Modern Methods in Software Engineering

Object Design - Interface Design

Literature used

Text book

Chapter 9

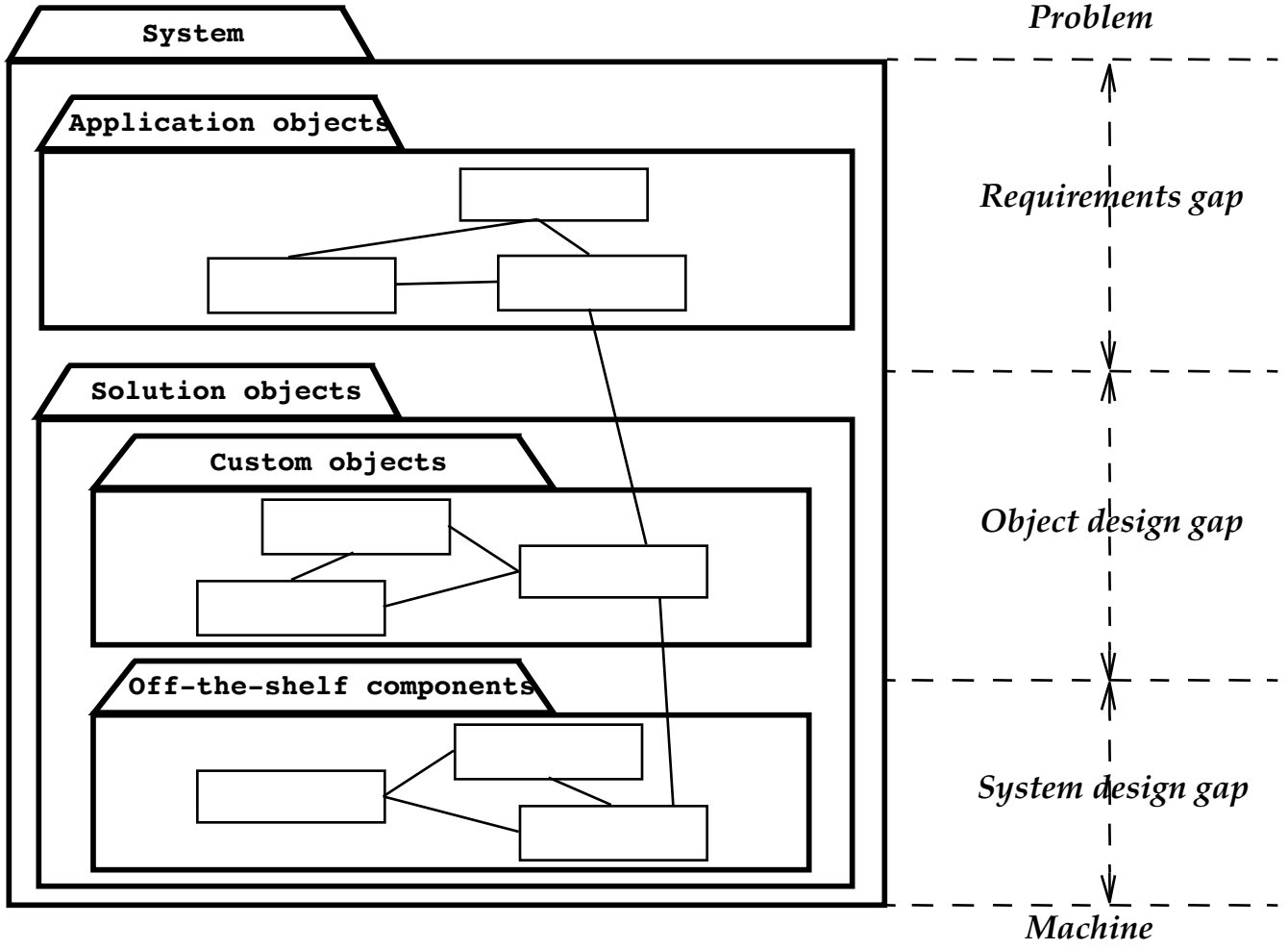
Recommended reading:

Jos Warmer, Anneke Kleppe. The Object Constraint Language, 1999

Content

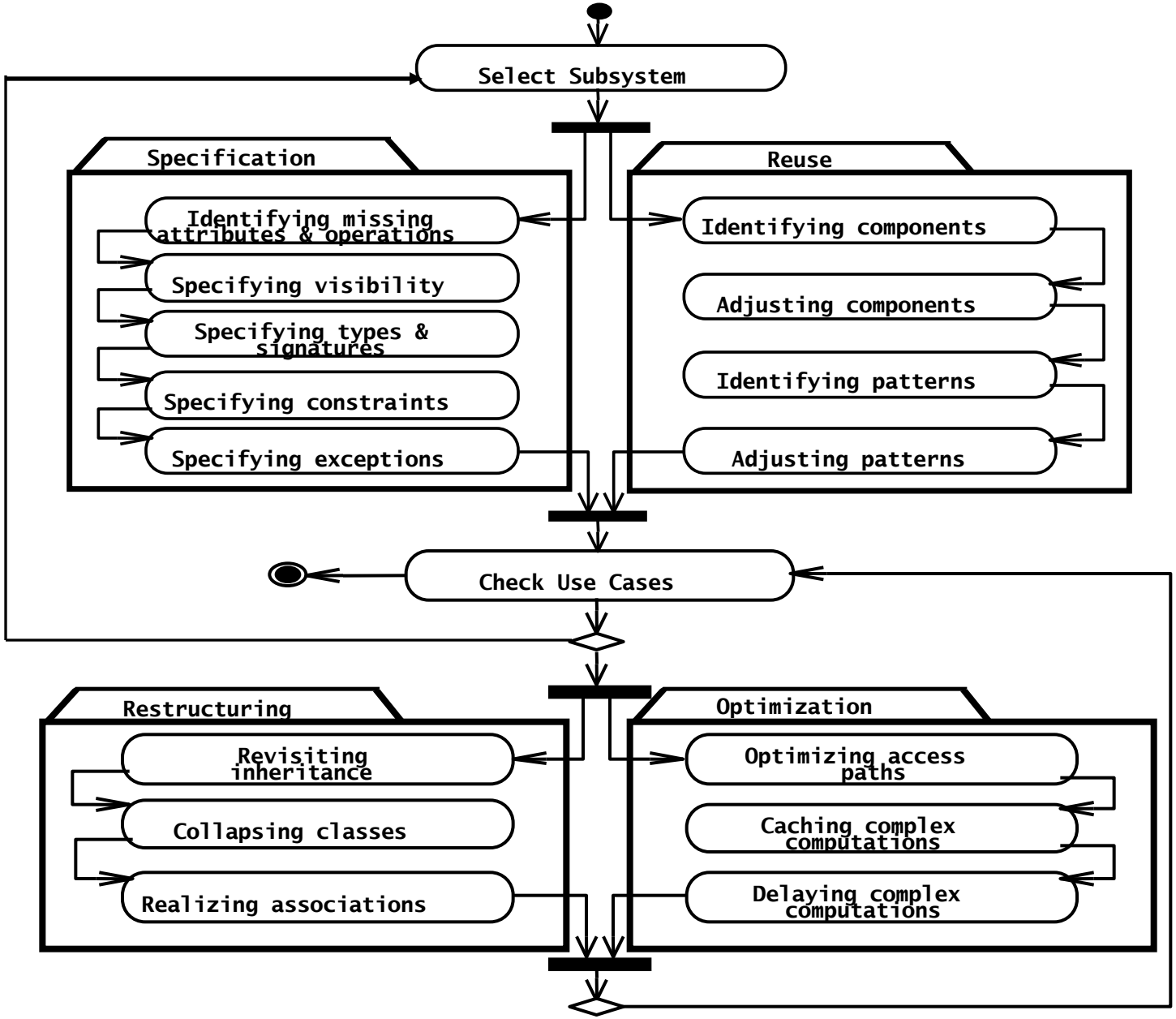
- Interface specification activities
- Visibility information
- Type signature information
- Contracts
- Constraints and Object Constraint Language (OCL)

Object Design: Closing the Gap



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

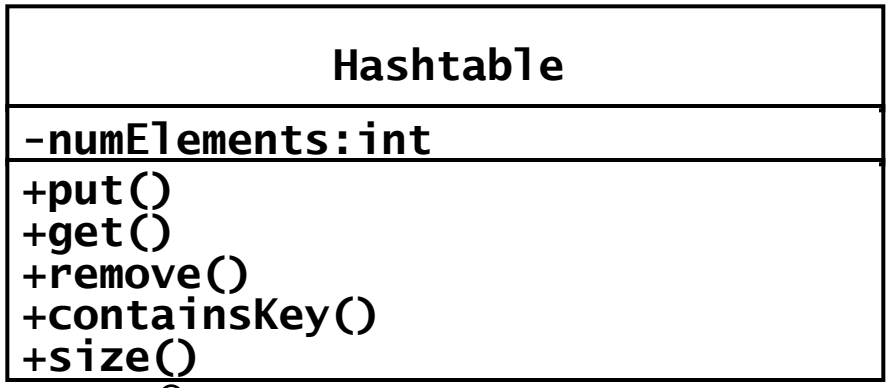
Object Design Activities

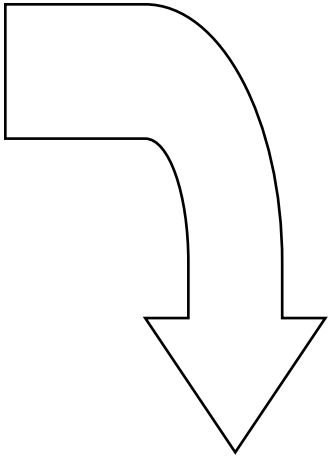


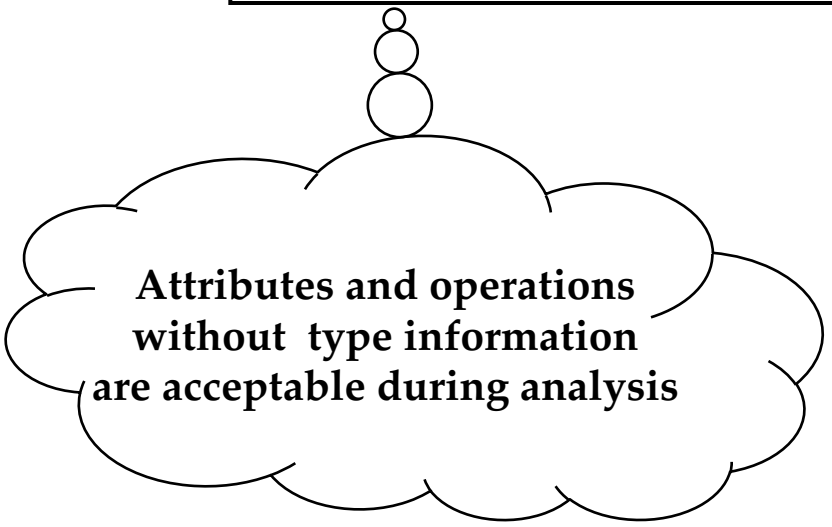
Some Interface Specification Activities

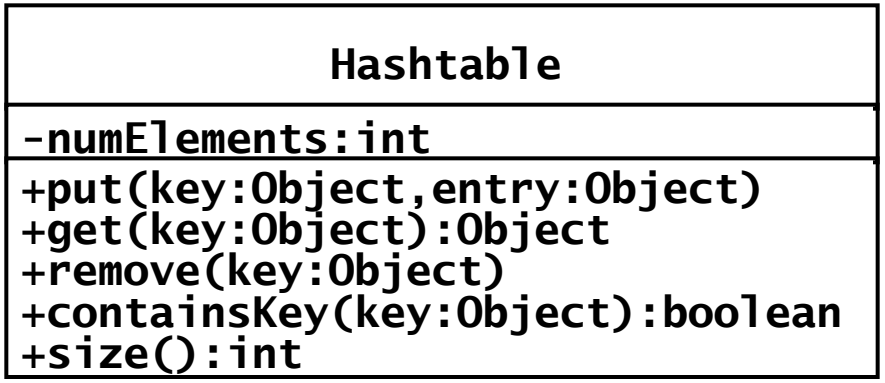
- Requirements analysis activities
  - Identifying attributes and operations without specifying their types or their parameters.
- Object design:
  - identifying missing attributes and operations
  - specifying type signatures and visibility
  - specifying invariants
  - specifying preconditions and postconditions.

Add Type Signature Information

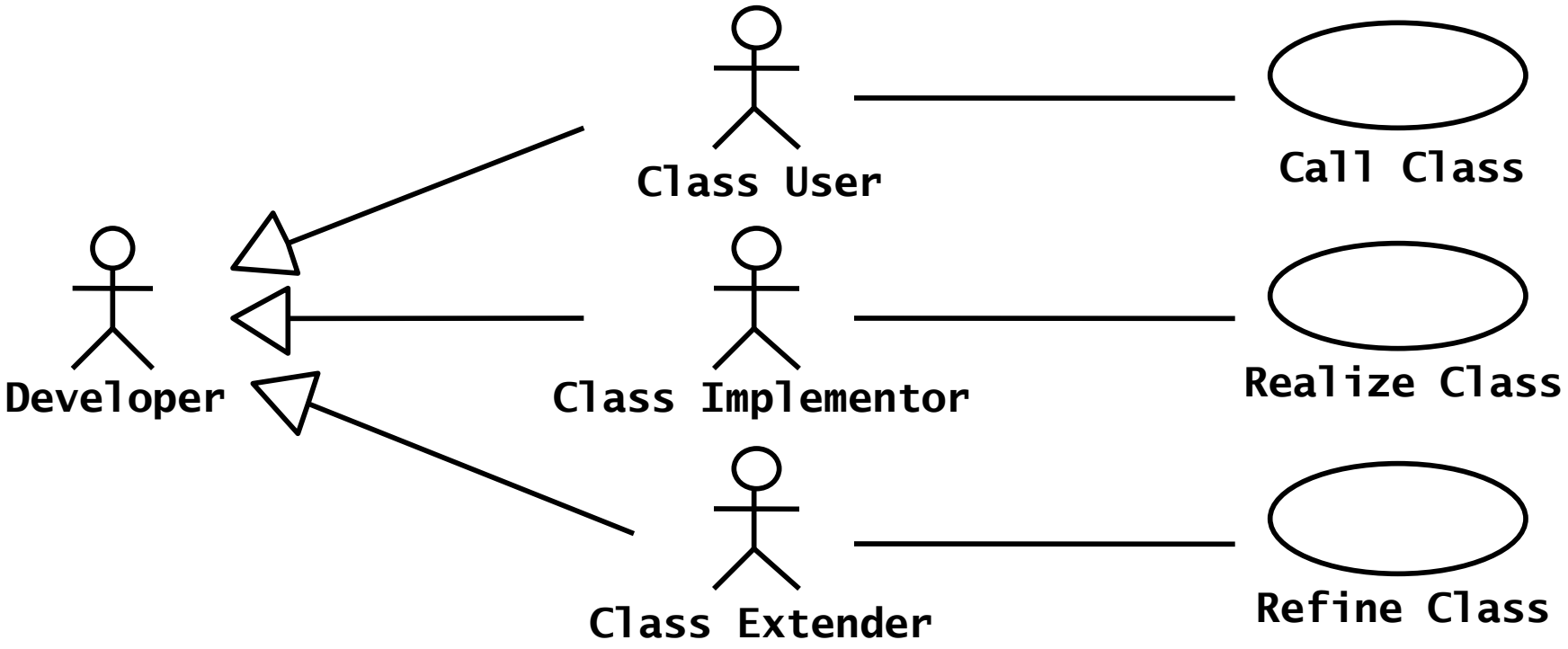




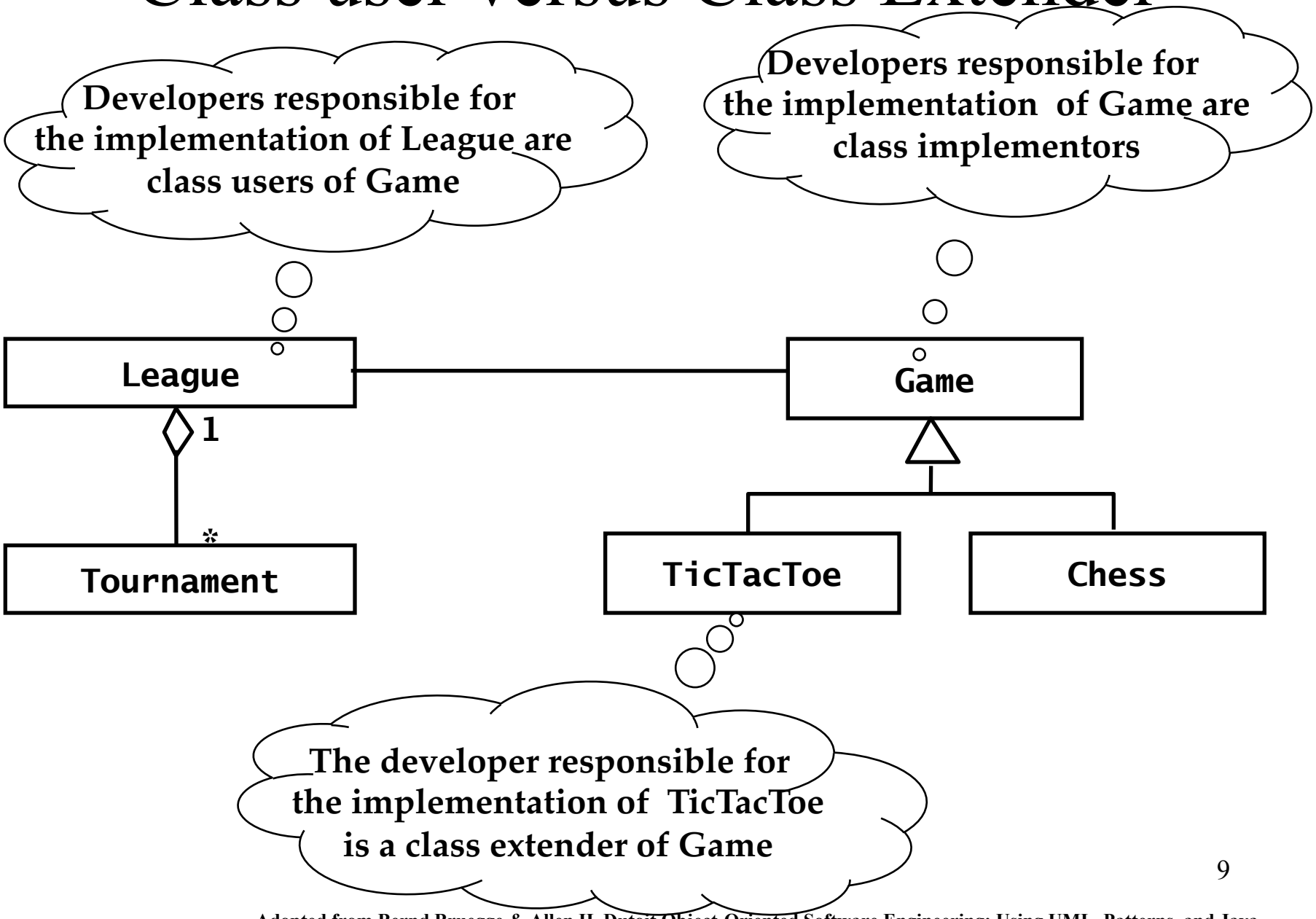




Developers play different Roles during Object Design



Class user versus Class Extender

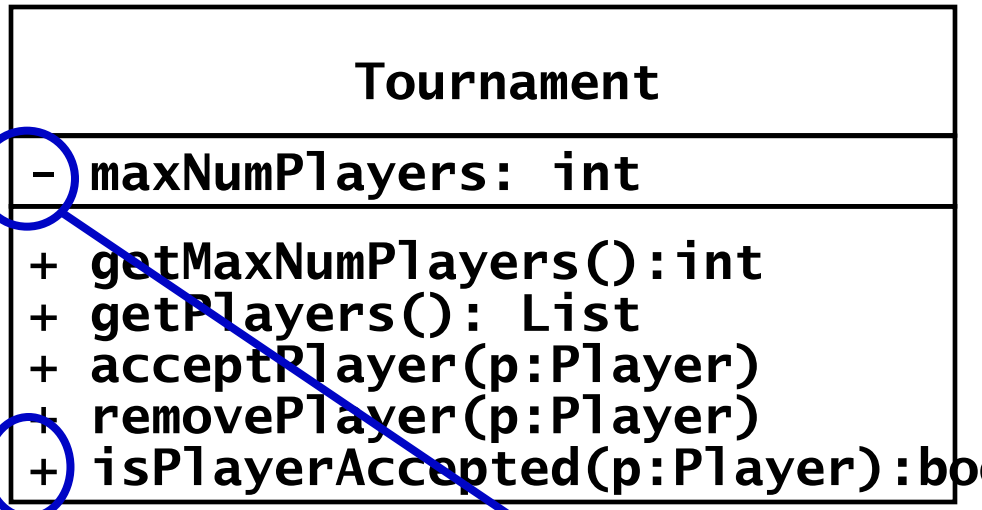


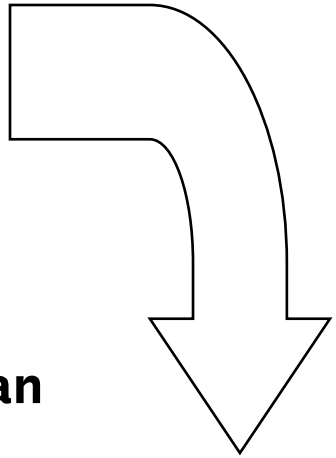
Add Visibility Information

UML defines three levels of visibility:

- Private (Class implementor):
- Protected (Class extender):
- Public (Class user):

Implementation of UML Visibility in Java





```
public class Tournament {
        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) {…};
```

Some Information Hiding Heuristics

- Carefully define the public interface for classes as well as subsystems (façade)
- Always apply the "Need to know" principle.
- The fewer an operation knows the better

Information Hiding Design Principles

- Only the operations of a class are allowed to manipulate its attributes
- Trade-off: Information hiding vs efficiency
- Do not apply an operation to the result of another operation.

Add Contracts

- Contracts are constraints on a class enable caller and callee to share the same assumptions about the class.

- Contracts include three types of constraints:
  - Invariant:
  - Precondition:
  - Postcondition:

Examples (for `Tournament` class)

- Invariant

```
t.getMaxNumPlayers() > 0
```

- Pre-conditions (for **`AcceptPlayer`**)

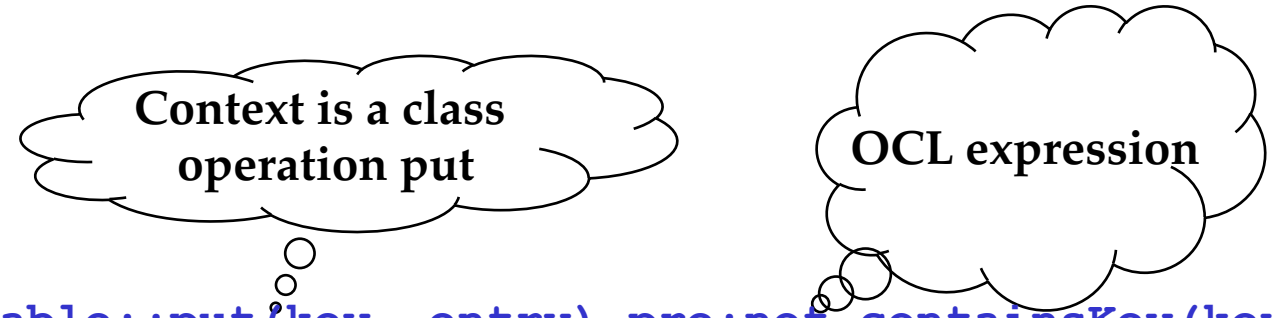
```
not isPlayerAccepted(p) and
getNumPlayers() < getMaxNumPlayers()
```

- Post-conditions (for **`AcceptPlayer`**)

```
getNumPlayers_afterAccept =
  getNumPlayers_beforeAccept + 1
```

Expressing contracts in UML Models OCL (Object Constraint Language)

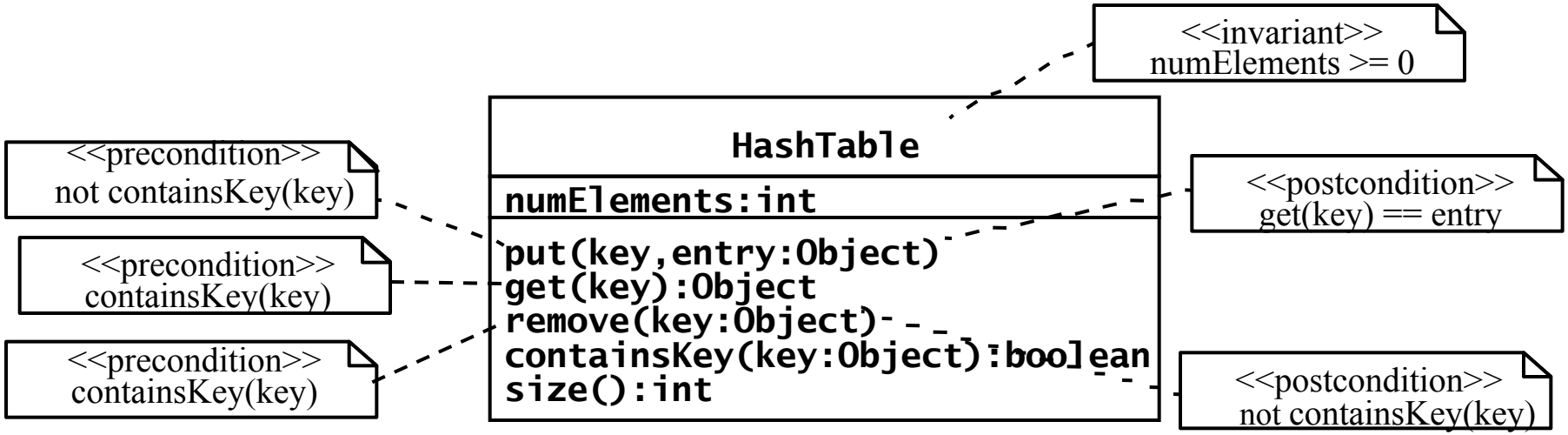
- OCL expressions for Hashtable operation **`put()`:**
  - Invariant:
    - `context Hashtable inv: numElements >= 0`



  - Precondition:
    - `context Hashtable::put(key, entry) pre:not containsKey(key)`
  - Post-condition:
    - `context Hashtable::put(key, entry) post: containsKey(key) and get(key) = entry`

Expressing Constraints in UML Models

- A constraint can also be depicted as a note attached to the constrained UML element by a dependency relationship.



Contract for acceptPlayer in Tournament

```
context Tournament inv:
  self.getMaxNumPlayers() > 0

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
```

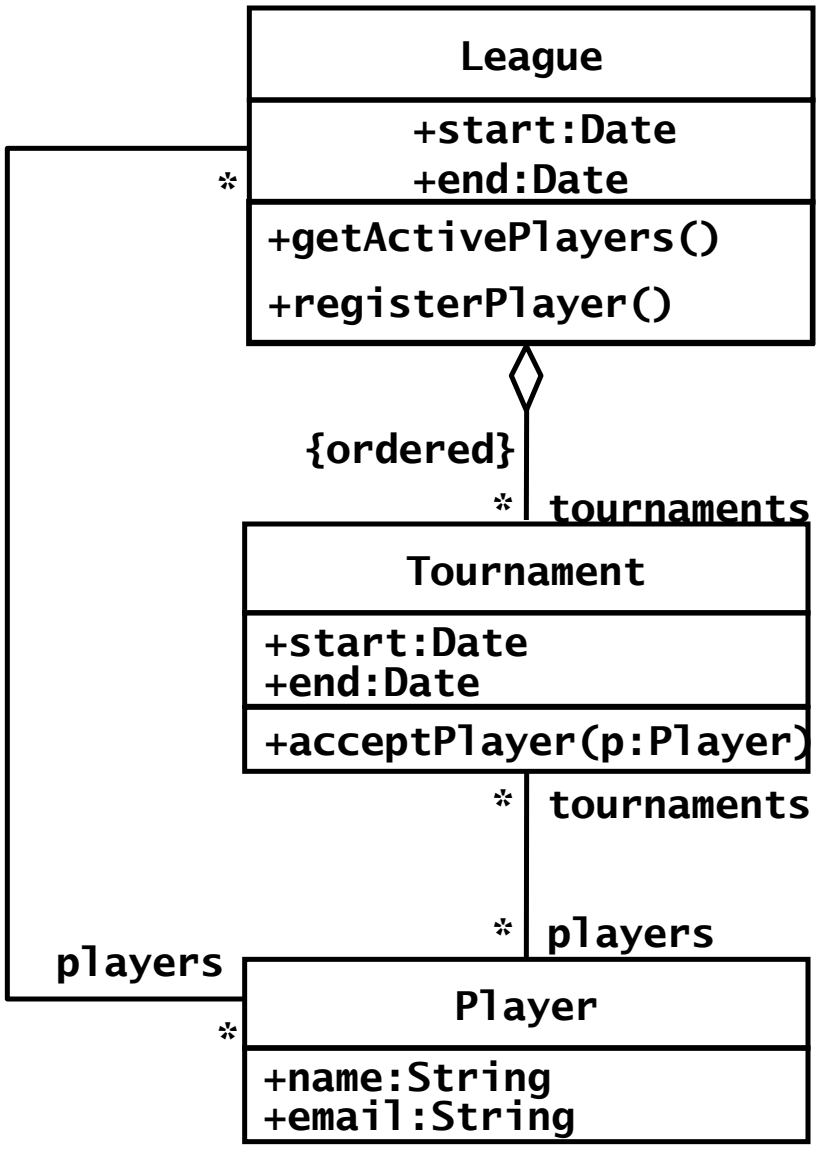
Associations between classes and constraints

- A Tournament's planned duration must be one week

- Players can be registered with a League only if they were not registered before

- Players can be accepted in a Tournament only if they already registered with League

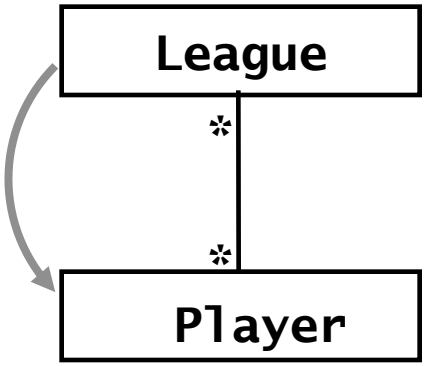
- The number of active players in a league are those that have taken part in at least one Tournament of the League



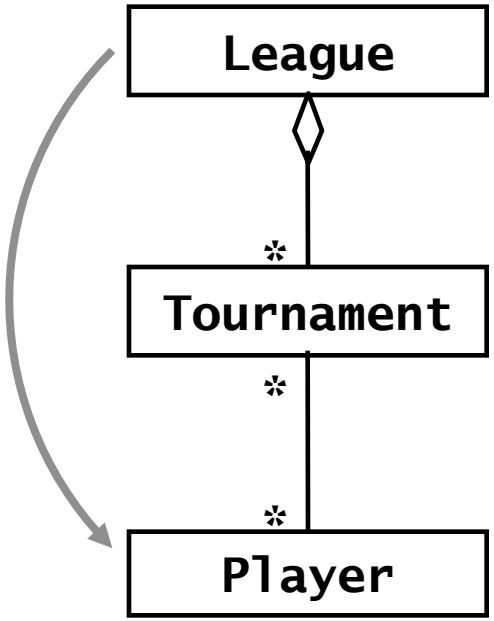
Types of Navigation through a Class Diagram

1. Local attribute

2. Directly related class

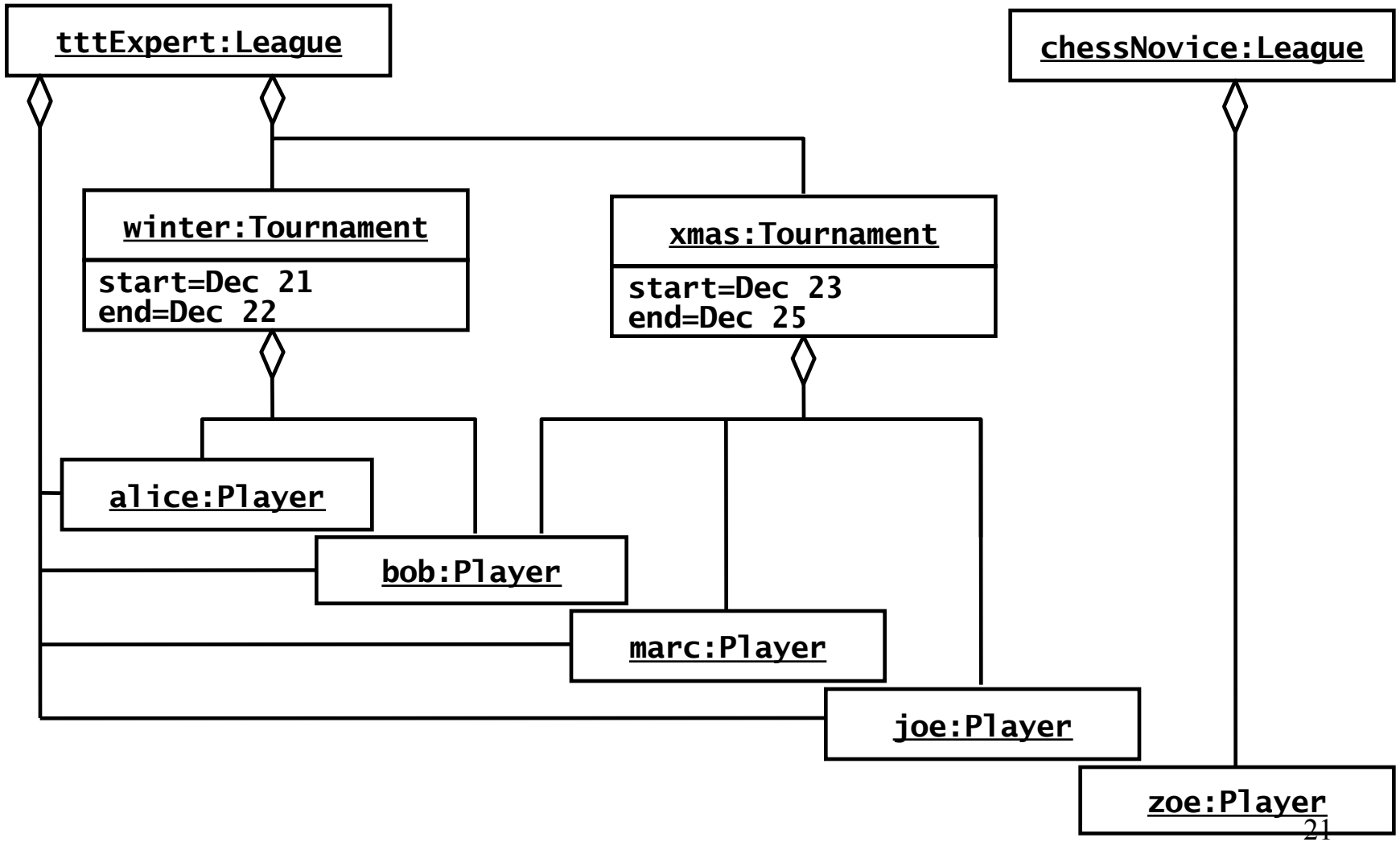


3. Indirectly related class



Any OCL constraint for any class diagram can be built using only a combination of these three navigation types!

Example situation with two Leagues, two Tournaments, and five Players



Sets, sequences and bags

- OCL sets are used when navigating a single association
- OCL sequences are used when navigating a single ordered association
- OCL bags are multisets: they can contain the same object multiple times

OCL provides operations for accessing collections

- **size**, which returns the number of elements in the collection
- **includes (object)**, which returns True if object is in the collection
- **select(expression)**, which returns a collection that contains only the elements of the original collection for which expression is True
- **union(collection**), which returns a collection containing elements from both the original collection and the collection specified as parameter
- **intersection(conection)**, which returns a collection that contains only the elements that are part of both the original collection and the collection specified as parameter
- **asSet (collection)**, which returns a set containing each element of collection only once.

Examples of constraints using each type of navigation

1. Local attribute

```
context Tournament inv:
   end - start <= Calendar.WEEK
```

2. Directly related class

```
context League::registerPlayer(p) pre:
   not players->includes(p)
```

3. Indirectly related classes

```
context Tournament::acceptPlayer(p) pre:
   league.players->includes(p)
```

```
context League::getActivePlayers post:
   result = tournaments.players->asSet
```

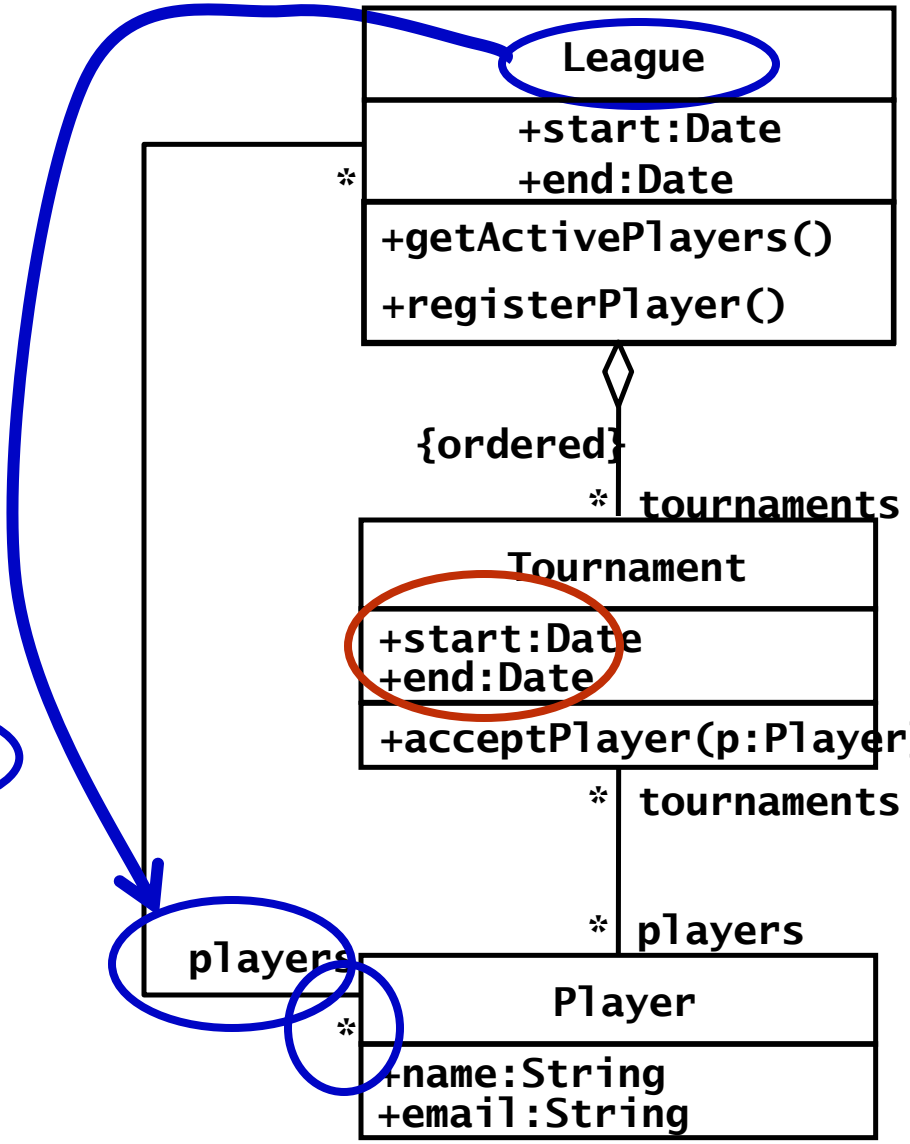
Specifying the Model Constraints

- **Local attribute navigation**

```
context Tournament inv:
end - start <= Calendar.WEEK
```

Directly related class navigation

```
context
  League::registerPlayer(p)
  pre:
  not players->includes(p)
```



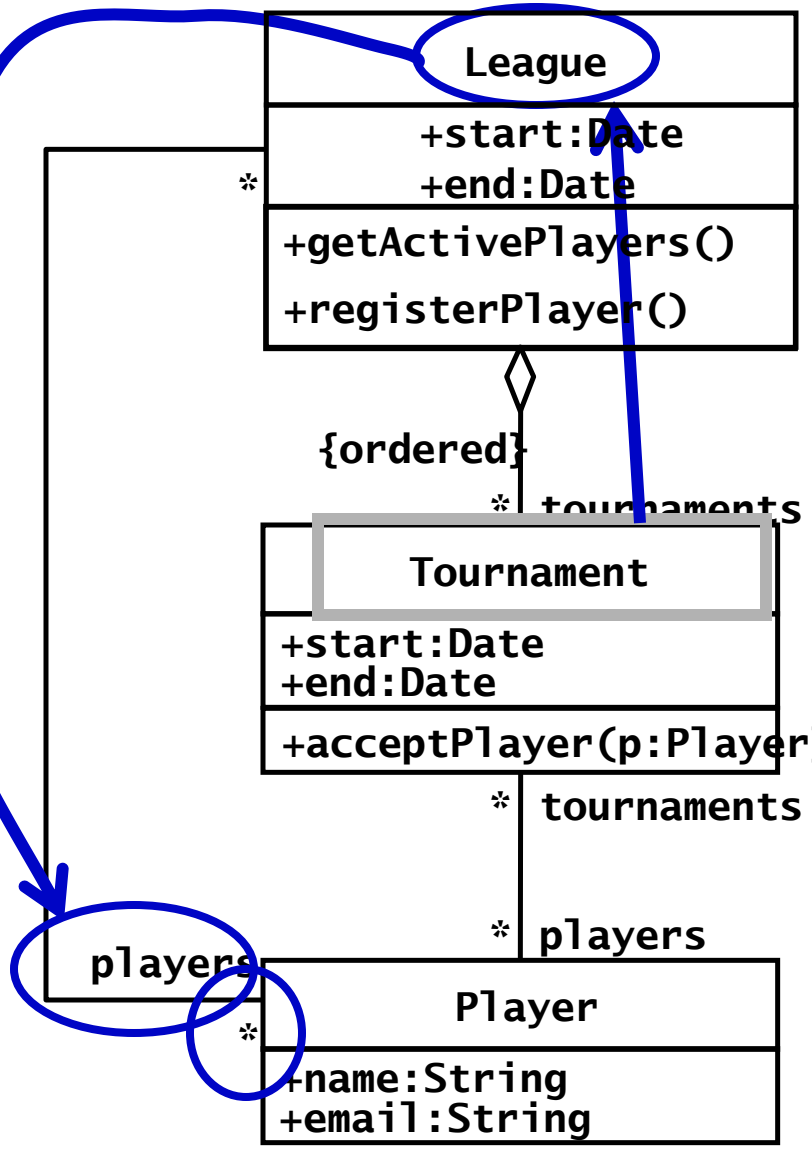
Specifying the Model Constraints

- **Local attribute navigation**

```
context Tournament inv:
end - start <= Calendar.WEEK
```

Indirectly related class navigation

```
context
    Tournament::acceptPlayer(p)
pre:
    league.players->includes(p)
```



Specifying the Model Constraints

- **Local attribute navigation**

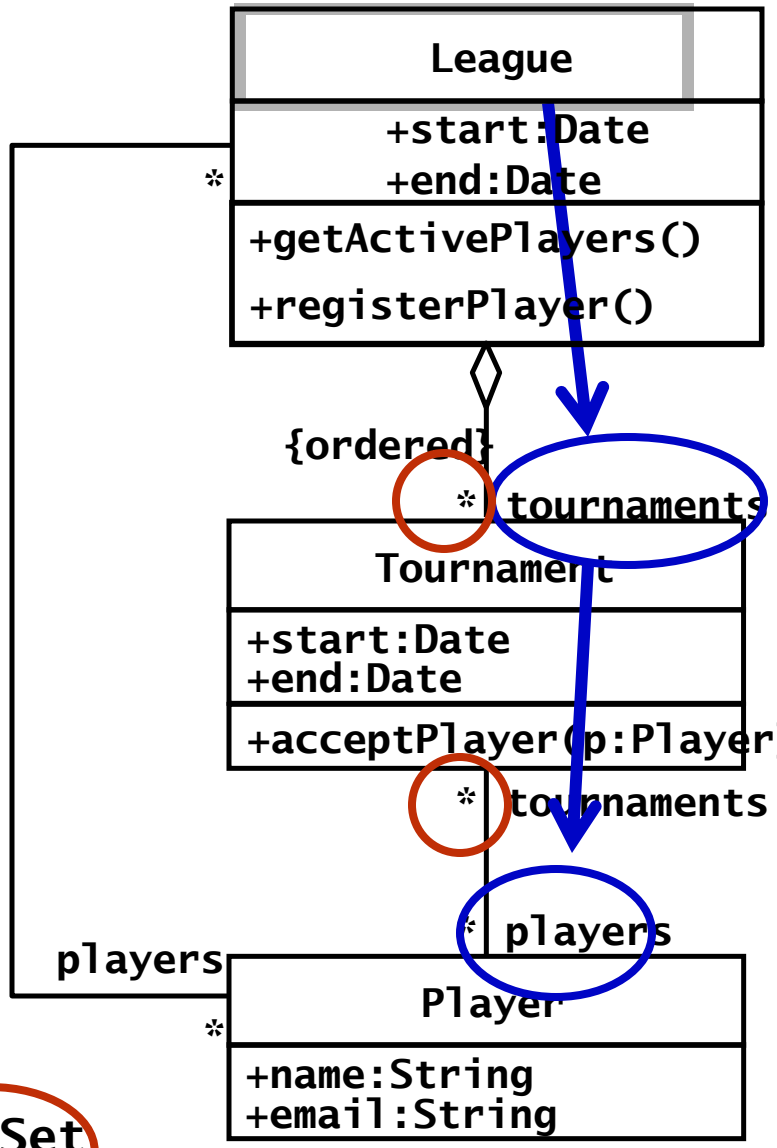
```
context Tournament inv:
-  end - start <= Calendar.WEEK
```

- **Indirectly related class navigation**

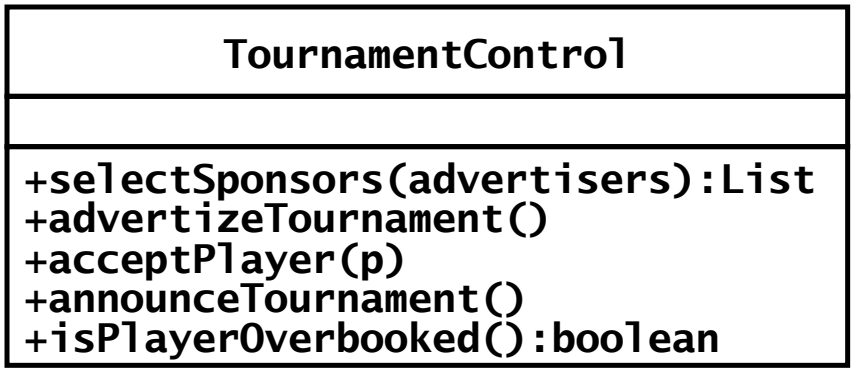
```
context Tournament::acceptPlayer(p)
   pre:league.players->includes(p)
```

- **Indirectly related class navigation**

```
context League::getActivePlayers
post:
   result = tournaments.players->asSet
```



Pre- and post-conditions for ordering operations on TournamentControl




```
context TournamentControl::selectSponsors(advertisers) pre:
  interestedSponsors->notEmpty and tournament.sponsors->isEmpty

context TournamentControl::advertiseTournament() pre:
  tournament.sponsors->notEmpty and not tournament.advertised

context TournamentControl::advertiseTournament() post:
  tournament.advertised

context TournamentControl::acceptPlayer(p) pre:
  tournament.advertised and interestedPlayers->includes(p) and
              not isPlayerOverbooked(p)

context TournamentControl::acceptPlayer(p) post:
  tournament.players->includes(p)
```

OCL supports Quantification

- **OCL forall quantifier**

```
/* All Matches in a Tournament occur within the
   Tournament's time frame */

context Tournament inv:
   matches->forAll(m:Match |
        m.start.after(start) and
        m.end.before(end))
```

- **OCL exists quantifier**

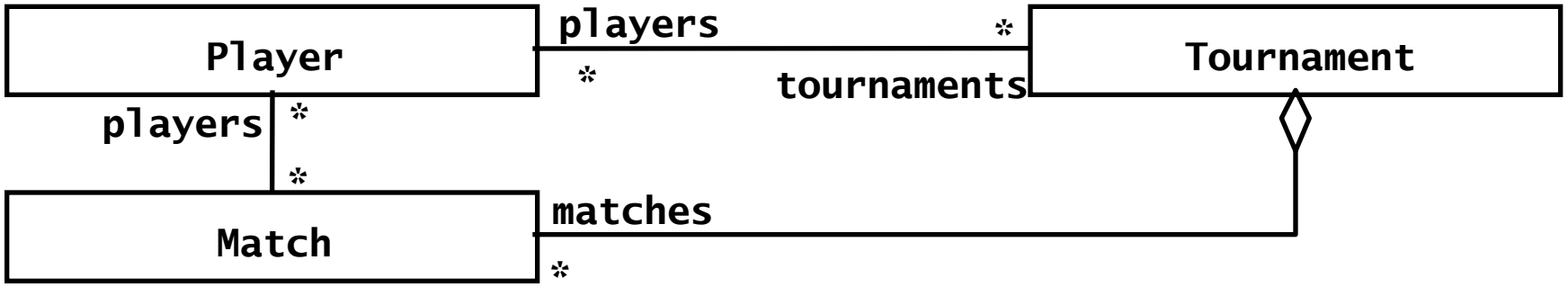
```
/* Each Tournament conducts at least one Match on
   the first day of the Tournament */

context Tournament inv:
        matches->exists(m:Match |
              m.start.equals(start))
```

Specifying invariants on Tournament and Tournament Control

- No Player can take part in two or more Tournaments that overlap

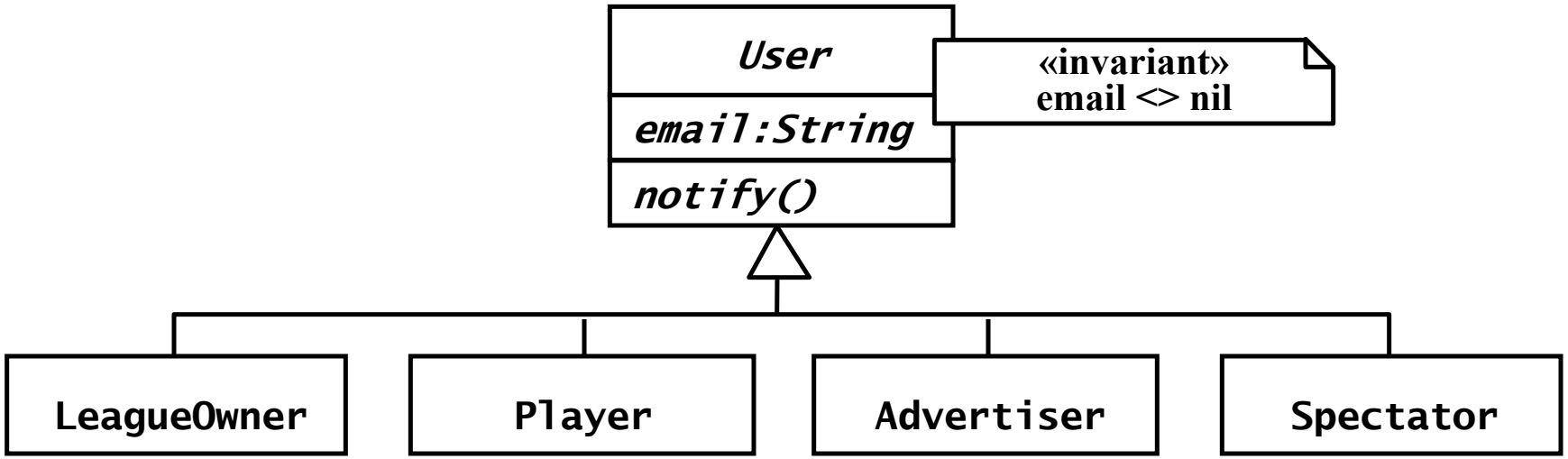
```
context TournamentControl inv:
  tournament.players->forAll(p|
    p.tournaments->forAll(t|
      t <> tournament implies
        not t.overlap(tournament)))
```

Specifying invariants on Match



- **A match can only involve players who are accepted in the tournament**
  - `context Match inv:`
    `players->forAll(p|`
    `p.tournaments->exists(t|`
    `t.matches->includes(self)))`
  - `context Match inv:`
    `players.tournaments.matches->includes(self)`

A contract inheritance



Contracts inheritance

- Preconditions.
  - A method of subclass is allowed to weaken the preconditions of the method it overrides
- Postconditions.
  - Methods must ensure the same postconditions as their ancestors or stricter ones.
- Invariants.
  - A subclass must respect all invariants of its superclasses. However, a subclass can strengthen the inherited invariants

Heuristics for writing readable constraints

- Focus on the lifetime of a class.
- Identify special values for each attribute
- Identify special cases for associations.
- Use helper methods to compute complex conditions.
- Avoid constraints that involve many association traversals.

Summary

- There are three different roles for developers during object design
  - Class user, class implementer and class extender
- During object design - and only during object design - we specify visibility rules
- Constraints are Boolean expressions on model elements
- Contracts are constraints on a class that enable class users, implementers and extenders to share the same assumption about the class (“Design by contract”)
- OCL is a language that allows us to express constraints on UML models
- Complex constrains involving more than one class, attribute or operation can be expressed with 3 basic navigation types.

Next lecture

- Text book

Chapters 10 and 11