

C. Examples of OCL in use, especially in "Object Design: Specifying Interfaces"

Note: This section is based heavily on Chapter 9 of the very useful text book “Object-Oriented Software Engineering” by Bruegge & Dutoit (Pearson, Prentice-Hall, 2nd edition).

C.1 Introduction	1
C.2 Constraints & OCL Basics	2
C.3 OCL Collections: Sets, Bags and Sequences	5
C.3.1 Examples of the OCL quantifiers “forAll” and “exists”	8
C.4 Interface Specification Activities-More OCL examples	9
C.4.1 Overview	9
C.4.2 Specifying preconditions & postconditions – More examples.....	10
C.4.3 Specifying invariants – More examples	11
C.4.4 Inheriting contracts - outline	12
C.5 Final note.....	12

C.1 Introduction

Bruegge & Dutoit present a number of example systems throughout their book, of which one (called ARENA) is used here to illustrate OCL and related matters. Overall, this system is concerned with gaming or sporting leagues and with the organisation of tournaments of matches between players within these leagues.

In terms of software, and particularly object design and implementation, three types of developer are identified. These are class implementor, responsible for realizing a given class, class extender, responsible for specializing a given class, and class user, responsible for realizing client classes of the given class.

Clearly, these types of developer will have different perspectives on the interface to the given class. Class implementors will require visibility on private attributes and operations (“-“ in UML), class extenders will require access to protected attributes and operations i.e. those that are accessible to the class in which they are defined and its descendant classes (“#” in UML), and class users will only require access to public attributes and operations (“+” in UML).

The following figures illustrate some of these ideas as well as introducing some of the ARENA classes.

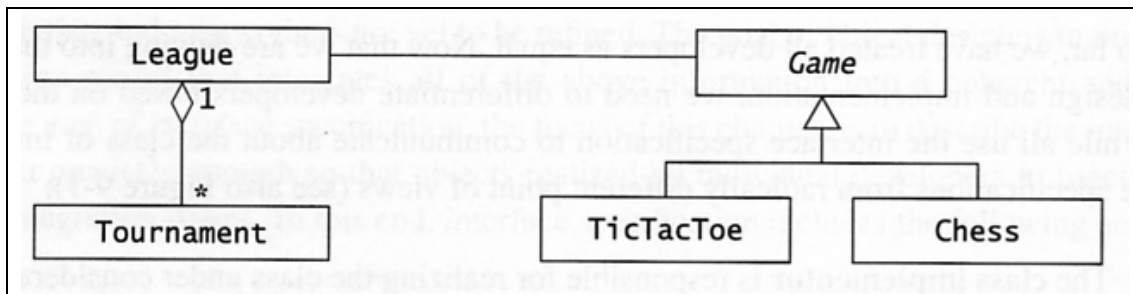
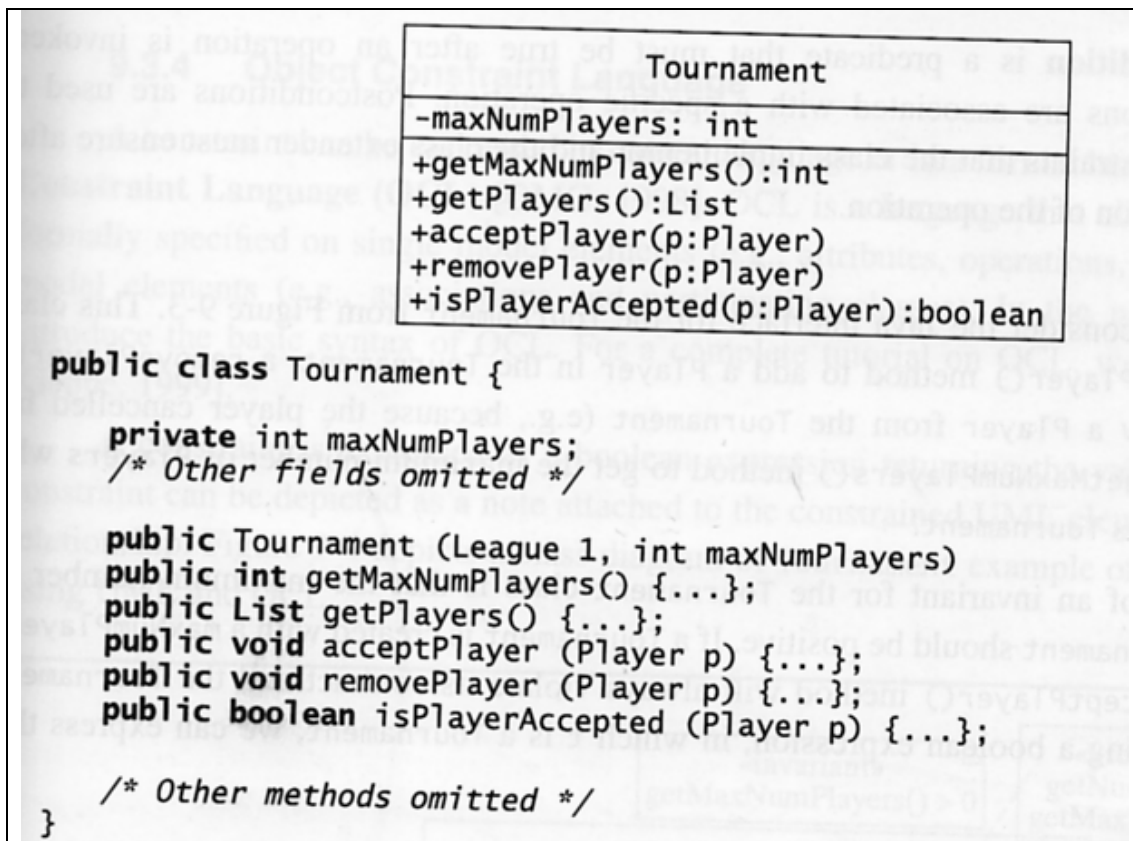
Figure 5C-1: ARENA *Game* abstract class with user classes and extender classes

Figure 5C-2: Declaration for the tournament class (UML class model and Java excerpts)

C.2 Constraints & OCL Basics

Typically, declarations such as those depicted in Figure 5C-2 are not sufficiently precise (e.g. the above does not preclude `maxNumPlayers` from being negative) and so the notion of contracts is introduced. These, as we have discussed before, are made up of

- operation preconditions: constraints that a *class user* must meet before calling the operation
- operation postconditions: constraints that a *class implementor* or a *class extender* must ensure after the operation call

- class invariants: Used to specify consistency constraints among class attributes and must always be true of all class instances.

The following figure illustrates how constraints (in OCL) may be attached as notes to a UML model, in this case a UML class diagram.

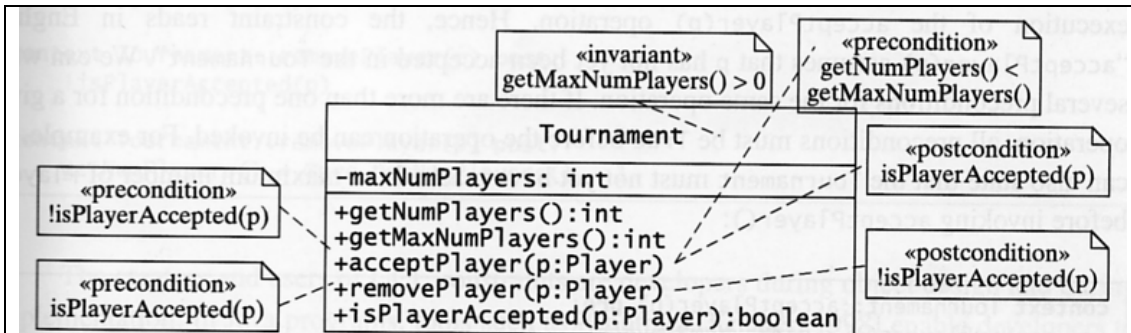


Figure 5C-3: Examples of invariants, preconditions and postconditions in OCL attached as notes to the UML model (UML class diagram)

As an alternative to displaying constraints on diagrams, which might become cluttered, it is possible to express them in textual form. Thus, as introduced previously, we have corresponding to Figure 5C-3 and noting that “self” does the same job as “this” does in Java,

context Tournament **inv**:

```
self.getMaxNumPlayers()>0
```

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

```
!isPlayerAccepted()
```

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

```
getNumPlayers()<getMaxNumPlayers()
```

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

```
isPlayerAccepted()
```

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

```
getNumPlayers()=getNumPlayers@pre()+1
```

Note: The above constraint does not appear in Figure 5.C-3. Also, Bruegge & Dutoit write “@pre.getNumPlayers ()” – either form is acceptable for CA422.

In similar fashion, the constraints on the operation “removePlayer()” may be written in textual form as

```

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

There are some tools available to allow OCL constraints to be documented systematically in the source code – the following figure is an illustration:

```

/** A Tournament is a series of Matches among a set of Players
 * which ends with a single winner. The Game and TournamentStyle of a
 * TournamentStyle is determined by the League in which the Tournament is
 * played.
 */
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;

    /* Constructors omitted */

    /** 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) {...}

    /* Other methods omitted */
}
  
```

Figure 5C-4: Method declarations for the Tournament class annotated with preconditions, postconditions, and invariants (Java, constraints using Javadoc style tags)

C.3 OCL Collections: Sets, Bags and Sequences

The previous section illustrated the case of specifying constraints on a single class. However, in general constraints may involve an arbitrary number of classes and attributes. This is perhaps the least obvious part of OCL and Bruegge & Dutoit provide the following example to explain the ideas involved. In particular, the following three constraints are to be specified:

1. A Tournament's planned duration must be under one week. [*Involves a single class – local attribute*]
2. Players can be accepted in a Tournament only if they are already registered with the corresponding League. [*Involves three classes and their associations – directly related*]
3. The number of active Players in a League are those that have taken part in at least one Tournament of the league. [*Involves an indirectly related class*]

The following figure depicts the associations among the League, Tournament and Player classes:

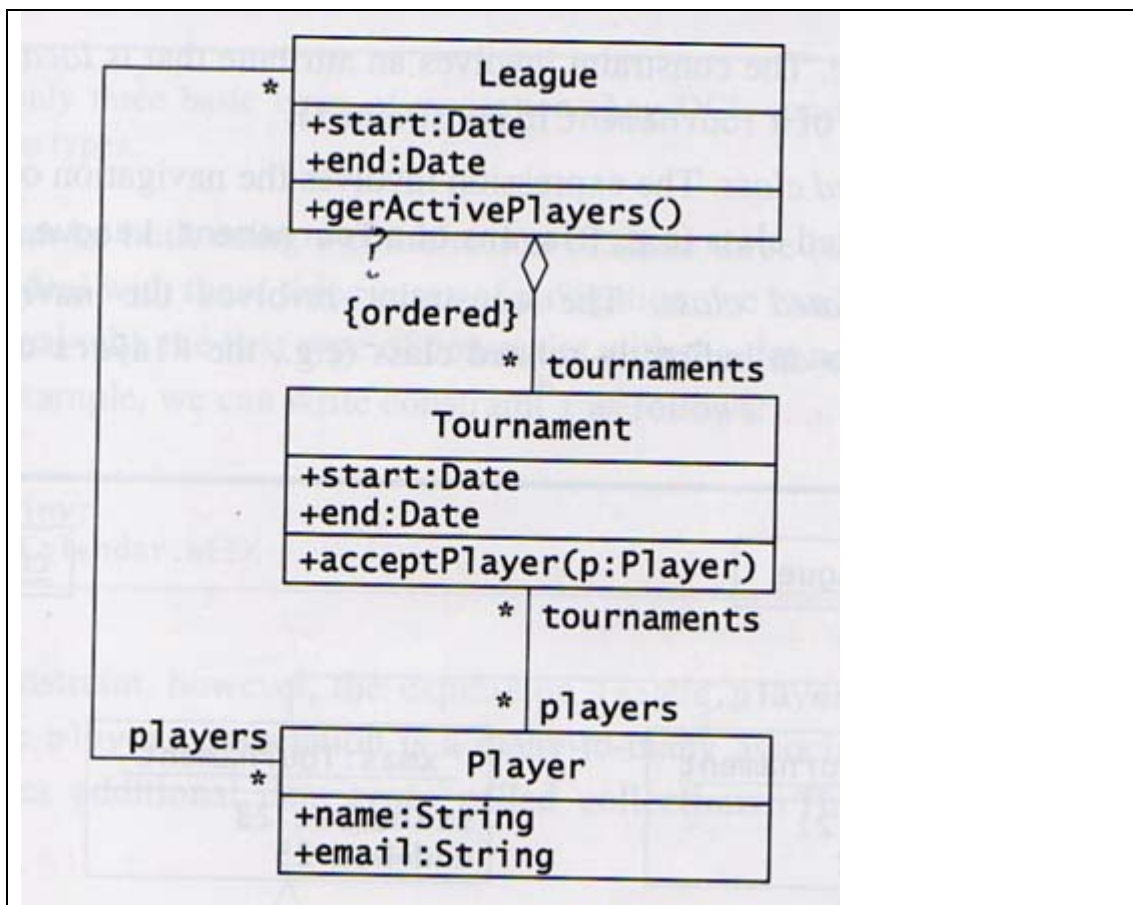
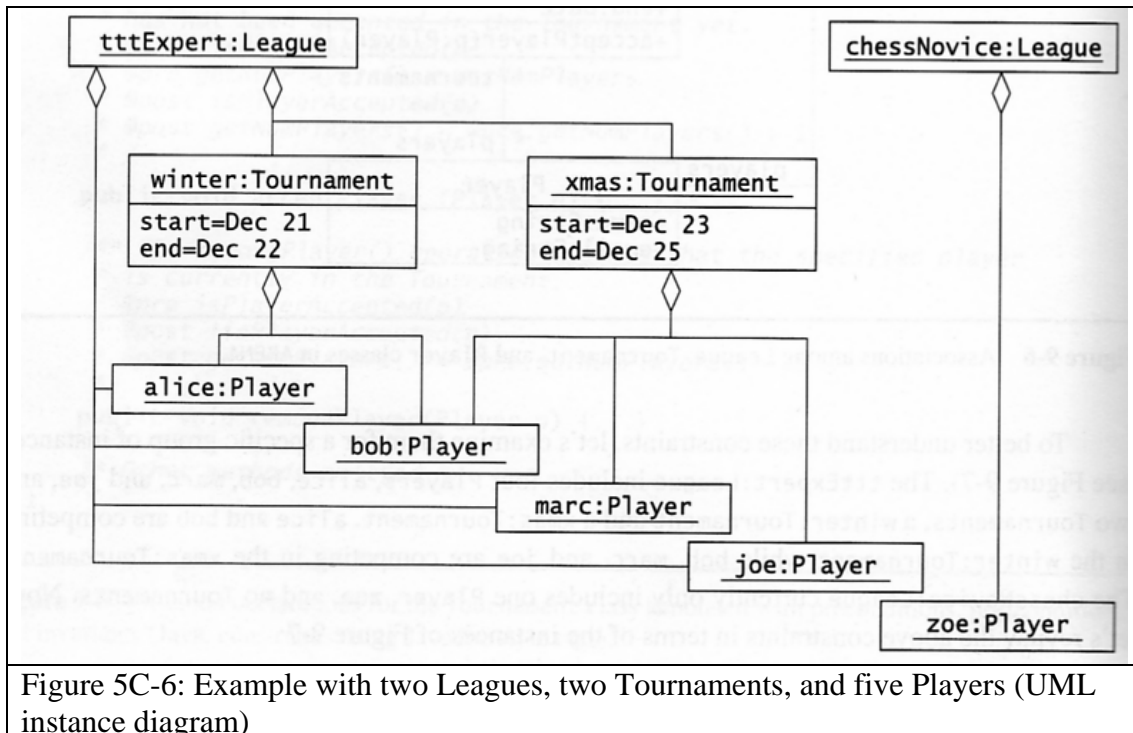


Figure 5C-5: ARENA – Associations among League, Tournament and Player classes
To further clarify understanding of the constraints, Figure 5C-6 depicts some specific instances. Thus,

tttExpert:League includes the four players **alice:Player**, **bob:Player**, **marc:Player**, and **joe:Player**, and two tournaments **winter:Tournament** and **xmas:Tournament**.

Players **alice** and **bob** are competing in the **winter:Tournament** while **bob**, **marc** and **joe** are competing in the **xmas:Tournament**.

chessNovice:League includes just **zoe:Player** and has no tournaments.



Review of constraints 1-3 for Figure 5C-6:

1. Both the **winter:Tournament** (2 days) and **xmas:Tournament** (3 days) last under a week so that this constraint is satisfied. [*Involves a single class – local attribute*]
2. All Players of the **winter:Tournament** and the **xmas:Tournament** are associated with **tttExpert:League**; also, **zoe:Player**, who is not in this League, does not take part in either tournament. Therefore this constraint is satisfied. [*Involves three classes and their associations – directly related*]
3. **tttExpert:League** has four active players whereas **chessNovice:League** has none as **zoe:Player** does not take part in a Tournament. [*Involves an indirectly related class*]

The procedure in all cases is to start with the class of interest and to navigate to one or more classes in the model. In general, there are 3 cases (Figure 5.C-7):

A• Local attribute: Constraint involves an attribute local to the class of interest (e.g. duration of a Tournament in constraint 1)

B• Directly related class: Expression involves the navigation of a single association to a directly related class (e.g. Players of a Tournament, League of a Tournament, Players of a League)

C• Indirectly related class: Involves the navigation of a series of associations to an indirectly related class (e.g. Players of all Tournaments of a League).

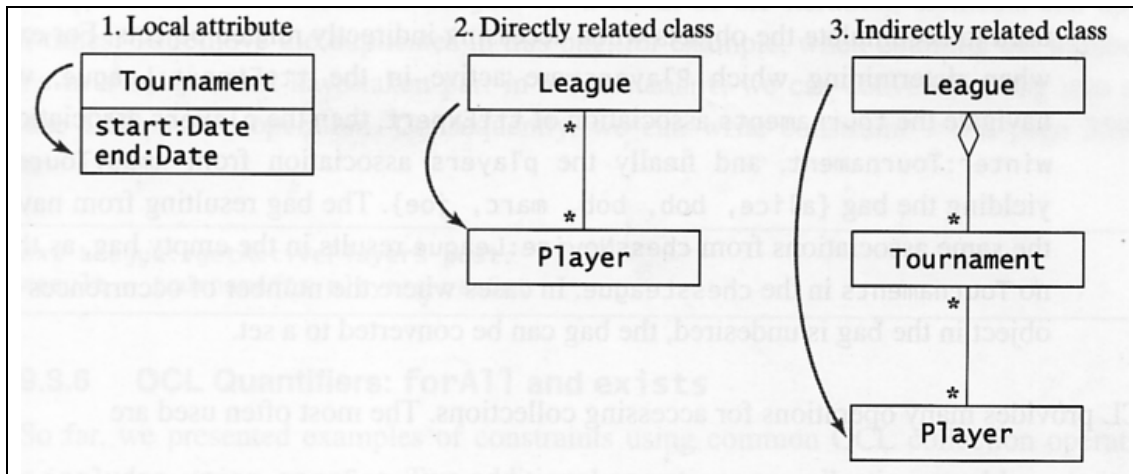


Figure 5C-7: There are only three basic types of navigation. Any OCL constraint can be built using a combination of the three basic types

All constraints can be built using a combination of **A**, **B** and **C**.

A: This is most familiar; for example, for constraint 1 we can write, using the dot notation,:

context Tournament **inv:**

```
self.end-self.start<= Calendar.week
```

In fact, “self” can be omitted if the context is clear.

Note: Cases B and C entail the use of OCL collections (set, sequence, bag) which we review before going on to expressing constraints 2 and 3.

OCL sets are used when navigating a single association. For example, navigating the players association of the winter:Tournament yields {alice, bob} and navigating the players association from tttExpert:League gives {alice, bob, marc, joe}. The only complication to recall is that navigating an association of multiplicity 1 yields an object directly, for example navigating the league association from winter:Tournament yields tttExpert:League (as opposed to {tttExpert:League}).

OCL sequences are used when navigating a single ordered association. For example, the association between League and Tournament is ordered so that navigating the tournaments association from tttExpert:League yields [winter:Tournament, xmas:tournament], with the index of the first member being 1 and of the second 2.

OCL bags are used to accumulate objects when accessing indirectly related objects:

For example, when determining which Players are active in the `tttExpert:League` we first navigate the `tournaments` association of `tttExpert`, then the `players` association from firstly `winter:Tournament` and from secondly `xmas:Tournament`. This results in the bag `{alice, bob, bob, marc, joe}`.

As indicated earlier, OCL provides many operations for accessing collections (e.g. `size`, `includes(object)`, `select(expression)`, `union(collection)`, `intersection(collection)`, and `asSet(collection)`).

Finally, recall that OCL uses the dot notation for accessing attributes and the `->` operator for accessing collections.

B: We can now see that constraint 2 may be written as

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

C: Finally, we can see that constraint 3 may be written as

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

Remark: The collection `tournaments.players` (in the context of `League`) is a bag (as we saw earlier) so we need to convert this to a set for the required result.

C.3.1 Examples of the OCL quantifiers “forAll” and “exists”

Note: Look ahead to Figures 5C-8 etc for the class `Match` mentioned in the following examples.

(1) “Ensure that all Matches in a Tournament occur within the Tournament’s time frame” can be expressed as

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

Note: `start` in `after(start)` and `before(start)` is the Tournament start.

(2) “Ensure that each Tournament conducts at least one Match on the first day of the Tournament” can be expressed as

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

C.4 Interface Specification Activities-More OCL examples

C.4.1 Overview

In their book, Bruegge & Dutoit address the activities involved in “interface specification”, namely (a) Identifying missing attributes and operations (b) Specifying type signatures and visibility, (c) Specifying preconditions and postconditions, (d) Specifying invariants and (e) Inheriting contracts.

Due to shortage of time we focus mainly on (c) and (d). However, to provide a context and material for examples, Figures 5.C-8 to Figure 5.C-10 do give an insight of what is involved in activities (a) and (b). Figure 5.C-8 is a class model after analysis and is incomplete. After some detailed design work, including development of a sequence diagram (Figure 5.C-9), a more complete class model (Figure 5.C-10) is derived; in particular, this includes a previously omitted operation (`isPlayerOverbooked()`) and type, signature and visibility information.

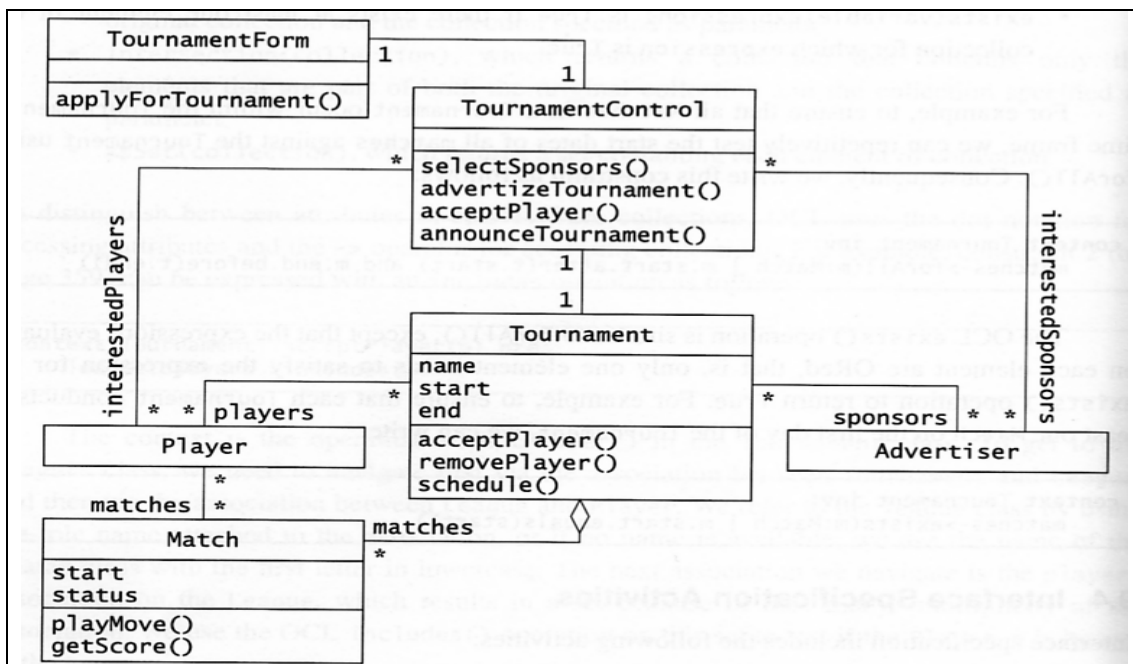


Figure 5C-8: Analysis objects of ARENA (identified during analysis) – UML class diagram. Only selected information is shown for brevity.

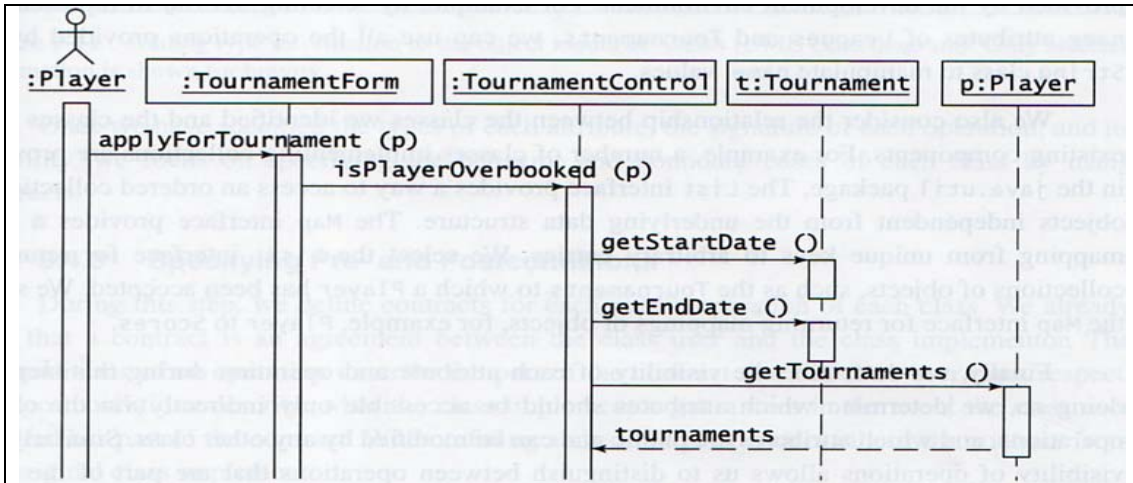


Figure 5C-9: A sequence diagram for the applyForTournament() operation (UML sequence diagram). The sequence diagram leads to the identification of a new operation, isPlayerOverbooked() to ensure that players are not assigned to Tournaments that take place simultaneously.

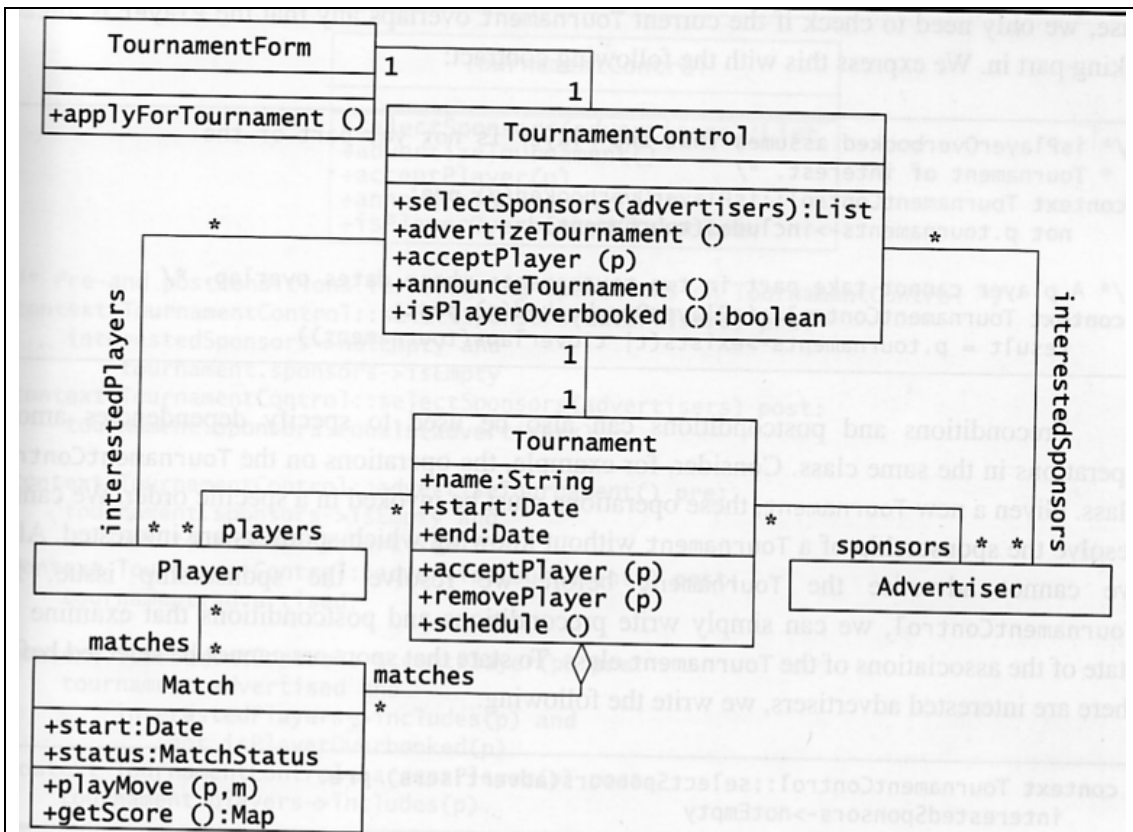


Figure 5C-10: Adding type information (see Figure 5C-8) to the object model of ARENA. Only selected information is shown for brevity.

C.4.2 Specifying preconditions & postconditions – More examples

Referring to the class TournamentControl in Figure 5C.10, the following are identified:

- (a) Operation isPlayerOverbooked

```
/* isPlayerOverbooked assumes that the player is not yet part of the Tournament of interest.*/
```

```
context TournamentControl::isPlayerOverbooked(p) pre:
```

```
    not p.tournaments->includes(tournament)
```

```
/* A Player cannot take part in 2 tournaments whose dates overlap*/
```

```
context TournamentControl::isPlayerOverbooked(p) post:
```

```
    result=p.tournaments->exists(t|t.overlaps(tournament))
```

(b) Specifying dependencies among operations in the same class

Given a new Tournament the operations on the TournamentControl class must be invoked in a certain order. Thus,

- Cannot resolve sponsorship if interested sponsors are not known:

```
context TournamentControl::selectSponsors(advertisers) pre:
```

```
    interestedSponsors->notEmpty
```

- Cannot select new sponsors once a sponsor has been selected:

```
context TournamentControl::selectSponsors(advertisers) pre:
```

```
    tournament.sponsors->isEmpty
```

- Ensure TournamentControl::selectSponsors is invoked once only:

```
context TournamentControl::selectSponsors(advertisers) post:
```

```
    tournament.sponsors.equals(advertisers)
```

C.4.3 Specifying invariants – More examples

The examples following also refer to classes in Figure 5C.10. In general, class invariants are more difficult than operation preconditions and postconditions which are fairly intuitive.

(a) An obvious invariant – all its Matches must occur in a Tournament's time frame

```
context Tournament inv:
```

```
    matches->forAll(m|m.start.after(start) and m.start.before(end))
```

(b) Less obvious – no Player can take part in 2 or more tournaments that overlap

```
context TournamentControl inv:
```

```
    tournament.players->forAll(p|
```

```
        p.tournaments->forAll(t|
```

```
            t<>tournament implies not t.overlap(tournament)))
```

C.4.4 Inheriting contracts - outline

Preconditions: A method of a subclass is allowed weaken the precondition of the method overrides i.e. it can handle more cases.

Postconditions: Methods must ensure the same or stricter postconditions than their ancestors.

Ex: Suppose it is proposed to implement a Set by inheriting from a List. The postcondition of List.add is that the size increases by one. However Set.add could not comply with this. Therefore, the proposal should be rejected.

Invariants: A subclass must respect all invariants of its superclasses though it can strengthen inherited invariants.

C.5 Final note

This section, 5C, has focussed on the use of contracts written in OCL during the object design process. It is possible to introduce constraints at an earlier stage, of course, but this will depend on the individual project (state of initial knowledge, degree of criticality, etc).

Constraints on requirements can be quite useful in developing system level tests also – again feasibility depends on the particular project.