

# Chapter 11: Object Design 3 (Specifying Interfaces)

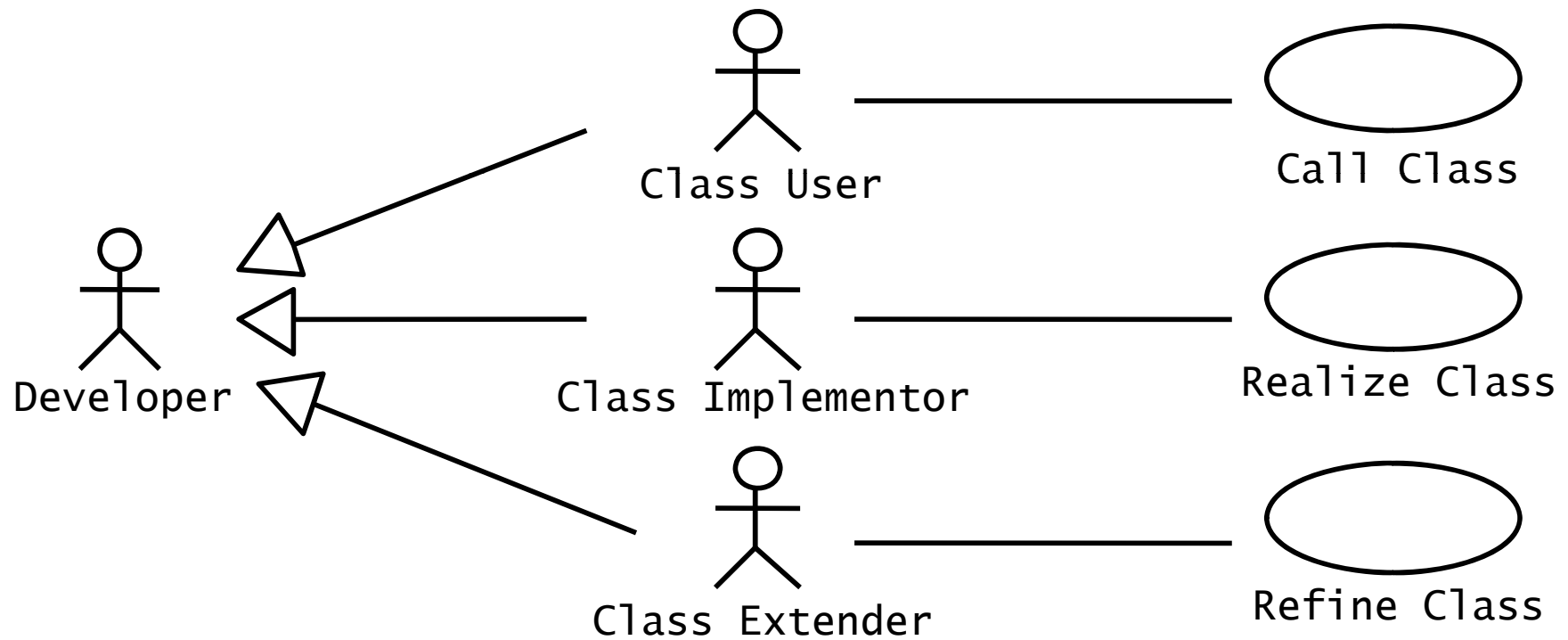
## Object-Oriented Software Construction

Armin B. Cremers, Tobias Rho, Daniel  
Speicher & Holger Mügge  
(based on Bruegge & Dutoit)

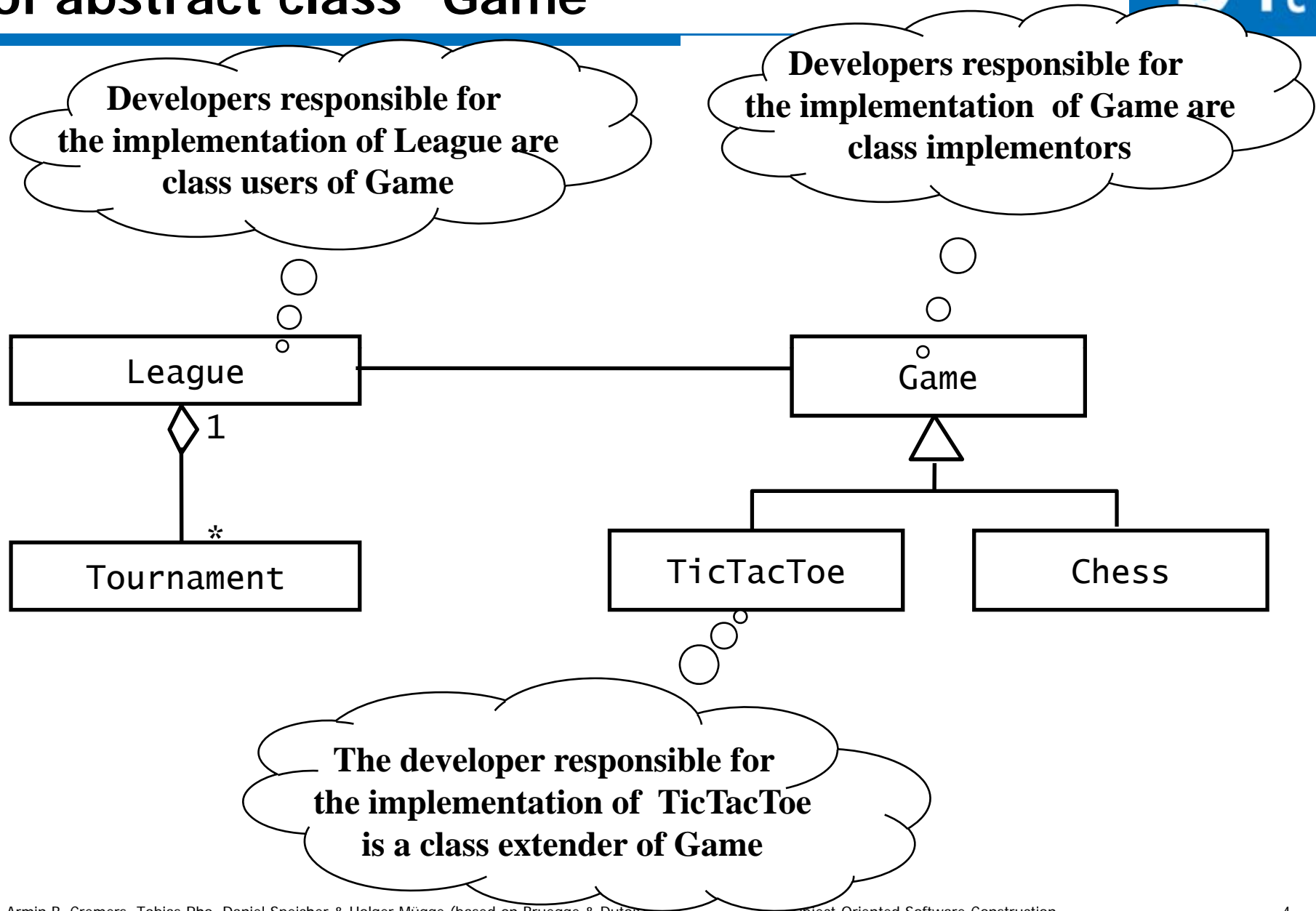


- ◆ Requirements analysis activities
  - ◆ Identifying first attributes and operations without specifying their types or their parameters.
  
- ◆ Goal of Interface Design:
  - ◆ Describe the Interface of each object precisely enough so that objects realized by individual developers fit together with minimal Integration problems
  
- ◆ Activities:
  - ◆ Identify missing attributes and operations from analysis object and subsystem service
  - ◆ Specify Visibility and Signatures
  - ◆ Specify Contracts (main focus in this lecture)

# Developers play different Roles during Object Design



# Example for the relationship from viewpoint of abstract class "Game"



# Add Visibility Information

UML defines four levels of visibility:

- ◆ Private (Class implementor):

- ◆ A private attribute can be accessed only by the class in which it is defined.
- ◆ A private operation can be invoked only by the class in which it is defined.
- ◆ Private attributes and operations cannot be accessed by subclasses or other classes.

- ◆ Protected (Class extender):

- ◆ A protected attribute or operation can be accessed by the class in which it is defined and on any descendent (subclass) of the class.

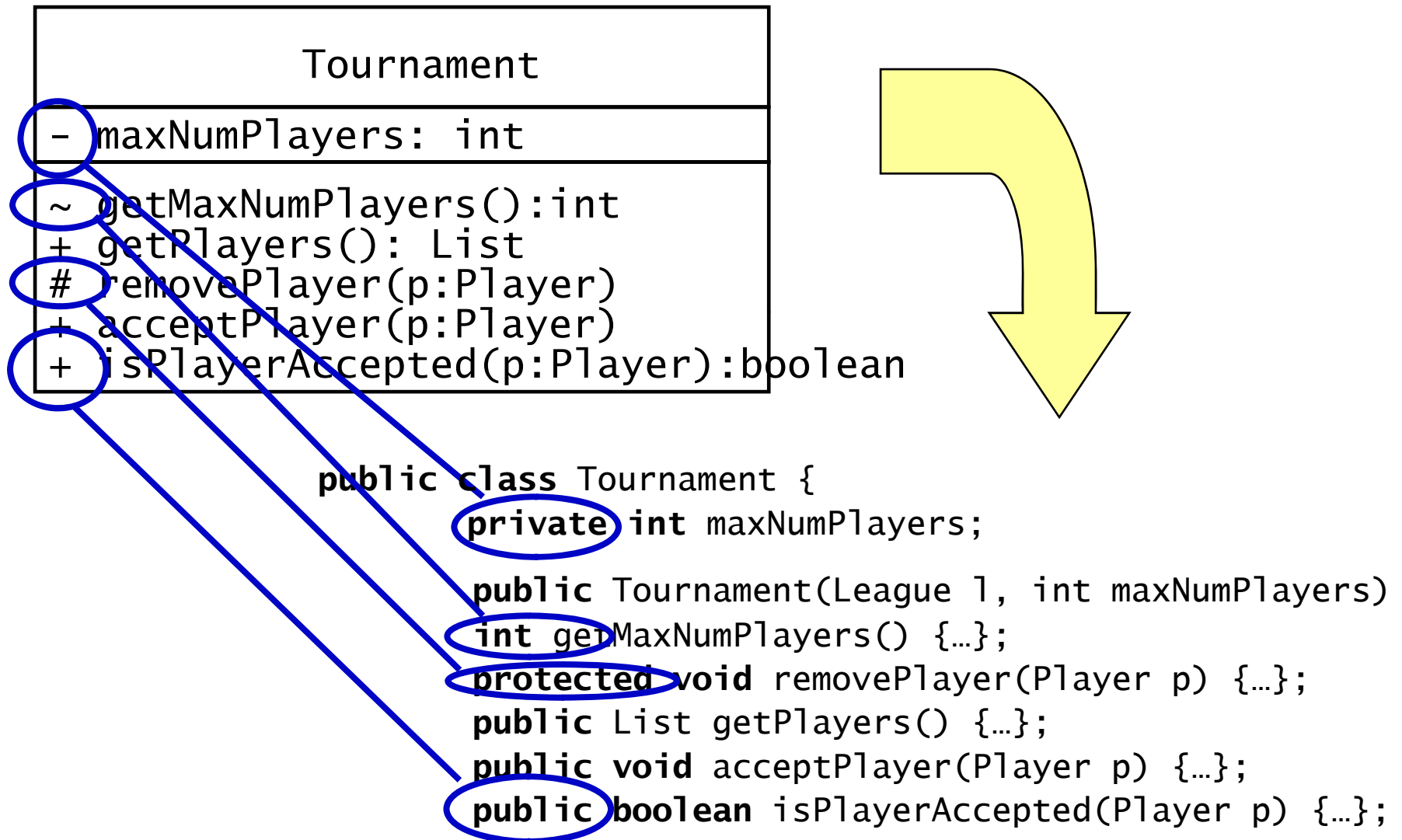
- ◆ Public (Class user):

- ◆ A public attribute or operation can be accessed by any class.

- ◆ Package (Class user)

- ◆ A package attribute or operation can be accessed only a by class, which is part of the same package (Java)

# Implementation of UML Visibility in Java



# Information Hiding Heuristics

- ◆ Carefully define the public interface for classes as well as subsystems (façade)
- ◆ Always apply the “Need to know” principle.
  - ◆ Only if somebody needs to access the information, make it publicly available, but then only through well defined channels, so you always control and verify the access.
- ◆ The fewer an operation knows
  - ◆ the less likely it will be affected by any changes
  - ◆ the easier the method can be changed
- ◆ Information hiding is not a major issue during analysis, however during design it is an issue.

# Information Hiding Design Principles

- ◆ Only the operations of a class are allowed to manipulate its attributes
  - ◆ Access attributes only via operations.

```
// Can be accessed directly  
public Object myAttr = ... ;
```

```
// Can't be accessed directly  
private Object myAttr = ... ;  
public Object getMyAttr(){  
    return this.myAttr;  
}  
public void setMyAttr( Object attr ){  
    doCheck( attr ); ...  
    this.myAttr = attr;  
}
```

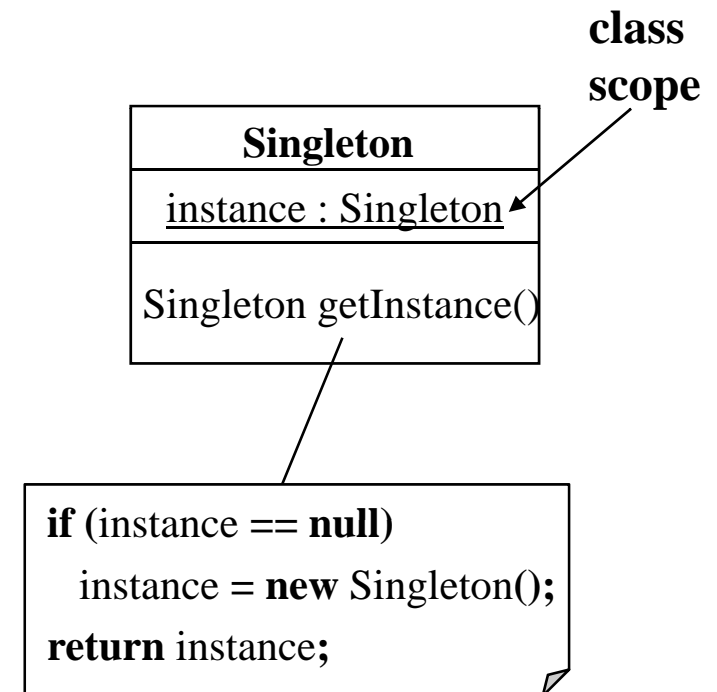
- ◆ Advantage of get and set methods: error checking



# Information Hiding Design Principles

- ◆ Sometimes a variable should be restricted to one single instance available in the system over all time (Java: virtual machine).
- ◆ Application of Singleton Pattern [Gamma, 1995]
- ◆ Sample Code (Java, usage of class fields):

```
public class Singleton {  
    private static Singleton instance = null;  
    public Singleton() { }  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```



- ◆ Idea: Contracts on a class enable caller (client) and callee (class itself) to share the same assumptions (semantics) about the class.
- ◆ Problem: Accurate Semantics cannot be added to standard UML models discussed so far
- ◆ Solution/Realization: **Design By Contract** by Bertrand Meyer (“Object-Oriented Software Construction”, Prentice Hall)
  - ◆ Process of developing software based on the notion of contracts between objects
  - ◆ Contracts are expressed as assertions (predicates) that make assumptions on the *cases*, a method can handle
  - ◆ Assertions comprise preconditions, postconditions and invariants

## ◆ Precondition:

- ◆ Preconditions are predicates associated with a specific operation and must be true *before* the operation is invoked.
- ◆ Preconditions are used to specify constraints that a caller must meet before calling an operation.

## ◆ Postcondition:

- ◆ Postconditions are predicates associated with a specific operation and must be true *after* an operation is invoked.
- ◆ Postconditions are used to specify constraints that the object (callee) must ensure after the invocation of the operation.

# Examples of Assertions

## Assertion of List method "add"

### PreCondition

list != null //List exists  
key.**instanceOf**(String)

List
-list:Collection
<b>+add(key:Object)</b> +create():void +remove(key:Object) +containsKey(key:Object):boolean +size():int

### PostCondition

size = size + 1;  
containsKey( key ) = true;

- ◆ Parties in the contract: class and client
  - ◆ PreCondition binds clients
  - ◆ PostCondition binds class
- ◆ A precondition expresses what the class requires of its calling client
- ◆ A postcondition expresses what the class guarantees to the calling code upon return
- ◆ Contract important for Class User (Client) and Class Implementor (class)
- ◆ Contract entails benefits and obligations for both parties

# Example

<b>Contract for add of class List</b>	<b>Obligations</b>	<b>Benefits</b>
<b>Class User</b>	Ensure that list exists but is not full before add() is called	Can retrieve item by containsKey( );
<b>Class Implementor</b>	Make sure that key is in the List; List has increased by one	Does not need to care about cases, where List is full or not created

- ◆ Preconditions and postconditions describe properties of individual methods
- ◆ Need for global properties of instances which must be preserved by all routines
- ◆ Examples (List)
  - ◆ Size of List must not be negative
  - ◆ Size of List must be lower than 1000 entries
  - ◆ The number of elements must be even

# Class invariants and class correctness

- ◆ A class invariant is an assertion to be satisfied by all instances of the class at all “stable” times:
  - ◆ on instance creation
  - ◆ before and after every remote call to a routine
  - ◆ may be violated during call
  
- ◆ A class invariant only applies to public methods; private methods are not required to maintain the invariant.



- ◆ An assertion  $I$  is a correct class invariant for a class  $C$  iff the following two conditions hold:
  - ◆ The constructor of  $C$ , when applied to arguments satisfying the constructor's precondition in a state where the attributes have their default values, yields a state satisfying  $I$ .
  - ◆ Every public method of the class, when applied to arguments and a state satisfying both  $I$  and the method's precondition, yields a state satisfying  $I$ .

# Design by Contract Inheritance

---

- ◆ What does inheritance mean in relation to Design by Contract?
  - ➔ Overridden methods must conceptually do the same job!
- ◆ Contract Inheritance
- ◆ Rule of Thumb: Demand no more, promise no less.

Demand no more:

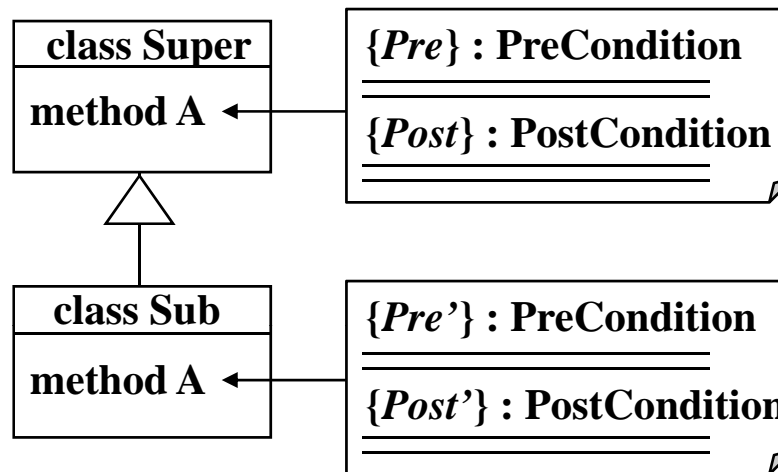
- ◆ A method of a subclass is allowed to **weaken** the **precondition** of the method it overrides
- ◆ An overridden method with a weaker precondition can handle more generic cases (i.e. at least the same or even more cases) than its superclass

Promise no less:

- ◆ A method of a subclass is allowed to **strengthen** the **postcondition** of the method it overrides
- ◆ An overridden method with a stronger postcondition ensures more specific cases (i.e. at most the same or even less cases) to the client upon return

# Design by Contract

## How to check pre and post conditions



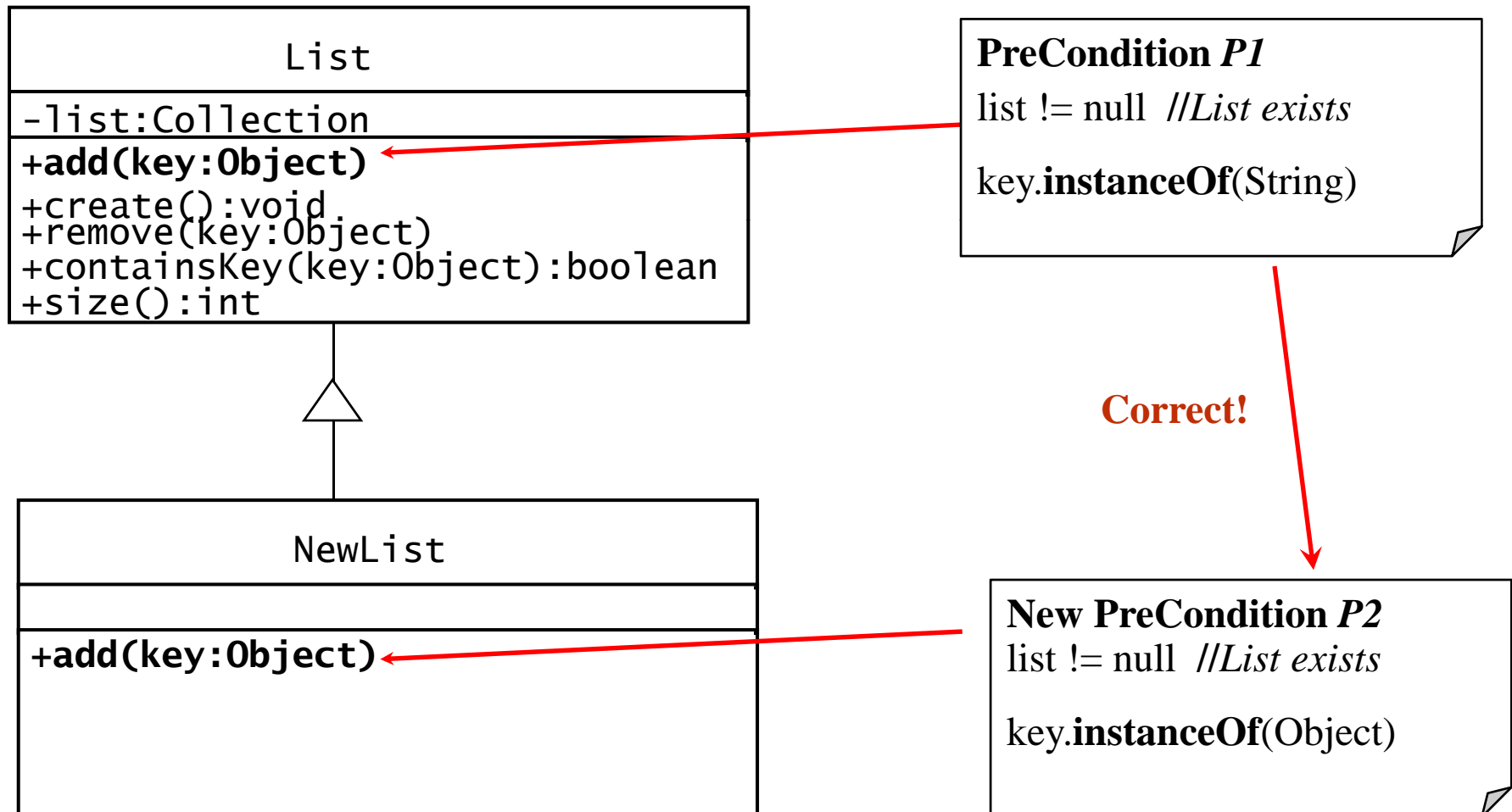
$\{A.Pre'\}$  is weaker than  $\{A.Pre\} \iff A.Pre \implies A.Pre'$

$\{A.Post'\}$  is stronger than  $\{A.Post\} \iff A.Post' \implies A.Post$

# Design by Contract

## Inheritance – Example 1

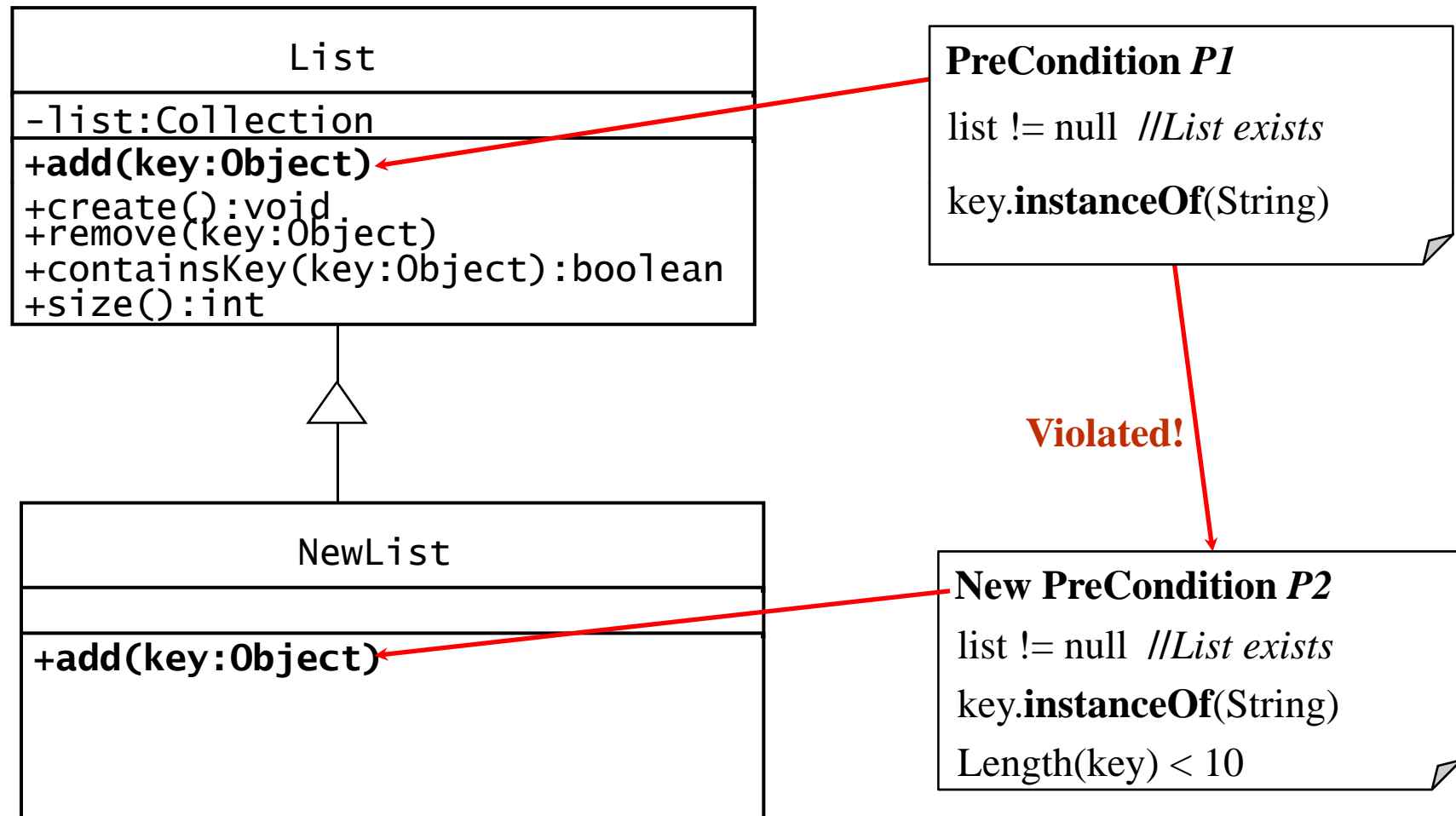
- ◆ A method of subclass is allowed to weaken the **preconditions** of the method it overrides. Example (1)



# Design by Contract

## Inheritance – Example 2

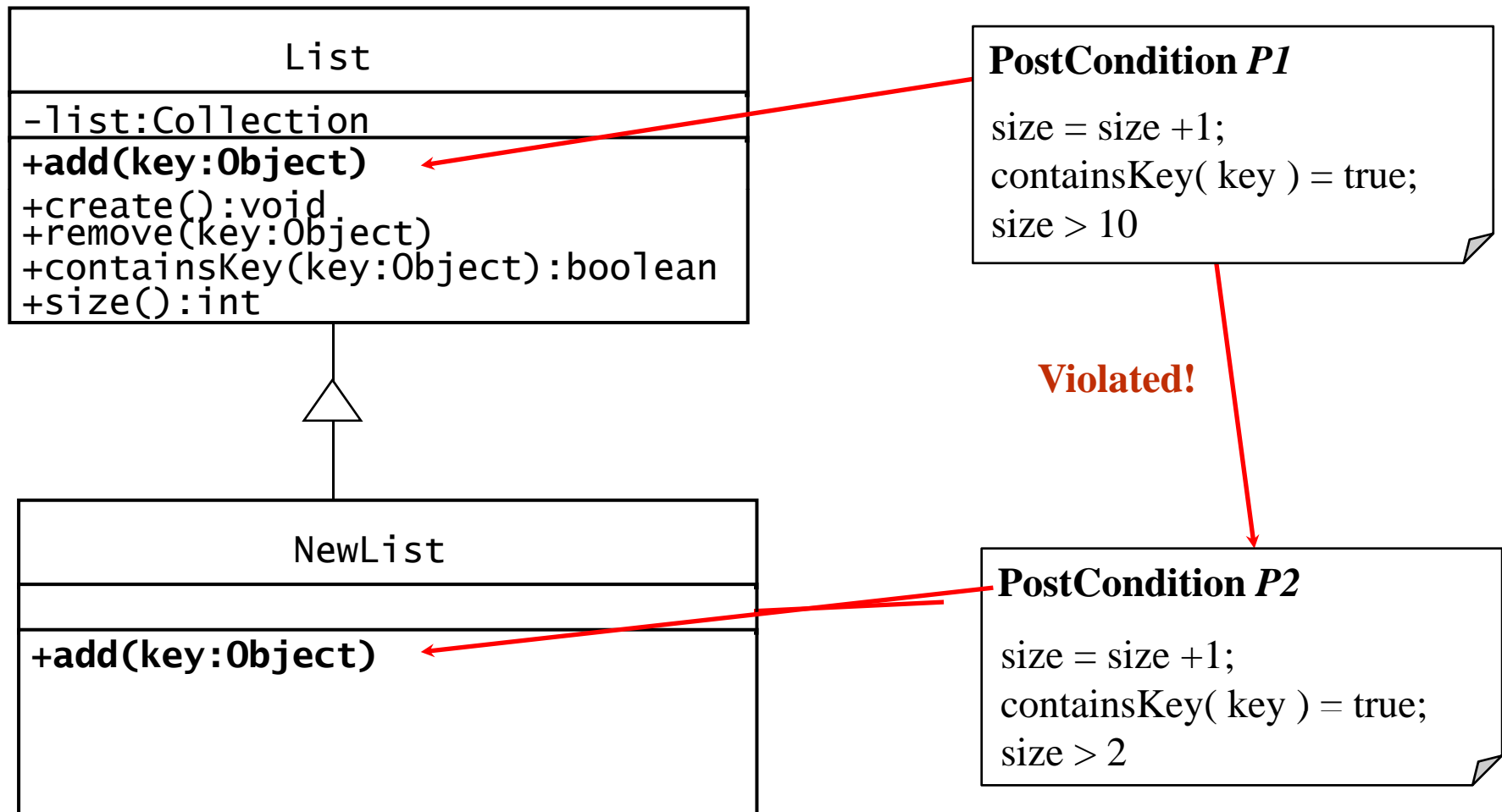
- ◆ A method of subclass is allowed to weaken the **preconditions** of the method it overrides. Example (2)



# Design by Contract

## Inheritance – Example 4

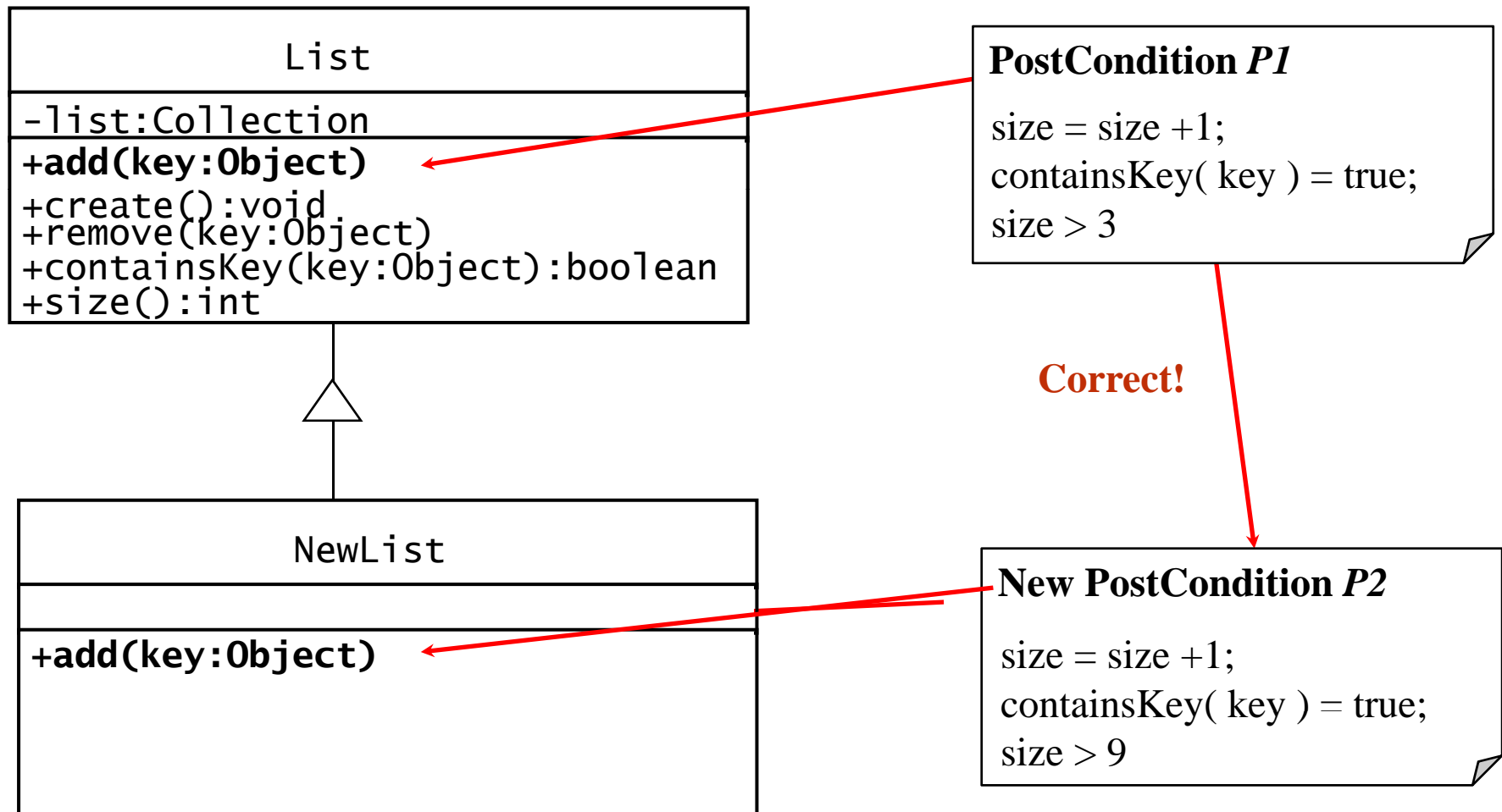
- ◆ A method must ensure a stronger **postcondition** as the method it overrides: Example:



# Design by Contract

## Inheritance – Example 3

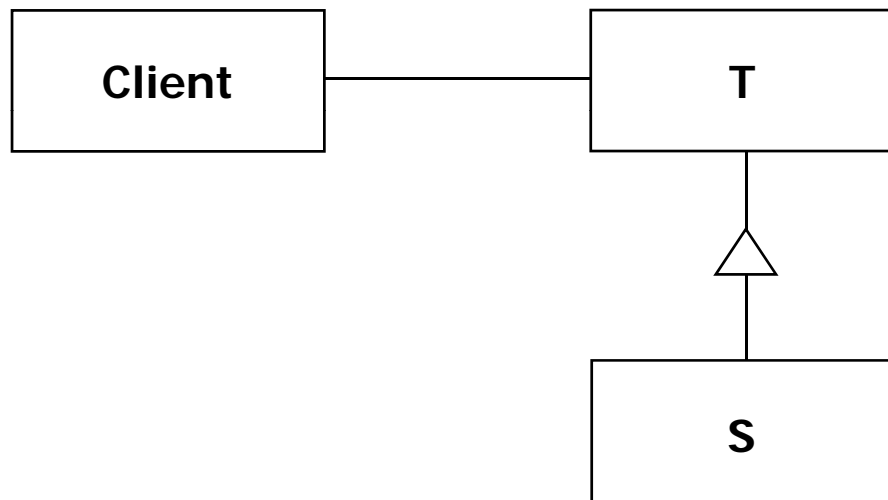
- ◆ A method must ensure a stronger **postcondition** as the method it overrides: Example:





## ◆ The Liskov Substitution Principle:

- ◆ “If an object of type S can be substituted in all places where an object of Type T is expected, then S is a subtype of T”



- ◆ If all preconditions of the methods of type S are weakened and all postconditions of the methods of type S are stronger than the methods of type T, then S is a subtype of type T
- ◆ Client do not has to change its code, if object T is substituted by S

# Advantages of Design By Contract

---

- ◆ Improves the reliability of programs
- ◆ Assertions provide accurate documentation for the implemented classes so that a programmer knows how to use the classes and what to expect from them;
- ◆ Assertions provide a way of controlling inheritance (substitutability)
- ◆ Assertions can be an important aid to developing critical systems.

# Expressing constraints in UML Models

- ◆ OCL 2.0 (Object Constraint Language) is part of UML 2.0
- ◆ OCL allows constraints to be formally specified on single model elements (attributes, operations, classes) or groups of model elements (associations and participating classes)
- ◆ A constraint is expressed as an OCL expression returning the value true or false.
- ◆ OCL is not a procedural language (cannot constrain control flow).
- ◆ More Info:
  - ◆ Introduction of OCL for Together
    - ◆ <http://bdn1.borland.com/devcon05/article/1,2006,33187,00.html>
  - ◆ Tutorial of University of Koblenz
    - ◆ <http://www.uni-koblenz.de/~beckert/Lehre/Praktikum-Formale-Entwicklung/oclIntro.pdf>

# Expressing constraints in UML Models

- ◆ OCL expressions for Hashtable operation put():

- ◆ The **context** keyword indicates the object to which the expression is valid



**Context is a class**

- ◆ Invariant:

- ◆ **context** Hashtable inv: numElements >= 0



**OCL expression**

- ◆ Precondition:

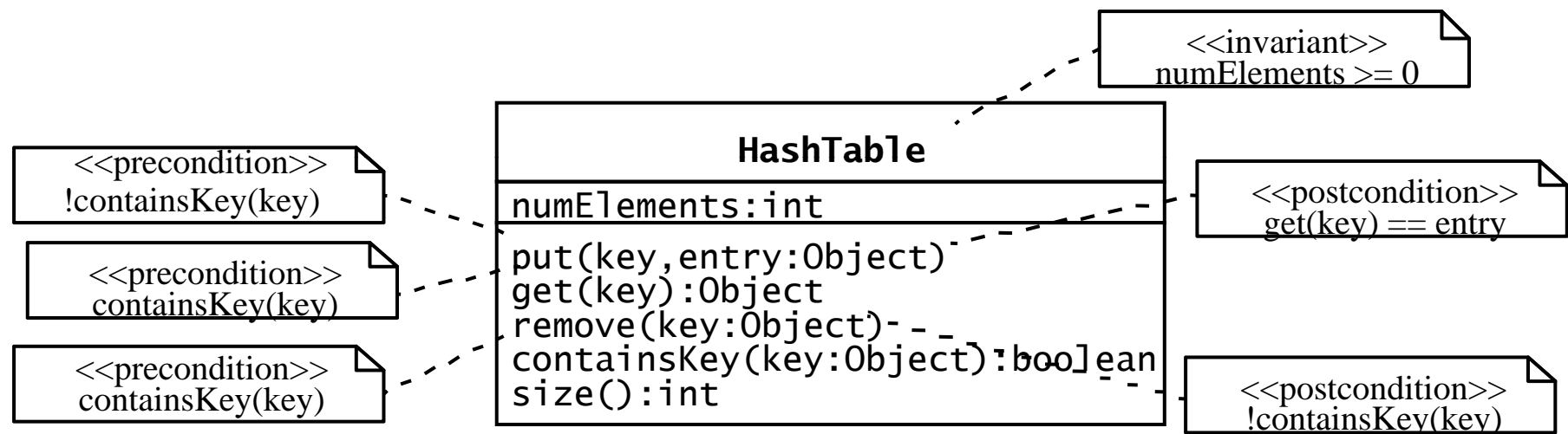
- ◆ **context** Hashtable::put(key, entry) pre: !containsKey(key)

- ◆ 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::removePlayer(p) **pre:**  
isPlayerAccepted(p)

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

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



value returned by  
method *after* invoking  
removePlayer



@pre.= value returned by  
method *before* invoking  
removePlayer

# OCL Annotations JavaDoc (iContract)

```
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
     * 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) {...}
```

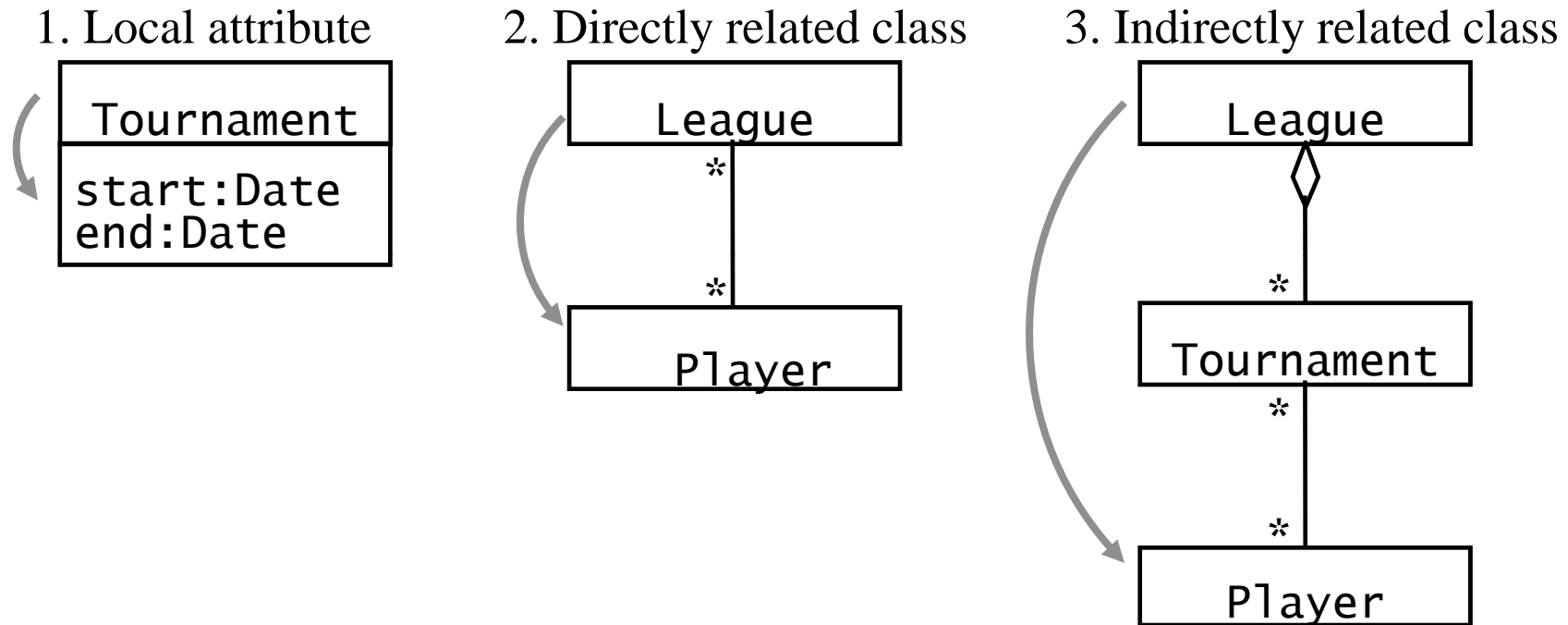
# Constraints can involve more than one class

---

- ◆ Until now: Constraints are valid within a single class
- ◆ Often, Constraints are necessary to declare assertions across many classes
  - ◆ Make Assumptions about associations between classes
  - ◆ Definition of global invariants
- ◆ Expressions for navigating between classes are necessary
- ◆ Set-based operations

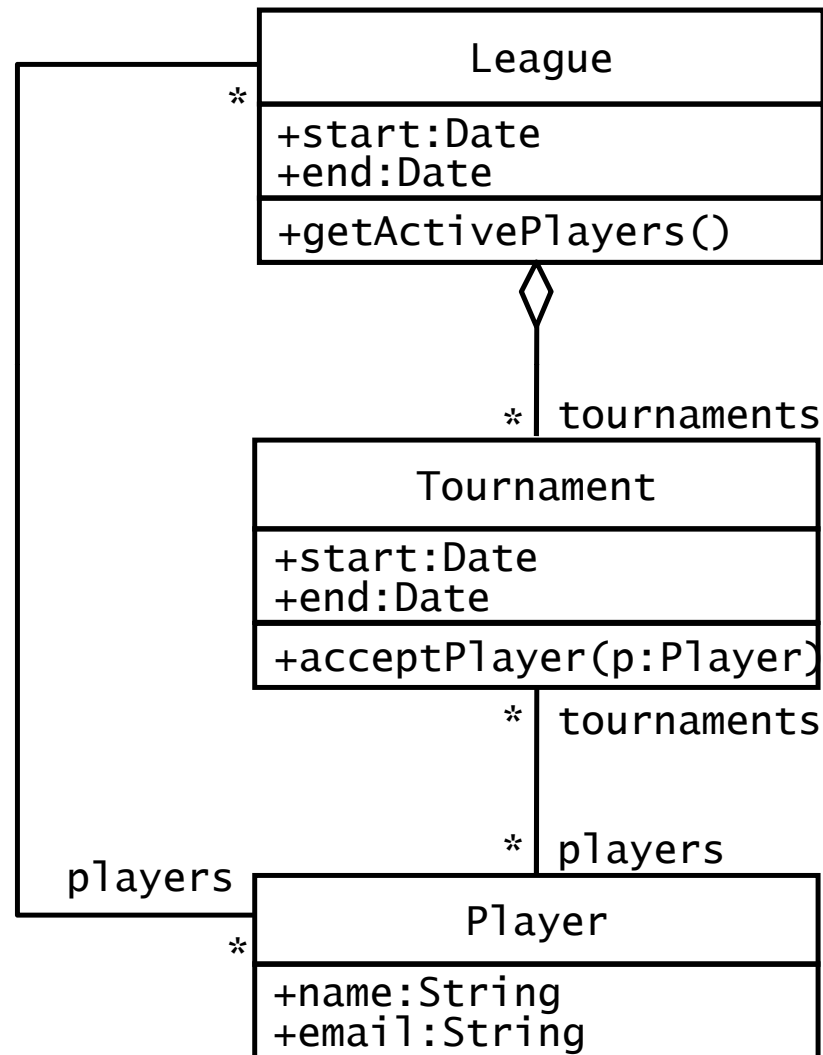


# Types of Navigation through a Class Diagram



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

# Example: League, Tournament and Player

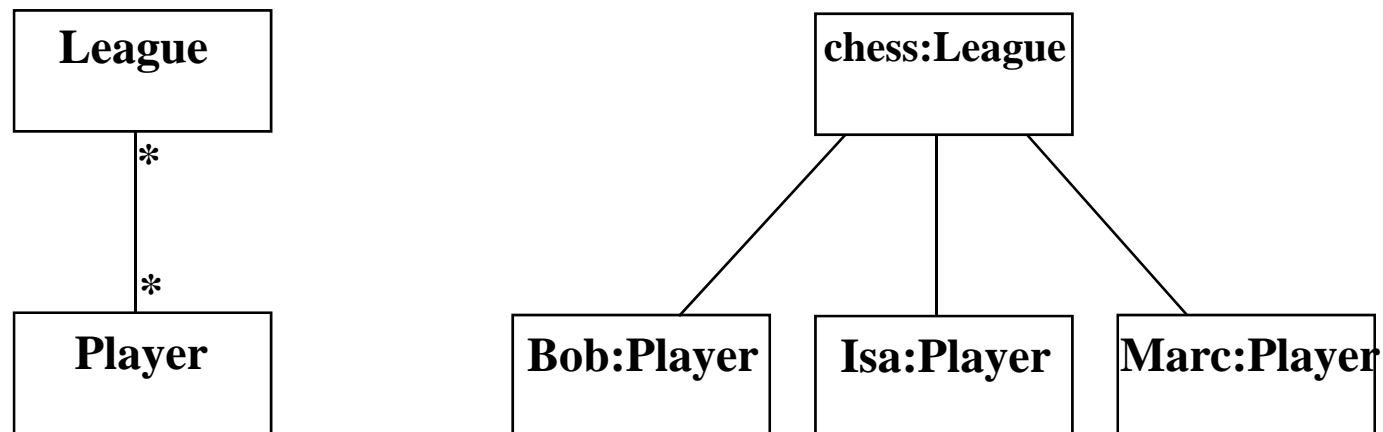


# Model Refinement with 2 additional Constraints

---

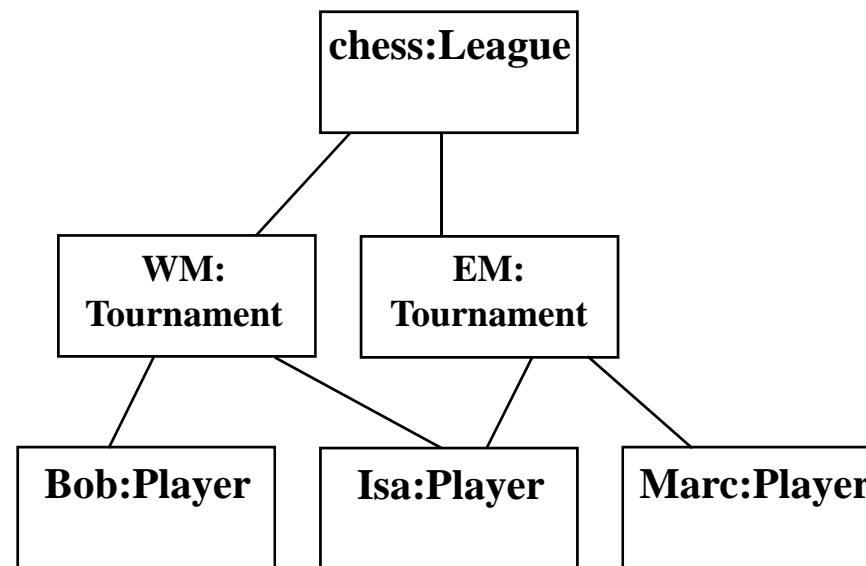
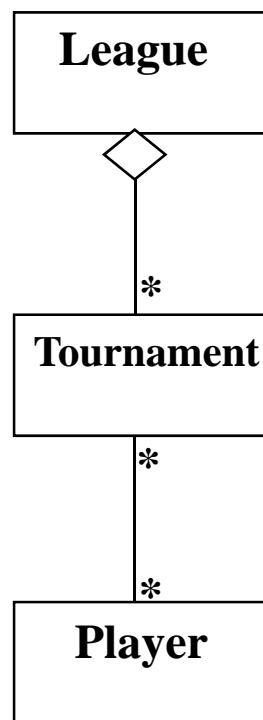
- ◆ A Tournament's planned duration must be under one week.
- ◆ Players can be accepted in a Tournament only if they are already registered with the corresponding League.

- ◆ Collections are data types used to navigate to associated classes within a constraint expression.
  - ◆ OCL Sets
    - ◆ Used to navigate a single (1:1, 1:n, or n:m) association



**league.player = {Bob, Isa, Marc}**

- ◆ Collections are data types used to navigate to associated classes within a constraint expression.
  - ◆ OCL Bags
    - ◆ Used to navigate through a series of at least two associations with one-to-many or many-to-many multiplicity



**league.tournament.player = {Bob, Isa, Marc,Isa}**

# OCL Collections

- ◆ OCL provides many operations for accessing collections (via Access Operator  $\rightarrow$ ). Examples:
  - ◆ `includes(Object)` – returns true if Object is in Collection
  - ◆ `union(Collection)` – returns a Collection containing elements from both the original collection and the collection specified as parameter
  - ◆ `asSet(Collection)` – returns a Set containing each element of the collection without any duplicate elements (as opposed to a bag)
- ◆ Examples:
  - ◆ `{Bob, Isa, Marc} \rightarrow includes( Timo ) = false;`
  - ◆ `league.tournament.player = {Bob, Isa, Marc, Isa}`  
`league.tournament.player \rightarrow asSet = {Bob, Isa, Marc}`

# Specifying the Model Constraints



## Local attribute navigation

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

## Directly related class navigation

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

