Section 4.3
Specifying Interfaces

1. Overview
2. Concepts
3. Activities

4.3.1 Overview

- Learning outcomes
  - construct the detailed object model

Overview (cont.)

- Goal
  - specify boundaries between objects
  - integrate all existing, partial models into one coherent whole

- Steps
  - identify missing attributes and operations
    - augment the object design model
  - specify visibility and signatures
    - decide on operations available to other objects and subsystems
    - determine operation signatures and return types
  - specify contracts
    - describe object and operation behaviour in terms of constraints

4.3.2 Interface Specification Concepts

- Class developer roles
- Contracts
- Object Constraint Language (OCL)
- OCL collections
- OCL qualifiers

Class Developer Roles

- Three possible roles for class developers:
  - class implementer
  - class user
  - class extender

- Class implementer
  - implements the class
  - designs the internal data structures
  - implements the code for the operations
  - designs the interface specification

Class Developer Roles (cont.)

- Class user
  - invokes the class operations from another class
    - this other class is called the *client class*
  - uses the interface specification as a boundary to the class

- Class extender
  - develops specializations of the class
    - the subclasses
  - uses the interface specification as the indication of:
    - the behaviour of the class
    - the constraints on the class

Class Developer Roles (cont.)

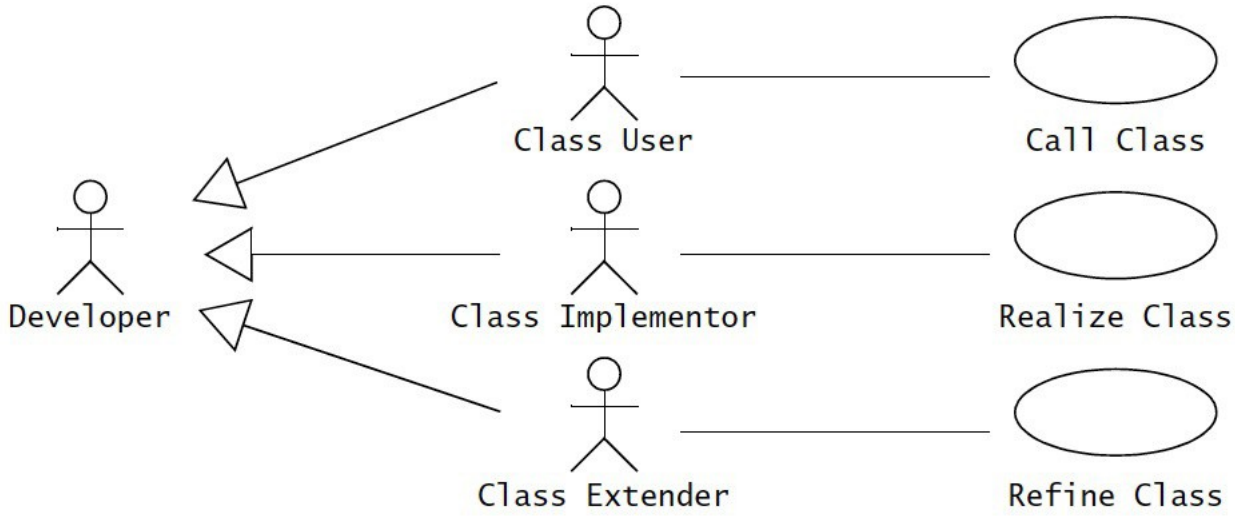

Figure 9-1 The Class Implementor, the Class Extender, and the Class User role (UML use case diagram).

Class Developer Roles (cont.)

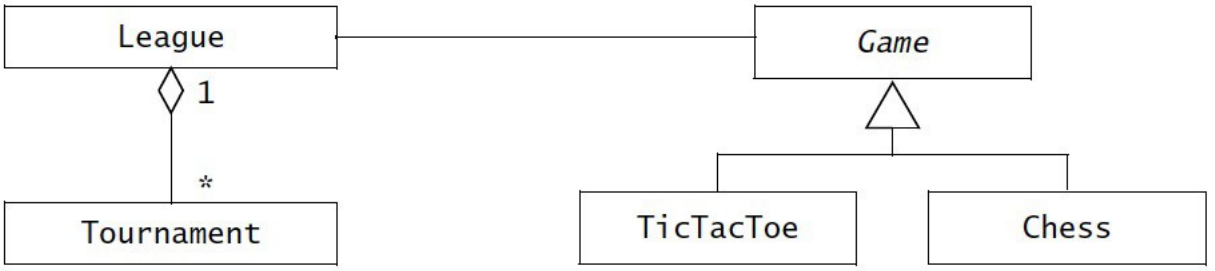

Figure 9-2 ARENA *Game* abstract class with user classes and extender classes.

Contracts

- What is a contract?
  - specifies the constraints on a class
  - these constraints:
    - must be ensured by:
      - class implementer
      - class extender
    - must be met by:
      - class user

- Contracts include three types of constraints:
  - invariant
  - precondition
  - postcondition

Contracts (cont.)

- What is an *invariant*?
  - it's a predicate that is always true for all instances of a class
  - it's associated with a class or an interface
  - it's used to specify consistency constraints among attributes
  - example:
    - maximum number of players in tournament must be greater than 0
    - given a `Tournament` object `t`: `t.getMaxNumPlayers() > 0`

Contracts (cont.)

- What is a *precondition*?
  - it's a predicate that must be true before an operation is invoked
  - it's associated with an operation
  - it's used to specify constraints that class user must meet before invoking the operation
  - example of a precondition for `acceptPlayer()` operation:
    - player must not already be accepted, and the current number of players must be less than the maximum
    - given a `Tournament` object `t` and player `p`:

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

Contracts (cont.)

- What is a *postcondition*?
  - it's a predicate that must be true after an operation executes
  - it's associated with an operation
  - it's used to specify constraints that the class implementer and the class extender must ensure after execution
  - example of a postcondition for `acceptPlayer()` operation:
    - accepting a player must increase the player count by 1
    - given a `Tournament` object `t` and player `p`:

```
t.getNumPlayers_afterAccept() =
        t.getNumPlayers_beforeAccept() + 1
```

Object Constraint Language (OCL)

- What is OCL?
  - it's a formal language used to specify constraints

- How is OCL used?
  - it may be used for constraints on:
    - single model elements
      - attributes, operations, classes
    - groups of model elements
      - associations, participating classes
  - its syntax is Pascal-like
  - it represents constraints as boolean expressions

OCL (cont.)

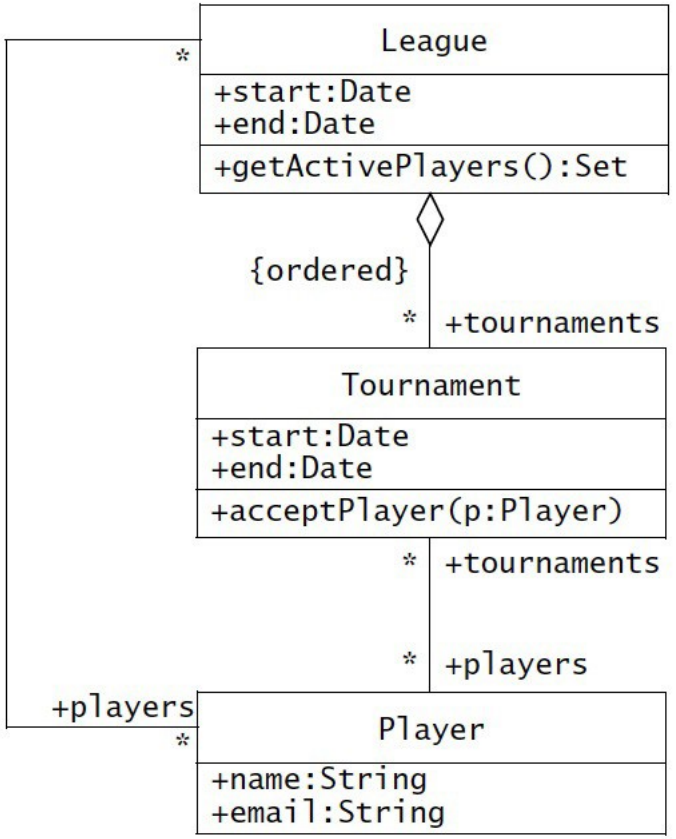

Figure 9-6 Associations among League, Tournament, and Player classes in ARENA.

OCL (cont.)

- Examples:
  - `Tournament` class

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

  - `Tournament` operations

```
context Tournament::acceptPlayer(p:Player)
        pre: !isPlayerAccepted(p)

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

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

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

OCL (cont.)

- Examples (cont.):
  - `Tournament` operations

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

context Tournament::removePlayer(p:Player)
        post: !isPlayerAccepted(p)

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

OCL Collections

- Constraints are based on navigation along associations
- Three different types of navigation:
  - local attribute
    - constraint uses an attribute local to the class
  - directly related class
    - constraint uses a single association to a directly related class
  - indirectly related class
    - constraint uses a series of associations to an indirectly related class

OCL Collections (cont.)

- Example of constraint using local attributes:
  - a tournament's duration must be under one week

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

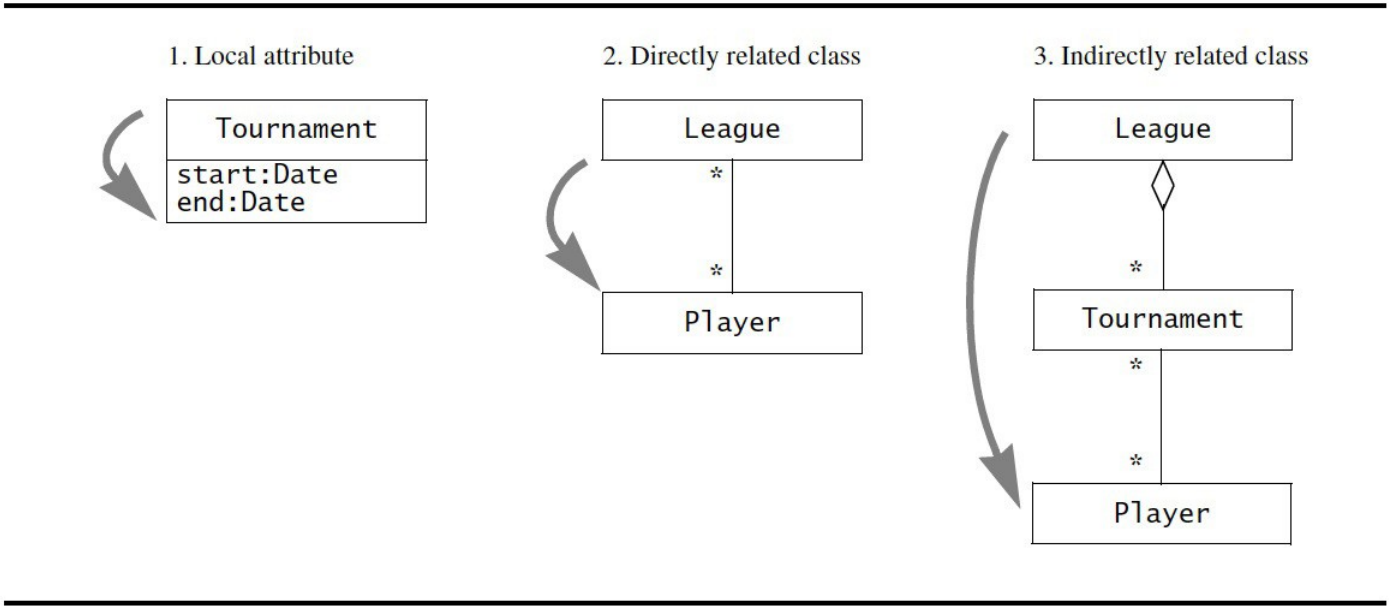

Figure 9-8 There are only three basic types of navigation. Any OCL constraint can be built using a combination of these three types.

OCL Collections (cont.)

- OCL collection: data type used in constraints
  - sets
    - used for single association
  - sequences:
    - used for single ordered association
  - bags
    - multisets used for indirectly related objects

- Collection operations

```
size                          union(collection)
includes(object)              intersection(collection)
select(expression)            aSet(collection)
```

OCL Collections (cont.)

- Collection operators
  - use dot operators to access attributes
  - use arrow operator to access collections

- Examples:
  - players can be accepted in a tournament only if they are already registered with the corresponding league

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

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

```
context Tournament::getActivePlayers:Set
        post: result = tournaments.players->asSet()
```

OCL Qualifiers

- Operations on collections allow for iteration
  - `forAll(variable|expression)`
    - true if `expression` is true for all elements in the collection
    - example:

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

  - `exists(variable|expression)`
    - true if there exists at least one element in the collection for which expression is true
    - example:

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

4.3.3 Interface Specification Activities

- Identifying missing attributes and operations
- Specifying types, signatures, visibility
- Specifying preconditions and postconditions
- Specifying invariants

Identifying Missing Attributes and Operations

- Attributes and operations during:
  - analysis
    - focus on application domain objects
  - detailed object design
    - focus on solution domain objects
    - fill in the blanks

- Strategy for identifying missing attributes and operations:
  - examine each subsystem
  - figure out what's missing to implement solution

Identifying Missing Attributes and Operations (cont.)

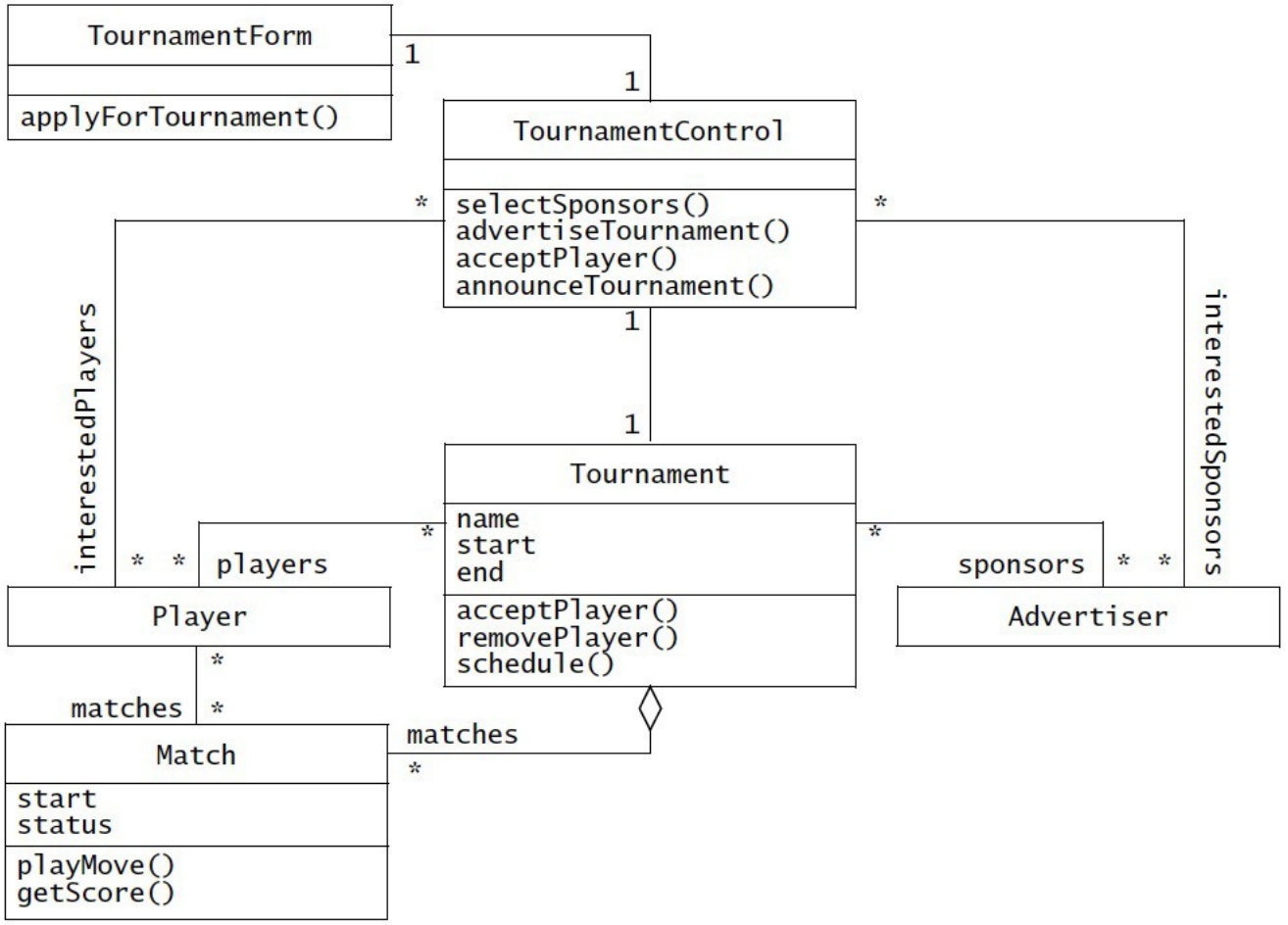

Figure 9-9 Analysis objects of ARENA identified during the analysis of AnnounceTournament use case (UML class diagram). Only selected information is shown for brevity.

Identifying Missing Attributes and Operations (cont.)

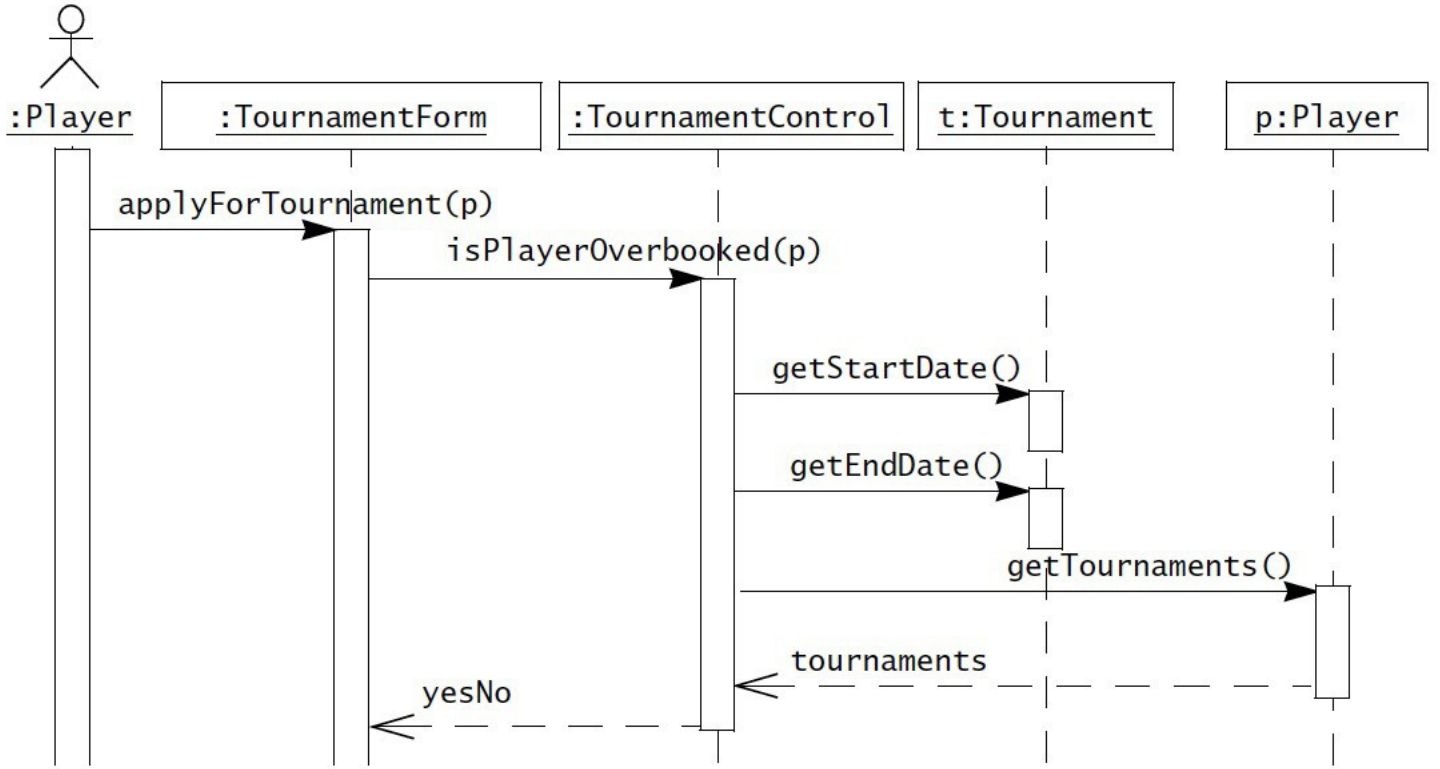

Figure 9-10 A sequence diagram for the `applyForTournament()` operation (UML sequence diagram). This sequence diagram leads to the identification of a new operation, `isPlayerOverbooked()` to ensure that players are not assigned to `Tournaments` that take place simultaneously.

Specifying Preconditions and Postconditions

- Characteristics of preconditions and postconditions
  - these are also known as *contracts*
  - they are associated with **class operations**
- Purpose of contracts
  - they are used to specify:
    - class behaviour
    - class boundary cases
  - they are required for each public operation of each class

Specifying Preconditions and Postconditions (cont.)

- Contracts are used to specify:
  - class behaviour
  - class boundary cases
- Required for each public operation of each class
- Form an agreement between class user and implementer
  - precondition:
    - it's the part of the contract that class user must respect
  - postcondition:
    - it's the part of the contract that the class implementer guarantees
    - the class user must fulfill their part of the contract

Specifying Invariants

- Characteristics of invariants
  - they are associated with a **class**
  - they identify the general properties of the class
  - they are a permanent contract that extends the operation-specific contracts
  - identifying invariants is a generalization process
    - similar to finding abstract classes

- Strategy for identifying invariants
  - start with the obvious class properties
  - extract the common properties from operation-specific contracts

Specifying Invariants (cont.)

- Heuristics
  - focus on the lifetime of the class
    - avoid constraints specific to operations
    - avoid constraints that hold only when an object is in certain states
  - identify special values for each attribute
    - e.g. zero or null, unique values
  - identify special cases for associations
    - special cases that are not specified by multiplicity

Specifying Invariants (cont.)

- Heuristics (cont.)
  - identify ordering among operations
  - use helper operations to compute complex conditions
  - avoid constraints that involve many association traversals
    - otherwise we get tighter coupling between unrelated classes

Recap

- What we learned:
  - to construct the detailed object model