. maxNumPlayers di tipo int può assumere valori negativi**
  - **Si possono aggiungere i contratti**
- Contratti su una classe consentono a class users, implementors e extenders di condividere le stesse assunzioni sulla classe.

3. Aggiungere Contratti

- Contratti includono 3 tipi di vincoli:
- Invariante:
  - Un predicato che è sempre vero per tutte le istanze di una classe. Invarianti sono vincoli associati a classi o interfacce.
- Precondizioni:
  - Precondizioni sono predicati associati con una **specifica operazione** e deve essere vera **prima** che l'operazione sia invocata. Precondizioni sono usate per specificare vincoli che un chiamante deve soddisfare prima di chiamare un'operazione.
- Postcondizione:
  - Postcondizioni sono predicati associati con **una specifica operazione** e devono essere vere **dopo** che l'operazione è stata invocata. Postcondizioni sono usate per specificare vincoli che l'oggetto deve assicurare dopo l'invocazione dell'operazione

Contratti: Esempi



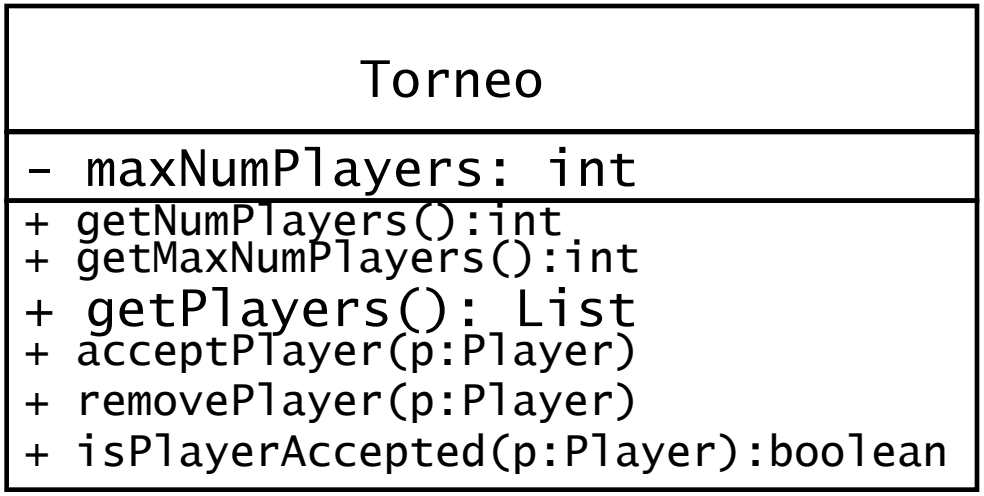
- **Invarianti**:

  il max numero di Players dovrebbe essere positivo poiché se fosse creato un Torneo con maxNumPlayers = 0, allora acceptPlayer() violerebbe sempre il suo contratto e il Torneo non potrebbe mai iniziare.

Indichiamo con t un Torneo

t.getMaxNumPlayers() > 0

Contratti: Esempi



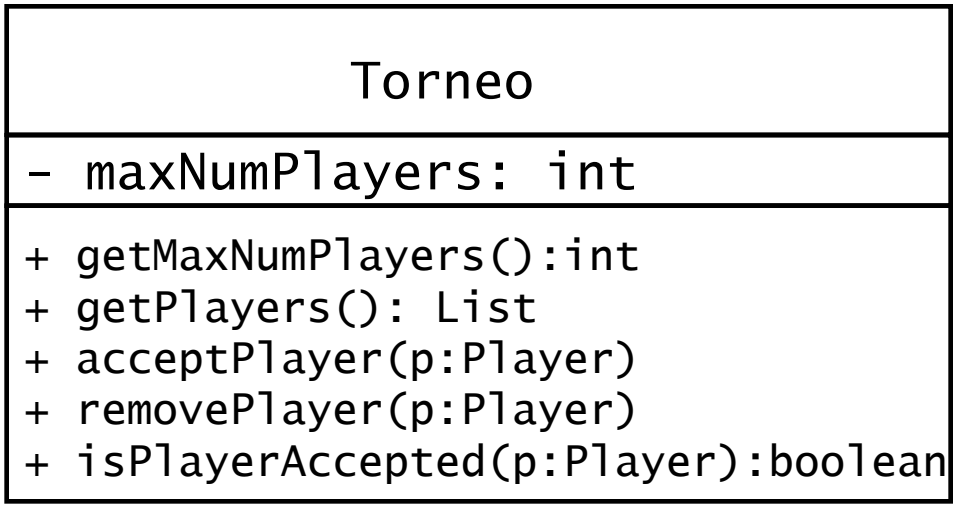
- **Precondizione**:

  ES. per acceptPlayer()

  Il Player da aggiungere non dovrebbe essere già stato accettato nel Torneo e che Torneo non abbia ancora raggiunto il numero max di Player

  ! t.isPlayerAccepted(p) and t.getNumPlayers < t.getMaxNumPlayers()

Contratti: Esempi



- **Postcondizione**:

  ES. per acceptPlayer()

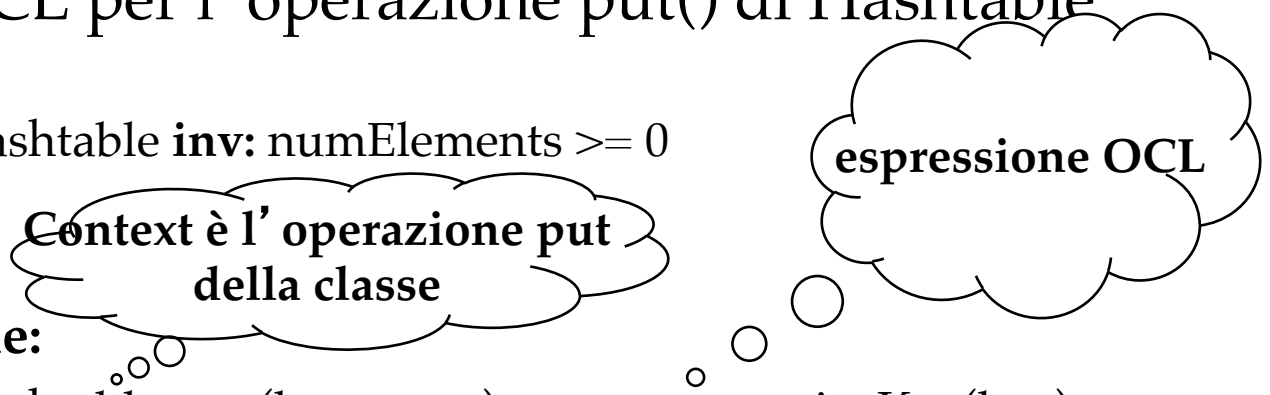
  Il numero corrente di Player deve essere esattamente 1 in più rispetto al numero di Player prima dell'invocazione del metodo

  t.getNumPlayers_afterAccept = t.getNumPlayers_beforeAccept + 1

- Invarianti, precondizioni e postcondizioni possono essere usati per specificare senza ambiguità casi speciali o eccezionali

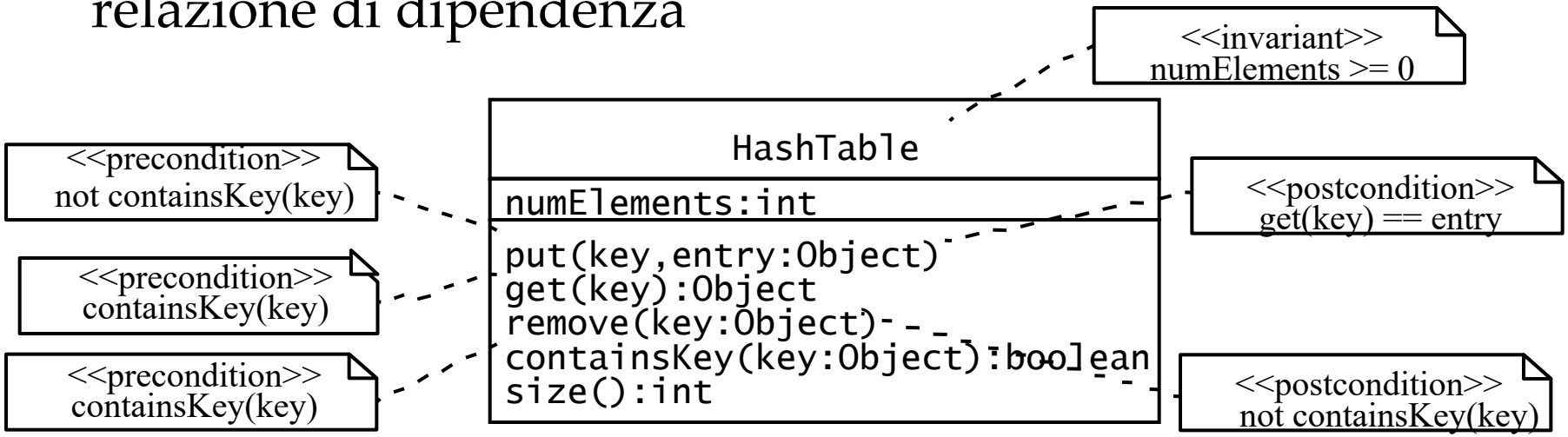
Esprimere Constraints nei Modelli UML

- OCL (Object Constraint Language)
  - OCL consente di specificare formalmente i vincoli sugli elementi di un singolo modello (attributi, operazioni, classi) o gruppi di elementi di modello (associazioni e classi partecipanti)
  - Un constraint è espresso come una espressione OCL che ritorna un valore vero o falso. OCL non è un linguaggio procedurale (non si può vincolare il control flow).
- Espressioni OCL per l'operazione put() di Hashtable
  - **Invariante:**
    - **context** Hashtable **inv:** numElements >= 0
  - **Precondizione:**
    - **context** Hashtable::put(key, entry) **pre:** not containsKey(key)
  - **Post-condizione:**
    - **context** Hashtable::put(key, entry) **post:** containsKey(key) and get(key) = entry



Esprimere Constraints nei Modelli UML

- Un constraint può anche essere illustrato come una nota attaccata all'elemento UML vincolato tramite una relazione di dipendenza



Contratti per acceptPlayer in Torneo

```
context Tournament::acceptPlayer(p) pre:
  not isPlayerAccepted(p)

context Tournament::acceptPlayer(p) pre:
  getNumPlayers() < getMaxNumPlayers()

context Tournament::acceptPlayer(p) post:
  isPlayerAccepted(p)

context Tournament::acceptPlayer(p) post:
  getNumPlayers() = @pre.getNumPlayers() + 1
```

Contract for removePlayer in Tournament

```
context Torneo::removePlayer(p) pre:
  isPlayerAccepted(p)

context Torneo::removePlayer(p) post:
  not isPlayerAccepted(p)

context Torneo::removePlayer(p) post:
  getNumPlayers() = @pre.getNumPlayers() - 1
```

Annotazione della classe Torneo

```
public class Torneo {
   /** Il Massimo numero di players
    * è sempre positivo.
    * @invariant maxNumPlayers > 0
    */
   private int maxNumPlayers;

   /** La List di players contiene
    * referimenti ai Players che
    * sono registrati
    * al Torneo. */
   private List players;

   /** Restituisce in numero
   corrente
    * di players nel torneo. */
   public int getNumPlayers() {…}

   /** Restituisce in numero massimo
    * di players nel torneo. */
   public int getMaxNumPlayers() {…}
```

```
   /** L'operazione acceptPlayer()
    * assume che il player
    * specificato non è stato ancora
    * accettato nel Torneo.
    * @pre not isPlayerAccepted(p)
    * @pre getNumPlayers()<maxNumPlayers
    * @post isPlayerAccepted(p)
    * @post getNumPlayers() =
    *       @pre.getNumPlayers() + 1
    */
   public void acceptPlayer (Player p)
   {…}

   /** L'operazione removePlayer()
    * assume che il player
    * specificato è al momento nel
   Torneo.
    * @pre isPlayerAccepted(p)
    * @post not isPlayerAccepted(p)
    * @post getNumPlayers() =
         @pre.getNumPlayers() - 1
    */
   public void removePlayer(Player p) {…}

}
```

Gestione dell'Object Design

- Due principali problemi di gestione durante OD
  - **Aumento della complessità di comunicazione**
    - Il numero di persone che prendono parte all'OD aumenta notevolmente. E' necessario **assicurare che le decisioni prese siano in accordo con gli obiettivi del progetto**
  - **Consistenza con le precedenti decisioni e documenti**
    - Dettagliando e raffinando il modello ad oggetti, gli sviluppatori possono rivedere alcune decisioni prese durante le fasi precedenti. **Occorre tener traccia di questi cambiamenti e assicurarsi che tutti i documenti li riflettano in modo consistente**

Object Design Document (ODD)

- ODD serve per scambiare informazione sulle interfacce tra i team e come riferimento per il testing. È rivolto a:
  - Architetti che partecipano al system design
  - Sviluppatori che realizzano ogni sottosistema
  - Tester

Object Design Document

1. **Introduzione**
   - 1.1 Object Design Trade-offs
   - 1.2 Linee Guida per la Documentazione delle Interfacce
   - 1.3 Definizioni, acronimi e abbreviazioni
   - 1.4 Riferimenti
2. **Packages**
3. **Class interfaces**
4. **Class Diagram**

Glossario

ODD

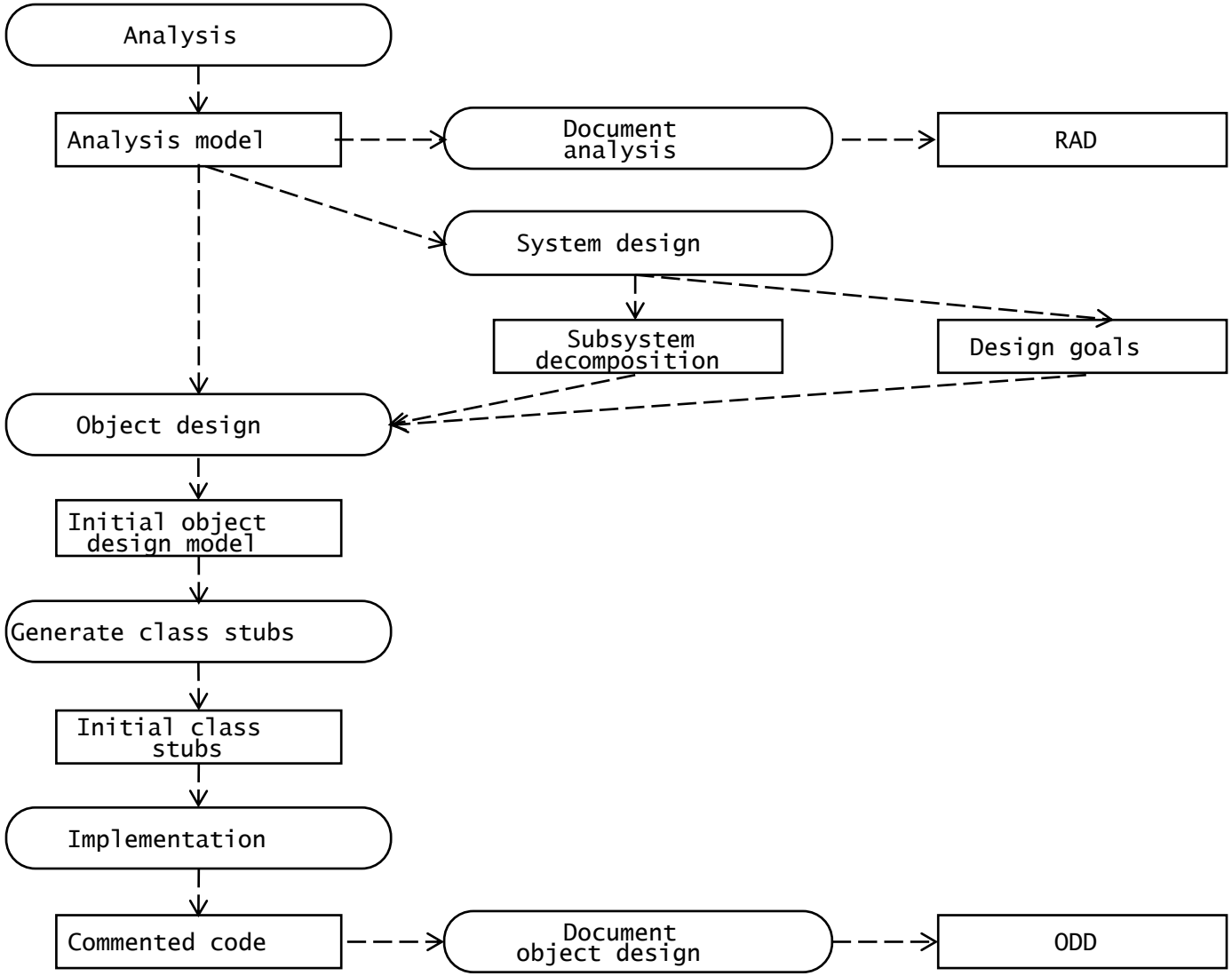
1. **Introduzione**
   - una descrizione dell'analisi dei trade-off realizzati dagli sviluppatori (comprare vs. costruire, spazio di memoria vs. tempo di risposta, ecc)
   - convenzioni e linee guida che servono a migliorare la comunicazione. Devono essere definite prima dell'inizio dell'OD e non devono variare
     - **alle classi sono assegnati nomi singolari**
     - **I metodi nominati con verbi, i campi e i parametri con i sostantivi**
     - **Lo status di un errore è restituito via un'eccezione, non con un valore di ritorno**
     - **…**
2. **Packages**
   - Descrive la decomposizione di sottosistemi in package e l'organizzazione in file del codice. Inoltre le dipendenze tra i package e il loro uso

3. **Class interfaces (Interfacce delle Classi)**
   - Descrive le classi e le loro interfacce pubbliche (overview di ogni classe, sue dipendenze con altre classi e package, i suoi attributi e operazioni pubblici, casi eccezionali)

Javadoc

- Le sezioni **Package** e **Class Interfaces** dell'ODD possono essere generati da un tool utilizzando i commenti del codice sorgente
- Javadoc genera pagine web dai commenti del codice
  - **Gli sviluppatori annotano le dichiarazioni di interfacce e classi con commenti tagged**
  - **Usando i vincoli è anche possibile includere pre e post condizioni nell'header dei metodi**
- tenendo insieme materiale per ODD e il codice sorgente consente di mantenere la consistenza più facilmente

Embedded ODD approach. Class stubs are generated from the object design model.



Javadoc

- Java SDK contiene uno strumento molto utile, chiamato javadoc, che genera documentazione HTML dal file sorgente
  - La documentazione delle classi predefinite dei package Java è prodotta in questo modo
  - I commenti devono cominciare con il delimitatore speciale /**

- Vantaggio: si mantiene codice e documentazione nello stesso file
  - Il codice e i commenti si possono aggiornare e la documentazione può essere riprodotta con javadoc

Documentazione di Javadoc

- Disponibile in linea alla URL
  - **http://java.sun.com/products/jdk/javadoc/index.html**

Informazioni estratte da Javadoc

- L'utility javadoc estrae informazioni relative ai seguenti elementi:
  - Package
  - Classi e interfacce pubbliche
  - Metodi pubblici e protetti
  - Campi pubblici e protetti

- E' possibile fornire commenti per ognuno di tali elementi
  - Ogni commento viene inserito immediatamente sopra la funzione che descrive
  - Comincia con /** e termina con */

Come inserire commenti

- Ogni commento di documentazione contiene testo formattato liberamente seguito da tag
  - **Un tag comincia con un segno @ come @author o @param**
- La prima frase del testo deve essere una frase di riepilogo
  - **L'utility javadoc genera automaticamente pagine di sommario che estraggono tali frasi**
  - **Nel testo libero è possibile utilizzare modificatori HTML**

Commenti alle classi

- Il commento alla classe deve essere inserito dopo ogni dichiarazione import prima della definizione di classe

```
/**
   Un oggetto <code>Card</code> rappresenta una carta da gioco,
   come "Regina di cuori". Una carta ha un seme (Cuori, Quadri, Fiori
   o Picche) e un valore (1 = Asso, 2 … 10, 11 = Fante, 12 = Regina,
   13 = Re).
*/

public class Card
{
…
}
```

Commenti ai metodi

- Ogni commento a un metodo deve trovarsi appena prima del metodo che descrive.
- Tag che si possono usare
  - **@param** ***variabile descrizione***
    - **Aggiunge una voce alla sezione "parametri" del metodo corrente.**
    - **La descrizione può essere su più righe e utilizzare tag HTML**
    - **Tutti i tag @param per un metodo devono stare insieme**
  - **@return** ***descrizione***
    - **Aggiunge una sezione "return" al metodo corrente**
    - **La descrizione può essere su più righe e utilizzare tag HTML**
  - **@throws** ***classe descrizione***
    - **Aggiunge una nota che dice che il metodo può generare un'eccezione**

Commenti ai metodi: esempio

```
/**
   Aumenta lo stipendio di un impiegato.
   @param byPercent la percentuale di cui aumentare
                    lo stipendio (es. 10 = 10%)
   @return la quantità di aumento
*/
public double raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

Commenti ai campi

- E’ necessario commentare solo i campi pubblici
  - **Nella maggior parte dei casi le costanti statiche**

```
/**
   Il seme “Cuori” delle carte
*/
public static final int CUORI = 1;
```

Placement of comments

- Documentation comments are recognized only when placed immediately before class, interface, constructor, method, or field declarations
- Documentation comments placed in the body of a method are ignored.
- A common mistake is to put an import statement between the class comment and the class declaration. Avoid this, as the Javadoc tool will ignore the class comment.

```
/**
 This is the class comment for the class Whatever.
*/
import com.sun; // MISTAKE - Important not to put import statement here

public class Whatever {
 }
```

Commenti generali

- @author *nome* - genera una voce autore
- @version *testo* - genera una voce versione
- @since *testo* - genera una voce "da"
  - **es. @since ver 1.7.1**

- @deprecated *testo*
  - aggiunge un commento che dice che l'elemento non dovrebbe più essere utilizzato. Il testo dovrebbe suggerire una sostituzione

- @see *collegamento*
  - Aggiunge un collegamento ipertestuale nella sezione "see also"
  - Collegamento può essere:
    - **"string"**
    - **etichetta**

Commenti generali

- **@see** "*string*"
  - Adds a text entry for *string*.
  - No link is generated.
  - The *string* is a book or other reference to information not available by URL.
  - The Javadoc tool distinguishes this from the previous cases by looking for a double-quote (") as the first character.
  - For example: @see "The Java Programming Language" This generates text such as:
  - **See Also:**
    - "The Java Programming Language"

Commenti generali

- **@see** *label*
  - Adds a link as defined by *URL#value*.
  - The Javadoc tool distinguishes this from other cases by looking for a less-than symbol (<) as the first character.
  - For example:

    @see Java Spec

    This generates a link such as:

    See Also:

    – Java Spec

Come estrarre i commenti

- Sia docDirectory la directory in cui si desiderano vadano i file HTML di documentazione
  - 1. Andare nella directory che contiene i file sorgente che si desidera documentare
  - 2. Eseguire il comando
    - Javadoc –d *docDirectory nomePackage*

    per un solo package, oppure eseguire
    - Javadoc –d *docDirectory nomePackage1 nomePackage2 …*

    per documentare più package
  - Se i file si trovano nel package predefinito eseguire invece
    - javadoc –d *docDirectory* *.java
  - Se si omette l'opzione –d directory i file HTML vengono estratti nella directory corrente

Documentazione di Javadoc

- Disponibile in linea alla URL
  - **http://java.sun.com/products/jdk/javadoc/index.html**
- Per vedere le possibili opzioni basta eseguire il comando senza argomenti
  - **javadoc**