

ARENA case study

4.6	ARENA Case Study	153
4.6.1	Initial Problem Statement	153
4.6.2	Identifying Actors and Scenarios	155
4.6.3	Identifying Use Cases	159
4.6.4	Refining Use Cases and Identifying Relationships	161
4.6.5	Identifying Nonfunctional Requirements	166
4.6.6	Lessons Learned	168
4.7	Further Readings	168
5.6	ARENA Case Study	206
5.6.1	Identifying Entity Objects	206
5.6.2	Identifying Boundary Objects	211
5.6.3	Identifying Control Objects	212
5.6.4	Modeling Interactions Among Objects	212
5.6.5	Reviewing and Consolidating the Analysis Model	213
5.6.6	Lessons Learned	217
5.7	Further Readings	218
7.6	ARENA Case Study	290
7.6.1	Identifying Design Goals	290
7.6.2	Identifying Subsystems	291
7.6.3	Mapping Subsystems to Processors and Components	292
7.6.4	Identifying and Storing Persistent Data	294
7.6.5	Providing Access Control	295
7.6.6	Designing the Global Control Flow	296
7.6.7	Identifying Services	297
7.6.8	Identifying Boundary Conditions	299
7.6.9	Lessons Learned	302
8.6	ARENA Case Study	341
8.6.1	Applying the Abstract Factory Design Pattern	341
8.6.2	Applying the Command Design Pattern	342
8.6.3	Applying the Observer Design Pattern	342
8.6.4	Lessons Learned	344

9.6	ARENA Case Study	382
9.6.1	Identifying Missing Operations in <i>TournamentStyle</i> and <i>Round</i>	383
9.6.2	Specifying the <i>TournamentStyle</i> and <i>Round</i> Contracts	384
9.6.3	Specifying the <i>KnockOutStyle</i> and <i>KnockOutRound</i> Contracts	386
9.6.4	Lessons Learned	387
10.6	ARENA Case Study	424
10.6.1	ARENA Statistics	424
10.6.2	Mapping Associations to Collections	426
10.6.3	Mapping Contracts to Exceptions	428
10.6.4	Mapping the Object Model to a Database Schema	430
10.6.5	Lessons Learned	431

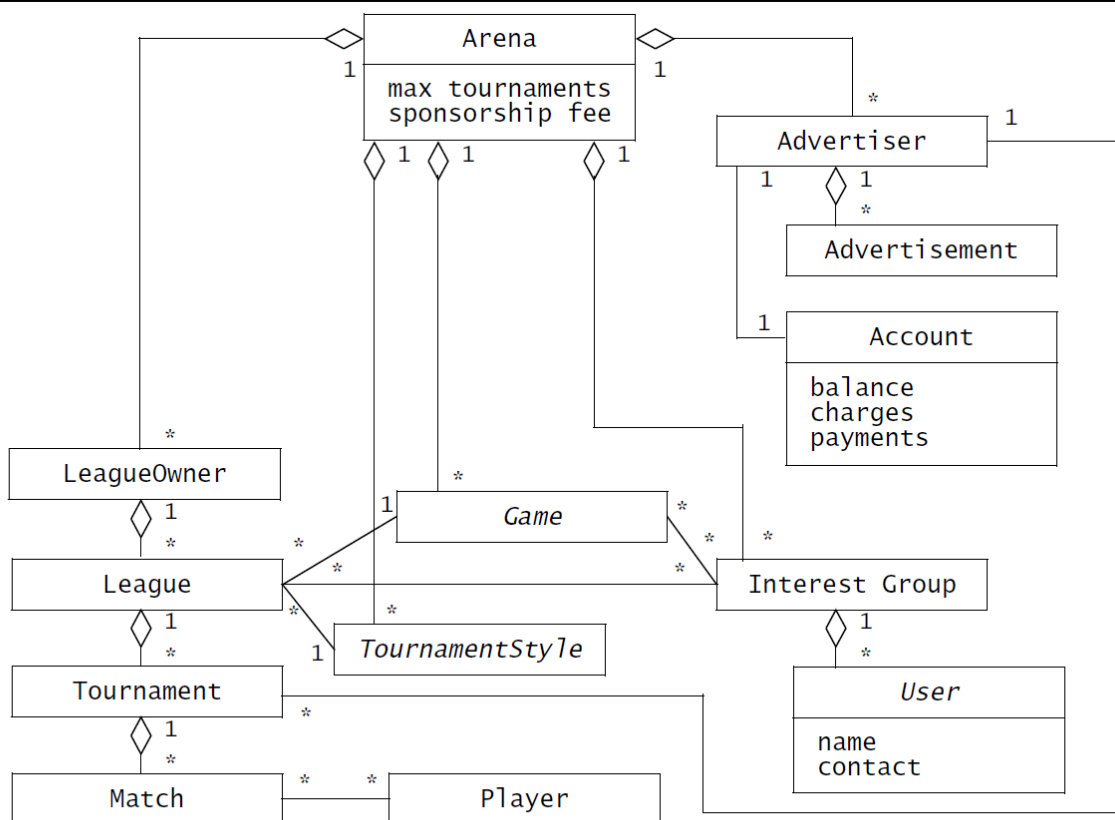


Figure 5-29 Entity objects identified after analyzing the AnnounceTournament use case.

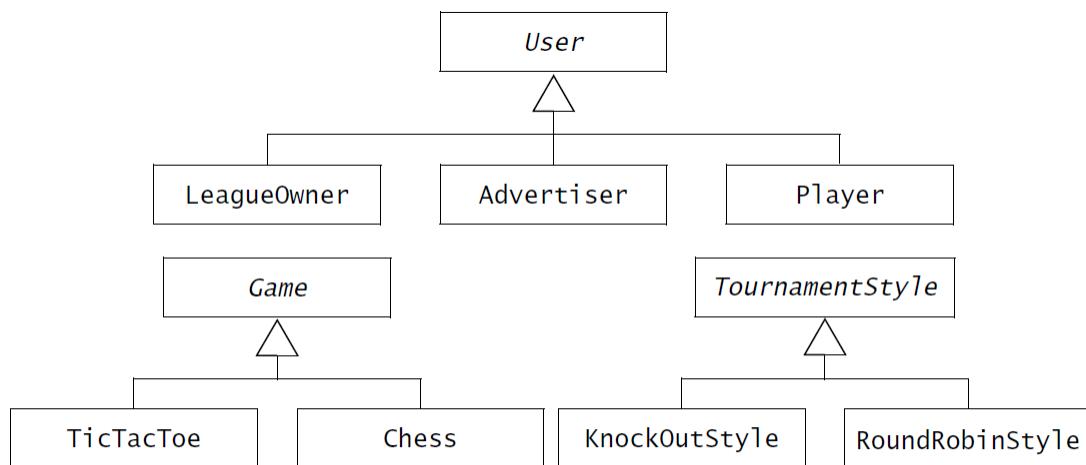


Figure 5-30 Inheritance hierarchy among entity objects of the AnnounceTournament use case.

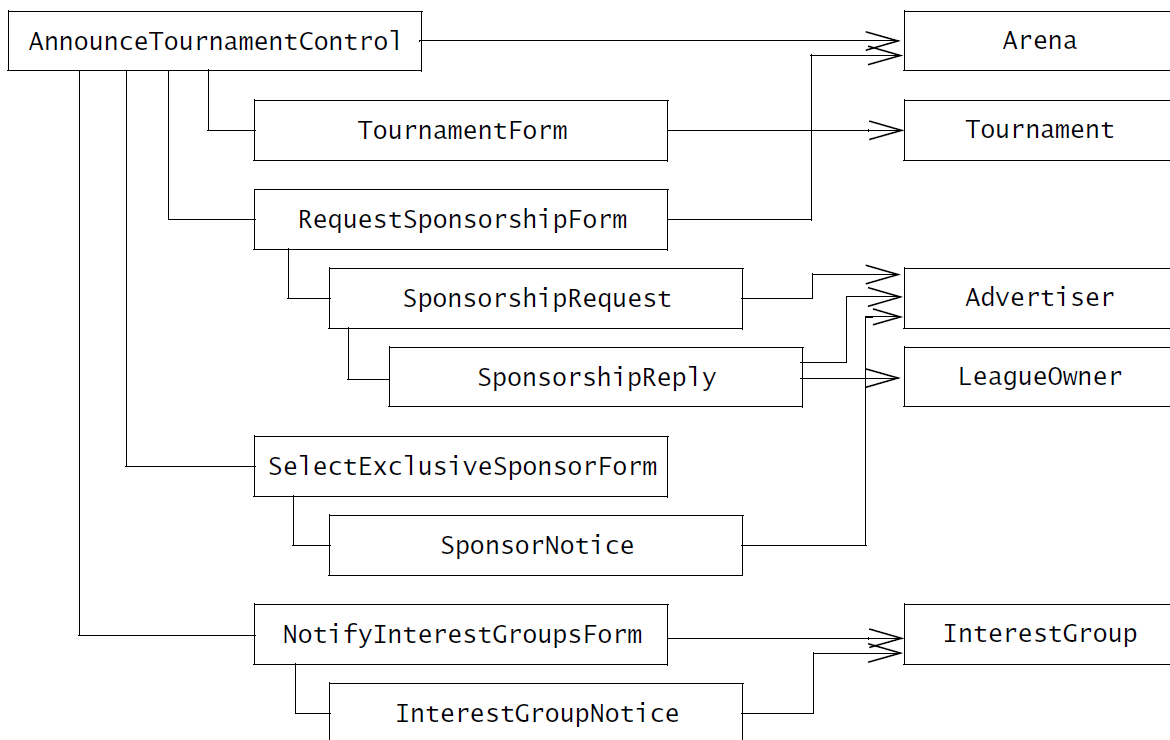


Figure 5-31 Associations among boundary, control, and selected entity objects participating in the AnnounceTournament use case.

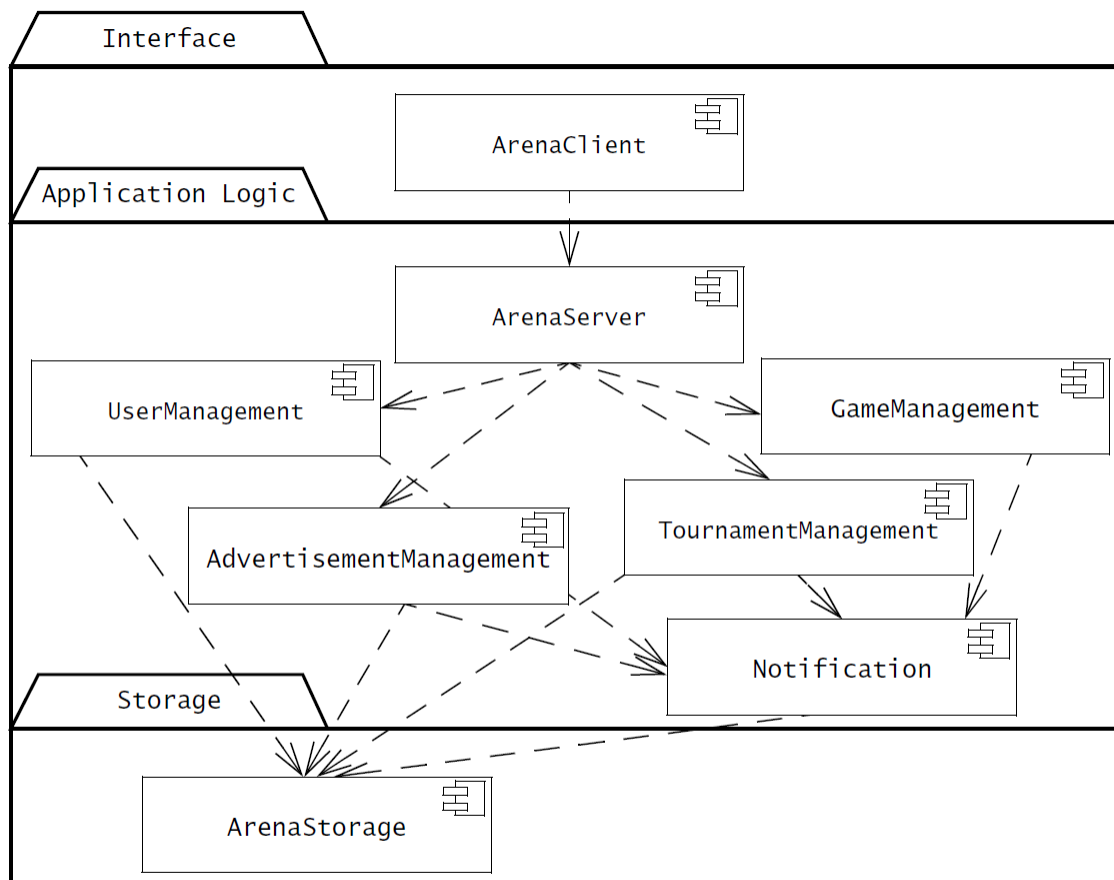


Figure 7-19 ARENA subsystem decomposition, game organization part (UML component diagram, layers shown as UML packages).

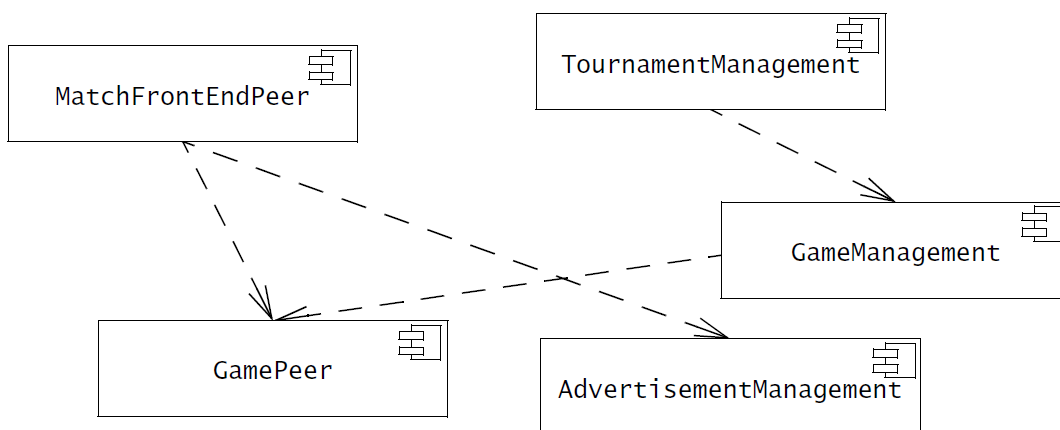


Figure 7-20 ARENA subsystem decomposition, game playing part (UML component diagram).

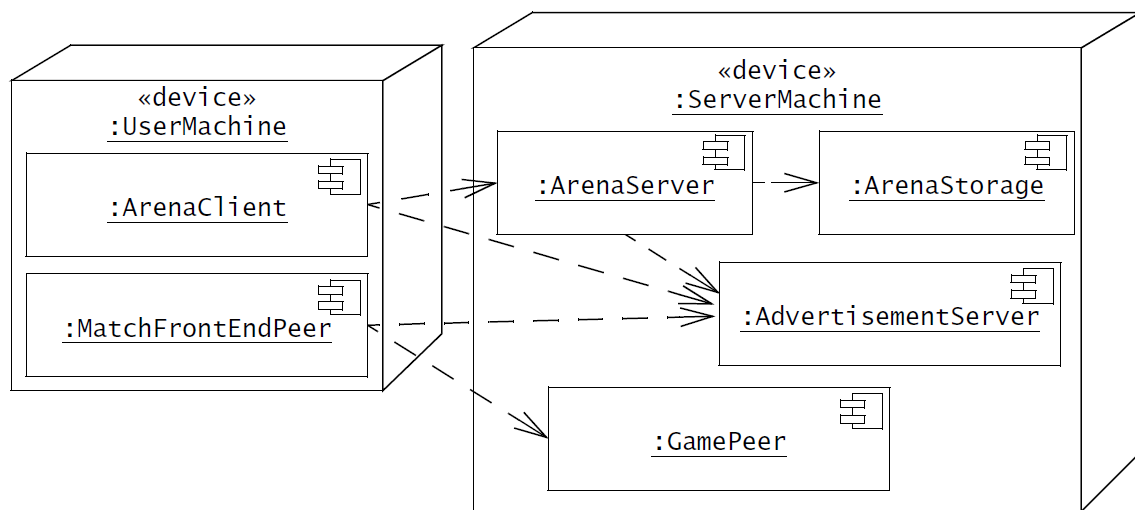


Figure 7-21 ARENA hardware/software mapping (UML deployment diagram). Note that each run-time component may support several subsystems.

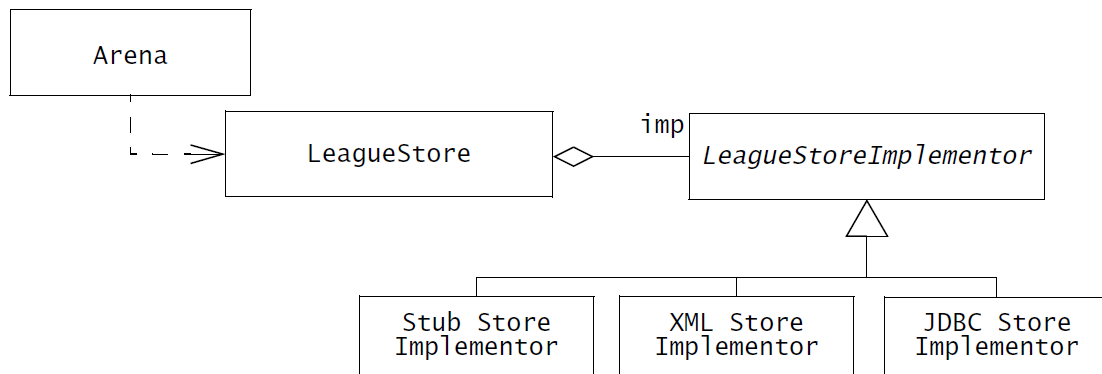


Figure 8-7 Applying the Bridge design pattern for abstracting database vendors (UML class diagram).

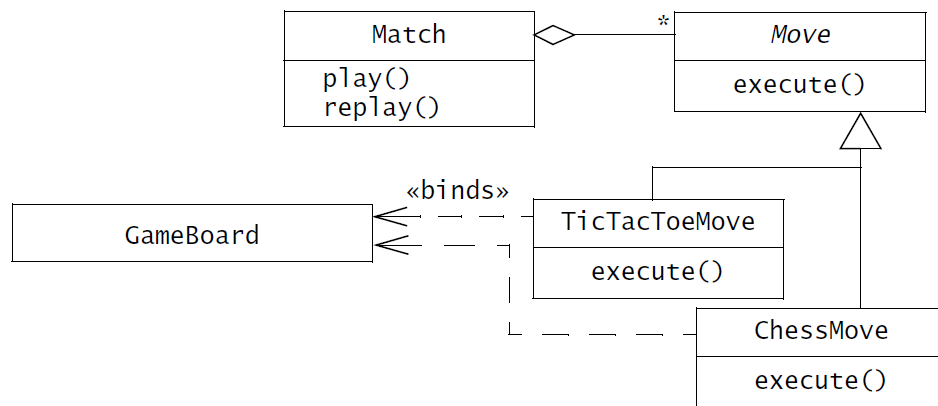


Figure 8-13 Applying the Command design pattern to Matches in ARENA (UML class diagram).

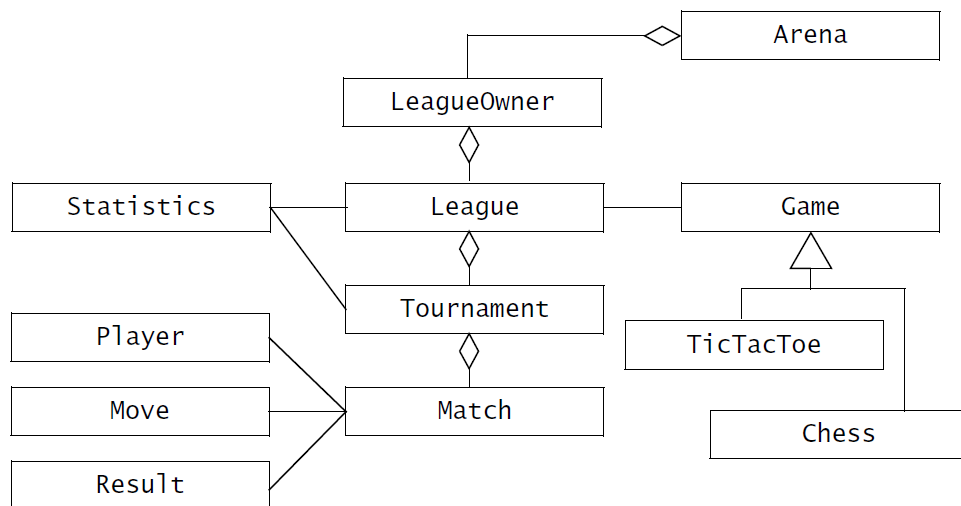


Figure 8-19 ARENA analysis objects related to Game independence (UML class diagram).

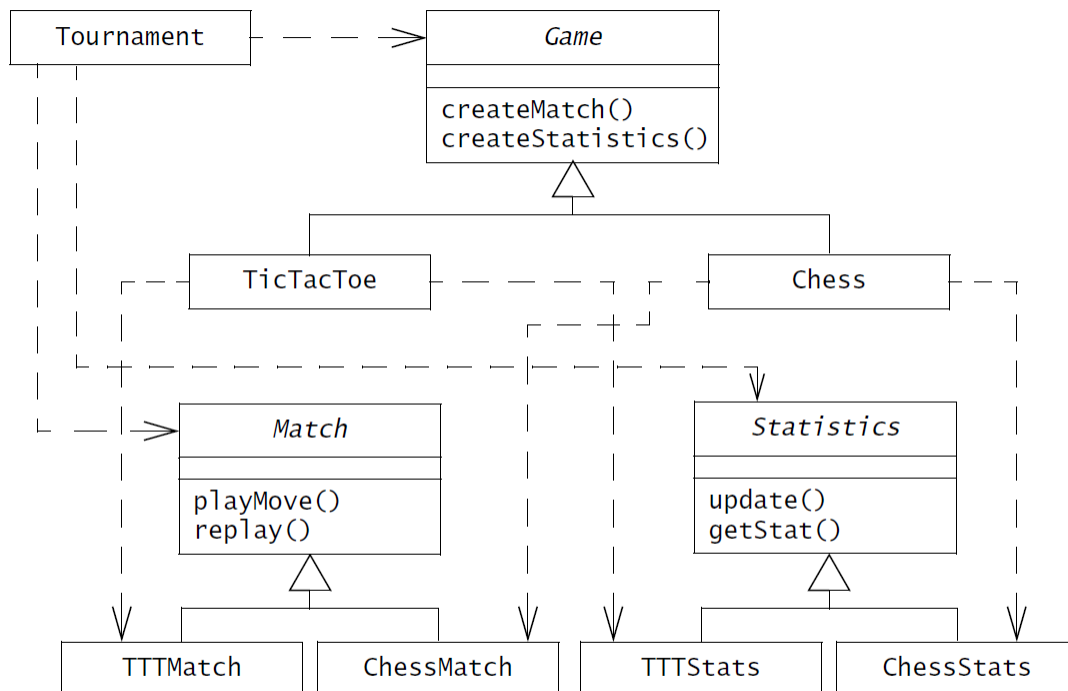


Figure 8-20 Applying the Abstract Factory design pattern to Games (UML class diagram).

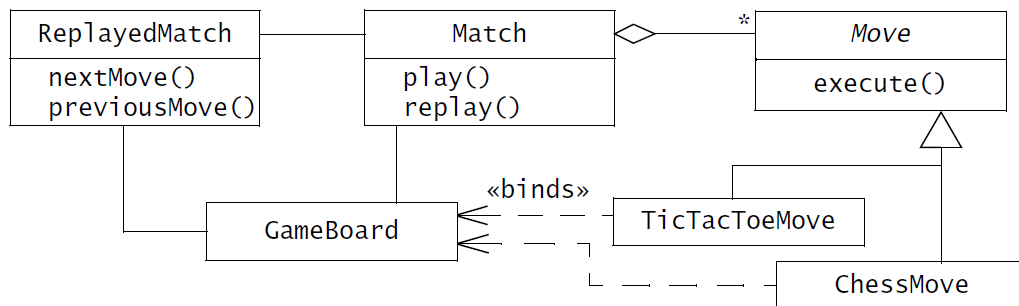


Figure 8-21 Applying the Command design pattern to Matches and ReplayedMatches in ARENA (UML class diagram).

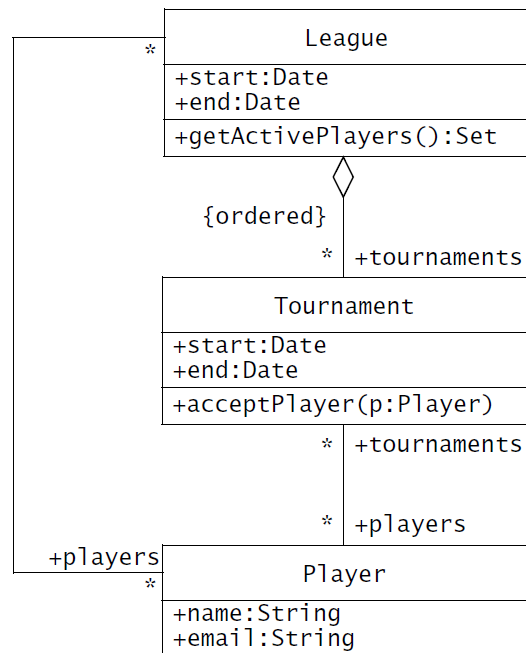


Figure 9-6 Associations among League, Tournament, and Player classes in ARENA.

```

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

```

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

```

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

```

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

```

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

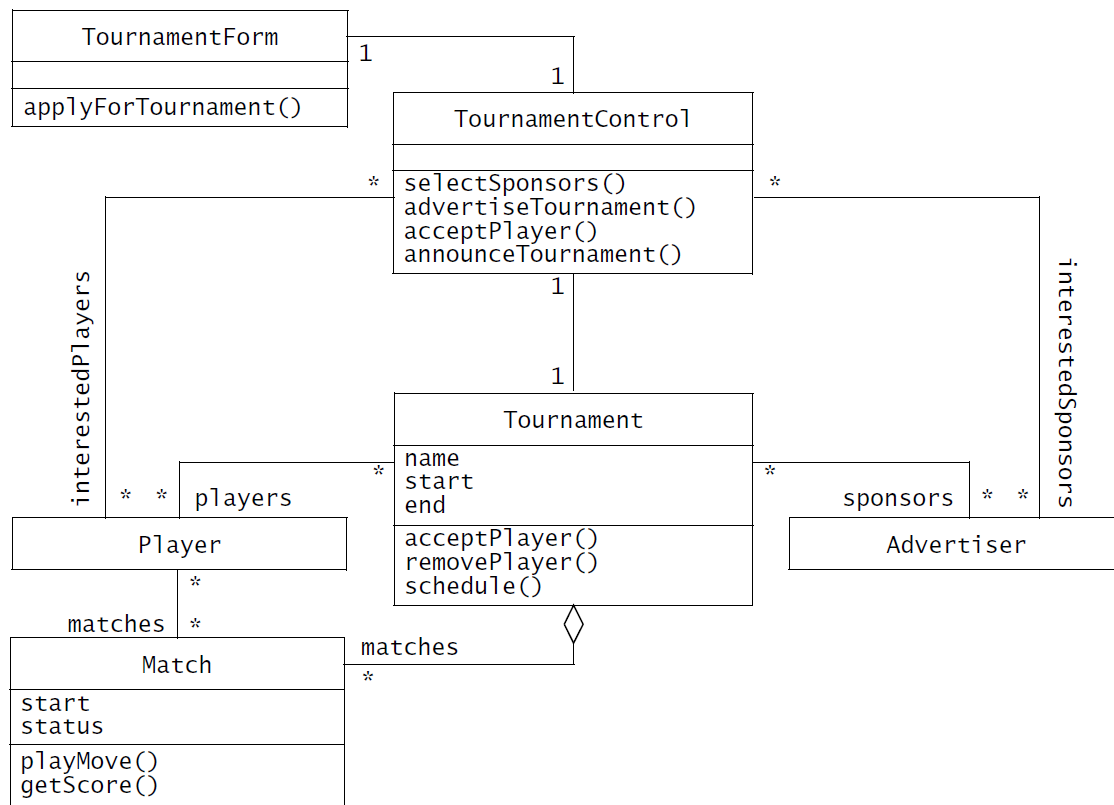



Figure 9-9 Analysis objects of ARENA identified during the analysis of AnnounceTournament use case (UML class diagram). Only selected information is shown for brevity.

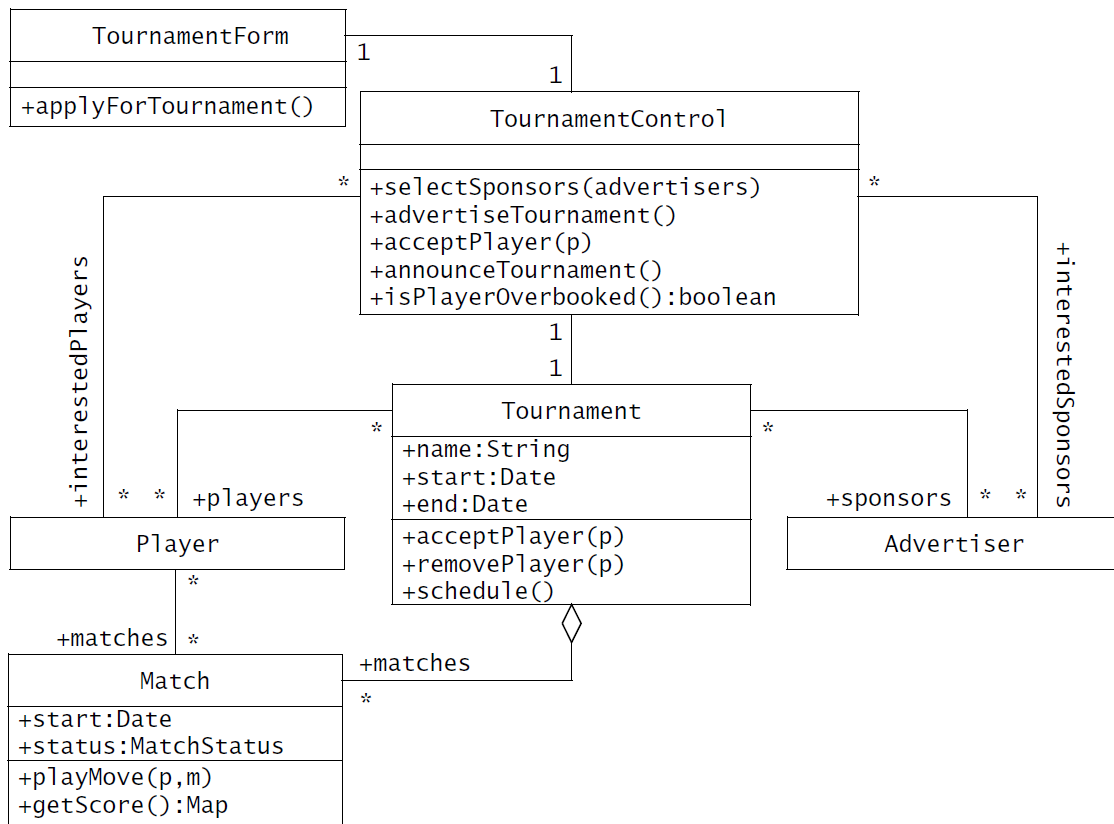


Figure 9-11 Adding type information to the object model of ARENA (UML class diagram). Only selected information is shown for brevity.

```

/* isPlayerOverbooked assumes that the Player is not yet part of the
 * Tournament of interest. */
context TournamentControl::isPlayerOverbooked(p) pre:
    not p.tournaments->includes(self.tournament)

/* A player cannot take part in two tournaments whose dates overlap. */
context TournamentControl::isPlayerOverbooked(p) post:
    result = p.tournaments->exists(t| t.overlaps(self.tournament))
  
```

```

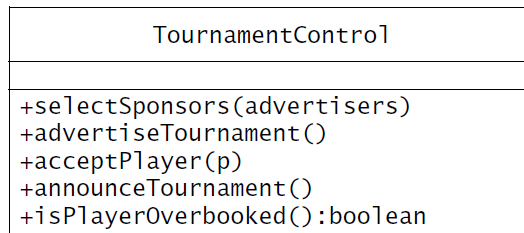
context TournamentControl::selectSponsors(advertisers) pre:
    interestedSponsors->notEmpty()
  
```

```

context TournamentControl::selectSponsors(advertisers) pre:
    tournament.sponsors->isEmpty()
  
```

```

context TournamentControl::selectSponsors(advertisers) post:
    tournament.sponsors.equals(advertisers)
  
```



/* Pre-and postconditions 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

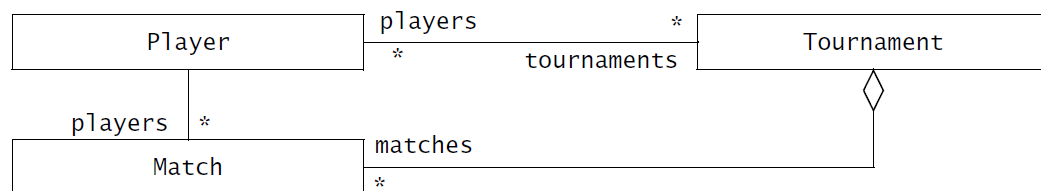
context TournamentControl::advertiseTournament() **post:**
 tournament.advertised

context TournamentControl::acceptPlayer(p) **pre:**
 tournament.advertised and
 interestedPlayers->includes(p) and
 not isPlayerOverbooked(p)

context TournamentControl::acceptPlayer(p) **post:**
 tournament.players->includes(p)

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

context TournamentControl **inv:**
 tournament.players->forAll(p|
 p.tournaments->forAll(t|
 t <> tournament implies not t.overlap(tournament)))



/* 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)

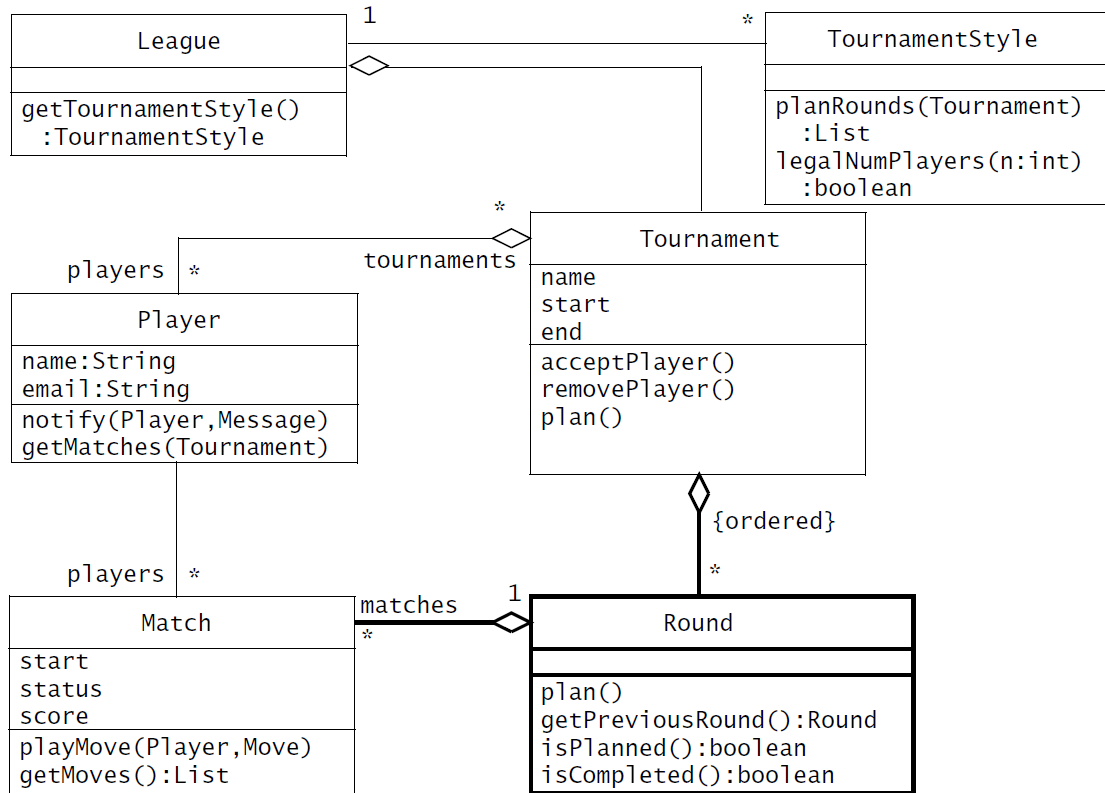


Figure 9-17 New *Round* class and changes in the *TournamentStyle*, *Tournament*, and *Round* APIs (UML class diagram). Thick lines indicate changes.

```

/* Only tournaments without rounds and with the right number of players
 * can be planned.*/
context TournamentStyle::planRounds(t:Tournament) pre:
  t <> null and t.rounds = null and legalNumPlayers(t.players->size)
  
```

```

/* All players are assigned to at least one match */
context TournamentStyle::planRounds(t:Tournament) post:
  t.getPlayers()->forall(p|
    p.getMatches(tournament)->notEmpty()

context TournamentStyle::planRounds(t:Tournament) post:
  result->forall(r1,r2| r1<>r2 implies
    r1.getEndDate().before(r2.getStartDate()) or
    r1.getStartDate().after(r2.getEndDate())
  
```

```
/* A player cannot be assigned to more than one match per round */
context Round inv:
  matches->forall(m1:Match|
    m1.players->forall(p:Player|
      p.matches->forall(m2:Match| m1 <> m2 implies m1.round <> m2.round)))
```

```
/* Invoking plan() on a Round whose previous Round is completed results
 * in a planned Round. */
context Round.plan() post:
  @pre.getPreviousRound().isCompleted() implies isPlanned()

/* A round is planned if all matches have players assigned to them. */
context Round.isPlanned() post:
  result implies
    matches->forall(m|
      m.players->size = tournament.league.game.numPlayersPerMatch)

/* A round is completed if all of its matches are completed. */
context Round.isCompleted() post:
  result implies
    matches->forall(m| m.winner <> null)
```

Figure 9-18 Contracts of the *Round* class.

```

/* The number of players should be a power of 2. */
context KnockOutStyle::legalNumPlayers(n:int) post:
    result = (floor(log(n)/log(2)) = (log(n)/log(2)))

/* The number of matches in a round is 1 for the last round. Otherwise,
 * the number of matches in a round is exactly twice the number of matches
 * in the subsequent round.
 */
context KnockOutStyle::planRounds(t:Tournament) post:
    result->forAll(index:Integer|
        if (index = result->size) then
            result->at(index).matches->size = 1
        else
            result->at(index).matches->size =
                (2*result->at(index+1).matches->size))
        endif)

/* A player can play in a round only if it is the first round or if it is the
 * winner of a previous round.
 */
context KnockOutRound inv:
    previousRound = null or
        matches.players->forAll(p|
            round.previousRound.matches->exists(m| m.winner = p))

/* If the previous round is not completed, this round cannot be planned. */
context KnockOutRound::plan() post:
    not self@pre.getPreviousRound().isCompleted() implies not isPlanned()

```

Figure 9-19 Refining contracts of the *Round* class.