Course "Softwaretechnik"

# Object Design: Specifying Interfaces, Model-to-implementation mapping

Lutz Prechelt, Bernd Bruegge & Allen H. Dutoit
Freie Universität Berlin, Institut für Informatik

- Visibility
- Type information
- Contracts: OCL
  - preconditions, postconditions, invariants
  - includes, asSet, forAll, exists
- Mapping associations to code

# Lernziele

- Detailerwägungen zum Geheimnisprinzip (information hiding) in Java machen
- Ein paar Einzelheiten von UML-Klassendiagrammen kennen lernen
- OCL verstehen und warum eine Nutzung sinnvoll sein kann
- Verstehen, wie man Elemente von UML-Klassendiagrammen schematisch in Code überführen kann
  - und warum das nicht unbedingt sinnvoll ist

# Wo sind wir?: Taxonomie "Die Welt der Softwaretechnik"

Freie Universität Berlin

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
  - Anforderungen (Problemraum)
  - Entwurf (Lösungsraum)
- Prozess (psycho-soziale P.)
  - Kognitive Beschränkungen
  - Mängel der Urteilskraft
  - **(Kommunikation, Koordination)**
  - Gruppendynamik
  - Verborgene Ziele
  - Fehler

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
  - **Abstraktion**
  - **Wiederverwendung**
  - **Automatisierung**
- Methodische Ansätze ("weich")
  - Anforderungsermittlung
  - **Entwurf**
  - Qualitätssicherung
  - Projektmanagement

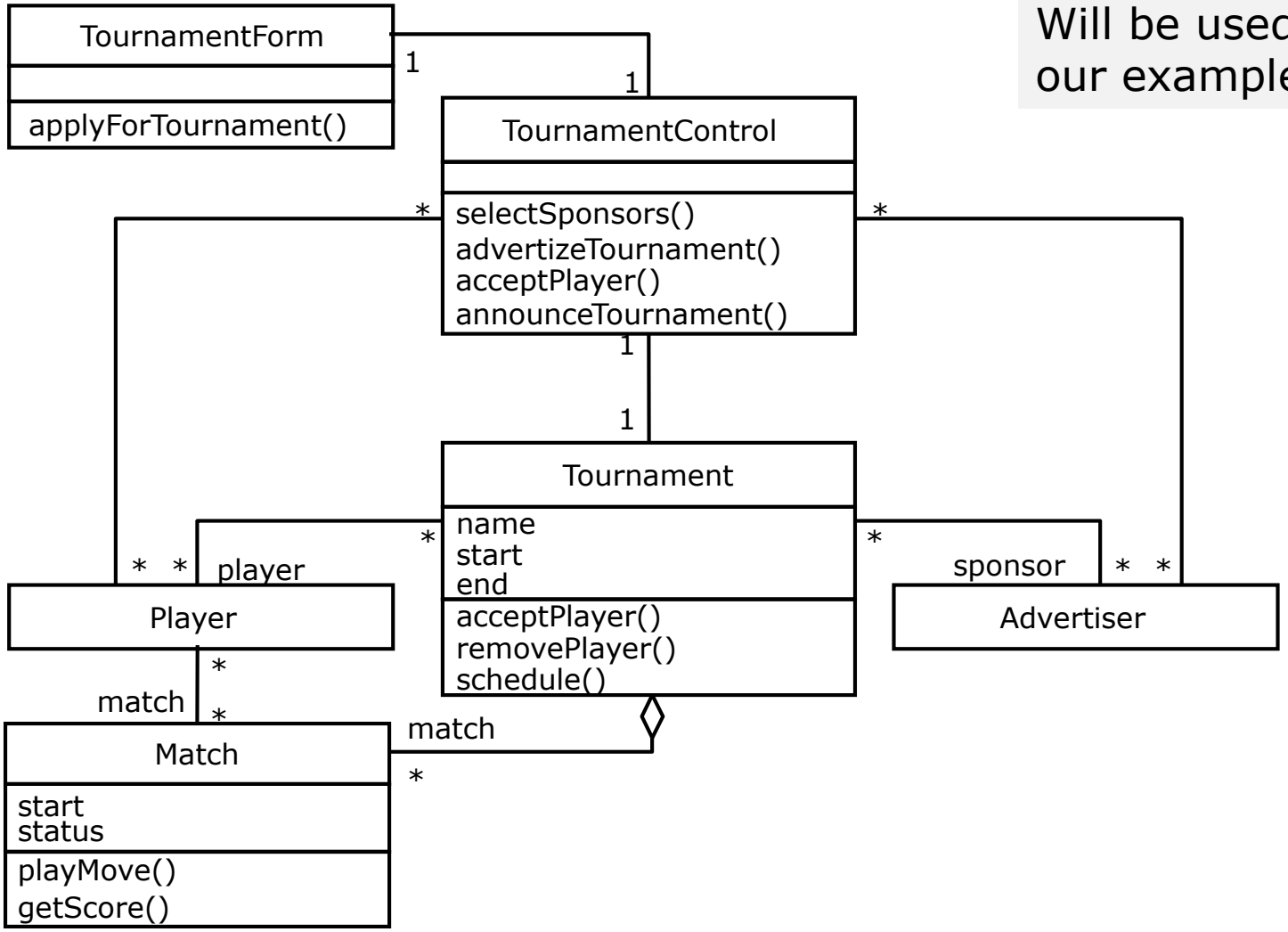
# Wo sind wir?: Entwurf

Freie Universität Berlin

- Einsicht: Man sollte *vor* dem Kodieren über eine günstige Struktur der Software nachdenken
  - und diese als Koordinationsgrundlage schriftlich festhalten

- Prinzipien:
  - **Trennung von Belangen**
  - **Architektur**: Globale Struktur festlegen (Grobentwurf), insbes. für das Erreichen der nichtfunktionalen Anforderungen
  - **Modularisierung**: Trennung von Belangen durch Modularisierung, Kombination der Teile durch Schnittstellen (information hiding, Lokalität)
  - **Wiederverwendung**: Erfinde Architekturen und Entwurfsmuster nicht immer wieder neu
  - **Dokumentation**: Halte sowohl Schnittstellen als auch zu Grunde liegende Entwurfsentscheidungen und deren Begründungen fest

# Part of ARENA's object model identified during the analysis




Will be used in our examples

# Specifying Interfaces

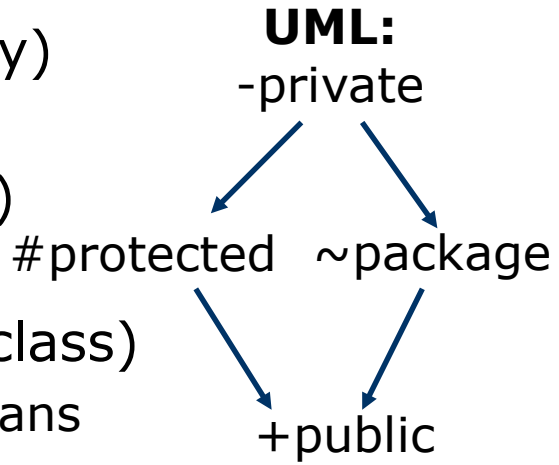


- Requirements analysis activities
  - Identifying attributes and operations without specifying their types or their parameters
    - Often not all attributes and operations are identified in this stage
- Object design: Four activities
  - 0. Identify remaining attributes and operations
  - 1. Add visibility information
  - 2. Add type signature information
  - 3. Add contracts
- Object design is a detail-level subtask of modularization

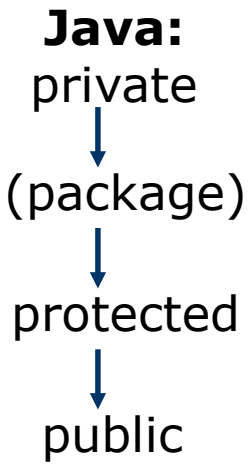
# 1. Add Visibility Information

UML defines four kinds of visibility:

- 1: Private (visible for class implementer only)
  - marked by '-' in diagrams
- 2a: Protected (visible also for class extender)
  - marked by '#' in diagrams
- 2b: Package (private to a package, not to a class)
  - when a package represents a module, this means 'publicly visible inside the module'
  - marked by '~' in diagrams
- 3: Public (fully visible)
  - marked by '+' in diagrams




- Difference to Java visibilities:
  - Java: 'protected' is also visible throughout the package. This is not true (and cannot be expressed) in UML
    - The 'package' default promotes creation of Facades



# Information Hiding heuristics at design time



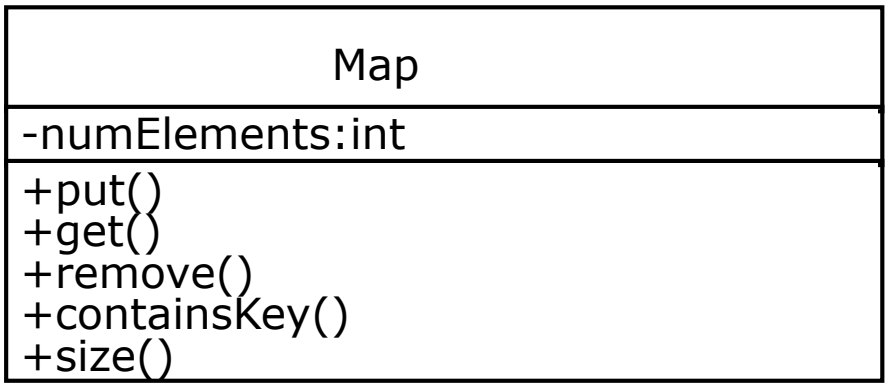
- In the analysis model, everything is considered public

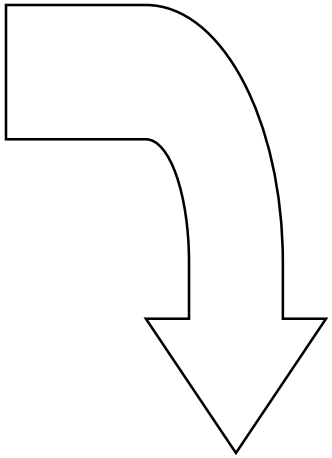
- Carefully define the public interface for classes as well as subsystems (façade)
- Export: Consider the "Need to know" principle
  - Only if somebody probably needs to access the information make it publicly possible,
  - preferably through well-defined channels, so the module can control the access (in particular changes to individual attributes).
- Import: The less an operation knows
  - the less likely it will be affected by any changes
  - the easier the module can often be changed
- Trade-off: Information hiding vs. efficiency or simplicity
  - In a few cases, accessing a private attribute might be better e.g. for speed reasons in real-time systems or games
    - BUT: *"Make it work first before you make it work fast"*
    - Low-ceremony languages rely on good judgment everywhere

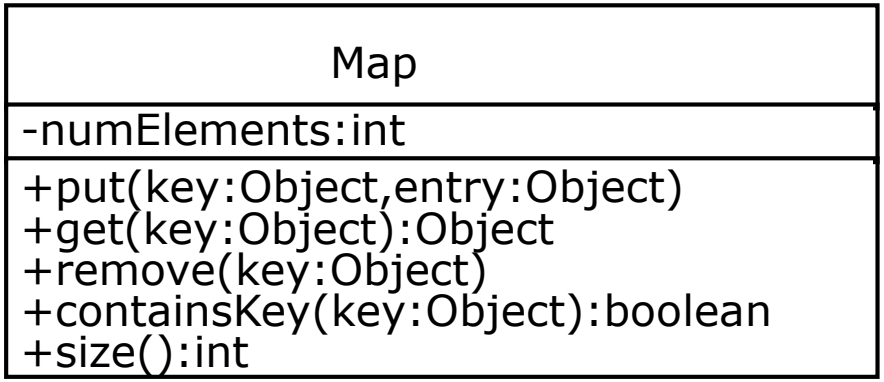
# Java: Packages as modules

- The module interface contains one *Facade* class (for methods) plus perhaps several data type classes (for data and methods)
  - perhaps interfaces only, not actual classes
- These classes or interfaces are *public*, all others have *package* visibility
  - and all members of these 'other' classes have *package* or *private* visibility (*public* and *protected* would not help)
  - *Package* (or *default*) visibility in Java has no visibility declarator
- Most members of *public* classes have *public* or *protected* visibility
  - protected members weaken the information hiding.
  - *private* should be used when the class is so complicated that *protected* would likely lead to integrity violations
  - *package* (for module-internal class-external access) is rarely needed, but may result in fewer changes over time

# 2. Add Type Signature Information






A thinking step, not necessarily in a class diagr.

# 3. Add Contracts



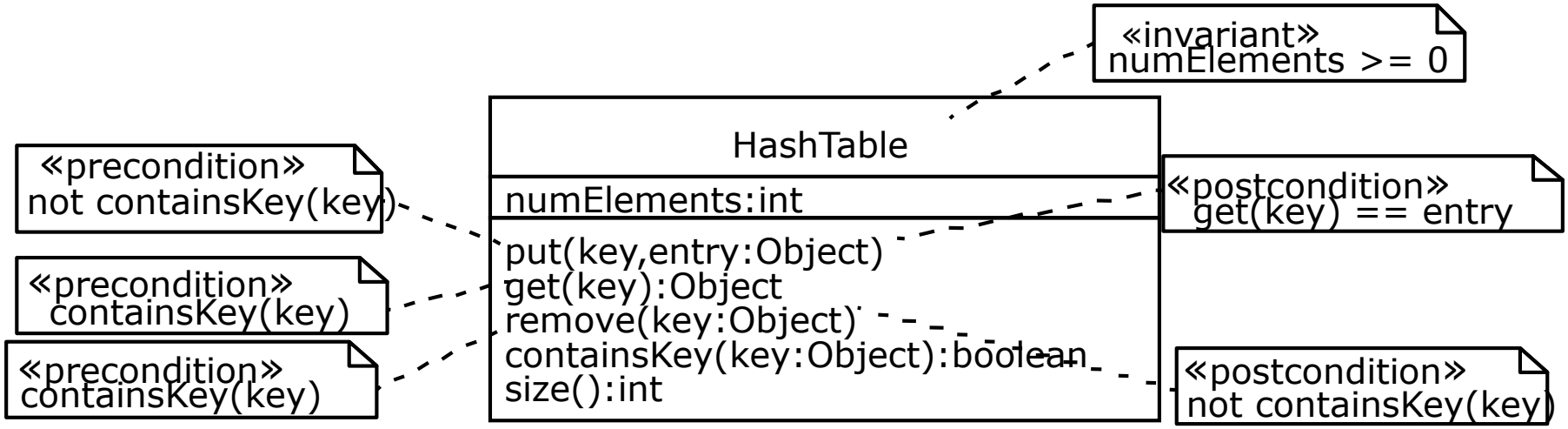
- Contracts on a class enable caller and callee to share the same assumptions about the class

Contracts include three types of constraints:

- Invariant:
  - A predicate that is true for an instance after any external call. Invariants are constraints associated with classes or interfaces
  - The invariant is an implicit part of each public postcondition
- Precondition:
  - Preconditions are predicates associated with a specific operation and must be true before the operation is invoked
  - They specify constraints that a caller must ensure before the call
- Postcondition:
  - Postconditions are predicates associated with a specific operation and must be true after the operation is invoked
  - They specify constraints that the class must ensure when the call returns

# OCL: Expressing Constraints in UML Models

- An OCL constraint can be depicted as a note attached to the constrained UML element by a dependency relationship




- Disadvantage?

- Or it can be specified textually outside the UML diagram:

# Contract for acceptPlayer in Tournament

Freie Universität Berlin

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() = getNumPlayers@pre() + 1

The value of the expression before the call

# Contract for removePlayer in Tournament

Freie Universität Berlin

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

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

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

**Is this contract complete?**

**No.** OCL specifications tend to make the tacit assumption that "everything else stays the same" -- they are very often incomplete.

# Annotation of Tournament class

```
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() {…}

    /** 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) {…}
}
```

Note: **`@pre`** etc. is not Javadoc syntax, but JContract (or similar) syntax.
See http://en.wikipedia.org/wiki/Design_by_contract for a list of tools.

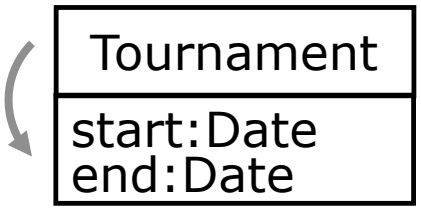
# Constraints
# can involve more than one class

Freie Universität Berlin

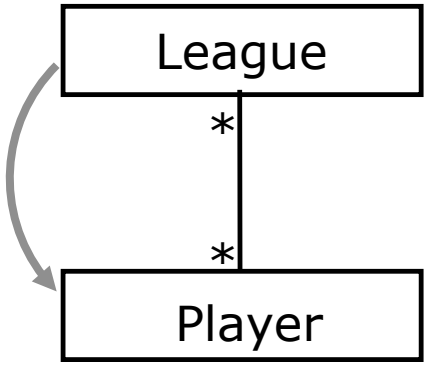
**How do we specify constraints on more than one object?**

# 3 Types of Navigation through a Class Diagram

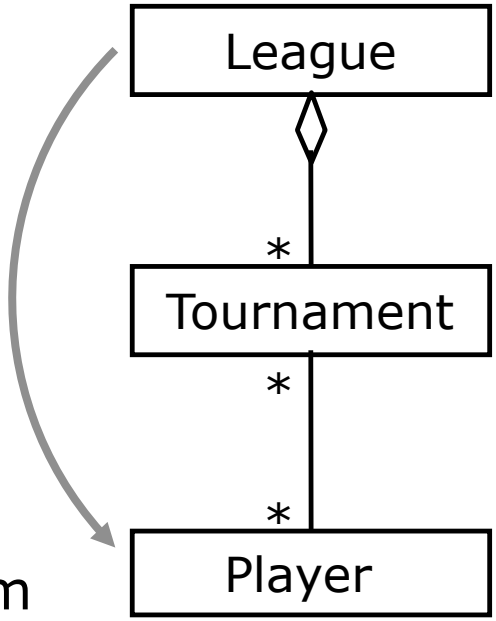
1. Local attribute




2. Directly related class

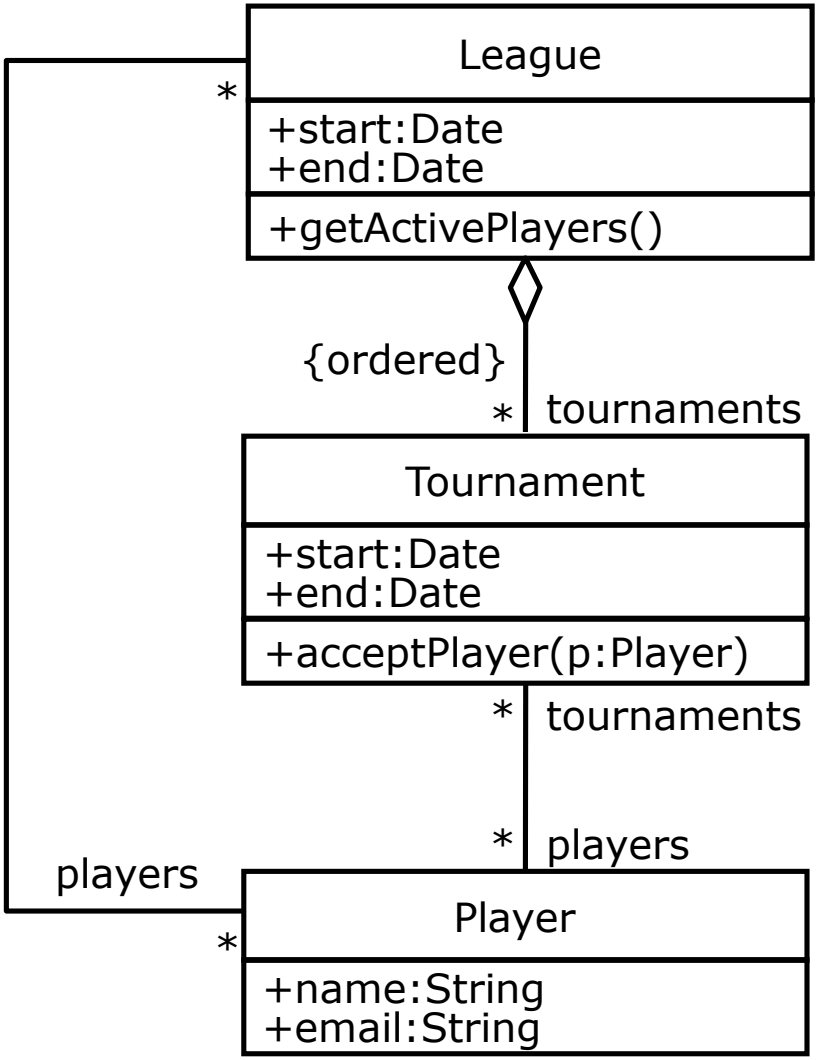



3. Indirectly related class




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

# ARENA Example: League, Tournament and Player



# Model refinement with 3 additional constraints

Freie Universität Berlin

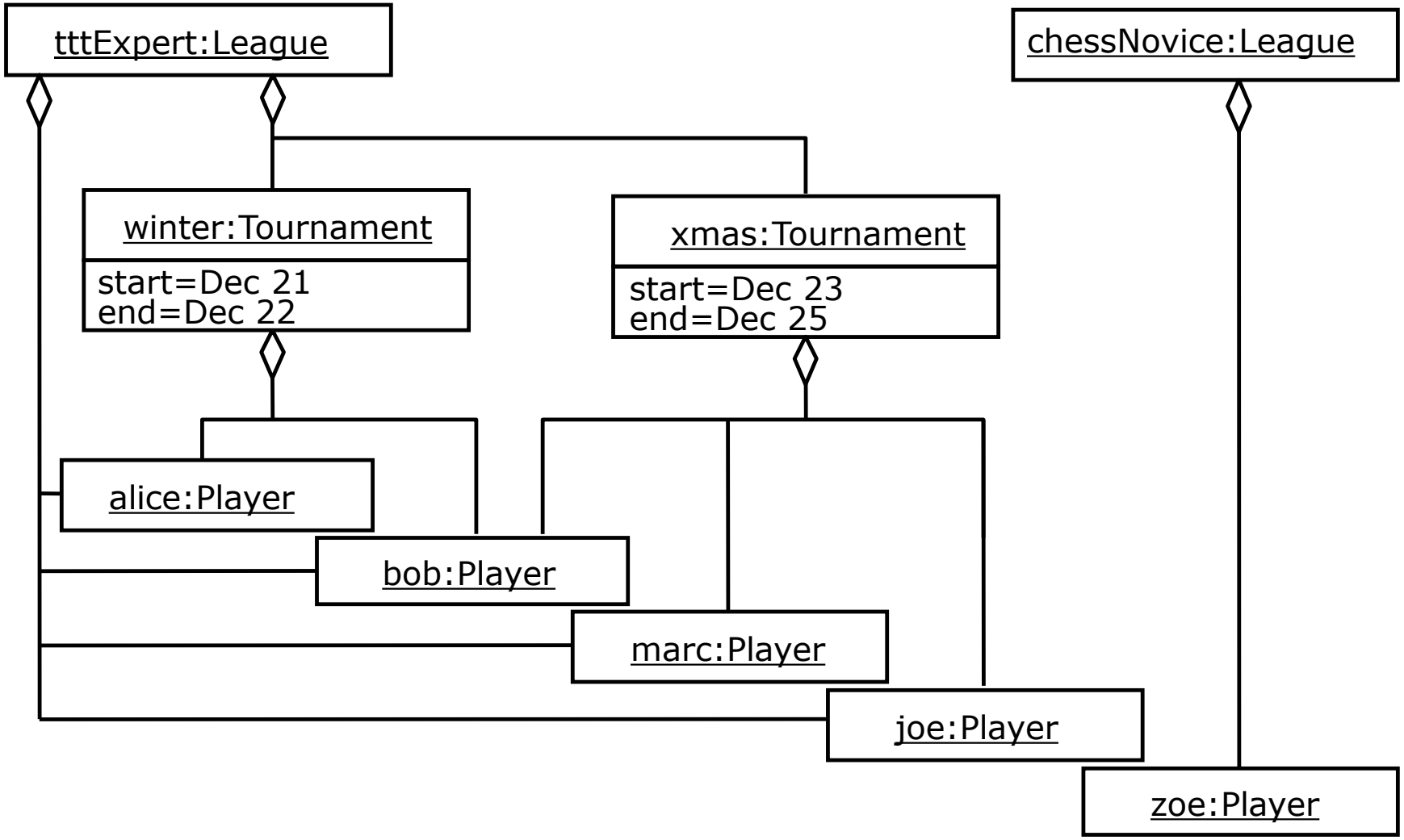
1. A Tournament's planned duration must be under one week

2. Players can be accepted in a Tournament only if they are already registered with the corresponding League

3. The Active Players in a League are those that have taken part in at least one Tournament of the League

- To better understand these constraints we instantiate the class diagram for a specific group of instances
  - 2 Leagues, 2 Tournaments and 5 Players

# Instance Diagram: 2 Leagues, 2 Tournaments, and 5 Players



# Specifying the Model Constraints

**Local attribute navigation**

context **Tournament** inv:

**end - start <= Calendar.WEEK**

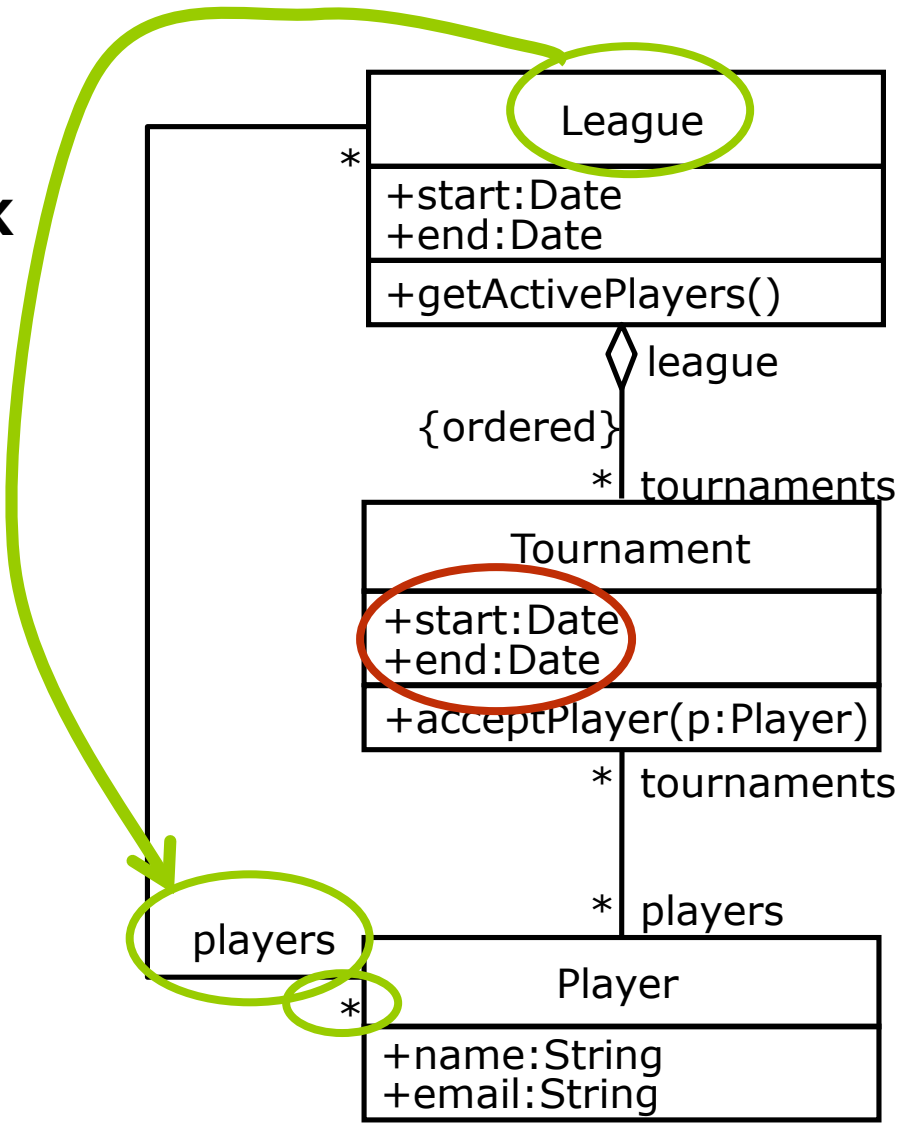
**Directly related class navigation**

context

**Tournament::acceptPlayer(p)**

pre:

**league.players->includes(p)**




Is the *League* arrow correct?

# Specifying the Model Constraints

**Local attribute navigation**

context **Tournament** inv:

**end - start <= Calendar.WEEK**

**Directly related class navigation**

context **Tournament::acceptPlayer(p)** pre:

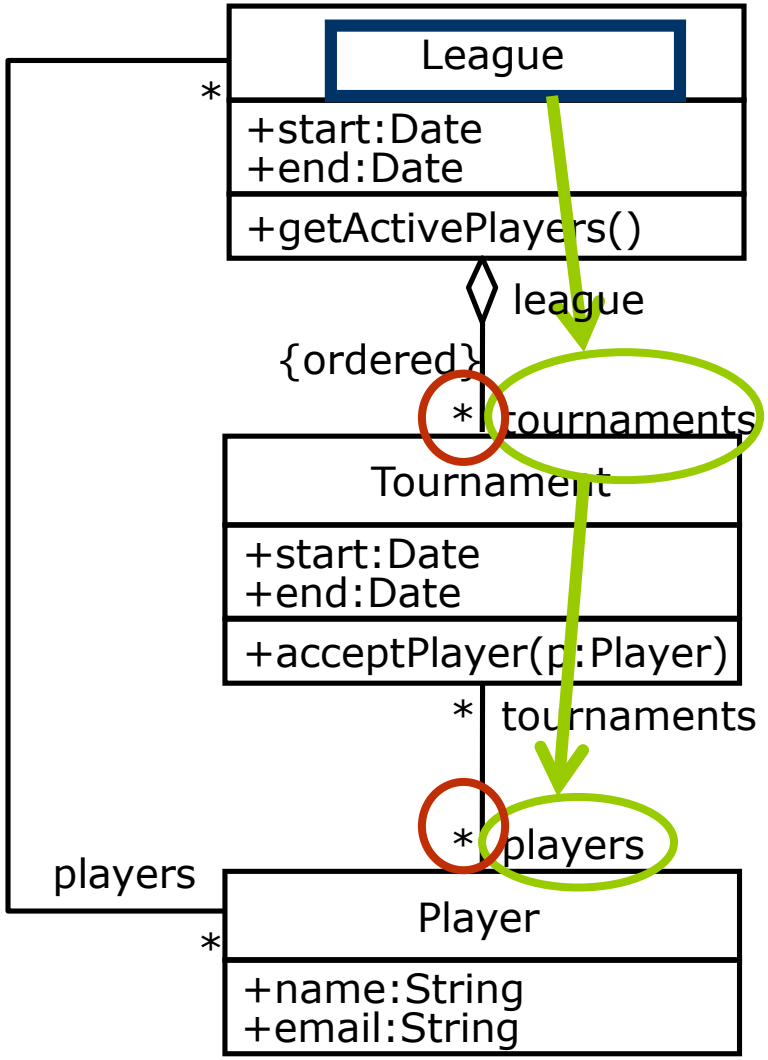
**league.players->includes(p)**

**Indirectly related class navigation**

context **League::getActivePlayers** post:

**result = tournaments->iterate(**

**t, p = {} | p union t.players )**



# Pre- and post-conditions for ordering operations on TournamentControl



1. Which order of calls will be enforced?
2. There are at least two dubious conditions here. Which?

| TournamentControl |
| --- |
| |
| +selectSponsors(advertisers):List<br>+advertizeTournament()<br>+acceptPlayer(p)<br>+announceTournament()<br>+isPlayerOverbooked():boolean |

**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)

# OCL supports Quantification

- OCL **forall** quantifier
  */* "All Matches in a Tournament occur within the Tournament's time frame": */*
  - context Tournament inv:
    matches->forAll(m |
    m.start.after(self.start) and m.end.before(self.end))

- OCL **exists** quantifier
  */* "Each Tournament conducts at least one Match on the first day of the Tournament": */*
  - context Tournament inv:
    matches->exists(m | m.start.equals(self.start))

There is at least one dubious condition here. Which?

# Specifying invariants on Tournament and Tournament Control

Freie Universität Berlin

- */* "All Matches in a Tournament occur within the Tournament's time frame": */*

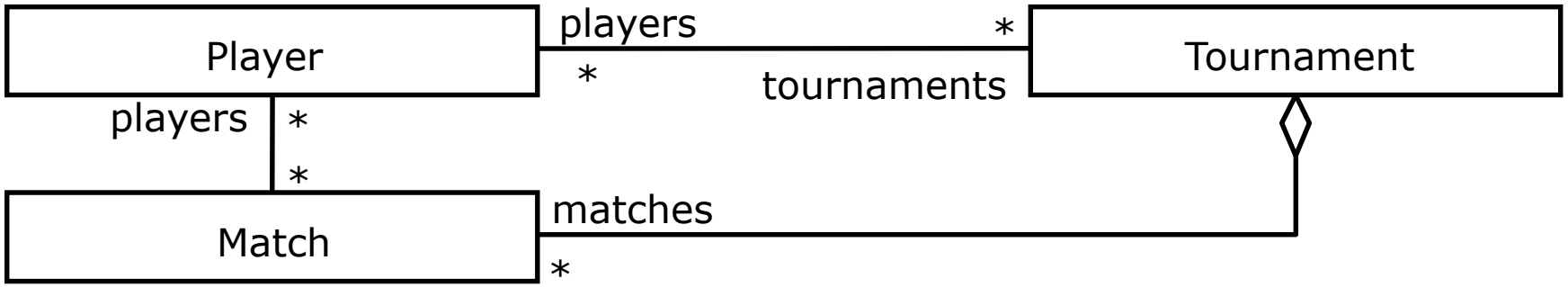
```
context Tournament inv:
    matches->forAll(m | m.start.after(self.start) and
                        m.end.before(self.end))
```

- */* "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

In this diagram, can Match m7 be among a Tournament's Matches without being among that Tournament's Players' Matches?




**Yes.** So we specify:

```
/* "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)  /* insufficient! */
/* this condition is too weak, as it requires only one player to be registered */
```

# OCL in practice: general comments

Freie Universität Berlin

Rules of thumb:

- Preconditions can often be expressed quite easily
- Invariants as well
- Postconditions are usually difficult to express in OCL
  - but even incomplete specifications can be useful
  - In that case, add a comment describing the rest
- It is often useful to introduce predicate methods in a class for simplifying the OCL expressions
  - see examples above

# OCL in practice: today



- OCL can be used to generate code which checks the behavior of classes at run time
  - Such implementations today often do not handle quantifiers
    - because their operationalization is often not practical
  - Similar mechanisms are available for Java by means of preprocessors
    - e.g. JContract
    - The constraints are expressed using Javadoc tags
    - The preprocessor inserts appropriate code

- Some languages have such (or perhaps simpler) mechanisms built-in
  - e.g. Eiffel: keywords *require, ensure, invariant*
  - Plain Java: **`assert`** expressions

https://en.wikipedia.org/wiki/Design_by_contract#Language_support

# OCL in practice: future

- In the future, tools (e.g. compilers) will often be able to check the consistency of code and OCL specifications
  - so no runtime checks are required
  - May often even be capable of checking quantified expressions
    - by applying compile-time verification (e.g. by model checking)
  - Will not be able to check all kinds of OCL specification, but many

- Consequence:
  Start specifying difficult contracts as soon as possible
  in your daily work

# Model-to-implementation mapping

Freie Universität Berlin

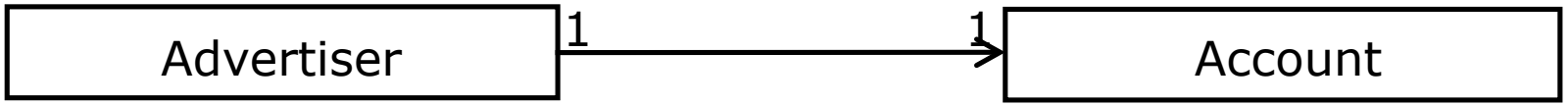
- Some aspects of detailed UML design models can be mapped into implementations schematically
  - Sometimes, this is done automatically by tools (**Model-driven architecture**, MDA)
- Example areas:
  1. Mapping associations to code
  2. Mapping contract violations to exceptions
  3. Mapping classes and associations to rDBMS database tables (**Object-relational mapping**, ORM)

- Let us look at association mapping as an example
  - and learn why manual coding is often preferable

# Realization of a unidirectional, one-to-one association

Freie Universität Berlin

## Object design model before transformation




## Source code after transformation

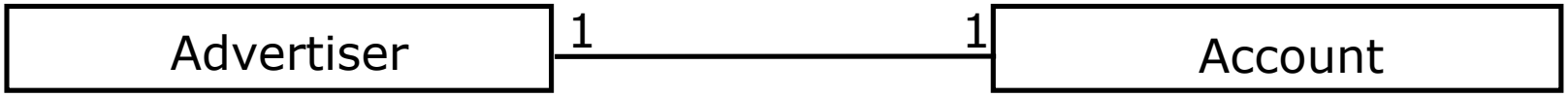
```
public class Advertiser {
    protected Account account;
    public Advertiser() {
        account = new Account();
    }
    public Account getAccount() {
        return account;
    }
}
```

create a *setAccount()* if the *Account* object is pre-existing

for bidirectional associations do likewise in *Account:*

# Bidirectional one-to-one association

## Object design model before transformation




## Source code after transformation

```
public class Advertiser {
    /* account is initialized in
     * constructor, never modified.
     */
    protected Account account;
    public Advertiser() {
      account = new Account(this);
    }
    public Account getAccount() {
            return account;
    }
}
```

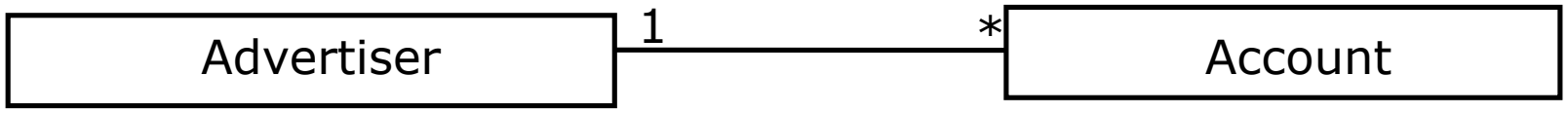
```
public class Account {
    /* owner is initialized in
     * constructor, never modified.
     */
    protected Advertiser owner;

    public Account(
                Advertiser owner) {
            this.owner = owner;
    }
    public Advertiser getOwner() {
            return owner;
    }
}
```

Does this work as intended? What can go wrong?

# Bidirectional, one-to-many association

Freie Universität Berlin

## Object design model before transformation




## Source code after transformation

```
public class Advertiser {
    protected Set accounts = new HashSet();
    public void addAccount(Account a) {
            accounts.add(a);
            if (a.getOwner() != this)
              a.setOwner(this);
    }
    public void removeAccount(Account a) {
            accounts.remove(a);
            if (a.getOwner() == this)
              a.setOwner(null);
    }
}
```

removeAccount breaks the UML model. Where?

For a good reason?

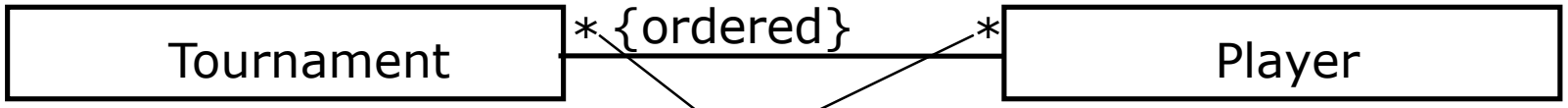
```
public class Account {
   protected Advertiser owner = null;
   public void setOwner(Advertiser
                                newOwner) {
      Advertiser oldOwner = owner;
      owner = null;  // cancel previous owner
      if (oldOwner != null)
         oldOwner.removeAccount(this);
      owner = newOwner;
      if (newOwner != null)
         newOwner.addAccount(this);
   }
   public Advertiser getOwner() {
      return owner;
   }
}
```

(beware of infinite recursion!)

# Bidirectional, many-to-many association

Freie Universität Berlin

## Object design model before transformation




## Source code after transformation

```
public class Tournament {
    protected List players;
    public Tournament() {
      players = new ArrayList();
    }
    public void addPlayer(Player p) {
     if (!players.contains(p)) {
         players.add(p);
         p.addTournament(this);
     }
    }
}
```

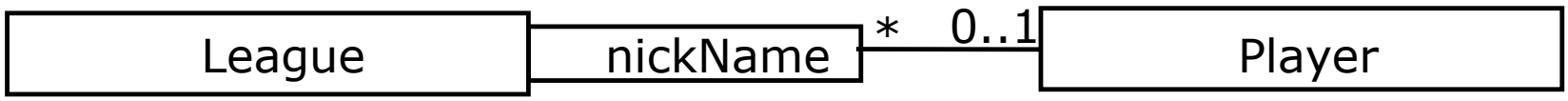
...and removePlayer (complicated!)

```
public class Player {
    protected List tournaments;
    public Player() {
     tournaments = new ArrayList();
    }
    public void addTournament(
          Tournament t) {
      if (!tournaments.contains(t)) {
          tournaments.add(t);
          t.addPlayer(this);
      }
    }
```

(beware of infinite recursion!)

Tools can do code generation. Disadvantage?

# Bidirectional qualified association (2)




**Source code after forward engineering:**

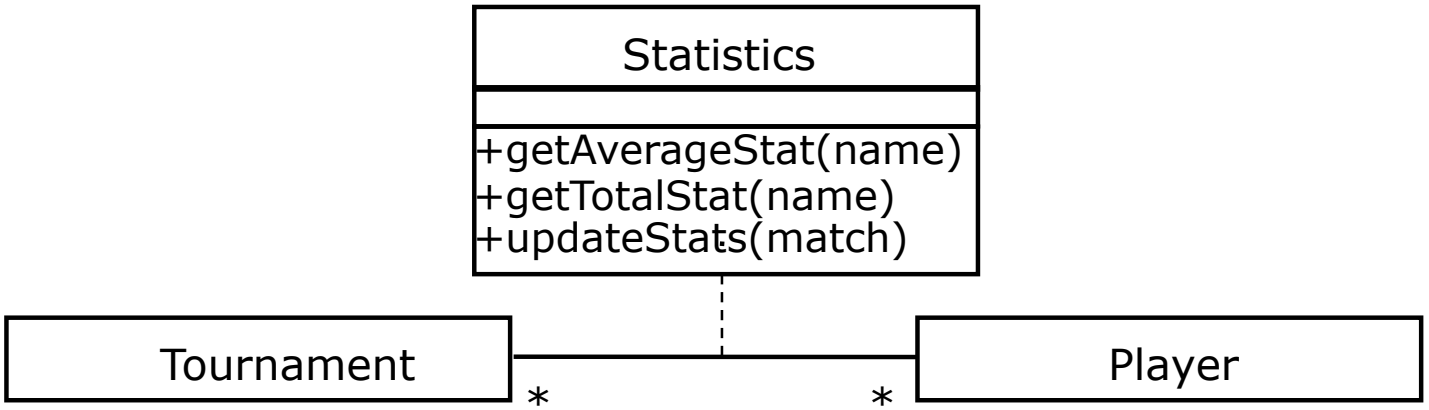
```
public class League {
  protected Map players;

  public void addPlayer
       (String nickName, Player p) {
     if (!players.
            containsKey(nickName)) {
       players.put(nickName, p);
       p.addLeague(nickName, this);
     }
    }
}
```

```
public class Player {
    protected Map leagues;

    public void addLeague
        (String nickName, League l) {
     if (!leagues.
              containsKey(l)) {
        leagues.put(l, nickName);
        l.addPlayer(nickName,this);
     }
    }
}
```

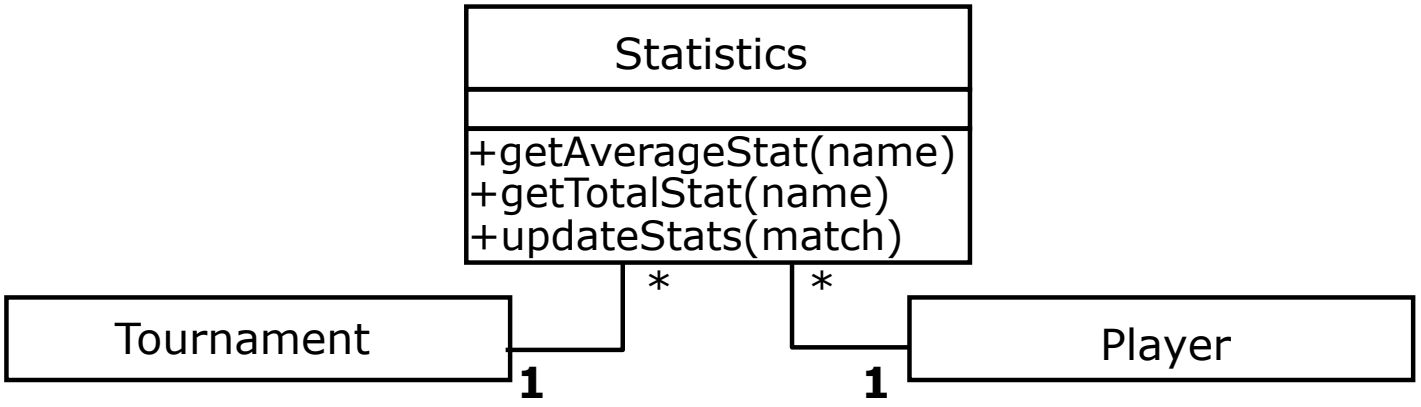
# Transformation of an association class

Freie Universität Berlin

## Object design model before transformation




## Object design model after transformation: A class and two binary associations



# Conventional vs. Agile processes

# Conventional vs. Agile processes wrt module specification / code generation

Freie Universität Berlin

- Sometimes inclined to detailed specifications
  - but often not
- Tend to like code generation
  - expect effort savings
  - (How common it is? Well, how common are conventional processes? What are conv. processes anyway?)
- Detailed specification is rare
  - is not lightweight
  - must be changed along with the code
- Skeptical of code generation except in the simplest cases
  - because it may get in the way of simple solutions

Precise and detailed UML design models are not common on either side!

# Summary



- During object design (and only then) we specify **visibility**
- **Contracts** are constraints on a class that enable class users, implementers, and extenders to share the same assumptions about the class ("Design by contract")
  - **Constraints** are boolean expressions on model elements
- **OCL** is a language that allows us to express constraints
  - OCL (object constraint language) is part of the UML world
    - but separate from UML proper
- Complicated constraints involving more than one class, attribute or operation can be expressed with 3 basic navigation types
- Various types of models can be mapped to code systematically

# Thank you!