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

Chapter 9, Object Design: Specifying Interfaces

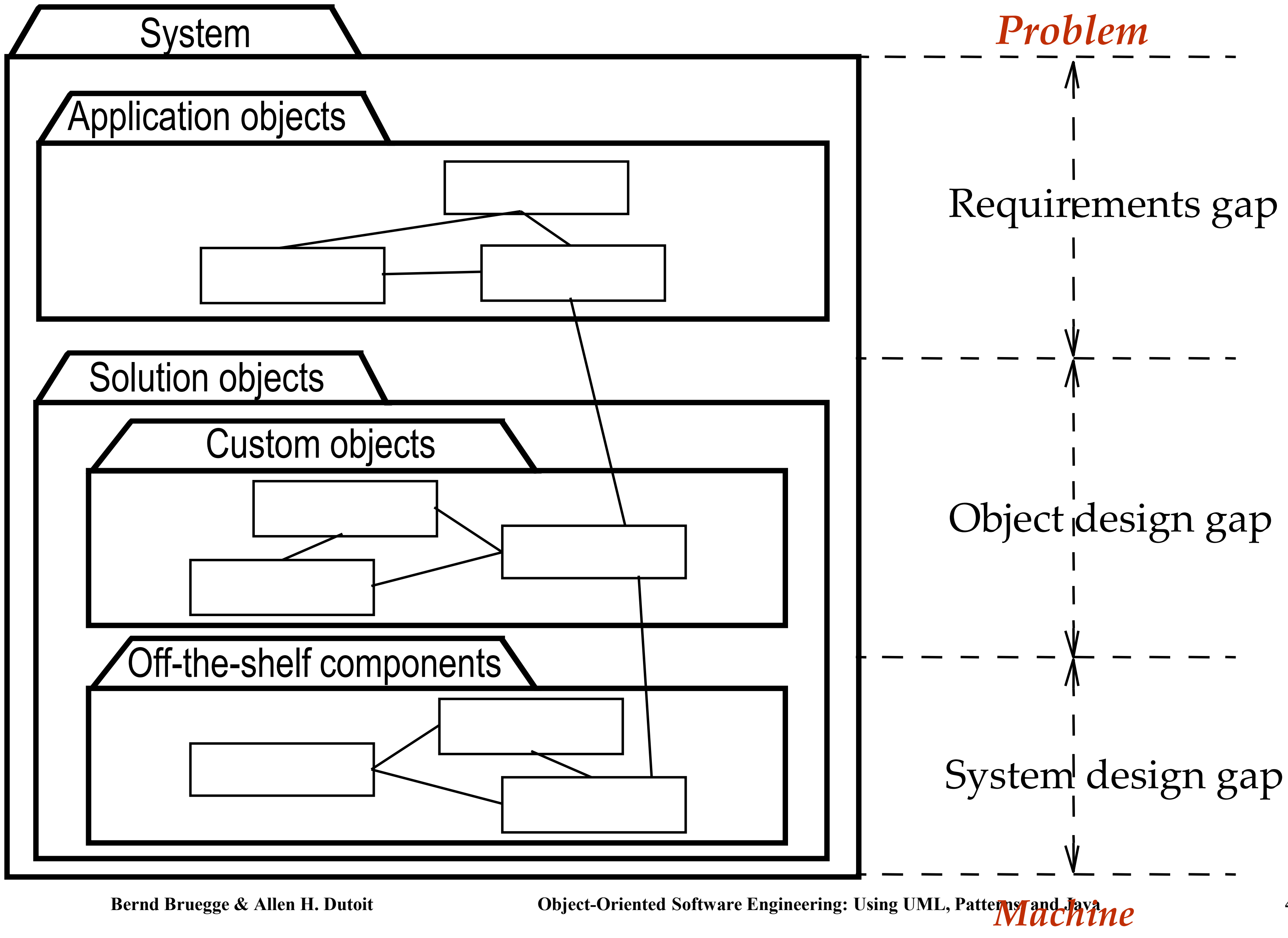
Lecture Plan

- Specifying Interfaces (Chapter 9)
  - Object Design Activities Visibilities and Information Hiding, Contracts
- Mapping Models to Java Code (Chapter 10)
  - Optimizations to address performance requirements
  - Implementation of class model components
    - Realization of associations
    - Realization of contracts
- Mapping Models to Relational Schema (Ch 10.4.4)
  - Realizing entity objects
  - Mapping the object model to a storage schema
  - Mapping class diagrams to tables.

Requirements Analysis vs. Object Design

- Requirements Analysis: The functional model and the dynamic model deliver operations for the object model
- Object Design: Decide where to put these operations in the object model
  - Object design is the process of
    - adding details to the requirements analysis
    - making implementation decisions
- Thus, object design serves as the basis of implementation
  - The object designer can choose among different ways to implement the system model obtained during requirements analysis.

Object Design: Closing the Final Gap



Object Design

As the focus of system design was on identifying large chunks of work that could be assigned to individual teams or developers, the focus of object design is on specifying the boundaries between objects. At this stage in the project, a large number of developers

concurrently refines and changes many objects and their interfaces. The pressure to deliver is increasing and the opportunity to introduce new, complex faults into the design is still there. The focus of interface specification is for developers to communicate clearly and precisely about increasingly lower-level details of the system.

The interface specification activities of object design include

- identifying missing attributes and operations
- specifying type signatures and visibility
- specifying invariants
- specifying preconditions and postconditions

Object Design Challenge

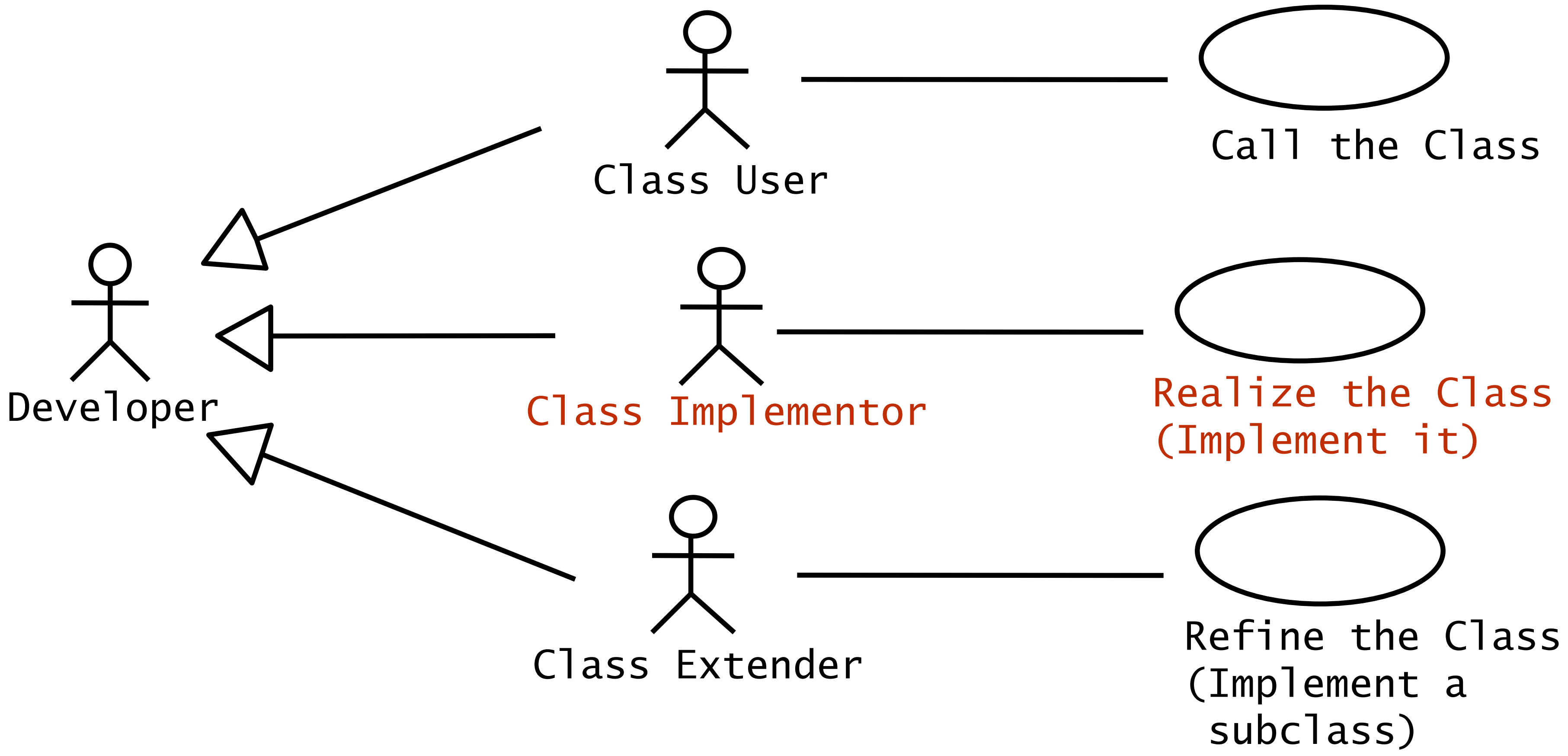
The large number of objects and developers, the high rate of change, and the concurrent number of decisions made during object design make object design much more complex than analysis or system design.

This represents a management challenge, as many important decisions tend to be resolved independently and are not communicated to the rest of the project.

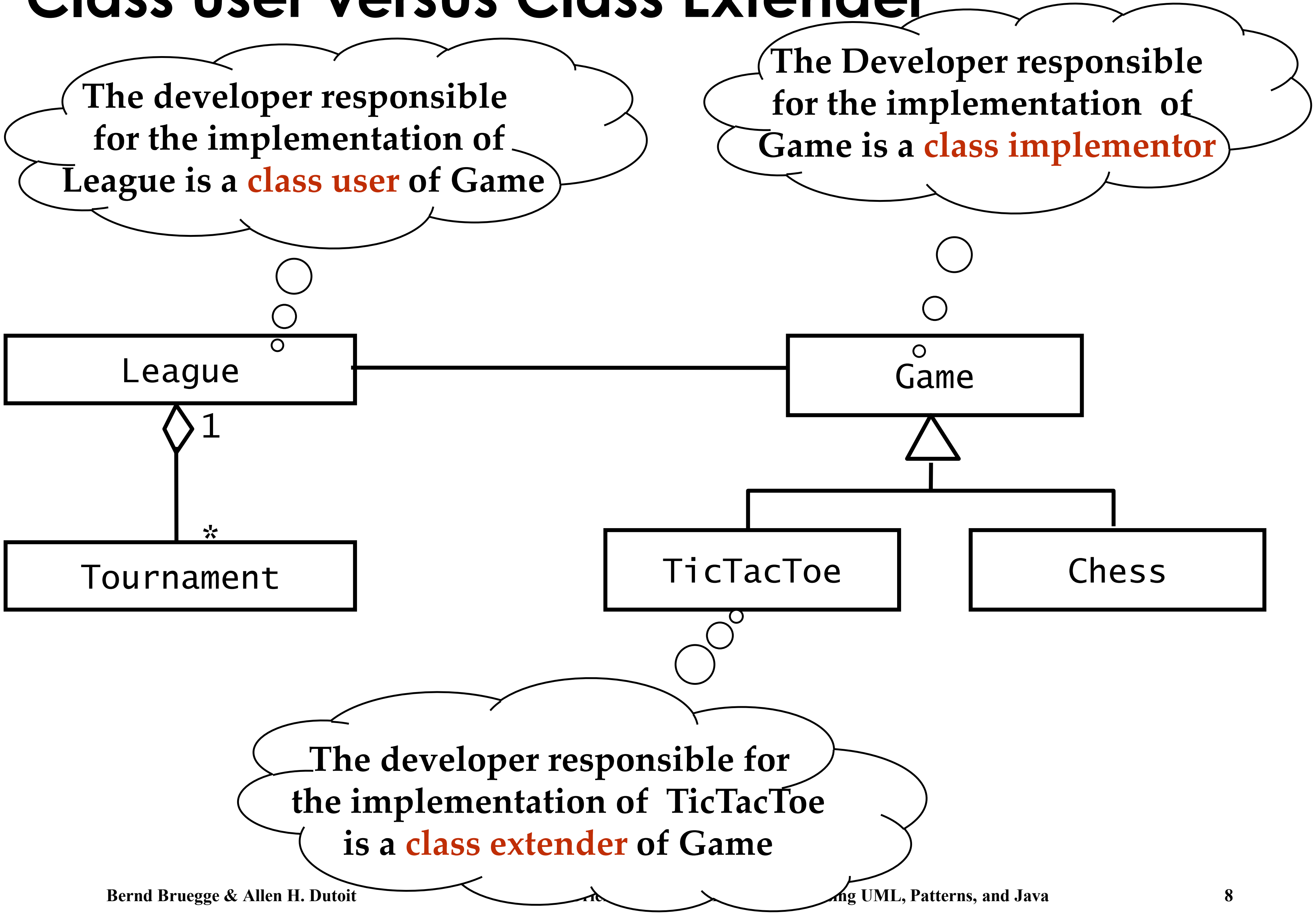
Object design requires much information to be made available among the developers so that decisions can be made consistent with decisions made by other developers and consistent with

design goals

Developers play 3 different Roles during Object Design of a Class


Class User
Call the Class
Developer
Class Implementor
Realize the Class (Implement it)
Class Extender
Refine the Class (Implement a subclass)

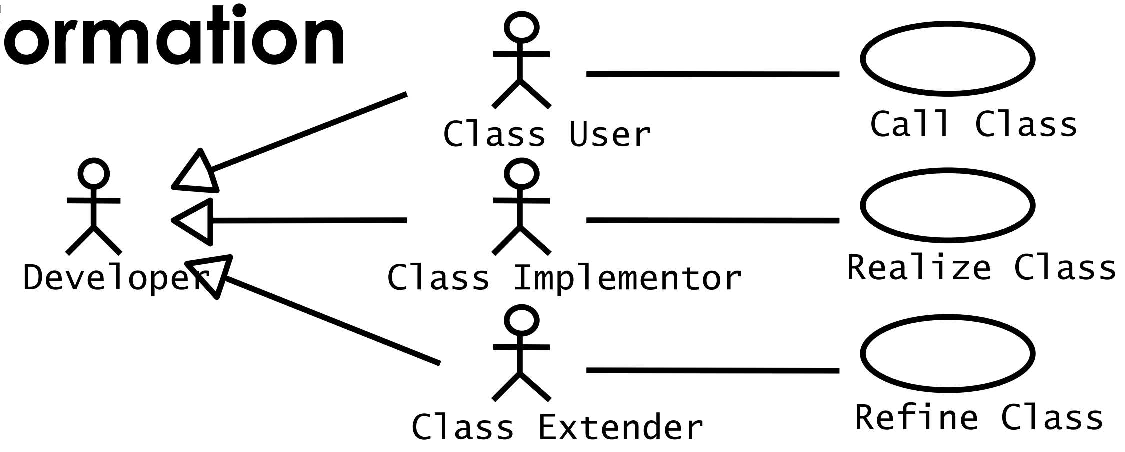
Class user versus Class Extender


The developer responsible for the implementation of League is a class user of Game
The Developer responsible for the implementation of Game is a class implementor
League
1
*
Tournament
Game
TicTacToe
Chess
The developer responsible for the implementation of TicTacToe is a class extender of Game

Specifying Interfaces

- Requirements analysis activities
  - Identify attributes and operations without specifying their types or their parameters
- Object design activities
  - ➡ Add visibility information
  - Add type signature information
  - Add contracts.

Add Visibility Information



Class user ("Public"): +

- Public attributes/operation can be accessed by any class

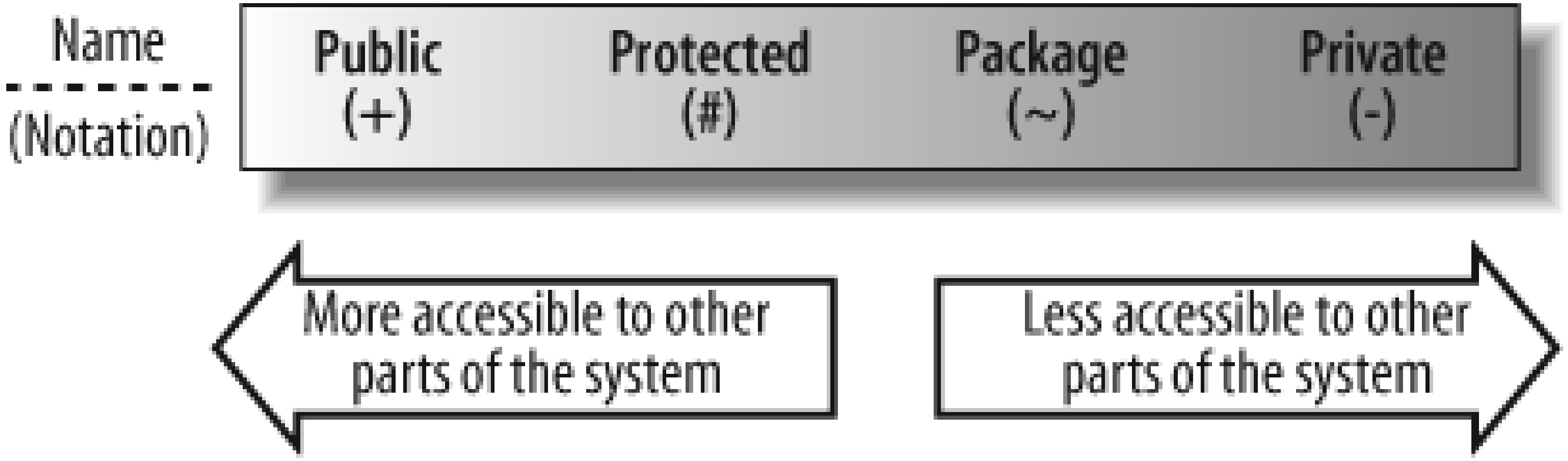
Class implementor ("Private"): -

- Private attributes and operations can be accessed only by the class in which they are defined
- They cannot be accessed by subclasses or other classes

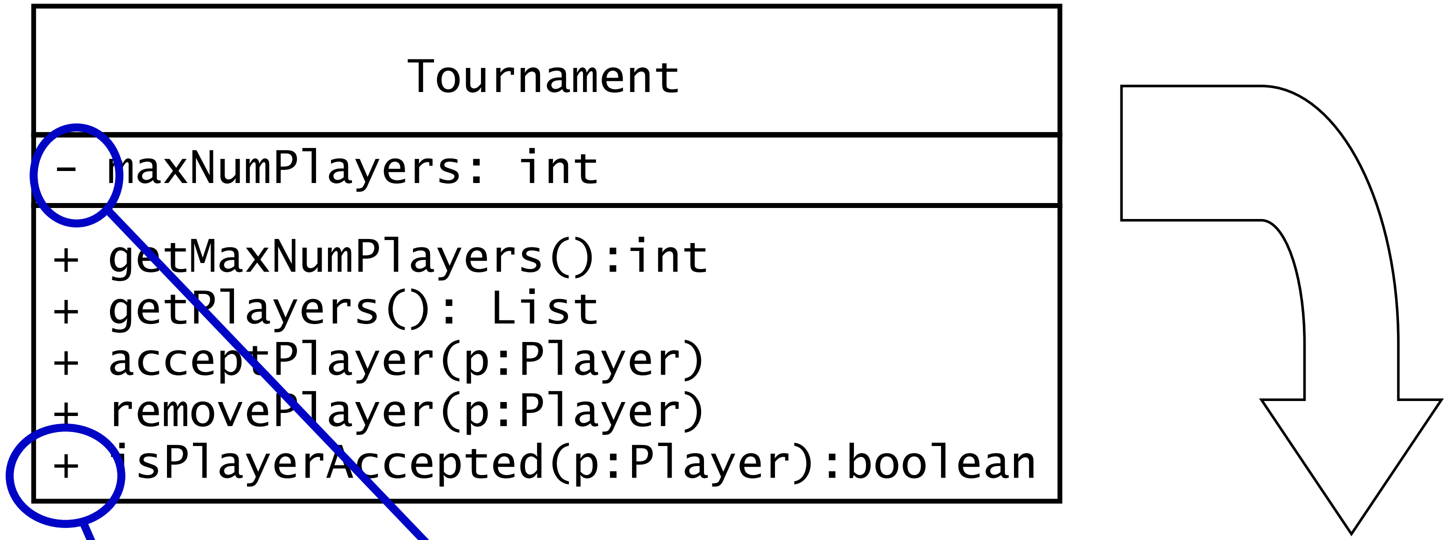
Class extender ("Protected"): #

- Protected attributes/operations can be accessed by the class in which they are defined and by any descendent of the class.

Visibility


Name
(Notation)
Public
(+)
Protected
(#)
Package
(~)
Private
(-)
More accessible to other parts of the system
Less accessible to other parts of the system

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) {…};
```

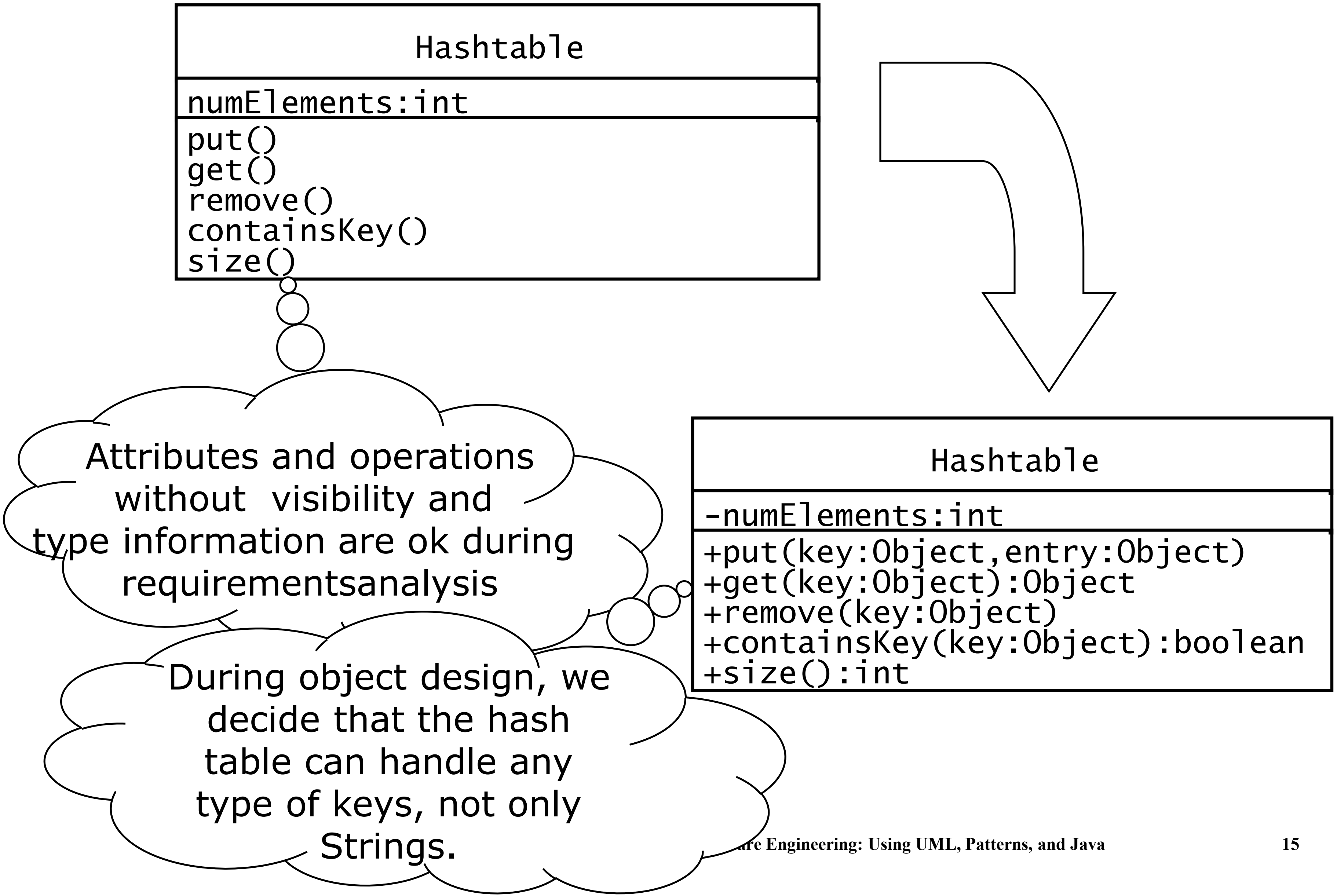
Information Hiding Heuristics

- Carefully define the public interface for classes as well as subsystems
  - For subsystems use a façade design pattern if possible
- Always apply the “Need to know” principle:
  - Only if somebody needs to access the information, make it publicly possible
    - Provide only well defined channels, so you always know the access
- The fewer details a class user has to know
  - the easier the class can be changed
  - the less likely they will be affected by any changes in the class implementation
- Trade-off: Information hiding vs. efficiency
  - Accessing a private attribute might be too slow.

Information Hiding Design Principles

- Only the operations of a class are allowed to manipulate its attributes
  - Access attributes only via operations
- Do not apply an operation to the result of another operation
  - Write a new operation that combines the two operations.

Add Type Signature Information



Attributes and operations without visibility and type information are ok during requirementsanalysis

During object design, we decide that the hash table can handle any type of keys, not only Strings.

Outline of Today's Lecture

- Object Design Activities
- Visibilities
- Information Hiding
- ➡ Contracts

Contract

- Contract: A lawful agreement between two parties in which both parties accept obligations and on which both parties can found their rights
    - The remedy for breach of a contract is usually an award of money to the injured party
- Object-oriented contract: Describes the services that are provided by an object if certain conditions are fulfilled
    - services = “obligations”, conditions = “rights”
    - The remedy for breach of an OO-contract is the generation of an exception.

Modeling Constraints with Contracts

- Example of constraints in Arena:
  - An already registered player cannot be registered again
  - The number of players in a tournament should not be more than maxNumPlayers
  - One can only remove players that have been registered
- These constraints cannot be modeled in UML
- We model them with contracts
- Contracts can be written in OCL.

Object-Oriented Contract

- An object-oriented contract describes the services that are provided by an object. For each service, it specifically describes two things:
  - The conditions under which the service will be provided
  - A specification of the result of the service
- Examples:
  - A letter posted before 18:00 will be delivered on the next working day to any address in Germany
  - For the price of 4 Euros a letter with a maximum weight of 80 grams will be delivered anywhere in the USA within 4 hours of pickup.

Object-Oriented Contract

- An object-oriented contract describes the services that are provided by an object. For each service, it specifically describes two things:
  - The conditions under which the service will be provided
  - A specification of the result of the service that is provided.
- Examples:
  - A letter posted before 18:00 will be delivered on the next working day to any address in Germany.
  - For the price of 4 Euros a letter with a maximum weight of 80 grams will be delivered anywhere in Germany within 4 hours of pickup.

Modeling OO-Contracts

- Natural Language
- Mathematical Notation
- Models and contracts:
  - A language for the formulation of constraints with the formal strength of the mathematical notation and the easiness of natural language:

    $\Rightarrow$ UML + OCL (Object Constraint Language)
  - Uses the abstractions of the UML model
  - OCL is based on predicate calculus

Contracts and Formal Specification

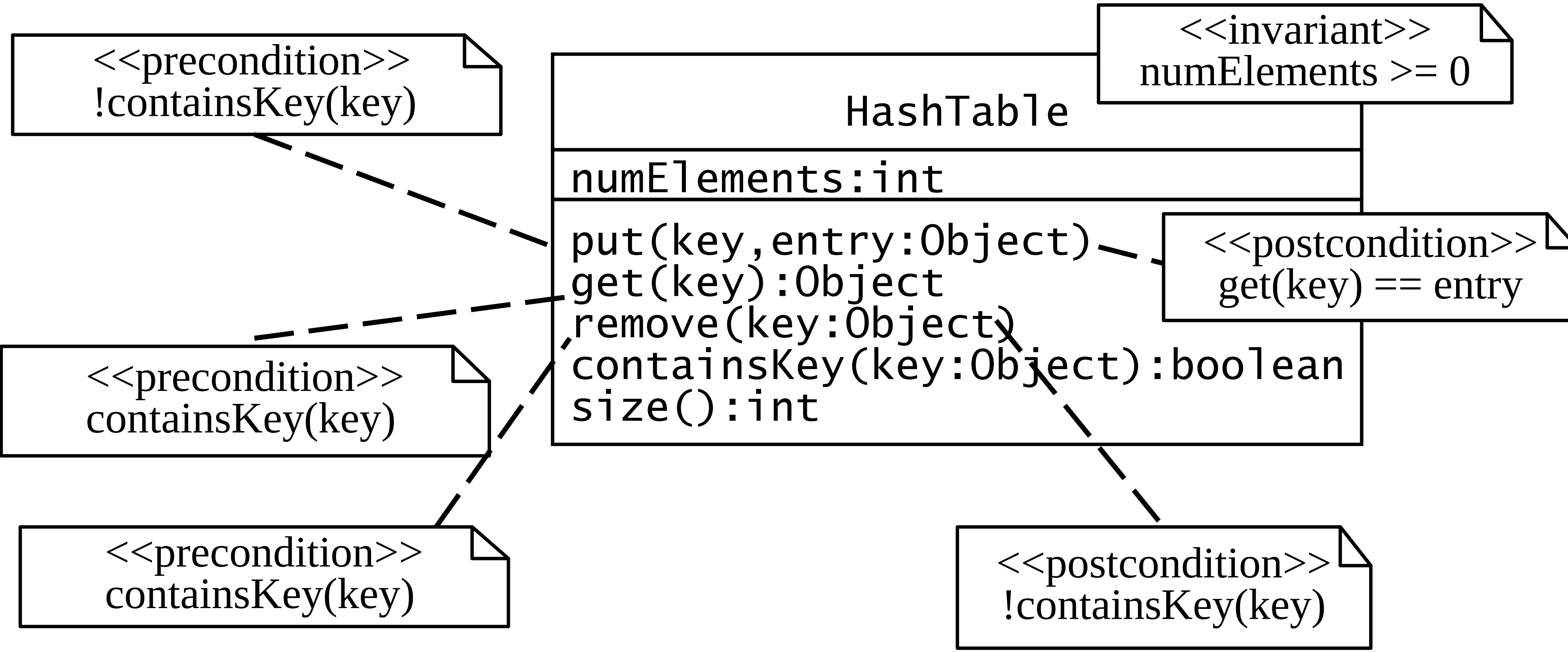
- Contracts enable the caller and the provider to share the same assumptions about the class
- A contract is an exact specification of the interface of an object
- A contract include three types of constraints:
  - Invariant:
    - A predicate that is always true for all instances of a class
  - Precondition ("rights"):
    - Must be true before an operation is invoked
  - Postcondition ("obligation"):
    - Must be true after an operation is invoked.

Formal Specification

- A contract is called a formal specification, if the invariants, rights and obligations in the contract are unambiguous.

Expressing Constraints in UML Models

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



Why not use Contracts already in Requirements Analysis?

- Many constraints represent domain level information
- Why not use them in requirements analysis?
  - Constraints increase the precision of requirements
  - Constraints can yield more questions for the end user
  - Constraints can clarify the relationships among several objects

- Constraints are sometimes used during requirements analysis, however there are trade offs

Requirements vs. Object Design Trade-offs

- Communication among stakeholders
  - Can the client understand formal constraints?
- Level of detail vs. rate of requirements change
  - Is it worth precisely specifying a concept that will change?
- Level of detail vs. elicitation effort
  - Is it worth the time interviewing the end user
  - Will these constraints be discovered during object design anyway?
- Testing constraints
  - If tests are generated early, do they require this level of precision?

OCL Simple Predicates

Example:

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

In English:

"The maximum number of players in any tournament should be a positive number."

Notes:

- "self" denotes all instances of "Tournament"
- OCL uses the same dot notation as Java.

OCL Preconditions

Example:

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

In English:

“The acceptPlayer(p) operation can only be invoked if player p has not yet been accepted in the tournament.”

Notes:

- The context of a precondtion is an operation
- isPlayerAccepted(p) is an operation defined by the class Tournament

OCL Postconditions

Example:

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

In English:

“The number of accepted player in a tournament increases by one after the completion of acceptPlayer()”

Notes:

- self@pre denotes the state of the tournament before the invocation of the operation.
- Self denotes the state of the tournament after the completion of the operation.

OCL Contract for acceptPlayer() in Tournament

```
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
```

OCL Contract for removePlayer() in Tournament

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

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

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

Example

- The age of a person is not negative.
- A person is younger than its parents.
- After a birthday, a person becomes one year older.
- A Person has 2 parents at max.
- Only an adult can be owner of a car.
- The first registration of a car can not be before it is built.
- Every Person that has a car has at least one car which is younger than the Person.
- Nobody can be his/her own parent.
- There's at least one Person which owns a car.

context Person inv: self.age >=0

context Person inv: self.parents->forAll(p|p.age>self.age)

context Person::hasBirthday() post:
self.age=self.age@pre+1

context Person inv: self.parents->size()<=2

context Person::getsChild() post: self.childs->notEmpty()
and self.childs->size() > self.childs@pre->size()

context Person inv: self.age<18 implies self.cars->isEmpty()

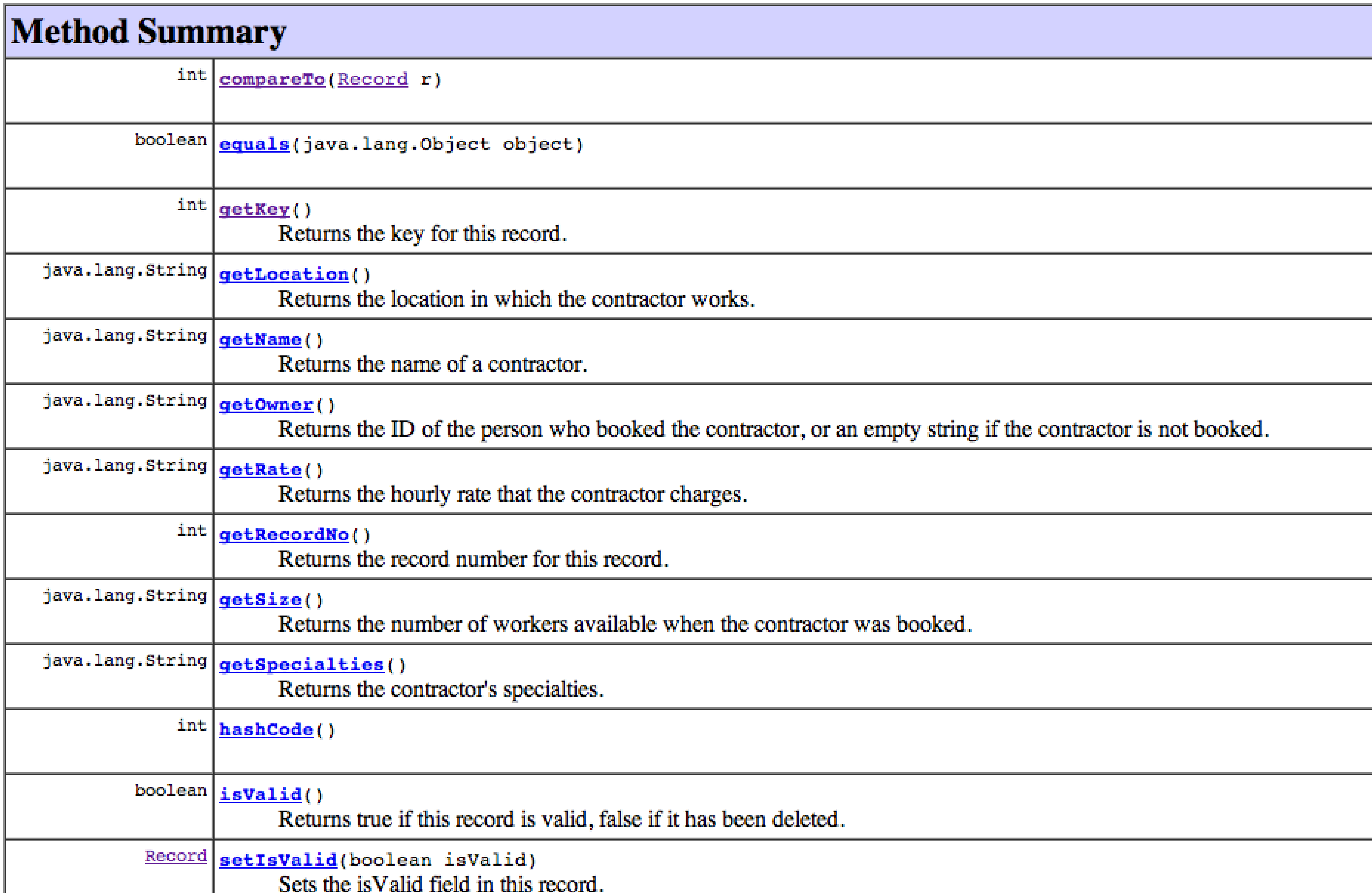
context Auto inv: self.registration>=self.constructionYear

context Person inv: self.cars->notEmpty() implies self.cars-
>exists(c | Calendar.YEAR - c.constructionYear < self.age)

context Person inv: self.parents->excludes(self)

context Person inv: Person.allInstances()->exists(p | p.cars-
>size() > 0)

javadoc



Method Summary	
int	compareTo(Record r)
boolean	equals(java.lang.Object object)
int	getKey() Returns the key for this record.
java.lang.String	getLocation() Returns the location in which the contractor works.
java.lang.String	getName() Returns the name of a contractor.
java.lang.String	getOwner() Returns the ID of the person who booked the contractor, or an empty string if the contractor is not booked.
java.lang.String	getRate() Returns the hourly rate that the contractor charges.
int	getRecordNo() Returns the record number for this record.
java.lang.String	getSize() Returns the number of workers available when the contractor was booked.
java.lang.String	getSpecialties() Returns the contractor's specialties.
int	hashCode()
boolean	isValid() Returns true if this record is valid, false if it has been deleted.
Record	setIsValid(boolean isValid) Sets the isValid field in this record.

Java Implementation of Tournament class (Contract as a set of JavaDoc comments)

```
public class Tournament {
/** The maximum number of players
 * is positive at all times.
 * @invariant maxNumPlayers > 0
 */
private int maxNumPlayers;

/** The players List contains
 *  references to Players who are
 *  are registered with the
 *  Tournament. */
private List players;

/** Returns the current number of
 * players in the tournament. */
public int getNumPlayers() {…}

/** Returns the maximum number of
 * players in the tournament. */
public int getMaxNumPlayers() {…}
```

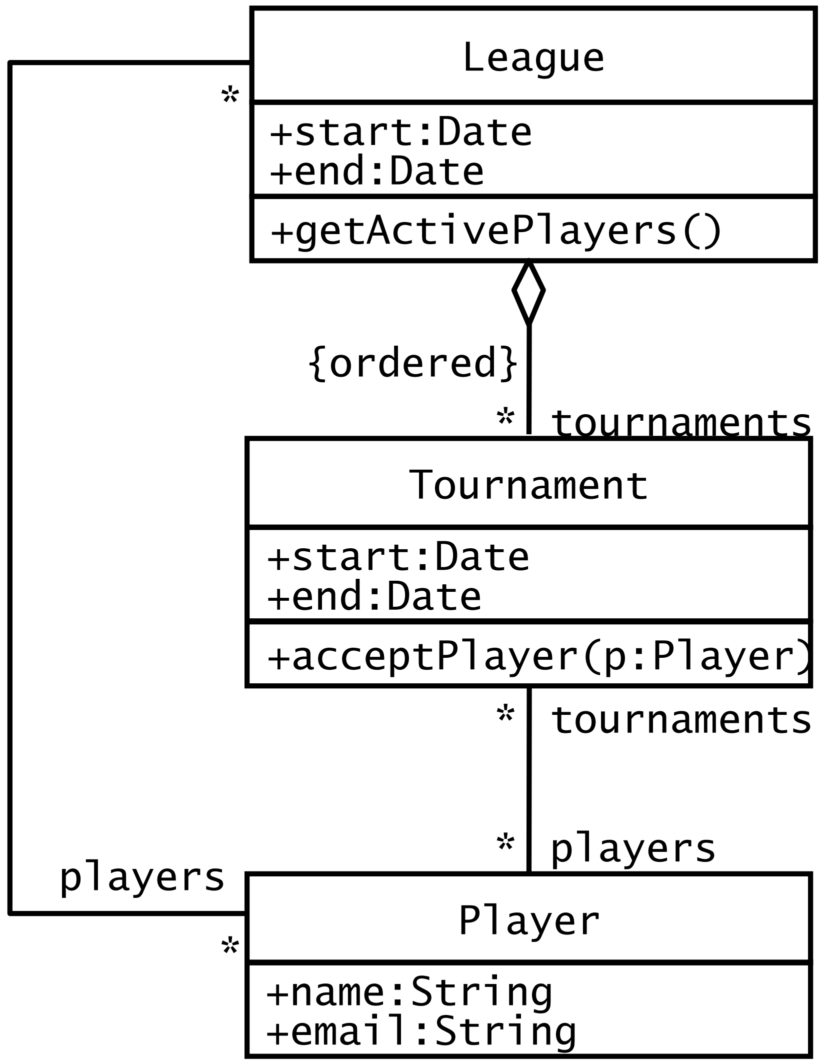
```
/** The acceptPlayer() operation
 * assumes that the specified
 * player has not been accepted
 * in the Tournament yet.
 * @pre !isPlayerAccepted(p)
 * @pre getNumPlayers()<maxNumPlayers
 * @post isPlayerAccepted(p)
 * @post getNumPlayers() =
 *     @pre.getNumPlayers() + 1
 */
public void acceptPlayer (Player p) {…}

/** The removePlayer() operation
 * assumes that the specified player
 * is currently in the Tournament.
 * @pre isPlayerAccepted(p)
 * @post !isPlayerAccepted(p)
 * @post getNumPlayers() =
 *     @pre.getNumPlayers() - 1
 */
public void removePlayer(Player p) {…}

}
```

Constraints can involve more than one class

How do we specify constraints on on a group of classes?

Starting from a specific class in the UML class diagram, we navigate the associations in the class diagram to refer to the other classes and their properties (attributes and Operations).

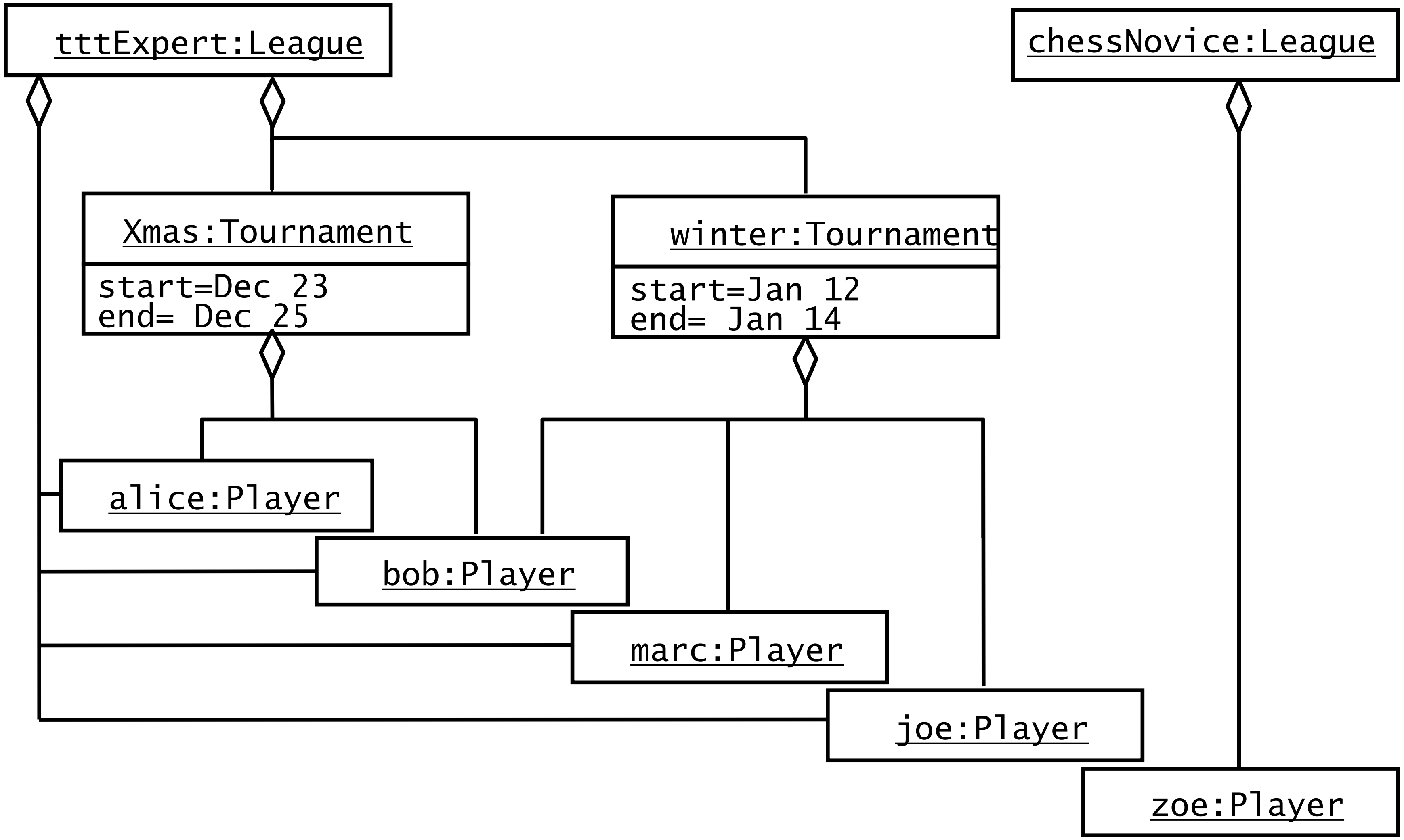
Example from ARENA: League, Tournament and Player



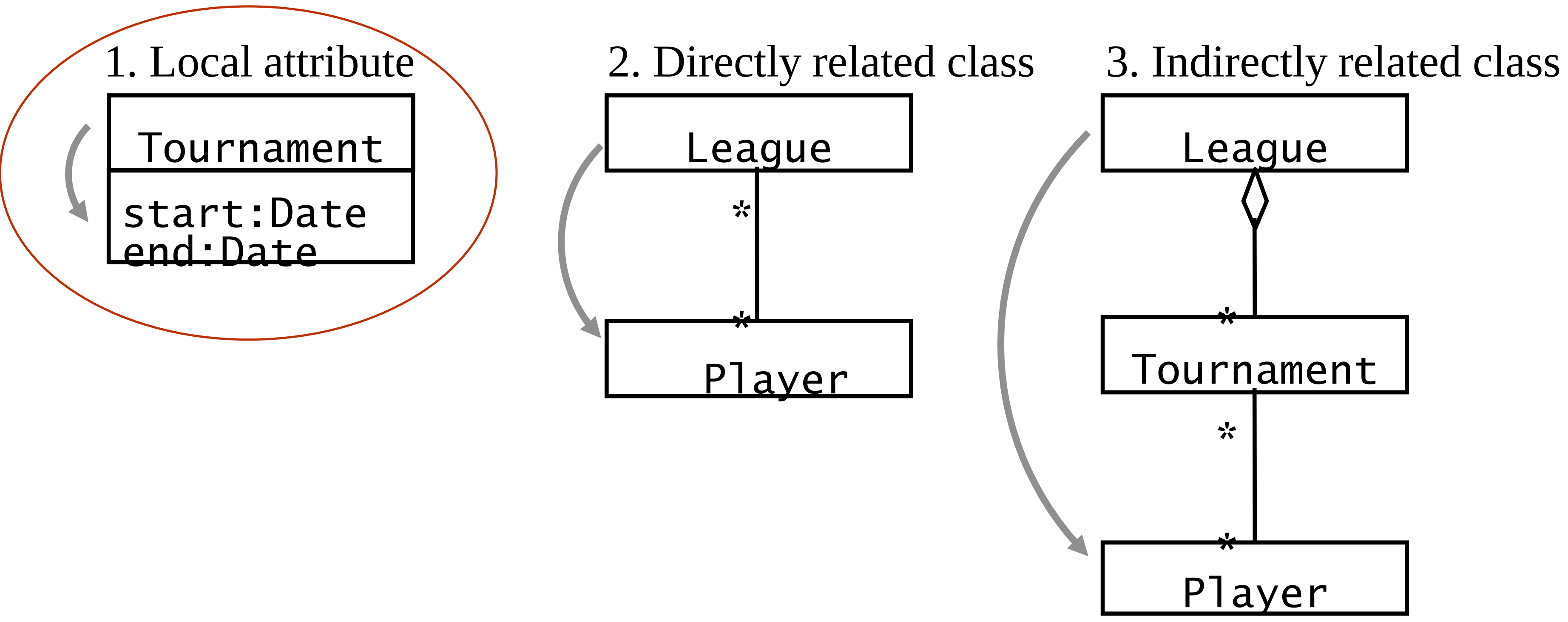
Constraints:

1. A Tournament's planned duration must be under one week.
2. Players can be accepted in a Tournament only if they are already registered with the corresponding League.
3. The number of active Players in a League are those that have taken part in at least one Tournament of the League.

Instance Diagram: 2 Leagues , 5 Players, 2 Tournaments



3 Types of Navigation through a Class Diagram



Any constraint for an arbitrary UML class diagram can be specified using only a combination of these 3 navigation types!

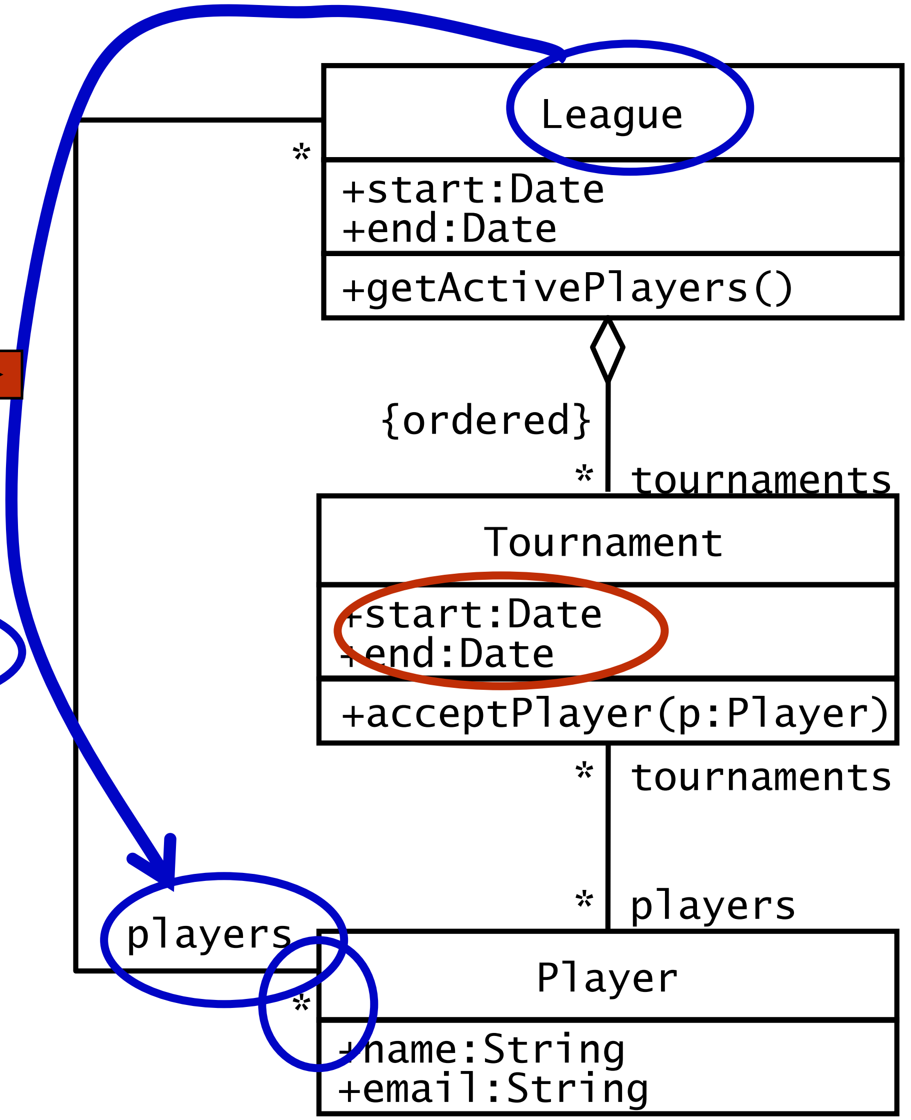
Specifying the Model Constraints in OCL

Local attribute navigation

```
    context Tournament inv:
      end - start <= 7
```

Directly related class navigation

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



OCL Sets, Bags and Sequences

- Sets, Bags and Sequences are predefined in OCL and subtypes of Collection. OCL offers a large number of predefined operations on collections. They are all of the form:

```
collection->operation(arguments)
```

OCL-Collection

- The OCL-Type Collection is the generic superclass of a collection of objects of Type T
- Subclasses of Collection are
  - Set: Set in the mathematical sense. Every element can appear only once
  - Bag: A collection, in which elements can appear more than once (also called multiset)
  - Sequence: A multiset, in which the elements are ordered
- Example for Collections:
  - Set(Integer): a set of integer numbers
  - Bag(Person): a multiset of persons
  - Sequence(Customer): a sequence of customers

OCL-Operations for OCL-Collections (1)

`size: Integer`
Number of elements in the collection

▶ **`includes(o:OclAny): Boolean`**
True, if the element **`o`** is in the collection

`count(o:OclAny): Integer`
Counts how many times an element is contained in the collection

`isEmpty: Boolean`
True, if the collection is empty

`notEmpty: Boolean`
True, if the collection is not empty

The OCL-Type **OclAny** is the most general OCL-Type

OCL-Operations for OCL-Collections(2)

`union(c1:Collection)`

Union with collection **`c1`**

`intersection(c2:Collection)`

Intersection with Collection **`c2`** (contains only elements, which appear in the collection as well as in collection **`c2`** auftreten)

`including(o:OclAny)`

Collection containing all elements of the Collection and element **`o`**

`select(expr:OclExpression)`

Subset of all elements of the collection, for which the OCL-expression **`expr`** is true

How do we get OCL-Collections?

- A collection can be generated by explicitly enumerating the elements
- A collection can be generated by navigating along one or more 1-N associations
  - Navigation along a single 1:n association yields a Set
  - Navigation along a couple of 1:n associations yields a Bag (Multiset)
  - Navigation along a single 1:n association labeled with the constraint {ordered } yields a Sequence

Summary

- Constraints are predicates (often boolean expressions) on UML model elements
- Contracts are constraints on a class that enable class users, implementors and extenders to share the same assumption about the class (“Design by contract”)
- OCL is the example of a formal language that allows us to express constraints on UML models
- Complicated constrains involving more than one class, attribute or operation can be expressed with 3 basic navigation types.

Backup and Additional Slides

OCL supports Quantification

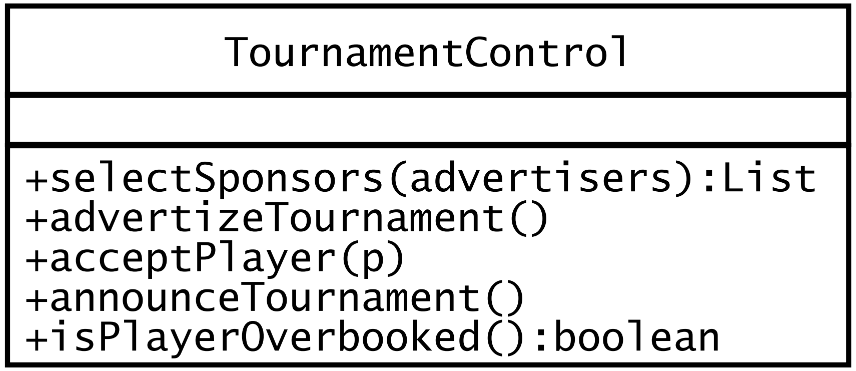
OCL forall quantifier

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

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

Pre- and post-conditions for ordering operations on TournamentControl



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

Specifying invariants on Tournament and Tournament Control

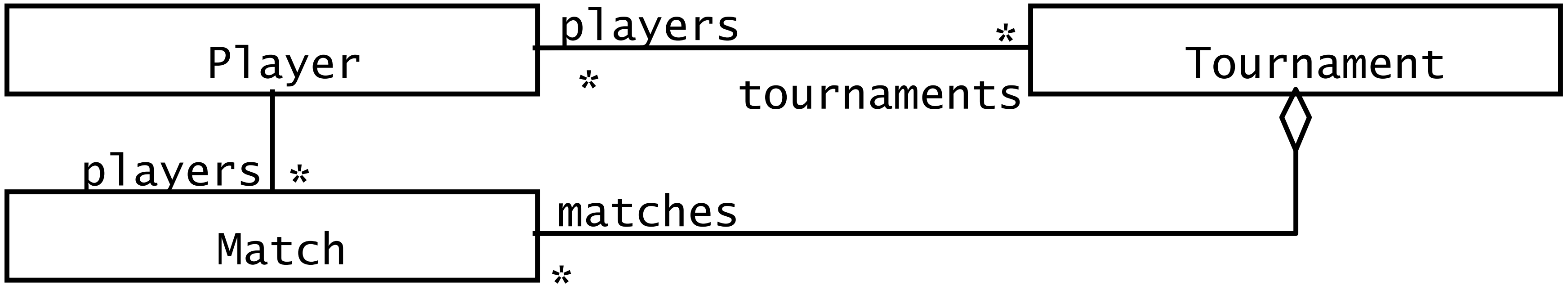
English: “All Matches of in a Tournament must occur within the time frame of the Tournament”

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

English: “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



English: "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)
```