## 1.6 ARENA Case Study

In each chapter, we introduce concepts and activities using increasingly complex examples, starting with toy examples from the classroom and moving to actual examples from project courses or from real systems. Moreover, to put the activities of each chapter in the context of the overall software engineering project, we also use a single, comprehensive case study throughout the book, describing the development of a system called ARENA.

ARENA is a multi-user, Web-based system for organizing and conducting tournaments. ARENA is game independent in the sense that organizers can adapt a new game to the ARENA game interface, upload it to the ARENA server, and immediately announce and conduct tournaments with players and spectators located anywhere on the Internet. Organizers can also define new tournament styles, describing how players are mapped to a set of matches and how to compute an overall ranking of players by adding up their victories and losses (hence, figuring out who won the tournament). To recoup their operational costs, organizers can also invite potential sponsors to display advertisement banners during games.

In the section entitled "ARENA Case Study" located at the end of each chapter, we discuss issues, design decisions, and trade-offs specific to the chapter in the context of ARENA. We also relate these issues to the parts of the case study presented in previous chapters, thereby emphasizing inter-chapter dependencies. For example:

- In Chapter 4, *Requirements Elicitation*, we describe how developers write an initial set of use cases based on information provided by a client. We define in more detail how tournaments should be organized and announced, and how players apply for new tournaments. In the process, we generate more questions for the client and uncover ambiguities and missing information about the system.
- In Chapter 5, *Analysis*, we describe how an object model and a behavior model are constructed from the use case model. We also examine how the development of these models leads to more refinements in the use case model and in the discovery of additional requirements. For example, we define more formally the concept of exclusive sponsorship, describe the workflow associated with deciding on the sponsorship of a tournament, and consolidate the object model.
- In Chapter 7, *System Design: Addressing Design Goals*, we select a client server architecture and a framework for realizing the system, and address issues such as data storage and access control. We examine different mechanisms for authenticating users on the Web, identify the persistent objects we need to store (e.g., game state, tournament results, player profiles), and decompose ARENA into smaller subsystems that can be handled by single programmers.
- In Chapter 8, *Object Design: Reusing Pattern Solutions*, and in Chapter 9, *Object Design: Specifying Interfaces*, we identify additional solution domain objects to fill the gap between the system design and the implementation. We reuse template solutions by selecting design patterns for addressing specific issues. For example, a strategy pattern is used to encapsulate different tournament styles.
- In Chapter 10, *Mapping Models to Code*, we translate the UML models we built so far into Java code, and reexamine the object design as new optimization issues are discovered. In this chapter, we illustrate the tight iteration between object design and implementation.

The work products associated with the ARENA system, along with a demonstration, are available from http://wwwbruegge.in.tum.de/OOSE/WebHome.

## 4.6 ARENA Case Study

In this section, we apply the concepts and methods described in this chapter to the ARENA system. We start with the initial problem statement provided by the client, and develop a use case model and an initial analysis object model. In previous sections, we selected examples for their illustrative value. In this section, we focus on a realistic example, describe artifacts as they are created and refined. This enables us to discuss more realistic trade-offs and design decisions and focus on operational details that are typically not visible in illustrative examples. In this discussion, "ARENA" denotes the system in general, whereas "arena" denotes a specific instantiation of the system.

### 4.6.1 Initial Problem Statement

After an initial meeting with the client, the problem statement is written (Figure 4-17).

Note that this brief text describes the problem and the requirements at a high level. This is not typically the stage at which we commit to a budget or a delivery date. First, we start developing the use case model by identifying actors and scenarios.

4. A **baseline** is a version of a work product that has been reviewed and formally approved. **Configuration management** is the process of tracking and approving changes to the baseline. We discuss configuration management in Chapter 13, *Configuration Management*.

**ARENA Problem Statement**

**1. Problem**

The popularity of the Internet and the World Wide Web has enabled the creation of a variety of virtual communities, groups of people sharing common interests, but who have never met each other in person. Such virtual communities can be short lived (e.g., a group of people meeting in a chat room or playing a tournament) or long lived (e.g., subscribers to a mailing list). They can include a small group of people or many thousands.

Many multi-player computer games now include support for the virtual communities that are players of the given game. Players can receive news about game upgrades, new game maps and characters; they can announce and organize matches, compare scores and exchange tips. The game company takes advantage of this infrastructure to generate revenue or to advertise its products.

Currently, however, each game company develops such community support in each individual game. Each company uses a different infrastructure, different concepts, and provides different levels of support. This redundancy and inconsistency results in many disadvantages, including a learning curve for players when joining each new community, for game companies who need to develop the support from scratch, and for advertisers who need to contact each individual community separately. Moreover, this solution does not provide much opportunity for cross-fertilization among different communities.

**2. Objectives**

The objectives of the ARENA project are to:

- provide an infrastructure for operating an arena, including registering new games and players, organizing tournaments, and keeping track of the players scores.
- provide a framework for league owners to customize the number and sequence of matches and the accumulation of expert rating points.
- provide a framework for game developers for developing new games, or for adapting existing games into the ARENA framework.
- provide an infrastructure for advertisers.

**3. Functional requirements**

ARENA supports five types of users:

- The *operator* should be able to define new games, define new tournament styles (e.g., knock-out tournaments, championships, best of series), define new expert rating formulas, and manage users.
- *League owners* should be able to define a new league, organize and announce new tournaments within a league, conduct a tournament, and declare a winner.
- *Players* should be able to register in an arena, apply for a league, play the matches that are assigned to the player, or drop out of the tournament.
- *Spectators* should be able to monitor any match in progress and check scores and statistics of past matches and players. Spectators do not need to register in an arena.
- The *advertiser* should be able to upload new advertisements, select an advertisement scheme (e.g., tournament sponsor, league sponsor), check balance due, and cancel advertisements.

**Figure 4-17** Initial ARENA problem statement.

**4. Nonfunctional requirements**

- *Low operating cost*. The operator must be able to install and administer an arena without purchasing additional software components and without the help of a full-time system administrator.
- *Extensibility.* The operator must be able to add new games, new tournament styles, and new expert rating formulas. Such additions may require the system to be temporarily shut down and new modules (e.g., Java classes) to be added to the system. However, no modifications of the existing system should be required.
- *Scalability.* The system must support the kick-off of many parallel tournaments (e.g., 10), each involving up to 64 players and several hundreds of simultaneous spectators.
- *Low-bandwidth network*. Players should be able to play matches via a 56K analog modem or faster.

**5. Target environment**

- All users should be able to access any arena with a web browser supporting cookies, Javascript, and Java applets. Administration functions (e.g., adding new games, tournament styles, and users) used by the operator should not be available through the web.
- `ARENA` should run on any Unix operating system (e.g., MacOS X, Linux, Solaris).

**Figure 4-17** *Continued.*

### 4.6.2 Identifying Actors and Scenarios

We identify five actors, one for each type of user in the problem statement (`Operator`, `LeagueOwner`, `Player`, `Spectator`, and `Advertiser`). As the core functionality of the system is to organize and play tournaments, we first develop an example scenario, `organize-TicTacToeTournament` (Figure 4-18) to elicit and explore this functionality in more detail. By first focusing on a narrow vertical slice of the system, we understand better the client's expectation of the system, including the boundary of the system and the kinds of interactions between the user and the system. Using the `organizeTicTacToeTournament` scenario of Figure 4-18, we produce a series of questions for the client depicted in (Figure 4-19). Based on the answers from the client, we refine the scenario accordingly.

Note that when asking questions of a client, our primary goal is to understand the client's needs and the application domain. Once we understand the domain and produce a first version of the requirements specification, we can start trading off features and cost with the client and prioritizing requirements. However, intertwining elicitation and negotiation too early is usually counterproductive.

After we refine the first scenario to the point that both we agree with the client on the system boundary (for that scenario), we focus on the overall scope of the system. This is done by identifying a number of shorter scenarios for each actor. Initially, these scenarios are not detailed, but instead, cover a broad range of functionality (Figure 4-20).

When we encounter disagreements or ambiguities, we detail specific scenarios further. In this example, the scenarios `defineKnockOutStyle` and `installTicTacToeGame` would be refined to a comparable level of detail as the `organizeTicTacToeTournament` (Figure 4-18).

| | |
|---|---|
| *Scenario name* | `organizeTicTacToeTournament` |
| *Participating actor instances* | `alice:Operator, joe:LeagueOwner, bill:Spectator, mary:Player` |
| *Flow of events* | 1. Joe, a friend of Alice, is a Tic Tac Toe aficionado and volunteers to organize a tournament.<br>2. Alice registers Joe in the arena as a league owner.<br>3. Joe first defines a Tic Tac Toe beginners league, in which any players can be admitted. This league, dedicated to Tic Tac Toe games, stipulates that tournaments played in this league will follow the knockout tournament style and "Winner Takes All" formula.<br>4. Joe schedules the first tournament in the league for 16 players starting the next day.<br>5. Joe announces the tournament in a variety of forums over the Web and sends mail to other Tic Tac Toe community members.<br>6. Bill and Mary receive the E-mail notification.<br>7. Mary is interested in playing the tournament and registers. 19 others apply.<br>8. Joe schedules 16 players for the tournament and rejects the 4 that applied last.<br>9. The 16 players, including Mary, receive an electronic token for entering the tournament and the time of their first match.<br>10. Other subscribers to the Tic Tac Toe mailing list, including Bill, receive a second notice about the Tournament, including the name of the players and the schedule of matches.<br>11. As Joe kicks off the tournament, the players have a limited amount of time to enter the match. If a player fails to show up, he loses the game.<br>12. Mary plays her first match and wins. She advances in the tournament and is scheduled for the next match against another winner of the first round.<br>13. After visiting the Tic Tac Toe Tournament's home page, Bill notices Mary's victory and decides to watch her next match. He selects the match, and sees the sequence of moves of each player as they occur. He also sees an advertisement banner at the bottom of his browser, advertising other tournaments and tic tac toe products.<br>14. The tournament continues until the last match, at which point the winner of the tournament is declared and his league record is credited with all the points associated with the tournament.<br>15. Also, the winner of the tournament accumulates expert rating points.<br>16. Joe can choose to schedule more tournaments in the league, in which case, known players are notified about the date and given priority over new players. |

**Figure 4-18** `organizeTicTacToeTournament` scenario for `ARENA`.

Typical scenarios, once refined, span several pages of text. We also start to maintain a glossary of important terms, to ensure consistency in the specification and to ensure that we use the client's terms. We quickly realize that the terms `Match`, `Game`, `Tournament`, and `League` represent application domain concepts that need to be defined precisely, as these terms could have a different interpretation in other gaming contexts. To accomplish this, we maintain a working glossary and revise our definitions as our exploratory work progresses (Table 4-4).

Steps 2, 7: Different actors register with the system. In the first case, the administrator registers Joe as a league owner; in the second case, a player registers herself with the system.

- Registration of users should follow the same paradigm. Who provides the registration information and how is the information reviewed, validated, and accepted?
- *Client: Two processes are confused in steps 2 & 7, the registration process, during which new users (e.g., a player or a league owner) establish their identity, and the application process, during which players indicate they want to take part in a specific tournament. During the registration process, the user provides information about themselves (name, nickname, E-mail) and their interests (types of games and tournaments they want to be informed about). The information is validated by the operator. During the application process, players indicate which tournament they want to participate in. This is used by the league owner during match scheduling.*
- Since the player information has already been validated by the operator, should the match scheduling be completely automated?
- *Client: Yes, of course.*

Step 5: Joe sends mail to the Tic Tac Toe community members:

- Does ARENA provide the opportunity to users to subscribe to individual mailing lists?
- *Client: Yes. There should be mailing lists for announcing new games, new leagues, new tournaments, etc.*
- Does ARENA store a user profile (e.g., game watched, games played, interests specified by a user survey) for the purpose of advertisement?
- *Client: Yes, but users should still be able to register without completing a user survey, if they want to. They should be encouraged to enter the survey, but this should not prevent them from entering. They will be exposed to advertisements anyway.*
- Should the profile be used to automatically subscribe to mailing lists?
- *Client: No, we think users in our community would prefer having complete control over their mailing list subscriptions. Guessing subscriptions would not give them the impression they are in control.*

Step 13: Bill browses match statistics and decides to see the next match in real time.

- How are players identified to the spectators? By real name, by E-mail, by nickname?
- *Client: This should be left to the user during the registration.*
- Can a spectator replay old matches?
- *Client: Games should be able to provide this ability, but some games (e.g., real-time, 3D action games) may choose not to do so because of resource constraints.*
- ARENA should support real-time games?
- *Client: Yes, these represent the largest share of our market. In general, ARENA should support as broad a range of games as possible.*
- ...

**Figure 4-19** Questions generated from the scenario of Figure 4-18. Answers from the client emphasized in *italics*. The interviewer can ask follow-up questions as new knowledge is accidentally stumbled upon.

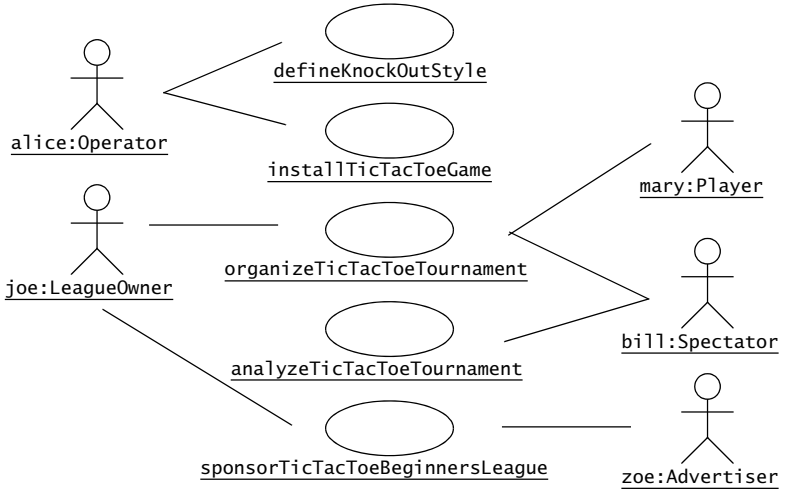



**Figure 4-20** High-level scenarios identified for ARENA. Clients and developers initially briefly describe scenarios. They refine them further to clarify ambiguities or uncover disagreements.

**Table 4-4** Working glossary for ARENA. Keeping track of important terms and their definitions ensures consistency in the specification and ensures that developers use the language of the client.

| | |
|---|---|
| **Game** | A Game is a competition among a number of Players that is conducted according to a set of rules. In ARENA, the term Game refers to a piece of software that enforces the set of rules, tracks the progress of each Player, and decides the winner. For example, tic tac toe and chess are Games. |
| **Match** | A Match is a contest between two or more Players following the rules of a Game. The outcome of a Match can be a single winner and a set of losers or a tie (in which their are no winners or losers). Some Games may disallow ties. |
| **Tournament** | A Tournament is a series of Matches among a set of Players. Tournaments end with a single winner. The way Players accumulate points and Matches are scheduled is dictated by the League in which the Tournament is organized. |
| **League** | A League represents a community for running Tournaments. A League is associated with a specific Game and TournamentStyle. Players registered with the League accumulate points according to the ExpertRating defined in the League. For example, a novice chess League has a different ExpertRating formula than an expert League. |
| **TournamentStyle** | The TournamentStyle defines the number of Matches and their sequence for a given set of Players. For example, Players face all other Players in the Tournament exactly once in a round robin TournamentStyle. |

Once we agree with the client on a general scope of the system, we formalize the knowledge acquired so far in the form of high-level use cases.

### 4.6.3 Identifying Use Cases

Generalizing scenarios into use cases enables developers to step back from concrete situations and consider the general case. Developers can then consolidate related functionality into single use cases and split unrelated functionality into several use cases.

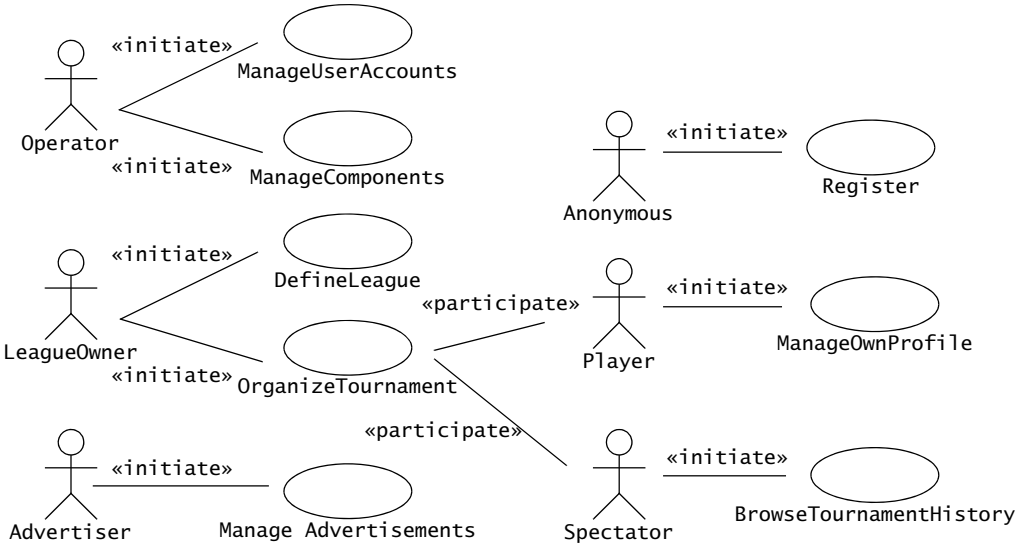
When inspecting the `organizeTicTacToeTournament` scenario closely, we realize that it covers a broad range of functionality initiated by many actors. We anticipate that generalizing this scenario would result in a use case of several dozen pages long, and attempt to split it into self-contained and independent use cases initiated by single actors. We first decide to split the functionality related to user accounts into two use cases, `ManageUserAccounts`, initiated by the `Operator`, and `Register`, initiated by potential players and league owners (Figure 4-21). We identify a new actor, `Anonymous`, representing these potential users who do not yet have an account. Similarly, we split the functionality with browsing past matches and with managing user profiles into separate use cases (`BrowseTournamentHistory` and `ManageOwnProfile`, initiated by the `Spectator` and the `Player`, respectively). Finally, to further shorten the use case `OrganizeTournament`, we split off the functionality for creating new leagues into the `DefineLeague` use case, as a `LeagueOwner` may create many tournaments within the scope of a single league. Conversely, we anticipate that the installation of new games and new styles requires similar steps from the `Operator`. Hence, we consolidate all functionality related to installing new components into the `ManageComponents` use case initiated by the `Operator`.

We capture these decisions by drawing an overview use case diagram and by briefly describing each use case (Figure 4-21). Note that a use case diagram alone does not describe much functionality. Instead, it is an index into the many descriptions produced during this phase.

Next, we describe the fields of each high-level use case, including the participating actors, entry and exit conditions, and a flow of events. Figure 4-22 depicts the high-level `OrganizeTournament` use case.

Note that all steps in this flow of events describe actor actions. High-level use cases focus primarily on the task accomplished by the actor. The detailed interaction with the system, and decisions about the boundaries of the system, are initially postponed to the refinement phase. This enables us to first describe the application domain with use cases, capturing, in particular, how different actors collaborate to accomplish their goals.

In Figure 4-22, we describe the sequence of actions that are performed by four actors to organize a tournament: the `LeagueOwner`, who facilitates the complete activity, the `Advertiser`, to resolve exclusive sponsorship issues, the potential `Players` who want to participate, and the `Spectators`. In the first step, we describe the handling of the sponsorship issue, thus making clear that any sponsorship issue needs to be resolved before the tournament is advertised and before the players apply for the tournament. Originally, the sponsorship issue was not described clearly in the scenarios of Figure 4-20 (which only described the sponsorships of leagues). After




**Register** — Anonymous users register with an Arena for a Player or a LeagueOwner account. User accounts are required before applying for a tournament or organizing a league. Spectators do not need accounts.

**ManageUserAccounts** — The Operator accepts registrations from LeagueOwners and for Players, cancels existing accounts, and interacts with users about extending their accounts.

**ManageComponents** — The Operator installs new games and defines new tournament styles (generalizes defineKnockOutStyle and installTicTacToeGame).

**DefineLeague** — The LeagueOwner defines a new league (generalizes the first steps of the scenario organizeTicTacToeTournament).

**OrganizeTournament** — The LeagueOwner creates and announces a new tournament, accepts player applications, schedules matches, and kicks off the tournament. During the tournament, players play matches and spectators follow matches. At the end of the tournament, players are credited with points (generalizes the scenario organizeTicTacToeTournament).

**ManageAdvertisements** — The Advertiser uploads banners and sponsors league or tournaments (generalizes sponsorTicTacToeBeginnersLeague).

**ManageOwnProfile** — The Players manage their subscriptions to mailing lists and answer a marketing survey.

**BrowseTournamentHistory** — Spectators examine tournament statistics and player statistics, and replay matches that have already been concluded (generalizes the scenario analyzeTicTacToeTournament).

**Figure 4-21** High-level use cases identified for ARENA.

| *Use case name* | `OrganizeTournament` |
|---|---|
| *Participating actors* | Initiated by `LeagueOwner`<br>Communicates with `Advertiser`, `Player`, and `Spectator` |
| *Flow of events* | 1. The `LeagueOwner` creates a `Tournament`, solicits sponsorships from `Advertisers`, and announces the `Tournament` (include use case `AnnounceTournament`).<br>2. The `Players` apply for the `Tournament` (include use case `ApplyForTournament`).<br>3. The `LeagueOwner` processes the `Player` applications and assigns them to matches (include use case `ProcessApplications`).<br>4. The `LeagueOwner` kicks off the `Tournament` (include use case `KickoffTournament`).<br>5. The `Players` compete in the matches as scheduled and `Spectators` view the matches (include use case `PlayMatch`).<br>6. The `LeagueOwner` declares the winner and archives the `Tournament` (include use case `ArchiveTournament`). |
| *Entry condition* | • The `LeagueOwner` is logged into `ARENA`. |
| *Exit conditions* | • The `LeagueOwner` archived a new tournament in the `ARENA` archive and the winner has accumulated new points in the league, OR<br>• The `LeagueOwner` cancelled the tournament and the players' standing in the league is unchanged. |

**Figure 4-22** An example of a high-level use case, `OrganizeTournament`.

discussions with the client, we decided to handle also tournament sponsorship, and to handle it at the beginning of each tournament. On the one hand, this enables new sponsors to be added to the system, and on the other hand, it allows the sponsor, in exchange, to advertise the tournament using his or her own resources. Finally, this enables the system to better select advertisement banners during the application process.

In this high-level use case, we boiled down the essentials of the `organize-TicTacToeTournament` scenario into six steps and left the details to the detailed use case. By describing each high-level use case in this manner, we capture all relationships among actors that the system must be aware of. This also results in a summary description of the system that is understandable to any newcomer to the project.

Next, we write the detailed use cases to specify the interactions between the actors and the system.

### 4.6.4 Refining Use Cases and Identifying Relationships

Refining use cases enables developers to define precisely the information exchanged among the actors and between the actors and the system. Refining use cases also enables the discovery of alternative flows of events and exceptions that the system should handle.

To keep the case study manageable, we do not show the complete refinement. We start by identifying one detailed use case for each step of the flow of events in the high-level `OrganizeTournament` use case. The resulting use case diagram is shown in Figure 4-23. We then focus on the use case, `AnnounceTournament`: Figure 4-24 contains a description of the flow of events, and Figure 4-25 identifies the exceptions that could occur in `AnnounceTournament`. The remaining use cases will be developed similarly.

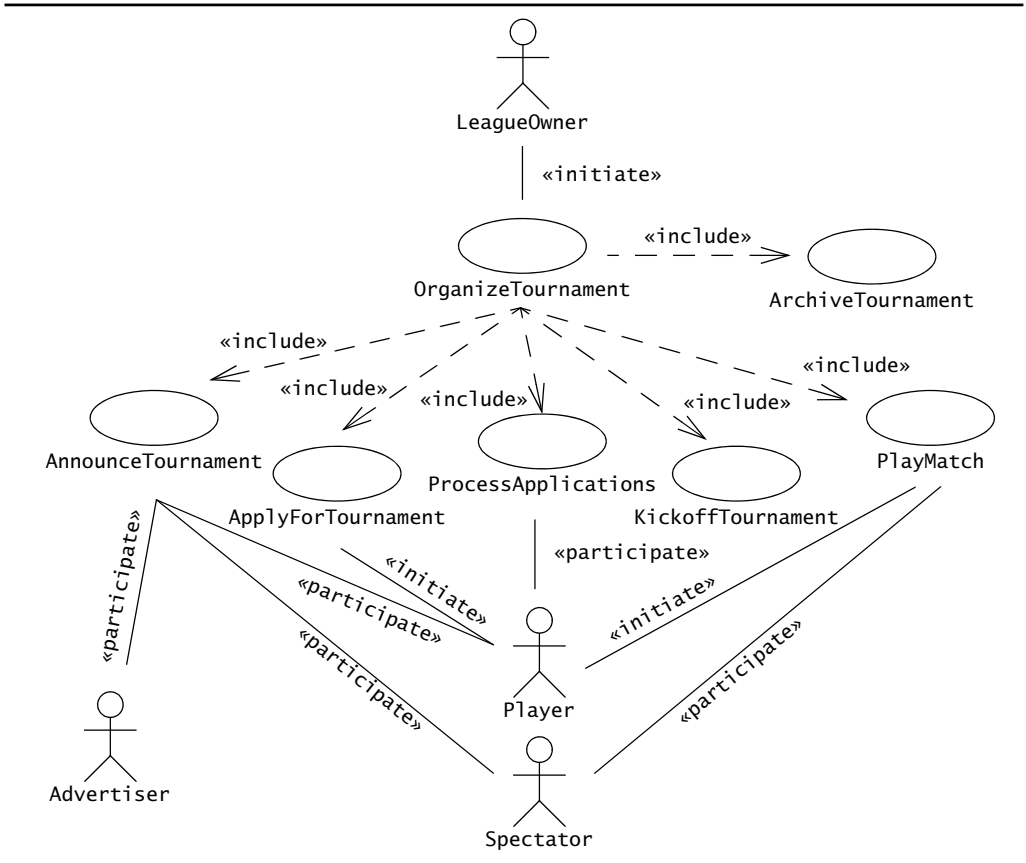



**Figure 4-23** Detailed use cases refining the `OrganizeTournament` high-level use case.

All of the use cases in Figure 4-23 are initiated by the `LeagueOwner`, except that the `ApplyForTournament` and `PlayMatch` are initiated by the `Player`. The `Advertiser` participates in `AnnounceTournament` and the `Spectator` participates in `AnnounceTournament` and `PlayMatch` use cases. The `Player` participates in all use cases that refine `OrganizeTournament`. To keep the use case diagram readable, we omitted the «`initiate`» relationships between the `LeagueOwner` and the refined use cases. When using a UML modeling tool, we would include those

| *Name* | `AnnounceTournament` |
|---|---|
| *Participating actors* | Initiated by `LeagueOwner`<br>Communicates with `Player`, `Advertiser`, `Spectator` |
| *Flow of events* | 1. The `LeagueOwner` requests the creation of a tournament.<br>2. The system checks if the `LeagueOwner` has exceeded the number of tournaments in the league or in the arena. If not, the system presents the `LeagueOwner` with a form.<br>3. The `LeagueOwner` specifies a name, application start and end dates during which `Players` can apply to the tournament, start and end dates for conducting the tournament, and a maximum number of `Players`.<br>4. The system asks the `LeagueOwner` whether an exclusive sponsorship should be sought and, if yes, presents a list of `Advertisers` who expressed the desire to be exclusive sponsors.<br>5. If the `LeagueOwner` decides to seek an exclusive sponsor, he selects a subset of the names of the proposed sponsors.<br>6. The system notifies the selected sponsors about the upcoming tournament and the flat fee for exclusive sponsorships.<br>7. The system communicates their answers to the `LeagueOwner`.<br>8. If there are interested sponsors, the `LeagueOwner` selects one of them.<br>9. The system records the name of the exclusive sponsor and charges the flat fee for sponsorships to the `Advertiser`'s account. From now on, all advertisement banners associated with the tournament are provided by the exclusive sponsor only.<br>10. Otherwise, if no sponsors were selected (either because no `Advertiser` was interested or the `LeagueOwner` did not select one), the advertisement banners are selected at random and charged to the `Advertiser's` account on a per unit basis.<br>11. Once the sponsorship issue is closed, the system prompts the `LeagueOwner` with a list of groups of `Players`, `Spectators`, and `Advertisers` that could be interested in the new tournament.<br>12. The `LeagueOwner` selects which groups to notify.<br>13. The system creates a home page in the arena for the tournament. This page is used as an entry point to the tournament (e.g., to provide interested `Players` with a form to apply for the tournament, and to interest `Spectators` in watching matches).<br>14. On the application start date, the system notifies each interested user by sending them a link to the main tournament page. The `Players` can then apply for the tournament with the `ApplyForTournament` use case until the application end date. |

**Figure 4-24** An example of a detailed use case, `AnnounceTournament`.

| | |
|---|---|
| *Entry condition* | • The `LeagueOwner` is logged into `ARENA`. |
| *Exit conditions* | • The sponsorship of the tournament is settled: either a single exclusive `Advertiser` paid a flat fee or banners are drawn at random from the common advertising pool of the `Arena`.<br>• Potential `Players` received a notice concerning the upcoming tournament and can apply for participation.<br>• Potential `Spectators` received a notice concerning the upcoming tournament and know when the tournament is about to start.<br>• The tournament home page is available for any to see, hence, other potential `Spectators` can find the tournament home page via web search engines, or by browsing the `Arena` home page. |
| *Quality requirements* | • Offers to and replies from `Advertisers` require secure authentication, so that `Advertisers` can be billed solely on their replies.<br>• `Advertisers` should be able to cancel sponsorship agreements within a fixed period, as required by local laws. |

**Figure 4-24** *Continued.*

relationships as well. We start by writing out the flow of events for the `AnnounceTournament` use case (Figure 4-24).

The steps in Figure 4-24 describe in detail the information exchanged between the actor and the system. Note, however, that we did not describe any details of the user interface (e.g., forms, buttons, layout of windows or web pages). It is much easier to design a usable user interface later, after we know the intent and responsibility of each actor. Hence, the focus on the refinement phase is to assign (or discover) the detailed intent and responsibilities of each actor.

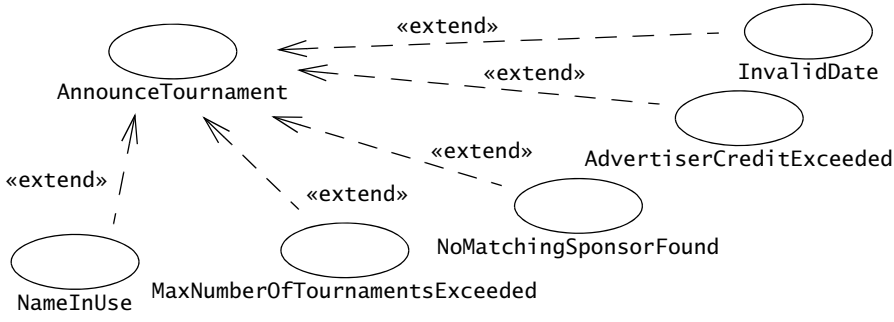
When describing the steps of the detailed `AnnounceTournament` use case, we and the client made more decisions about the boundaries of the system:

- We introduced start and end dates for the application process and for executing the tournament (Step 3 in Figure 4-24). This enables us to communicate deadlines to all actors involved to ensure that the tournament happens within a reasonable time frame.
- We decided that advertisers indicate in their profile whether they are interested in exclusive sponsorships or not. This enables the `LeagueOwner` to target `Advertisers` more specifically (Step 4 in Figure 4-24).
- We also decided to enable advertisers to commit to sponsorship deals through the system and automated the accounting of advertisement and the billing. This entails security and legal requirements on the system, which we document in the "quality requirements" field of the use case (Step 9 and 10 in Figure 4-24).

Note that these decisions are validated with the client. Different clients and environments can lead to evaluating trade-offs differently for the same system. For example, the decision about soliciting `Advertisers` and obtaining a commitment through the system results in a more complex and expensive system. An alternative would have been to solicit `Advertisers` via E-mail, but obtain their commitment via phone. This would have resulted in a simpler system, but more work on the part of the `LeagueOwner`. The client is the person who decides between such alternatives, understanding, of course, that these decisions have an impact on the cost and the delivery date of the system.

Next, we identify the exceptions that could occur during the detailed use case. This is done by reviewing every step in the use case and identifying all the events that could go wrong. We briefly describe the handling of each exception and depict the exception handling use cases as extensions of the `AnnounceTournament` use case (Figure 4-25).

Note that not all exceptions are equal, and different kinds of exceptions are best addressed at different stages of development. In Figure 4-25, we identify exceptions caused by resource constraints (`MaxNumberOfTournamentsExceeded`), invalid user input (`InvalidDate`, `NameInUse`), or application domain constraints (`AdvertiserCreditExceeded`, `NoMatchingSponsorFound`). Exceptions associated with resource constraints are best handled during system design. Only during system design will it become clear which resources are limited and how to best share




| | |
|---|---|
| **`AdvertiserCreditExceeded`** | The system removes the `Advertiser` from the list of potential sponsors. |
| **`InvalidDate`** | The system informs the `LeagueOwner` and prompts for a new date. |
| **`MaxNumberOfTournaments Exceeded`** | The `AnnounceTournament` use case is terminated. |
| **`NameInUse`** | The system informs the `LeagueOwner` and prompts for a new name. |
| **`NoMatchingSponsorFound`** | The system skips the exclusive sponsor steps and chooses random advertisements from the advertisement pool. |

**Figure 4-25** Exceptions occurring in `AnnounceTournament` represented as extending use cases. (Note that `AnnounceTournament` in this figure is the same as the use case in Figure 4-23).

them among different users which may, in turn, trigger further requirements activities during system design to validate with the client the handling of such exceptions. Exceptions associated with invalid user input are best handled during user interface design, when developers will be able to decide at which point to check for invalid input, how to display error messages, and how to prevent invalid inputs in the first place. The third category of exceptions—application domain constraints—should receive the focus of the client and developer early. These are exceptions that are usually not obvious to the developer. When missed, they require substantial rework and changes to the system. A systematic way to elicit those exceptions is to walk through the use case step by step with the client or a domain expert.

Many exceptional events can be represented either as an exception (e.g., `AdvertiserCreditExceeded`) or as a nonfunctional requirement (e.g., "An `Advertiser` should not be able to spend more advertisement money than a fixed limit agreed beforehand with the `Operator` during the registration"). The latter representation is more appropriate for global constraints that apply to several use cases. Conversely, the former is more appropriate for events that can occur only in one use case (e.g., "`NoMatchingSponsorFound`").

Writing each detailed use case, including their exceptions, constitutes the lion's share of the requirements elicitation effort. Ideally, developers write every detailed use case and address all application domain issues before committing to the project and initiating the realization of the system. In practice, this never happens. For large systems, the developers produce a large amount of documentation in which it is difficult, if not impossible, to maintain consistency. Worse, the requirements elicitation activity of large projects should already be financed, as this phase requires a lot of resources from both the client and the development organization. Moreover, completeness at an early stage can be counterproductive: use case steps change during development as new domain facts are discovered. The decision about how many use cases to detail and how much to leave implicit is as much a question of trust as of economics: the client and the developers should share a sufficiently good understanding of the system to be ready to commit to a schedule, a budget, and a process for handling future changes (including changes in requirements, schedule, and budget).

In `ARENA`, we focus on specifying in detail the interactions that involve the `Advertisers` and the `Players`, since they have critical roles in generating revenue. Use cases associated with the administration of the system or the installation of new games or tournament styles are left for later, since they also include more technical issues that are dependent on the solution domain.

### 4.6.5 Identifying Nonfunctional Requirements

Nonfunctional requirements come from a variety of sources during the elicitation. The problem statement we started with in Figure 4-17 already specified performance and implementation requirements. When detailing the `AnnounceTournament` use case, we identified further legal requirements for billing `Advertisers`. When reviewing exceptions in the previous section, we identified a constraint on the amount of money `Advertisers` can spend. Although we encounter many nonfunctional requirements while writing use cases and refining them, we cannot ensure

that we identify all the essential nonfunctional requirements. To ensure completeness, we use the FURPS+ categories we described in Section 4.3.2 (or any other systematic taxonomy of nonfunctional requirements) as a checklist for asking questions of the client. Table 4-5 depicts the nonfunctional requirements we identified in ARENA after detailing the AnnounceTournament use case.

**Table 4-5** Consolidated nonfunctional requirements for ARENA, after the first version of the detailed AnnounceTournament use case.

| Category | Nonfunctional requirements |
|---|---|
| **Usability** | • Spectators must be able to access games in progress without prior registration and without prior knowledge of the Game. |
| **Reliability** | • Crashes due to software bugs in game components should interrupt at most one Tournament using the Game. The other Tournaments in progress should proceed normally.<br>• When a Tournament is interrupted because of a crash, its LeagueOwner should be able to restart the Tournament. At most, only the last move of each interrupted Match can be lost. |
| **Performance** | • The system must support the kick-off of many parallel Tournaments (e.g., 10), each involving up to 64 Players and several hundreds of simultaneous Spectators.<br>• Players should be able to play matches via an analog modem. |
| **Supportability** | • The Operator must be able to add new Games and new TournamentStyles. Such additions may require the system to be temporarily shut down and new modules (e.g., Java classes) to be added to the system. However, no modifications of the existing system should be required. |
| **Implementation** | • All users should be able to access an Arena with a web browser supporting cookies, Javascript, and Java applets. Administration functions used by the operator are not available through the web.<br>• ARENA should run on any Unix operating system (e.g., MacOS X, Linux, Solaris). |
| **Operation** | • An Advertiser should not be able to spend more advertisement money than a fixed limit agreed beforehand with the Operator during the registration. |
| **Legal** | • Offers to and replies from Advertisers require secure authentication, so that agreements can be built solely on their replies.<br>• Advertisers should be able to cancel sponsorship agreements within a fixed period, as required by local laws. |

### 4.6.6 Lessons Learned

In this section, we developed an initial use case and analysis object model based on a problem statement provided by the client. We used scenarios and questions as elicitation tools to clarify ambiguous concepts and uncover missing information. We also elicited a number of nonfunctional requirements. We learned that

- Requirements elicitation involves constant switching between perspectives (e.g., high-level vs. detailed, client vs. developer, activity vs. entity).
- Requirements elicitation requires a substantial involvement from the client.
- Developers should not assume that they know what the client wants.
- Eliciting nonfunctional requirements forces stakeholders to make and document trade-offs.

## 5.6 ARENA Case Study

In this section, we apply the concepts and methods described in this chapter to the ARENA system. We start with the use case model and the glossary developed in the previous chapter. We identify participating entity, boundary, and control objects, and refine them by adding attributes and associations to the analysis object model. Finally, we identify inheritance relationships and consolidate the analysis object model. In this section, we focus primarily on the AnnounceTournament use case.

### 5.6.1 Identifying Entity Objects

Entity objects represent concepts in the application domain that are tracked by the system. We use the glossary produced during elicitation as a starting point for identifying entity objects in ARENA. We identify additional entity objects and their attributes by applying Abbott's heuristics on the use cases. We initially focus only on noun phrases that denote concepts of the application domain. Figure 5-23 depicts the AnnounceTournament use case with the first occurrence of noun phrases we identified in **bold**.

Note that we identify entity objects corresponding to actors in the use case model. Actors are concepts in the application domain and are relevant to the system (e.g., for access control or for documenting responsibilities or authorship). In ARENA, each legitimate LeagueOwner is represented with an object that is used to store data specific to that LeagueOwner, such as her contact information, the leagues that she manages, and so on.

Note, also, that not all noun phrases we identified correspond to classes. For example, name of a tournament is a noun phrase referring to an attribute of the Tournament class. List of Advertisers is an association, in this case, between the League class and the Advertiser class. We can use a few simple heuristics to distinguish between noun phrases that correspond to objects, attributes, and associations:

| *Name* | AnnounceTournament |
|---|---|
| *Flow of events* | 1. The **LeagueOwner** requests the creation of a **tournament**.<br>2. The system checks if the LeagueOwner has exceeded the **number of tournaments** in the **league** or in the **arena**. If not, the system presents the LeagueOwner with a form.<br>3. The LeagueOwner specifies a **name**, **application start and end dates** during which Players can apply to the tournament, **start and end dates** for conducting the tournament, and a **maximum number of Players**.<br>4. The system asks the LeagueOwner whether an exclusive sponsorship should be sought and, if yes, presents a **list of Advertisers** who expressed the desire to be **exclusive sponsors**.<br>5. If the LeagueOwner decides to seek an exclusive sponsor, he selects a subset of the **names** of the **proposed sponsors**.<br>6. The system notifies the selected sponsors about the upcoming tournament and the **flat fee** for exclusive sponsorships.<br>7. The system communicates their **answers** to the LeagueOwner.<br>8. If there are interested sponsors, the LeagueOwner selects one of them.<br>9. The system records the **name** of the exclusive sponsor and charges the flat fee for sponsorships to the **Advertiser's account**. From now on, all **advertisement banners** associated with the tournament are provided by the exclusive sponsor only.<br>10. If no sponsors were selected (either because no Advertisers were interested or the LeagueOwner did not select any), the advertisement banners are selected at random and charged to each Advertiser's account on a per unit basis.<br>11. Once the sponsorship issues is closed, the system prompts the LeagueOwner with a **list of groups of Players, Spectators, and Advertisers** that could be interested in the new tournament.<br>12. The LeagueOwner selects which groups to notify.<br>13. The system creates a home page in the arena for the tournament. This page is used as an entry point to the tournament (e.g., to provide interested Players with a form to apply for the tournament, and to interest Spectators into watching **matches**).<br>14. At the application start date, the system notifies each interested user by sending them a link to the main tournament page. The Players can then apply for the tournament with the ApplyForTournament use case until the application end date. |

**Figure 5-23** Applying Abbott's heuristics for identifying entity objects in the AnnounceTournament use case. The first occurrence of a noun phrase is emphasized in **bold**.

- *Attributes are properties.* Attributes represent a single property of an object. They represent a partial aspect of an object and are incomplete. For example, the name of an `Advertiser` is an attribute that identifies an `Advertiser`. However, it does not include other relevant information about the `Advertiser` (e.g., her current account balance, the type of banners she advertises, etc.) that are represented by other attributes or associations of the `Advertiser` class.
- *Attributes have simple types.* Attributes are properties that often have types such as a number (e.g., maximum number of `Tournaments`), string (e.g., the name of an `Advertiser`), dates (e.g., the application start and end date of a `Tournament`). Properties such as an address, a social security number, and a vehicle identification number are also usually considered as simple types (and hence represented as attributes) because users treat those as simple, atomic concepts. Complex concepts are represented as objects that are related to other objects with associations. For example, an `Account` is an object that is related to the corresponding `Advertiser` and can include a balance, a history of transactions, a credit limit, and other similar properties.
- *Nouns referring to collections are associations, often with implicit ends*. Lists, groups, tables, and sets are represented by associations. For example, `ARENA` prompts the `LeagueOwner` with a list of `Advertisers` that are potentially interested in exclusive sponsorships. This concept can be represented with an association between the `Arena` class and the `Advertiser` class, denoting which `Advertisers` are interested in exclusive sponsorships. Often, the association end is implicit. For example, when sponsorship issues are closed, `ARENA` prompts the `LeagueOwner` with a list of groups of `Players`, `Spectators`, and `Advertisers`. We identify a new class, `InterestGroup`, representing collections of users interested in new events about a league or a game. Then, we identify an association between the `Arena` class and the `InterestGroup` class (corresponding to the word "list") representing all `InterestGroups`. Then, we identify an association between the `InterestGroup` class and the `Player`, `Spectator`, and `Advertisers` classes (corresponding to the word "group"). Finally, we identify additional associations originating from the `InterestGroup` class to other classes representing the interest of the users in the `InterestGroup` (i.e., `League`, `Game`).

Table 5-6 lists the entity objects, their attributes, and their associations that we identified so far from the `AnnounceTournament` use case. We attach the attributes and associations to their relevant classes and write definitions for new classes. Writing definitions has several purposes. First, a name is not specific enough for all stakeholders to share the same understanding about the concept. For example, terms such as `Game` and `Match` can be interchanged in many contexts. In `ARENA`, however, they refer to distinct concepts (i.e., a `Game` represents a set of rules enforced by a piece of software, a `Match` represent a competition among a set of `Players`). Second, objects identified during analysis correspond also to terms in the glossary we started during elicitation. Stakeholders use the glossary throughout development to resolve ambiguities and

establish a standard terminology. Writing short definitions as classes are identified is the best way to prevent ambiguities and misunderstandings. Postponing the writing of definitions results in loss of information and in incomplete definitions.

**Table 5-6** Entity objects participating in the `AnnounceTournament` use case identified from noun phrases in the use case. "(?)" denote areas of uncertainty that lead to the questions in Figure 5-24.

| Entity Object | Attributes & Associations | Definition |
|---|---|---|
| **`Account`** | • balance<br>• history of charges (?)<br>• history of payments (?) | An `Account` represents the amount currently owed by an `Advertiser`, a history of charges, and payments. |
| **`Advertiser`** | • name<br>• leagues of interest for exclusive sponsorships (?)<br>• sponsored tournaments<br>• account | Actor interested in displaying advertisement banners during the `Matches`. |
| **`Advertisement`** | • associated game (?) | Image provided by an `Advertiser` for display during matches. |
| **`Arena`** | • max number of tournaments<br>• flat fee for sponsorships (?)<br>• leagues *(implied)*<br>• interest groups *(implied)* | An instantiation of the `ARENA` system. |
| **`Game`** | | A `Game` is a competition among a number of `Players` that is conducted according to a set of rules. In `ARENA`, the term `Game` refers to a piece of software that enforces the set of rules, tracks the progress of each `Player`, and decides the winner. |
| **`InterestGroup`** | • list of players, spectators, or advertisers<br>• games and leagues of interests *(implied)* | `InterestGroups` are lists of users in the `ARENA` which share an interest (e.g, for a game or a league). `InterestGroups` are used as mailing lists for notifying potential actors of new events. |
| **`League`** | • max number of tournament<br>• game | A `League` represents a community for running `Tournaments`. A `League` is associated with a specific `Game` and `TournamentStyle`. `Players` registered with the `League` accumulate points according to the `ExpertRating` of the `League`. |

**Table 5-6** *Continued.*

| Entity Object | Attributes & Associations | Definition |
|---|---|---|
| `LeagueOwner` | • name *(implied)* | The actor creating a `League` and responsible for organizing `Tournaments` within the `League`. |
| `Match` | • tournament<br>• players | A `Match` is a contest between two or more `Players` within the scope of a `Game`. The outcome of a `Match` can be a single winner and a set of losers or a tie (in which their are no winners or losers). Some `TournamentStyles` may disallow ties. |
| `Player` | • name *(implied)* | |
| `Tournament` | • name<br>• application start date<br>• application end date<br>• play start date<br>• play end date<br>• max number of players<br>• exclusive sponsor | A `Tournament` is a series of `Matches` among a set of `Players`. `Tournaments` end with a single winner. The way `Players` accumulate points and `Matches` are scheduled is dictated by the `League` in which the `Tournament` is organized. |

The identification of entity objects and their related attributes usually triggers additional questions for the client. For example, when we identify implicit attributes and associations, we should double-check with the client to confirm whether our intuition was correct. In other cases, the ends of an association are ambiguous. We collect all the questions generated by the identification of objects and go back to the client (or the domain expert). Figure 5-24 depicts the questions we have after identifying entity objects participating in the `AnnounceTournament` use case.

**Questions for the ARENA client**

- What information should be recorded in the advertisers' accounts? For example, should a complete log of the display of each advertisement banner be recorded?
- Do advertisers express the interest for exclusive sponsorships for specific leagues or for the complete arena?
- Should advertisement banners be associated to games (to enable a more intelligent selection of banners when there is no exclusive sponsorship)?
- Does the flat fee for exclusive sponsorship vary across leagues or tournaments?

**Figure 5-24** Questions triggered by the identification of entity objects.

### 5.6.2 Identifying Boundary Objects

Boundary objects represent the interface between the system and the actors. They are identified from the use cases and usually represent the user interface at a coarse level. Do not represent layout information or user interface details such as menus and buttons. User interface mock-ups are much better suited for this type of information. Instead, boundary objects represent concepts such as windows, forms, or hardware artifacts such as workstations. This enables stakeholders to visualize where functionality is available in the system.

Abbott's heuristics do not identify many boundary objects, as they are often left implicit initially. Instead, we scan the `AnnounceTournament` use case (Figure 5-23) and identify where information is exchanged between the actors and the system. We focus both on forms in which actors provide information to the system (e.g., the form used by the `LeagueOwner` to create a `Tournament`) and on notices in which the system provides information to the actors (e.g., a notice received by `Advertisers` requesting sponsorship). As with other objects, we briefly define each class as we identify it. Table 5-7 depicts the boundary objects we identified for `AnnounceTournament` with their definitions. Figure 5-25 depicts our additional questions.

Note that `AnnounceTournament` is a relatively complex use case involving several actors. This yields a relatively large number boundary objects. In practice, a use case can have as few as a single boundary object to represent the interface between the initiating actor and the system. In all cases, however, each use case should have at least one participating boundary object (possibly shared with other use cases).

**Table 5-7** Boundary objects participating in the `AnnounceTournament` use case.

| Boundary Object | Definition |
|---|---|
| **`TournamentForm`** | Form used by the `LeagueOwner` to specify the properties of a `Tournament` during creation or editing. |
| **`RequestSponsorshipForm`** | Form used by the `LeagueOwner` to request sponsorships from interested `Advertisers`. |
| **`SponsorshipRequest`** | Notice received by `Advertisers` requesting sponsorship. |
| **`SponsorshipReply`** | Notice received by `LeagueOwner` indicating whether an `Advertiser` wants the exclusive sponsorship of the tournament. |
| **`SelectExclusiveSponsorForm`** | Form used by the `LeagueOwner` to close the sponsorship issue. |
| **`NotifyInterestGroupsForm`** | Form used by the `LeagueOwner` to notify interested users. |
| **`InterestGroupNotice`** | Notice received by interested users about the creation of a new `Tournament`. |

**More questions for the ARENA client**

- What should we do about sponsors who do not answer?
- How should we advertise a new tournament if there are no relevant interest groups?
- How should users be notified (e.g., E-mail, cell phone, ARENA notice box)?

**Figure 5-25** Questions triggered by the identification of boundary objects.

### 5.6.3 Identifying Control Objects

Control objects represent the coordination among boundary and entity objects. In the common case, a single control object is created at the beginning of the use case and accumulates all the information needed to complete the use case. The control object is then destroyed with the completion of the use case.

In AnnounceTournament, we identify a single control object called AnnounceTournamentControl, which is responsible for sending and collecting notices to Advertisers, checking resource availability, and, finally, notifying interested users. Note that, in the general case, several control objects could participate in the same use case, if, for example, there are alternate flows of events to be coordinated, multiple workstations operating asynchronously, or if some control information survives the completion of the use case.

### 5.6.4 Modeling Interactions Among Objects

We have identified a number of entity, boundary, and control objects participating in the AnnounceTournament use case. Along the way, we also identified some of their attributes and associations. We represent these objects in a sequence diagram, depicting the interactions that occur during the use case to identify additional associations and attributes.

In the sequence diagram, we arrange the objects we identified along the top row. We place left-most the initiating actor (i.e., LeagueOwner), followed by the boundary object responsible for initiating the use case (i.e., TournamentForm), followed by the main control object (i.e., AnnounceTournamentControl), and the entity objects (i.e., Arena, League, and Tournament). Any other participating actors and their corresponding boundary objects are on the right of the diagram. We split the sequence diagram associated with AnnounceTournament into three figures for space reasons. Figure 5-26 depicts the sequence of interactions leading to the creation of a tournament. Figure 5-27 depicts the workflow for requesting and selecting an exclusive sponsor. Figure 5-28 focuses on the notification of interest groups.

The sequence diagram in Figure 5-26 is straightforward. The LeagueOwner requests the creation of the tournament and specifies its initial parameter (e.g., name, maximum number of players). The AnnounceTournamentControl instance is created and, if resources allow, a Tournament entity instance is created.

The sequence diagram in Figure 5-27 is more interesting as it leads to the identification of additional associations and attributes. When requesting sponsorships, the control object must

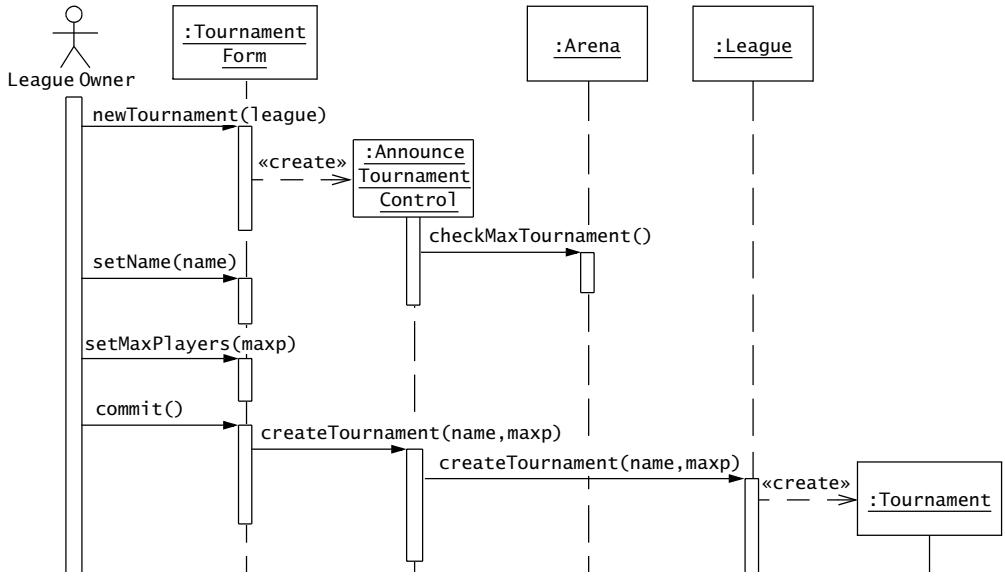



**Figure 5-26** UML sequence diagram for `AnnounceTournament`, tournament creation workflow.

first obtain a list of interested sponsors. It requests it from the `Arena` class, which maintains the list of interested sponsors. This entails that the `Arena` class maintains at all times the list of all `Advertisers`, so that it can return this list to the `AnnounceTournamentControl` object (or control objects for other use cases that require the list of all `Advertisers`). To notify an `Advertiser`, we also may need contact information, such as E-mail address, or we may need to create a mailbox for notices within `ARENA`. Consequently, we add an contact attribute to the `Advertiser` class, which initially stores the E-mail address of the `Advertiser` until further devices are supported. Anticipating similar needs for other actors, we also add contact attributes to the `LeagueOwner` and `Player` classes.

When constructing the sequence diagram for notifying interest groups (Figure 5-28), we realize that the use case does not specify how the selected sponsor is notified. Consequently, we add a step in the use case to notify all sponsors who replied about the sponsorship decisions before interest groups are notified. This requires the identification of a new boundary object, `SponsorNotice`. The rest of the interaction does not yield any new discovery, as we already anticipated the need for the `InterestGroup` and the `InterestGroupNotice` classes.

### 5.6.5 Reviewing and Consolidating the Analysis Model

Now that we have identified most participating objects, their associations, and their attributes, we draw UML class diagrams documenting the results of our analysis so far. As we have identified many objects, we use several class diagrams to depict the analysis object model. We

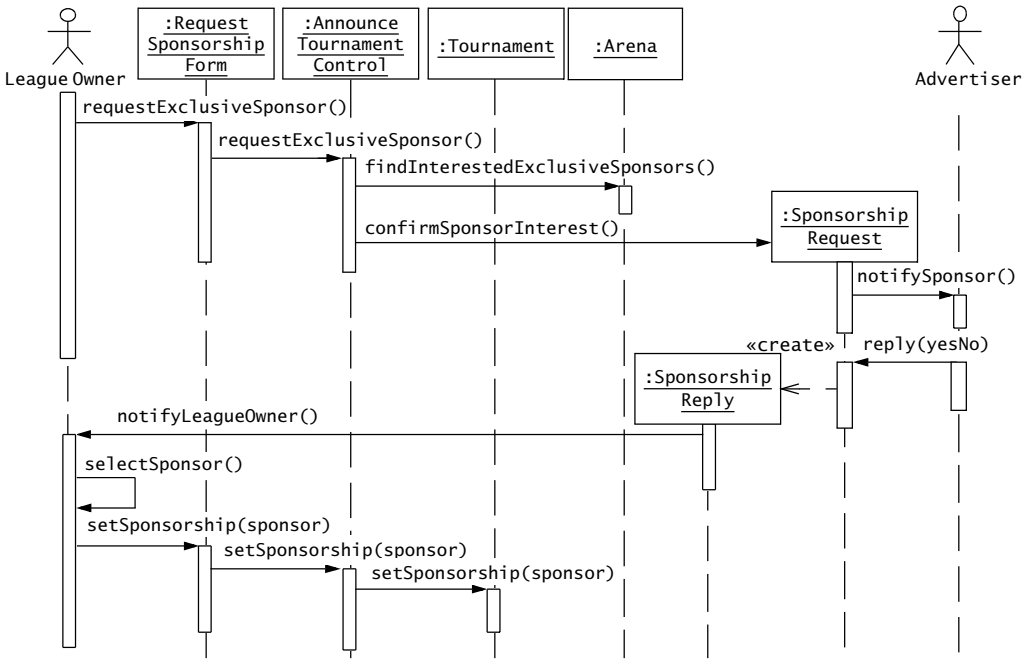



**Figure 5-27** UML sequence diagram for `AnnounceTournament` use case, sponsorship workflow.

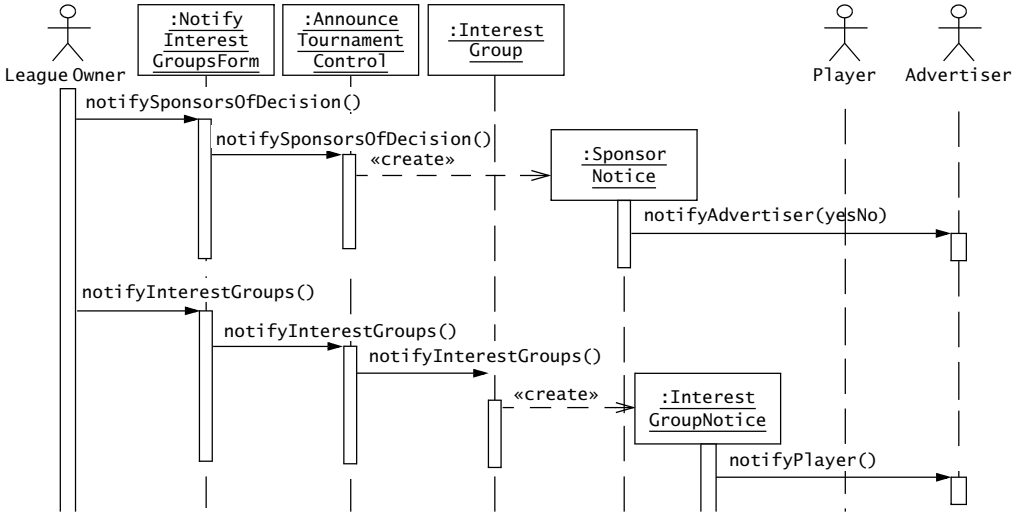



**Figure 5-28** UML sequence diagram for `AnnounceTournament` use case, interest group workflow.

use these class diagrams as a visual index into the glossary we developed. Although we should not expect the client or the users to be able to review class diagrams, we can use class diagrams for generating more questions for interviews with the client.

We first focus on the entity objects, since these need to be carefully reviewed by the client as they represent application domain concepts (Figure 5-29). Note that we use the `Arena` class as a root object in the system; the `Arena` class represents a specific instantiation. For example, given an instantiation, it is possible to get a list of all `InterestGroups`, `Advertisers`, `LeagueOwners`, `Games`, and `TournamentStyles` by querying the `Arena` class. Moreover, note that objects are not shared among instantiations. For example, `LeagueOwners` belong to exactly one instantiation of the system. If a user is a `LeagueOwner` in several `ARENA` instantiations of the system, she holds a `LeagueOwner` account in each instantiation. We make these type of choices during analysis based on our interpretation of the problem statement, based on our experience, and based on resources available to build in the system. In all cases, these decisions need to be reviewed and confirmed by the client.

Next, we draw a class diagram depicting the inheritance hierarchies (Figure 5-30). Although UML allows inheritance relationships and associations to coexist in the same diagram, it is good practice during analysis to draw two separate diagrams to depict each type of

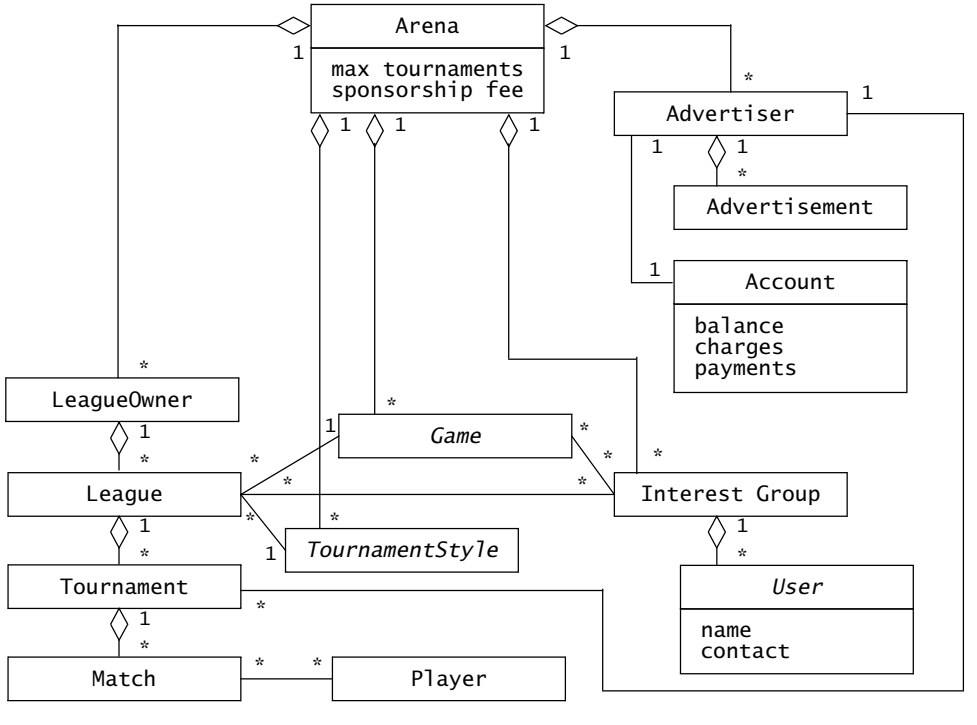



**Figure 5-29** Entity objects identified after analyzing the `AnnounceTournament` use case.

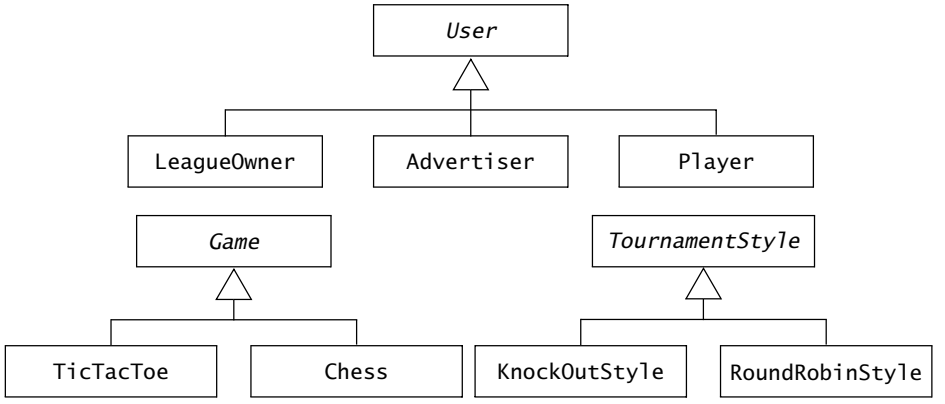



**Figure 5-30** Inheritance hierarchy among entity objects of the `AnnounceTournament` use case.

relationship. First, the UML symbols used to denote each type are similar and easily confused. Second, analysts usually focus on inheritance and associations at different times. We will see later, in Chapters 6 through 10, that this is not the case during system design and object design, where it is often necessary to consider both relationships to understand how different classes are related.

Figure 5-30 shows three inheritance hierarchies. First, we identified an abstract class *`User`* through generalization. This enables us to treat common attributes of various users in a more general fashion, including contact information and registration procedures. Note that in the problem statement and in the use cases, we already used the term "user," so we are simply formalizing a concept that was already in use. We identified two other inheritance hierarchies, *`Game`* and *`TournamentStyle`*, identified through specialization to provide examples for both concepts and to provide traceability to the problem statement. The `TicTacToe` and the `Chess` classes are concrete specializations of *`Game`* that embody rules for the games called "tic tac toe" and "chess," respectively. The `KnockOutStyle` and the `RoundRobinStyle` classes are concrete specializations of the *`TournamentStyle`* that provide algorithms for assigning `Players` to knock-out tournaments (in which players need to win to remain in the tournament) and round robin tournaments (in which each player plays all other players exactly once), respectively.

Finally, we draw a class diagram that depicts the associations among the boundary, control, and selected entity objects associated with the use case (Figure 5-31). To generate this diagram from the sequence diagrams, we draw the equivalent communication diagram, with the control object to the left, the boundary objects in the center, and the entity objects on the right. We then replace the iterations with associations, where necessary, so that the objects in the workflow can carry send messages to objects depicted in the sequence diagrams. We then add navigation to the associations to denote the direction of the dependencies: control and boundary

objects usually know about each other, but entity objects do not depend on any control or boundary objects.

Whereas the class diagram in Figure 5-29 focused primarily on the relationships among application domain concepts, the class diagram of Figure 5-31 focuses on the concepts associated with workflow of the use case at a coarse level. The control object acts as the glue among boundary and entity objects, since it represents the coordination and the sequencing among the forms and notices. As indicated in the sequence diagrams in Figures 5-26 through 5-28, the control object also creates several of the boundary objects. The class diagram in Figure 5-31 provides a summary of the objects participating in the use case and the associations traversed during the use case. However, the sequence diagrams provide the complete sequencing information of the workflow.

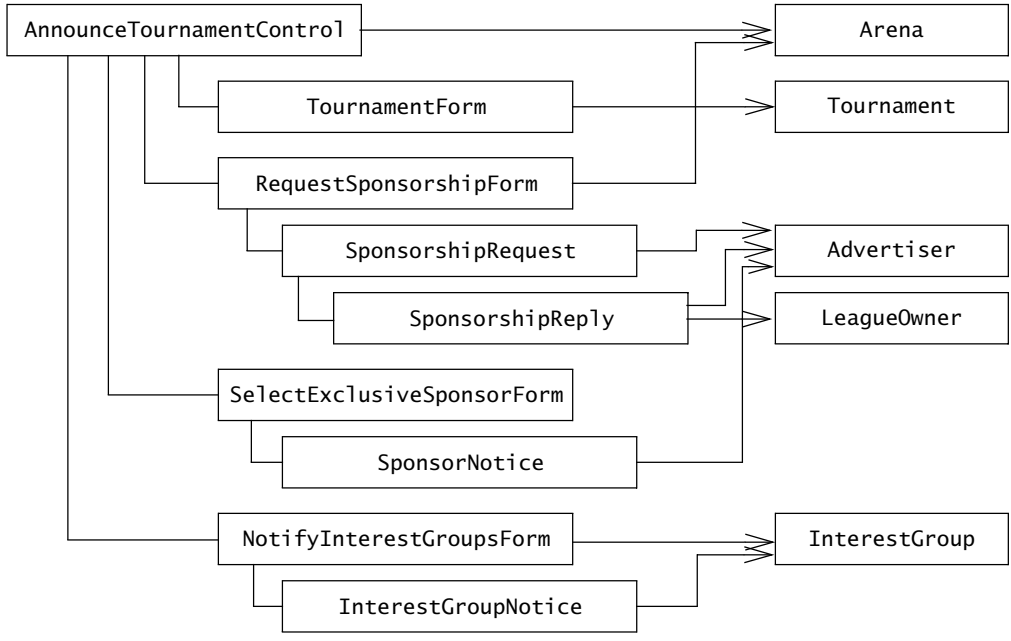



**Figure 5-31** Associations among boundary, control, and selected entity objects participating in the `AnnounceTournament` use case.

### 5.6.6 Lessons Learned

In this section, we developed the part of the analysis object model relevant to the `AnnounceTournament` use case of `ARENA`. We started by identifying entity objects using Abbott's heuristics, then identified boundary and control objects, and used sequence diagrams to find additional associations, attributes, and objects. Finally, we consolidated the object model and depicted it with a series of class diagrams. We learned that:

- Identifying objects, their attributes and associations, takes many iterations, often with the client.
- Object identification uses many sources, including the problem statement, use case model, the glossary, and the event flows of the use cases.
- A nontrivial use case can require many sequence diagrams and several class diagrams. It is unrealistic to represent all discovered objects in a single diagram. Instead, each diagram serves a specific purpose—for example, depicting associations among entity objects, or depicting associations among participating objects in one use case.
- Key deliverables, such as the glossary, should be kept up to date as the analysis model is revised. Others, such as sequence diagrams, can be redone later if necessary. Maintaining consistency at all times, however, is unrealistic.
- There are many different ways to model the same application domain or the same system, based on the personal style and experience of the analyst. This calls for developing style guides and conventions within a project, so that all analysts can communicate effectively.

## 7.6 ARENA Case Study

In this section, we apply the concepts and methods described in this chapter to the ARENA system. We start with identifying the design goals for ARENA and design an initial subsystem decomposition. We then select a software and hardware platform and define the persistent stores, access control, and global control flow. Finally, we look at the boundary conditions of ARENA.

### 7.6.1 Identifying Design Goals

Design goals are qualities that enable us to prioritize the development of the system. Design goals originate from the nonfunctional requirements specified during requirements elicitation and from technical and management goals specified by the project.

In ARENA, the main client is the ArenaOperator, who provides the resources for setting up an Arena for a particular community. ArenaOperators are themselves Players who may have system administration or even programming skills. The advertisement features allow them to recoup some of their costs. Moreover, we anticipate that ArenaOperators will form a community and that the integration of new games into ARENA and improvements to ARENA will be mostly contributed by ArenaOperators. However, advertisement is not the main purpose of ARENA. From these observations and from the ARENA problem statement (Figure 4-17), we identify the following design goals:

- *Low operating cost*. To minimize the need for advertisement, the cost of running the system (e.g., hardware resources, network resources, administration costs, etc.) should be minimized. This also leads us to select free or open-source components. This design goal is a refinement of the nonfunctional requirement "low operating cost" of the ARENA problem statement (Figure 4-17).
- *High availability.* The value of an Arena increases with the number of players available for playing tournaments. Unexpected crashes and interruptions in tournaments will create a lot of frustration for the players and discourage them from attending other tournaments. This design goal is not explicitly stated in the problem statement or the requirements, but is necessary if an Arena is to attract and keep a sufficiently large number of players.
- *Scalability in terms of number of players and concurrent tournaments*. The response time of the Arena may not degrade dramatically with the number of Players. When needed, an ArenaOperator should have the option of increasing the capacity of an Arena by adding hardware nodes. This design goal is a refinement of the nonfunctional requirement "scalability" in the ARENA problem statement (Figure 4-17).
- *Ease of adding new games*. Some games, such as chess, are timeless. However, the computer game industry evolves with different fashions and hardware improvements. Consequently, to keep an Arena active, it should be relatively easy to adapt and install

new games. This design goal is a refinement of the nonfunctional requirement "extensibility" in the ARENA problem statement (Figure 4-17).

- *Documentation for open source development*. The organization and documentation of the ARENA game framework should then make it easier for new developers to contribute features to the code. This includes source code documentation that supports low-level changes and improvements, as well as a good architecture-level documentation that supports the addition of new features. This design goal originated from the developers and management of ARENA (as opposed to the client). Note that such design goals may require additional interaction with the client, as they might interfere with implicit client goals that have not yet been made explicit.

### 7.6.2 Identifying Subsystems

We first identify subsystems from the functional requirements of ARENA and from the analysis model. The purpose of this activity is to divide the system in self-contained components that can be managed by individuals. As we address other design issues, such as access control and persistency management, we will refine or modify this initial subsystem decomposition.

We first distinguish two main parts of the ARENA subsystem: the game organization part of the system, which is responsible for coordinating Users when organizing an Arena, a League, or a Tournament, and the game playing part, in which Players conduct individual Matches in the scope of a Tournament.

For the game organization part, we select a three-tier architectural style (Figure 7-19) in which an ArenaClient subsystem provides a front end for users to initiate all organization-related use cases (e.g., AnnounceTournament, ApplyForTournament, RegisterPlayer). The ArenaServer subsystem is responsible for access control and concurrency control, and delegates to nested subsystems for the application logic. Different subsystems are dedicated to the user management of users, advertisements, tournaments, and games. The bottom tier is realized by the ArenaStorage subsystem, responsible for storing any persistent objects, except for those representing Match states.

For the game playing part, the client server architecture may not be sufficient for synchronous games in which the action of one player can trigger events for another player within a relatively short time. Synchronous behavior could be simulated with polling; however, because of scalability and responsiveness goals, we select a peer-to-peer architecture in which MatchFrontEndPeer subsystems provide the user interface and a GamePeer maintains the state of the matches currently under way and enforces the game rules. MatchFrontEndPeers may also communicate directly with each other for real-time games. To achieve the game independence design goal, ARENA provides a framework for both the MatchFrontEndPeer and the GamePeer, while the bulk of the game logic is provided by customized game-dependent components. Adding a game consists of developing adapters for existing games or ARENA-compliant components for new games. The TournamentManagement subsystem uses the GameManagement subsystem to initiate a GamePeer and to collect the results of the individual Matches. The

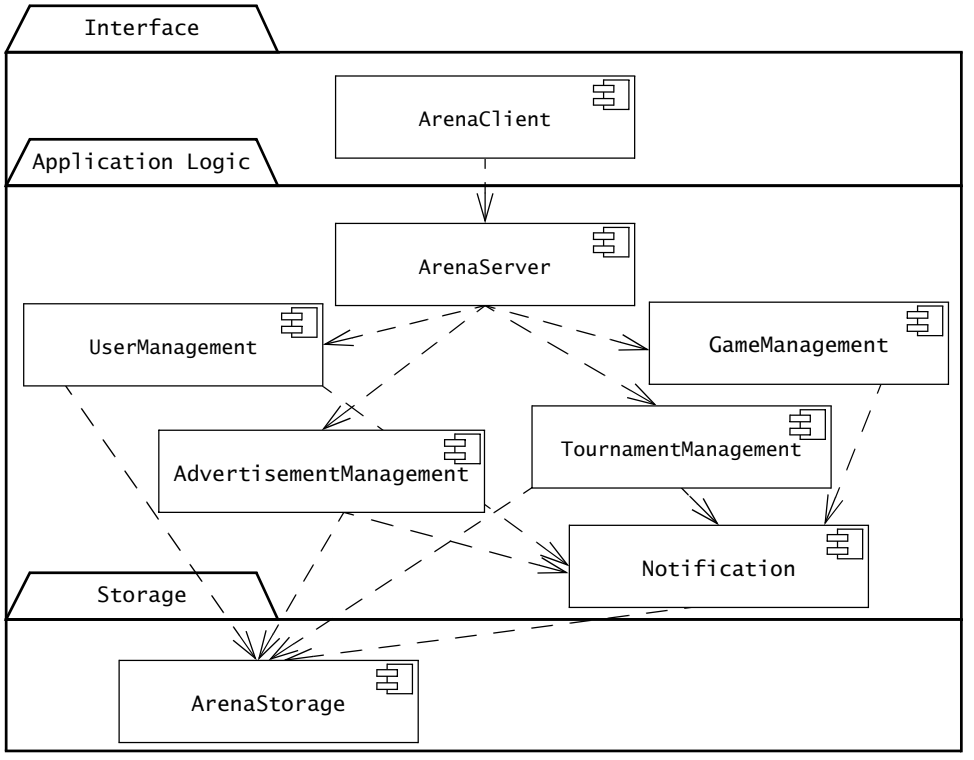



**Figure 7-19** ARENA subsystem decomposition, game organization part (UML component diagram, layers shown as UML packages).

MatchFrontEndPeer uses the AdvertisementManagement subsystem to retrieve Advertisements (Figure 7-20). Note that for turn-based games, a client server architectural style would be sufficient, as the response time for such games is less critical. The selection of the peer-to-peer style does not prevent specific games from following a client server style.

### 7.6.3 Mapping Subsystems to Processors and Components

Mapping subsystems to processors and components enables us to identify potential concurrency among subsystems and to address performance and reliability goals.

ARENA is inherently a distributed system, as users sit in front of different machines, possibly several time zones apart. However, we distinguish only between two types of nodes: the UserMachine to provide a user interface and the ServerMachines to run the application logic and storage and, more generally, to provide the ARENA services. ArenaClient and the MatchFrontEndPeer subsystems run on the UserMachine. In an installation of ARENA with few

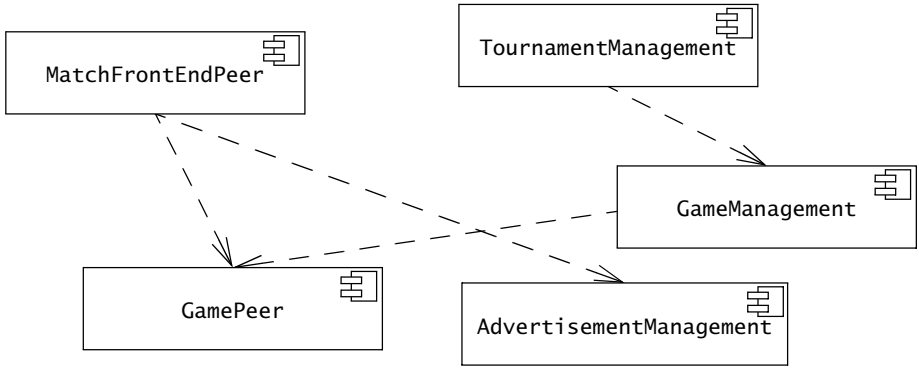



**Figure 7-20** ARENA subsystem decomposition, game playing part (UML component diagram).

players, all other subsystems can be collocated onto a single ServerMachine. However, to ensure scalability, we identify an additional subsystem dedicated to send advertisement banners to the browser, and assign the AdvertisementServer, the GamePeer, the ArenaStorage, and the ArenaServer subsystems to different processes that can run on different ServerMachines. The ArenaServer component includes the nested, TournamentManagement, UserManagement, and GameManagement subsystems (Figure 7-21).

For the realization of the game organization part of ARENA, we select the Java EE framework. Java EE is a collection of interfaces and standards developed by Sun Microsystems and community efforts for developing portable web-based information systems in Java. The

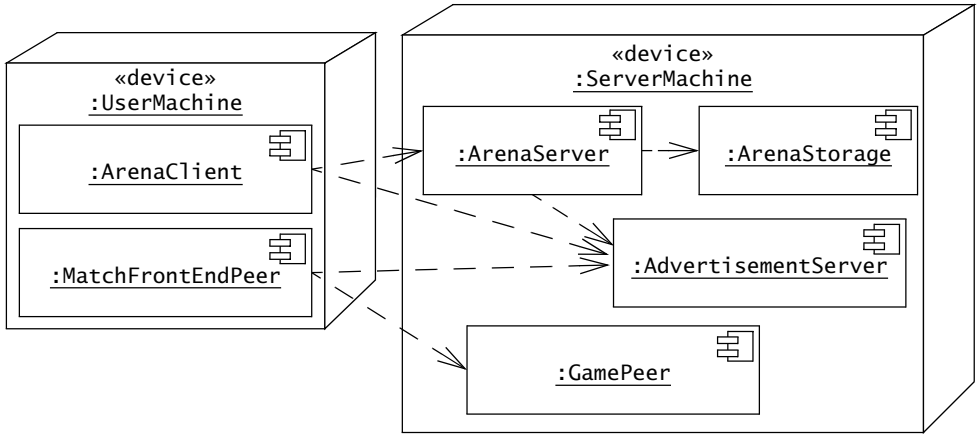



**Figure 7-21** ARENA hardware/software mapping (UML deployment diagram). Note that each run-time component may support several subsystems.

advantage of this standard is that it is implemented by several open-source and commercial products, thus enabling an ArenaOperator to trade off scale (e.g., number of players, tournaments, and leagues) for set-up cost (e.g., licensing and run-time costs). Also, in their simplest form, open-source components of this framework are easy to install and require little prior administration knowledge.

Consequently, the ArenaClient is a Web browser and the ArenaServer and the other game organization subsystems are accessed through a Web server. To realize the ArenaServer and related subsystems, we select Java Servlets and Java Server Pages (JSP), components of Java EE, as the main technology for implementing the boundary objects. Servlets are classes that are located on the ServerMachine. Servlets receive, process, and respond to requests from a Web browser by generating an HTML page. JSPs provide a compact way of specifying servlets using a language similar to HTML. A preprocessor then generates a servlet from a JSP. We use JSPs to realize the boundary and control objects of ARENA. JSPs in turn invoke methods on entity objects and storage objects, which are also realized using the Java Foundation classes.

Having identified the subsystems, the concurrency, and the hardware/software mapping, we now turn our attention to persistency management.

### 7.6.4 Identifying and Storing Persistent Data

***Identifying persistent objects***

ARENA deals with two sets of objects that must be stored. The first set includes the objects that are created and accessed by the game organization subsystems (e.g., Tournament, Game, Match, Player) and that need to be persistent to track the progress of Leagues, Matches, Tournaments, and their Players. The second set includes the objects that are created and accessed by the GamePeer and the MatchFrontEndPeer during Matches, which are used to replay matches for Spectators and to resume Matches that were interrupted by system crashes. The first set of objects is well defined and will probably not change much during the lifetime of ARENA. The second set of objects are specific to each Game and are defined by the Game developers. Hence, we decide to manage the first set of persistent objects with the ArenaStorage subsystem of ARENA and let game developers decide how to manage the state of the Matches in game-specific components. The persistent objects of each Game will then only be accessed through a generic Game interface implemented by each individual Game.

***Selecting a storage strategy***

Selecting a persistent storage strategy during system design enables us to deal with other issues related to storage management, such as concurrency control and crash recovery. For example, many database management systems allow concurrent queries and provide transaction mechanisms to ensure data integrity.

Our highest priority design goal in ARENA is to minimize operating costs, so we first consider using flat files for storage. Such a system can be installed easily, since there is no database management system to configure or to manage. However, a system based solely on flat files would not scale to large installations with dozens of games and thousands of players.

To accommodate both goals, we select a mixed strategy. The storage subsystem will provide an abstract interface that enables both a flat file and a relational database implementation. When installing an Arena the first time, the ArenaOperator selects which implementation fits the goals best. The ArenaOperator will not be able to switch strategies at run time, but will be able to convert persistent objects from flat files to the database and back during system reconfiguration. This will increase the development cost of ARENA, but provide more flexibility to the ArenaOperator. To reduce development risks, the initial prototype of ARENA will only use flat files. A second prototype will use a database-independent API (e.g., JDBC [JDBC, 2009]) to store persistent objects in a relational database, thus enabling ArenaOperators to use different relational products.

Game developers will address game storage issues individually. Given the sequential nature of game data, we anticipate games to use flat files for storage as well.

### 7.6.5 Providing Access Control

As ARENA is a multi-user system, different actors are allowed to view different sets of objects and invoke different types of operations on them. To succinctly document access rights, we draw an access control matrix (Table 7-4) depicting the allowed operations on the entity objects for each actor. In summary, ArenaOperator can create Users and Leagues, LeagueOwners can create Tournaments and Matches. Advertisers can upload and remove advertisements as well as apply for sponsorship of Leagues or Tournaments. Note that the LeagueOwner makes the final sponsorship decision, as documented in the analysis. Players can subscribe to a League (to receive announcements), apply for a Tournament, and play Matches they have been scheduled for. Finally, Spectators can view player statistics, view League and Tournament schedules, and subscribe to receive notifications and view Matches.

Note that most of the access control information is already available in the use case model. The access control matrix, however, presents a more detailed and compact view, thus enabling a client to review access control more easily and a developer to implement it correctly.

Spectators are actors that are not authenticated to the system. All other actors must first authenticate before they can modify any object in the system. We select a user name/password mechanism for initiating sessions. We then use access control lists on each object (e.g., Leagues, Tournaments, and Matches) to check the access privileges of the user. A Session object per authenticated user tracks currently logged in users.

**Table 7-4** Access matrix for main ARENA objects.

| Objects / Actors | Arena | User | League | Tournament | Match |
|---|---|---|---|---|---|
| ArenaOperator | «create»<br>createUser | «create»<br>deactivate | «create»<br>archive | | |
| LeagueOwner | | getStats<br>getInfo | archive<br>setSponsor | «create»<br>archive<br>setSponsor | «create»<br>end |
| Advertiser | uploadAds<br>removeAds | | apply for<br>sponsorship | apply for<br>sponsorship | |
| Player | apply for<br>LeagueOwner | setInfo | view<br>subscribe | apply for<br>tournament<br>view<br>subscribe | play<br>end |
| Spectator | apply for<br>Player<br>apply for<br>Advertiser | getStats | view<br>subscribe | view<br>subscribe | subscribe<br>replay |

### 7.6.6 Designing the Global Control Flow

As described in Section 7.4.4, there are three types of control flow paradigms: procedure-driven, event-driven, or threaded control flow paradigm. The selection of one paradigm over another depends on response time and throughput requirements on the system and development complexity. Moreover, in a system with multiple components, it is possible to select different control paradigms for different components.

In Sections 7.6.3 and 7.6.4, when selecting components for the interface and storage subsystems of ARENA, we effectively restricted the alternatives for control flow mechanisms for the game organization part. The WebServer waits for requests from the WebBrowser. Upon the receipt of a request, the WebServer processes it and dispatches it to the appropriate servlet or JSP, thus resulting in an event-based control flow. The WebServer allocates a new thread for each request, allowing the parallel handling of requests. This results in a more responsive system by enabling the WebServer to respond to individual WebBrowser requests before other requests have been completely processed, and can increase throughput by enabling the processing of one request while another is waiting for the database to respond. The drawback of threads is the higher complexity of the system resulting from synchronizing parallel threads. To ensure a robust design with respect to concurrency, we define the following strategy for dealing with concurrent accesses to shared data:

- *Boundary objects should not define any fields*. Instead, boundary objects hold temporary data associated with the current request in local variables. As boundary objects are shared among threads, this prevents concurrency hazards at that level.
- *Control objects should not be shared among threads*. Instead, there should be at most one control object associated with each session, and users should not be able to issue concurrent requests involving the same control object within the same session. This should especially be enforced when control objects survive the processing of a single request.
- *Entity objects should not provide direct access to their fields*. Instead, all changes and accesses to the object state should be done through dedicated methods. Moreover, these methods should only access the fields of the receiver object (i.e., `this`), and not of other instances of the same class. If classes are realized as abstract data types (see Section 2.3.2), all fields should already be private.
- *Methods for accessing state in entity objects should be synchronized*. That is, the synchronized mechanism provided by Java should be used so that only one thread at a time can be active in the access method.
- *Nested calls to synchronized methods should be avoided*. Developers of synchronized methods should investigate if a nested method call can result in calling another synchronized method. This could lead to deadlocks and should be avoided. If such nested calls cannot be avoided, developers should either reallocate class behavior among methods to avoid such nested calls, or impose a strict ordering of synchronized method calls.
- *Redundant state should be time-stamped*. The state of an object can occasionally be duplicated. One example of duplication is when the state of an object is stored in a Web form in the `WebBrowser` and in storage subsystem. To detect situations in which concurrent changes to the same object can lead to a conflict, a time-stamp should be added to the duplicated data to represent the last modification time.

For the game part of `ARENA`, the `MatchFrontEndPeer` and the `GamePeer` run in separate processes. These processes are started by the `GameManagement` subsystem as required by `Tournament` schedules and `Player` attendance. The internal control flow of the `MatchFrontEndPeer` and the `GamePeer` can be event driven or threaded, depending on the needs of the specific game.

### 7.6.7 Identifying Services

We have now defined a subsytem decomposition and decided on the control flow paradgms for each run-time component and for the system as a whole. We are now ready to identify services provided by each subsystem.

In this example, we focus on the dependencies among ArenaServer, UserManagement, AdvertisementManagement, and TournamentManagement, in the context of the Organize-Tournament use case (Figure 4-24).

We first notice that all requests handled by the ArenaServer must be authorized according to the access control policy defined in Section 7.6.5. This leads us to define an Authentication service to check a user's credentials upon login, and an Authorization service to check if the request is allowed for the role of the requesting user. We assign both services to UserManagement.

During the first steps in the OrganizeTournament use case, the LeagueOwner creates a Tournament in the context of a League. TournamentManagement therefore needs to provide services for creating and getting information about Tournaments and Leagues. These services are trivial but are needed for every class in the analysis model. To denote that the TournamentManagement subsystem owns the Tournament and League classes, we define the Tournament and League services. In practice, these types of services are left implicit in the model to avoid overcrowding, as they add little information.

In the next steps, the LeagueOwner invites Advertisers to sponsor the new Tournament. We add the Sponsorship service to AdvertisementManagement, allowing the LeagueOwner to invite Advertisers and Advertisers to respond. Defining both sets of actions in the same service allows us to keep all operations related to sponsorship in one place.

When the LeagueOwner selects Advertisers to sponsor a Tournament, we are presented with the option of defining a new service either in AdvertisementManagement or in TournamentManagement, as each subsystem owns an end of the Advertiser-Tournament association. We decide to assign the responsibility to track the state of the OrganizeTournament activity to TournamentManagement, as the use case centers on the definition of Tournaments. In analysis terms, TournamentManagement owns the control object associated with OrganizeTournament. Consequently, we add a SponsorSelection service to the TournamentManagement subsystem.

In the next steps of OrganizeTournament, the LeagueOwner advertises the Tournament and interested Players apply and are selected. We are faced with the question of allocating the Player service to the UserManagement or the TournamentManagement subsystem. We decide to assign Player to the TournamentManagement, as Player is strongly connected with League and Tournament. As accepting Players in Tournaments goes beyond simply creating Players and getting information about them, we define a PlayerAcceptance service to support this step.

Figure 7-22 depicts the services identified so far. We observe two trends during the above discussion:

- When trading off services between two subsystems, functionality tends to aggregate in the subsystem where the control object corresponding to the use case is defined. While this results in subsystem with high coherence, we need to be careful that the resulting complexity of the subsystem is not too high.

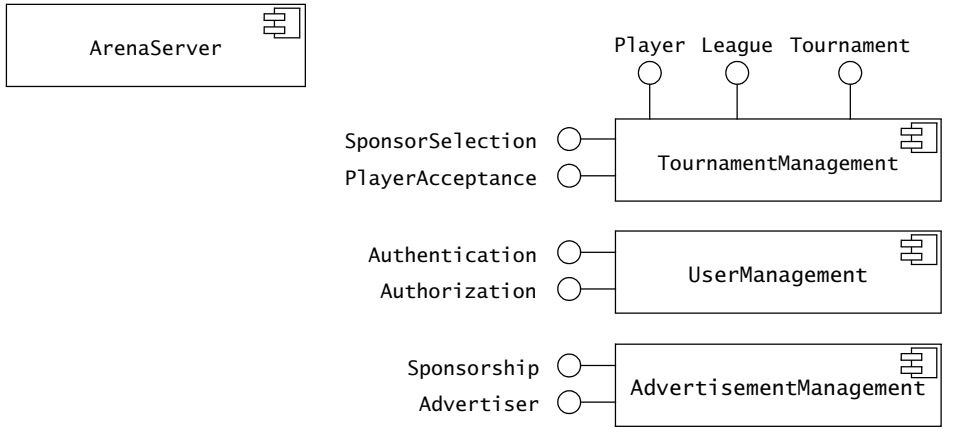



**Figure 7-22** ARENA subsystem decomposition, game organization part with services identified (UML component diagram, ball-and-socket notation, dependencies omitted for clarity).

- Defining services based on steps in a use case tend to result in fine grained services. While this allows us to validate the subsystem decomposition, it may result in many interfaces that each define a single operation. This is a sign that we are moving too quickly towards object design. During a second pass, we may choose to consolidate several related services into single services to keep the design at the architectural level of abstraction and the subsystem decomposition understandable. Naming services with noun phrases that denote a collection of operations also helps us in avoiding this pittfall.

### 7.6.8 Identifying Boundary Conditions

During this activity, we review the design decisions we made so far and identify additional administrator use cases. We first examine the life time of the persistent objects of ARENA, the life time of each run-time component, and the types of system failures.

***Configuration use cases***

The handling of most persistent objects is already described in the use cases developed during analysis (Figure 4-21) and in the access control matrix (Table 7-4). For example, ArenaOperators create and deactivate Users. LeagueOwners create and archive Leagues and Tournaments. Players initiate and end Matches. Advertisers manage Advertisement Banners. However, the handling of the Arena and Game objects has not been described in the use case model so far, as these objects have been refined during system design. Arena is created with the installation of the system. Games are created and destroyed whenever Games are added or deleted from the system. Hence, we identify two additional use cases invoked by the

ArenaOperator, InstallArena and ManageGames. Moreover, we decided, when discussing persistent storage (Section 7.6.4), that an ArenaOperator could convert the persistent objects between a flat file representation and a database representation. This results in an additional configuration use case, ConvertPersistentStorage (Table 7-5).

**Table 7-5** Additional ARENA boundary use cases identified when reviewing persistent objects.

| | |
|---|---|
| **InstallArena** | The ArenaOperator creates an Arena, gives it a name, selects a persistent storage subsystem (either flat file or database), and configures resource parameters (e.g., maximum number of concurrent tournaments, file path for storage). |
| **ManageGames** | The ArenaOperator installs or removes a Game, including custom code for the GamePeer and MatchFrontEndPeer. The list of Games is updated for the next time a LeagueOwner creates a League. |
| **Convert Persistent Storage** | When the ArenaServer is shut down, the ArenaOperator can convert the persistent storage from a flat file storage to a database storage or from a database storage to a flat file storage. |

### *Start-up and shutdown use cases*

As depicted in the UML deployment diagram in Figure 7-21, ARENA includes five run-time components: the WebBrowser, the ArenaServer (which includes the subsystems UserManagement, GameManagement, TournamentManagement, Notification, and Arena-Storage), the MatchFrontEndPeer, the GamePeer, and for the second prototype, the DatabaseServer. The WebBrowser and the DatabaseServer are off-the-shelf components and are started and shut down individually. The MatchFrontEndPeer and the GamePeer are started and shut down by the WebBrowser and the ArenaServer, respectively. The start-up and shutdown of ArenaServer is currently not described in the use case model. Hence, we identify two additional use cases invoked by the ArenaOperator (Table 7-6).

**Table 7-6** Additional ARENA boundary use cases identified when reviewing runtime components.

| | |
|---|---|
| **StartArenaServer** | The ArenaOperator starts the ArenaServer. If the server was not cleanly shut down, this use case invokes the Check Data Integrity use case described in the next section. As soon as the initialization of the server is complete, LeagueOwners, Players, Spectators, and Advertisers can initiate any of their use cases. |
| **ShutDownArenaServer** | The ArenaOperator stops the ArenaServer. The server terminates any ongoing Matches and stores any cached data. MatchFrontEndPeers and GamePeers are shut down. Once this use case is completed, the LeagueOwners, Players, Spectators, and Advertisers cannot access or modify the Arena. |

### *Exception use cases*

ARENA can experience four major classes of system failures:

- A network failure in which one or more connections among MatchFrontEndPeers and GamePeers are interrupted
- A host or a component failure in which one or more MatchFrontEndPeers or GamePeers are unexpectedly terminated
- A network failure in which one or more connections between a WebBrowser and the ArenaServer are interrupted
- A server failure in which the ArenaServer is unexpectedly terminated.

We decide to handle the first two classes of exceptions in the custom Game components. We will provide generic methods for MatchFrontEndPeers and GamePeers to re-establish connection after a network failure or to restore the state of Matches after a crash. However, the handling of the exception itself depends on the type of game. For example, a real-time simulation game will not tolerate network failures and should be interrupted or restarted, whereas a board game can tolerate short interruptions transparently from its Players. Hence, we leave the flexibility to the game developers to decide how to handle those exceptions.

We handle network failures interrupting connections between the WebBrowser and the ArenaServer by notifying the user of the network failure, similar to current WebBrowsers. We expect the actors to retry later, at the cost of loosing the data that was already entered in a form. Consequently, we will design forms in such a way that little data can be lost in any one failure.

We decide to handle the last type of failure by a use case for checking the integrity of the persistent data after an unexpected termination of the ArenaServer (see Table 7-7). This use case can be invoked automatically by the system upon start-up (see StartArenaServer in Table 7-6) or manually by the ArenaOperator. We also identify an additional use case to restart interrupted GamePeers and notify relevant players.

**Table 7-7** Additional ARENA boundary use cases identified when reviewing persistent objects.

| | |
|---|---|
| **CheckDataIntegrity** | ARENA checks the integrity of the persistent data. For file-based storage, this may include checking if the last logged transactions were saved to disk. For database storage, this may include invoking tools providing by the database system to re-index the tables. |
| **RestartGamePeers** | ARENA starts any interrupted Matches and notifies any running MatchFrontEndPeer that GamePeer is back on-line. |

### 7.6.9 Lessons Learned

In this section, we examined the system design issues for the ARENA system. We identified and prioritized design goals, we decomposed the system into subsystems, we mapped the subsystems to components and platforms, we selected a persistent data storage strategy, we described the access control mechanisms for the system, we examined control flow issues, and we identified use cases for handling boundary cases. We learned that

- Most system design issues are interrelated. For example, selecting a component for boundary or storage objects has implications on the global control flow of the system.
- Some system design issues have different solutions in different parts of the system. For example, we dealt with issues related to architecture, control flow, crash recovery, and storage issues differently for the organization part than for the game playing part of ARENA.
- Some system design issues can be postponed until the object design phase (e.g., decisions about `GamePeers`) or to a later release (e.g., storage implementation).
- In all cases, design goals serve to prioritize and evaluate different design alternatives.

## 8.6 ARENA Case Study

In this section, we apply three design patterns to the object design of ARENA. As specified during requirements analysis, we anticipate ARENA to support many different types of games. Hence, in this section, we focus on the classes related to Games, Matches, and their respective boundary objects. In particular, we focus on the following design patterns:

- *Abstract Factory design pattern* (Section 8.6.1). We shield the Tournament and League objects from Game specifics by turning the Game abstract interface into an AbstractFactory. This way, only concrete Products need to be supplied for a new concrete Game.
- *Command design pattern* (Section 8.6.2). We shield the objects related to playing and replaying Matches by encapsulating concrete Moves for each Game.
- *Observer design pattern* (Section 8.6.3). We standardize the interactions between Match and Move entity objects with MatchView objects across all Games with a subscriber/publisher paradigm.

In this section, we only focus on Games that involve a sequence of Moves performed by Players who take turns. We do not consider Games that involve simultaneous or concurrent actions at this point.

### 8.6.1 Applying the Abstract Factory Design Pattern

Achieving game independence in ARENA is not as straightforward as it initially appears. In the analysis model (see Figure 8-19), we define an abstract Game interface to shield the Tournament and League objects from the specifics of each game. However, supporting a specific Game

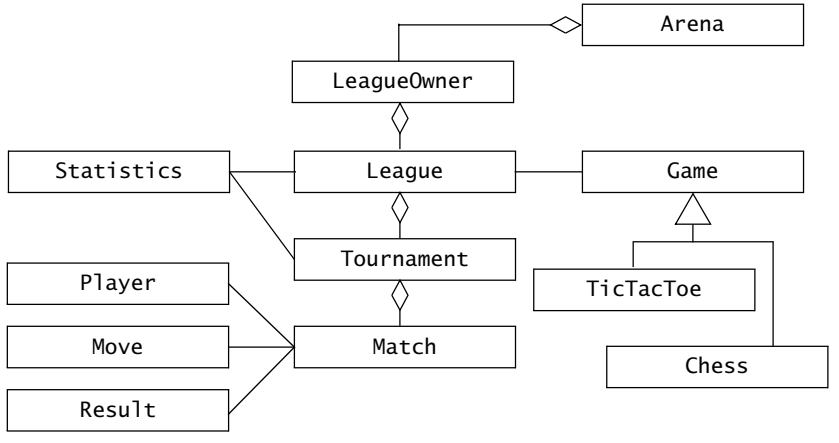



**Figure 8-19** ARENA analysis objects related to Game independence (UML class diagram).

involves the tight collaboration of several objects representing the rules of Game, the state of a Match under progress, specific Moves played by the different contenders, and the Result of the Match. At the end of a Match, the Results are accumulated in Tournament and League-wide Statistics. We need to define a Game framework within ARENA that shields the Tournament and League objects from all the Game specifics, while supporting standard interactions among specialized Game, Match, Move, Result, and Statistics objects.

As several specialized objects need to collaborate, we first select the **Abstract Factory design pattern** to address the game independence design issue (see Figure 8-20). The abstract Game interface is an abstract factory that provides methods for creating Matches and Statistics. Each concrete Game (e.g., TicTacToe and Chess) realizes the abstract Game interface and provides implementations for the Match and Statistics objects. For example, the TicTacToe Game implementation returns TTTMatches and TTTStats objects when the createMatch() and the createStatistics() methods are invoked. The concrete Match objects (e.g., TTTMatch and ChessMatch) track the current state of the Match and enforce the Game rules. Each concrete Game also provides a concrete Statistics object for accumulating average statistics (e.g., average Match length, average number of Moves, number of wins and losses per player, as well as Game specific Statistics). The League and the Tournament objects each use a concrete Statistics object to accumulate statistics for the League and the Tournament scope, respectively. Because the League and Tournament objects only access the abstract Game, Match, Statistics interfaces, the League and Tournaments work transparently for all Games that comply with this framework.

### 8.6.2 Applying the Command Design Pattern

Although Spectators can watch Matches as they occur, we anticipate that many Matches will be viewed asynchronously, after the fact. Hence, we need to store the sequence of moves in each Match, so that it can be replayed at a later time.

As described in Section 8.4.5, we apply the **Command design pattern** and represent each move as a Command object. The abstract Move object (corresponding to the Command object in the design pattern) provides the interface to the League and Tournament objects to manipulate Moves independently from the concrete Games and Matches. The concrete Moves are created by and stored in a queue in the concrete Match object (Figure 8-21).

To deal with concurrent Spectators replaying the same Match, we need to refine this solution further. For each request to replay an archived Match, ARENA creates a new ReplayedMatch that includes its own GameBoard to hold the current state of the replayed Match and feeds it the Move objects of the archived Match, one at the time. This enables the same Match to be replayed by many different Spectators independently.

### 8.6.3 Applying the Observer Design Pattern

ARENA supports multi-player games, such as TicTacToe and Chess. Each Player accesses a Match in progress through a client application running on his local machine. Consequently,

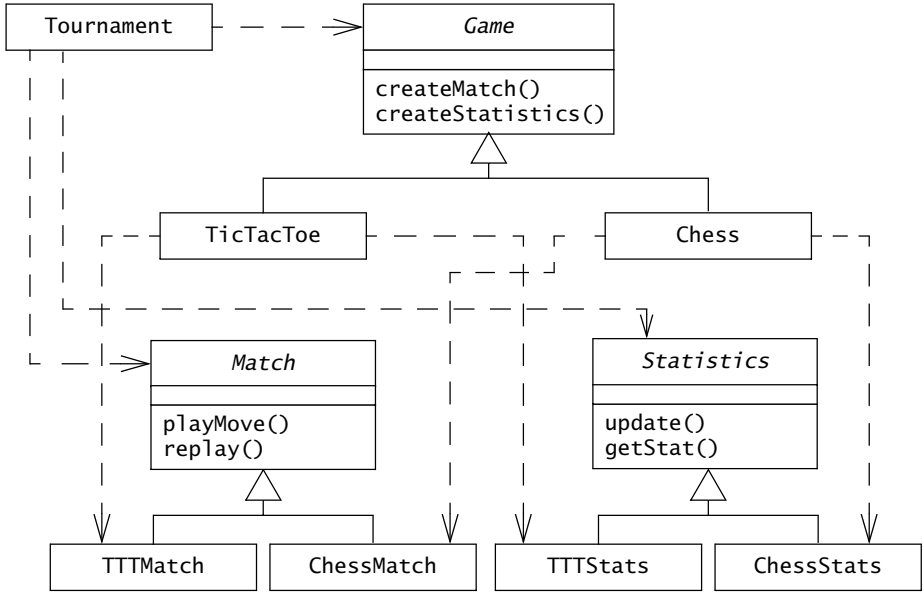



**Figure 8-20** Applying the Abstract Factory design pattern to `Games` (UML class diagram).

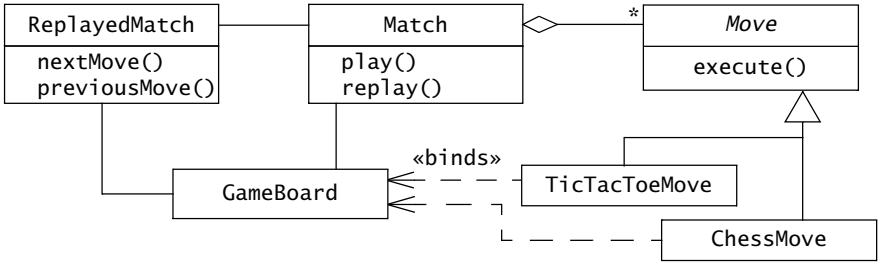



**Figure 8-21** Applying the Command design pattern to `Matches` and `ReplayedMatches` in `ARENA` (UML class diagram).

potentially many views of the same `Match` in progress must be kept consistent. To address this problem, we use a distributed version of the **Observer design pattern** (Figure 8-22), in which the `Observers` are the boundary objects in each client, the `Subjects` are the `GameBoard` objects that maintain the current state of each `Match`. References between `Subjects` and `Observers` are remote object references provided by Java RMI. In addition to maintaining consistency among different views of the same `Match`, this enables us to use the same pattern for `ReplayedMatches`.

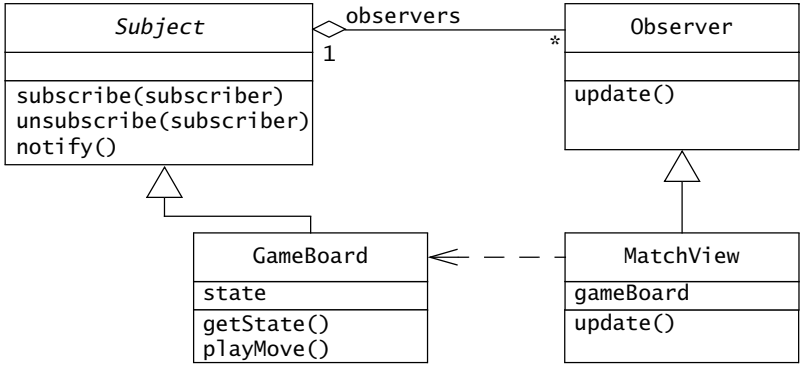



**Figure 8-22** Applying the Observer design pattern to maintain consistency across MatchViews (UML class diagram).

### 8.6.4 Lessons Learned

In this section, we applied three different patterns to address the problem of decoupling `ARENA` from specific `Games`. In this short example, we learned that

- *Design patterns mesh and overlap*. For example, the `Match` class participates in two patterns. It is an `AbstractProduct` in the Abstract Factory and an `Invoker` in the Command pattern.
- *Selecting the right pattern is not trivial*. There are many published catalogs of design patterns. Unless a developer is familiar with them and has used them in the past, it is difficult to assess which pattern to apply in which context. This emphasizes the importance of documenting patterns with examples, which can then be used by developers to assess whether a specific pattern is applicable to their problem.
- *Design patterns must be refined*. Patterns are template solutions, and most often must be adapted to the problem at hand. When refining a pattern, the use of specification inheritance and delegation must be examined carefully so that the extensibility advantages provided by the pattern are not destroyed.

## 9.6 ARENA Case Study

In this section, we apply the concepts and methods described in this chapter to a more extensive example from the ARENA system. We focus on the classes surrounding *TournamentStyle*, which is responsible for creating a set of Matches, assigning Players from the Tournaments, and deciding on start and end dates for each Match. Specifying contracts for these classes is particularly critical, as *TournamentStyle* provides an open interface for other developers to create new *TournamentStyles*. Hence, the boundary between *TournamentStyle* and the rest of

ARENA should be sufficiently clear so that developers can write new *TournamentStyles* without having to understand the ARENA source code.

First, we identify any missing operations or classes. Next, we specify contracts for the abstract classes related to *TournamentStyle*. Finally, we show how these contracts can be strengthened when writing a new *TournamentStyle* (e.g., KnockOutStyle).

### 9.6.1 Identifying Missing Operations in *TournamentStyle* and *Round*

From the ARENA analysis model, we know that the responsibility of the *TournamentStyle* class is to map a list of Players participating in a Tournament onto a series of Matches. Examples of concrete *TournamentStyles* are the KnockOutStyle, in which only winners can move to the next Match, and the RoundRobinStyle, in which each Player competes against all of the other Players exactly once.

During this activity, we focus on the Application Programmer Interface (API) of *TournamentStyle* that a Tournament invokes to generate the series of Matches. First, we observe that *TournamentStyles* generate series of Matches that can be played in parallel (e.g., the first round of a championship) and series of Matches that have precedence constraints (e.g., both semifinals of a knock-out tournament must be completed before the final can start). We also observe that not all Matches will be assigned with Players from the start. In the KnockOutStyle, for example, it is only as Matches are won that subsequent Matches are assigned. This leads us to the following set of requirements:

- *Schedule representation*. We need to represent the graph of Matches to be played in the Tournament in such a way that all *TournamentStyles* can be represented.
- *Incremental planning*. We need to define a set of methods on *TournamentStyle* that enables Matches to be incrementally planned as the Tournament progresses.
- *Game independence*. The graph of Matches and the planning needs to be independent from the *Game*. In particular, the Match class that is part of the *Game* abstract factory should not be constrained by the above requirements.

This leads us to represent the graph of Matches with a new class, *Round* (Figure 9-17). A *Round* corresponds to a set of Matches that can be held concurrently. Hence, a schedule for a Tournament is simply a list of *Rounds*. Next, we define a *TournamentStyle.planRounds()* operation, returning a list of *Rounds*, that is responsible for creating all the Matches in the Tournament and organizing them into a sequence of *Rounds*. *TournamentStyle.planRounds()* also assigns Players to Matches of the first *Round* (and possibly to other *Rounds* as well). Then, we define a *Round.plan()* operation class that is responsible for incrementally assigning Players to Matches of the next *Round*, if necessary. To enable class extenders to define new styles, *TournamentStyle* and *Round* are abstract classes that are refined for each style. For example, the KnockOutStyle class extends *TournamentStyle*, while KnockOutRound extends *Round*. The Tournament class only accesses the abstract classes, hence decoupling it from the concrete style that was selected for the League.

We now specify the contracts of the *TournamentStyle* and *Round* abstract classes.

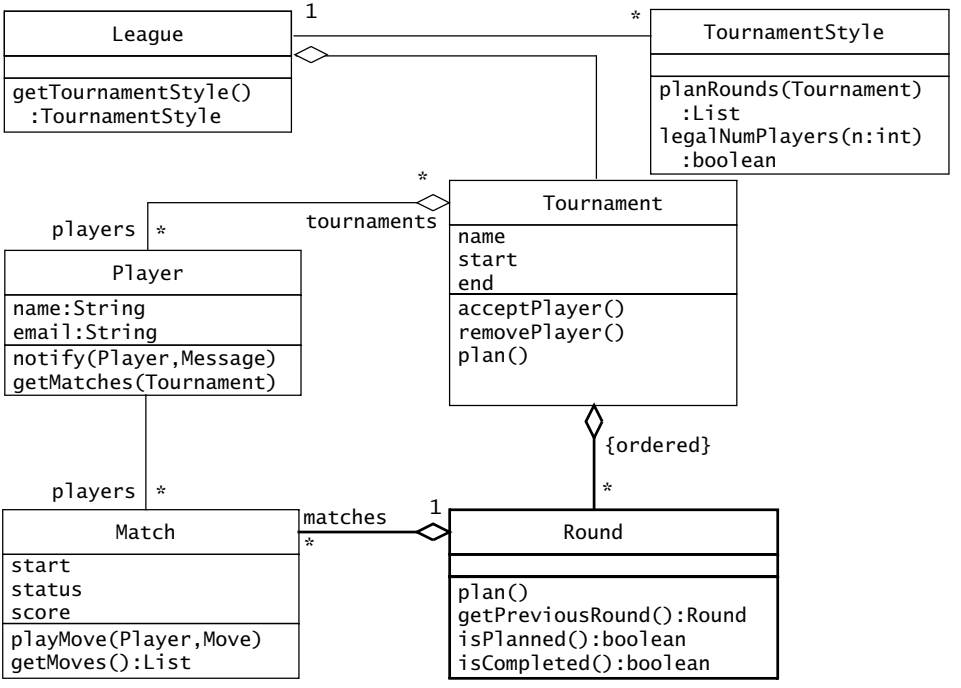



**Figure 9-17** New *Round* class and changes in the *TournamentStyle*, Tournament, and *Round* APIs (UML class diagram). Thick lines indicate changes.

### 9.6.2 Specifying the *TournamentStyle* and *Round* Contracts

The *TournamentStyle* abstract class creates a list of *Rounds* for a given Tournament, and the Tournament stores the *Rounds*. Hence, the *TournamentStyle* does not have any state and, consequently, *TournamentStyle* does not have any invariants.

The *TournamentStyle.planRounds()* operation has one parameter, the Tournament being planned. A general precondition is that Tournaments can only be planned once. Hence, *TournamentStyle.planRounds()* only accepts Tournaments that do not include any *Rounds* yet. A more specific precondition is that concrete *TournamentStyles* may require a specific number of Players. For example, the KnockOutStyle requires a number of Players that is a power of 2. Since the number of Players required by different *TournamentStyles* can vary, we address this issue by adding an operation to the *TournamentStyle* class that returns the set of legal numbers of Players. Note that this set will be bounded by the maximum number of Players per Tournament allowed in the Arena.

```
/* 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)
```

Another approach would be to express the number of Players expected by *TournamentStyles* as a precondition in the subclasses. KnockOutStyle.planRounds(), for example, can have a precondition stating that the number of Players should be a power of 2. While associating such specific preconditions with concrete classes is a good idea, we also need to ensure that the preconditions associated with the superclass are at least as strict as those associated with the subclasses, as explained in Section 9.4.5. Otherwise, we would violate the Liskov Substitution Principle. In concrete terms, it would mean that a class extender writing a subclass of *TournamentStyle* could introduce constraints that the Arena system would violate, and consequently crash the system, while respecting all existing contracts.

There are relatively few post conditions on the *TournamentStyle.planRounds()* operation, since, by design, we do not want to restrict the possible concrete *TournamentStyles*. First, we ensure that each Player is assigned to at least one Match after the initial invocation of planRounds(). Second, we ensure that Matches in different *Rounds* do not overlap, as it is possible for the same Player to take part in more than one *Round*. To express this second constraint, we add the getStartDate() and getEndDate() operations to the *Round* class that compute the start of the earliest Match in the *Round* and the latest end of a Match, respectively.

```
/* 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())
```

To make it easier to write the above constraint, we added a helper operation, Player.getMatches(t:Tournament), to retrieve all the Matches in which a Player takes part within the scope of one Tournament. As the *Round* class is part of the *TournamentStyle* API, we also need to specify the contract of the *Round* class. A *Round* is essentially a list of Matches that can occur concurrently. Consequently, a given Player can take part in at most one Match per *Round*. We specify this with an invariant:

```
/* 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)))
```

Note that the above invariant is still valid if no Players have been assigned to a Match yet.

Next, we focus on the planning status of the *Round*. A *Round* can have one of three states:

- Initially, a *Round* is not planned. It only contains Matches that may or may not have Players assigned.
- Its status becomes planned if the plan() operation manages to assign Players to all Matches.
- Its status becomes completed when all of its Matches have been played and have winners.

This leads to adding two methods, *Round.isPlanned()* and *Round.isCompleted()*, to make it easier for a *Round* to decide when to assign Players to Matches and for Tournaments to decide when a *Round* can be started. We represent the above as three OCL constraints (Figure 9-18) relating the status of the *Round* with the status of its Matches.

```
/* 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.

As we see in the next section, some *TournamentStyles* require that the *Round* be planned only when the previous *Round* is completed.

### 9.6.3 Specifying the KnockOutStyle and KnockOutRound Contracts

Subclasses inherit the contracts of their ancestors (Section 9.4.5). The KnockOutStyle and the KnockOutRound concrete classes inherit the contracts of *TournamentStyle* and *Round*, respectively. However, subclasses can strengthen the invariants and postconditions of the inherited contract and weaken its preconditions. This enables us to specify the refinement of the subclass more precisely and to document in detail how its behavior from other refinements differs from other subclasses (e.g., RoundRobinStyle).

First, we define the return value of the operation `KnockOutStyle.legalNumPlayers()` with a postcondition (Figure 9-19). The number of `Players` can be any power of 2 between 1 and the maximum number of `Players` in the `Tournament`. Next, we define the number of `Matches` in each *Round:* the final *Round* has exactly one `Match`, and all other *Rounds* have exactly twice as many `Matches` as the next *Round*. Next, we add a constraint stating that `Players` can be part of a *Round* only if it is the first *Round* or if they won all other `Matches` in previous *Rounds*. Finally, we strengthen the post condition of `KnockOut.plan()` by requiring that the previous *Round* be completed before the current *Round* can be planned.

```
/* 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.

### 9.6.4 Lessons Learned

In the previous sections, we wrote contracts for the *TournamentStyle*, *Round*, `KnockOutStyle`, and `KnockOutRound` classes. In doing so, we learned that

- *Identifying missing operations and writing contracts are overlapping activities.* Writing contracts forces us to look carefully at each object, and might result in finding new behaviors and boundary cases. In our example, we identified a new class, *Round*,

for modeling the relationship between `Matches` in a `Tournament`. While this concept is from the application domain, refining the definition of *`TournamentStyle`* leads to its identification.

- *Writing contracts leads to additional helper operations for inspecting objects*. Although the relevant object state is accessible by examining attributes and navigating relevant associations, writing simple contracts encourages us to add helper methods for examining state. This results in simpler constraints and classes that are more easily testable (see Chapter 11, *Testing*).
- *Writing contracts leads to better abstractions*. When inheriting contracts, only preconditions can be weakened. Postconditions and invariants can only be strengthened. Consequently, when writing contracts for abstract classes, we add abstract operations for describing properties of the concrete classes. For example, the *`TournamentStyle.legalNumPlayers()`* operation enables us to get around the problem that some of the concrete *`TournamentStyles`* may have constraints on the number of `Players`. In the end, adding such methods results in a better defined meta-model of the concrete classes.

Writing contracts leads to a specification of the legal parameter values. The next step is to define the behavior of the operation when the caller does not respect the contract. We examine the role of exceptions in Chapter 10, *Mapping Models to Code*.

### 10.6.1 ARENA Statistics

A `Statistics` instance is responsible for tracking a number of running counters within the context of a *`Game`* and for a number of scopes. For example, a `Spectator` should be able to view the `Statistics` associated with a specific `Player` within the scope of a single `Tournament` (e.g., the average number of moves per `Match` by player John in the winter Tic Tac Toe tournament) or within the more general scope of a `League` (e.g., the average number of moves per match by

player John in the Tic Tac Toe novice league). Similarly, a Spectator should also be able to view average statistics over all Players in a League. The scope of a statistic are, from most general to most specific:

- All Matches in a *Game*
- All Matches in a League
- All Matches in a League played by a specific Player
- All Matches in a Tournament
- All Matches in a Tournament played by a specific Player.

Earlier (see Section 8.6.1), we addressed the issue of *Game* independence by defining Statistics as a product in an Abstract Factory pattern. For each new *Game* added to ARENA, developers are asked to refine the Statistics interface. For *Games* that do not need specialized Statistics, we provide a default implementation of the Statistics interface called DefaultStatistics. ARENA does not access the concrete *Game* and Statistics classes, thereby ensuring *Game* independence (Figure 10-23).

Our design goal is to keep the Statistics interface simple so that it can be easily specialized for different *Games*. In other words, the concrete Statistics class for a specific *Game* should only have to define the formulae for computing the *Game*-specific Statistics. Scopes

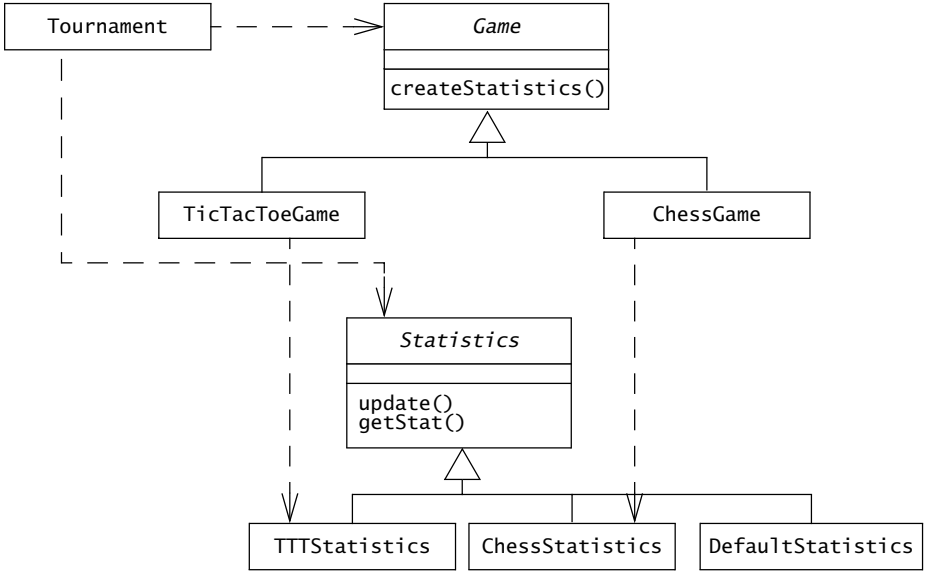



**Figure 10-23** Statistics as a product in the *Game* Abstract Factory (UML class diagram).

should be handled by ARENA. Moreover, response time in ARENA has a higher priority than memory consumption.

This leads us, during object design, to the decision of computing all Statistics incrementally as each Match completes. We organize the individual counters we compute into Statistics objects, each representing a scope. Hence, there is a Statistics object for each *Game*, League, Tournament, and for each combination of Player/*Game*, Player/League, and Player/Tournament.

For example, let us assume that Player John takes part in the two Tournaments, t1 and t2, in the Tic Tac Toe novice League. He then moves on to the expert Tic Tac Toe League and takes part in one more Tournament, t3. Let us further assume that the Tic Tac Toe Statistics object tracks his win ratio, the ratio of the number of the *Games* he has won over all the *Games* he has played. Under these circumstances, six Statistics objects track John's win ratio:

- Three Statistics objects track John's win ratio for the Tournaments t1, t2, and t3.
- Two Statistics objects tracks John's win ratio over all his Tournaments within the novice and the expert Leagues, respectively.
- One for the Tic Tac Toe *Game*, tracking John's lifetime win ratio.

In UML, we can represent compactly this complex set of interactions with an N-ary association Statistics class relating the Player, the *Game*, the League, and the Tournament classes (Figure 10-24). For any given Statistics association, only one of *Game*, League, or Tournament is involved in the association, denoting whether the Statistics is value for all Leagues in a specific *Game*, for a single League, or for a single Tournament, respectively.

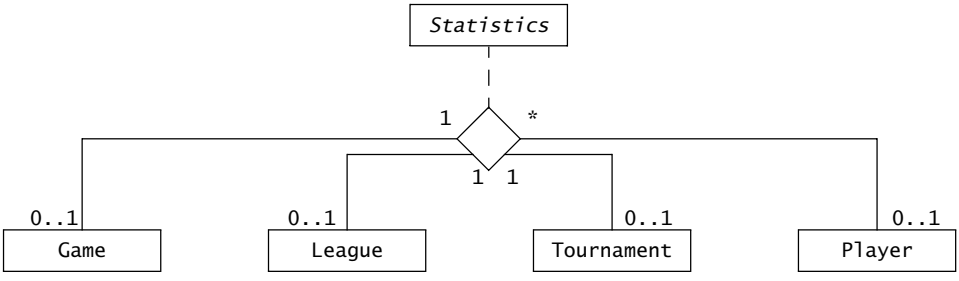



**Figure 10-24** N-ary association class Statistics relating League, Tournament, and Player (UML class diagram).

### 10.6.2 Mapping Associations to Collections

We map the N-ary Statistics association to a Statistics Java class, whose sole purpose is to hold the values of the counters, and a singleton StatisticsVault, responsible for tracking the

links between the Statistics objects and the relevant *Game*, League, Tournament, and Player instances (Figure 10-25). We add operations to StatisticsVault for retrieving a specific Statistics given a scope. If the Statistics object of interest does not exist, the StatisticsVault creates it using the corresponding *Game.createStatistics()* method. Internally, the StatisticsVault uses a private HashMap to store the relationship between the combination of Player, *Game*, League, and Tournament and the corresponding Statistics object.

SimpleStatisticsVault depicted in Figure 10-25 is a direct mapping of the N-ary association of Figure 10-24. SimpleStatisticsVault does not accomplish any other task beyond maintaining the state of the association. TournamentControl invokes methods of SimpleStatisticsVault to retrieve the needed Statistics objects, and then invokes the Statistics.update() method as Matches complete. StatisticsView (a boundary object displaying statistics to the user) retrieves the needed Statistics based on the user selection and invokes the Statistics.getStat() method to retrieve the individual values of the counters. In both cases, two steps are needed, one to retrieve the correct Statistics object, the other to invoke the method that does the actual work.

Ideally, we would like to avoid this situation and design an interface that enables TournamentControl and StatisticsView to invoke only one method. This would simplify the method calls related to Statistics in both objects. It would also centralize the scope look-up logic into a single object, making future changes easier. To accomplish this, we refine the SimpleStatisticsVault into a Facade design pattern (see Figure 6-30). We merge the getStatisticsObject() methods with the methods offered by the Statistics object; that is, we provide an update() method for TournamentControl and a set of getStat() methods for

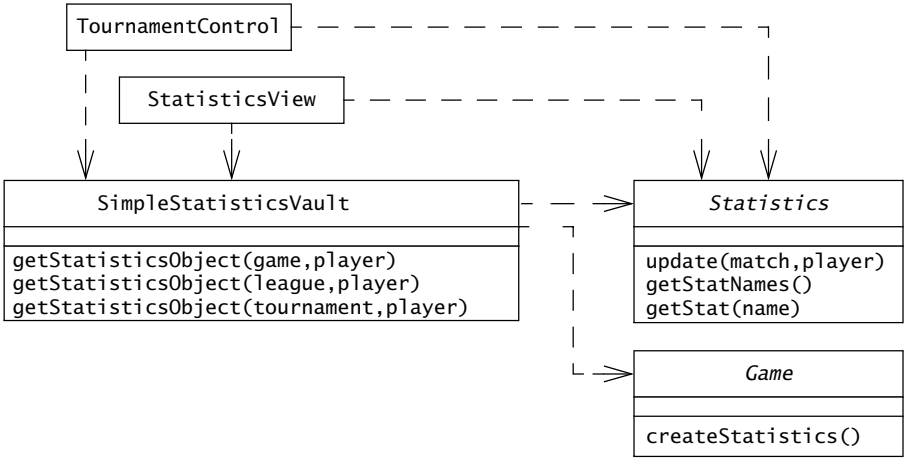



**Figure 10-25** SimpleStatisticsVault object realizing the N-ary association of Figure 10-24.

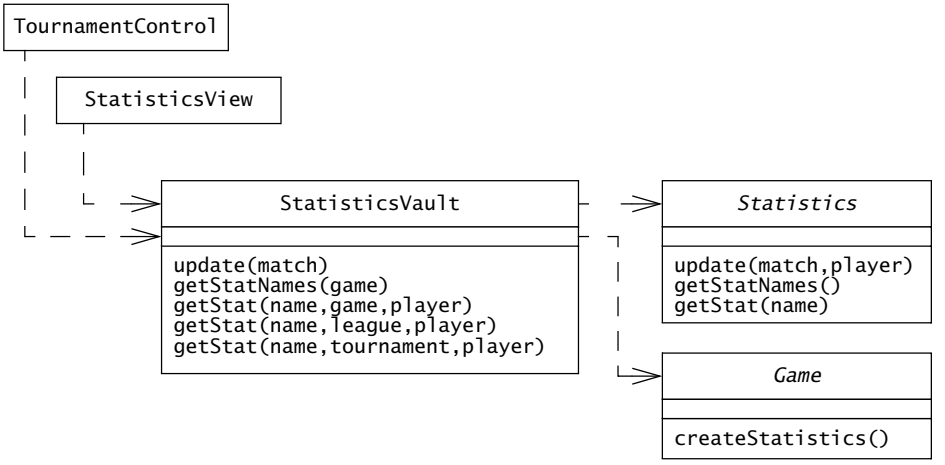



**Figure 10-26** `StatisticsVault` as a Facade shielding the control and boundary objects from the Statistics storage and computation (UML class diagram).

`StatisticsView`. This results in a slightly more complex interface (Figure 10-26). However, the interface is still within 7 ± 2 methods, and all handling of `Statistics` objects is now centralized.

If we decide to add different scopes (e.g., match statistics), we will only need to add a method to `StatisticsVault`. Moreover, `TournamentControl` and `StatisticsView` are now completely decoupled from the individual `Statistics` objects, giving us the option of changing the way `Statistics` are stored, for example, by computing them on the fly or by caching them with a Proxy design pattern (Figure 10-7).

The role of the `Statistics` object is reduced to tracking counters (basically, name/value pairs). None of the classes `Statistics`, `Player`, `Tournament`, or `League` store data about the N-ary association. Consequently, when writing a specialized `Statistics` object for a new *`Game`*, the class extender need only focus on the formula for computing a statistic given a `Match` and a `Player`.

### 10.6.3 Mapping Contracts to Exceptions

Because we introduced a new interface, the `StatisticsVault` facade, in the `ARENA` object design model, we need to write its contract and relate it to the contracts of the existing classes. First, we focus on the constraints associated with the Statistics class and propagate them to the `StatisticsVault` class. The `Statistcs.getStat()` method assumes that the name passed in parameter is that of a statistic known by this type of object. Hence, we add one such constraint for each `getStat()` method in the `StatisticsVault`:

```
context StatisticsVault::getStat(name,game,player) pre:
    getStatNames()->includes(name)
context StatisticsVault::getStat(name,league,player) pre:
    getStatNames()->includes(name)
context StatisticsVault::getStat(name,tournament,player) pre:
    getStatNames()->includes(name)
```

Similarly, we add constraints to the `StatisticsVault.update()` method to reflect the constraint on the `Statistics.update()` method. In this case, we only need to ensure that the `match` is not `null` and has been completed.

```
context StatisticsVault::update(match) pre:
    match <> null and match.isCompleted()
```

Next, we need to examine the remaining parameters of the `StatisticsVault` methods and document any additional preconditions. We stipulate that the `player` parameter can be `null` (denoting the `Statistics` object for all players), but parameters specifying a `game`, `league`, and `tournament` cannot be `null`, because a `null` value, in this case, does not correspond to an application domain concept (e.g., no statistics are valid across all *Games*). In addition, if a player is specified, she must be related to the scope object (e.g., a player does not have statistics for a tournament in which she did not play).

```
context StatisticsVault::getStat(name,game,player) pre:
    game <> null and
        player <> null implies player.leagues.game->includes(game)
context StatisticsVault::getStat(name,league,player) pre:
    league <> null and
        player <> null implies league.players->includes(player)
context StatisticsVault::getStat(name,tournament,player) pre:
    tournament <> null and
        player <> null implies tournament.players->includes(player)
```

Finally, we map the above constraints to exceptions and checking code. For constraints that we propagated from the `Statistics` object, we simply forward the exceptions we receive from `Statistics` and omit the checking code. The `UnknownStatistic` is raised by `Statistics.getStat()` and forwarded by `StatisticsVault.getStat()` methods. The `InvalidMatch` and `MatchNotCompleted` exceptions are raised by `Statistics.update()` and forwarded by `StatisticsVault.update()` method. For the other constraints, we define a new exception, `InvalidScope`, for the cases where the `game`, `league`, or `tournament` parameters are `null` or are not related to the specified `player`. Finally, we add checking code at the beginning of the `StatisticsVault.getStat()` methods to check for `null` references and raise the `InvalidScope` exception, if needed. Figure 10-27 depicts the public interface of the `StatisticsVault` class.

```
public class StatisticsVault {
    public void update(Match m)
        throws InvalidMatch, MatchNotCompleted {...}

    public List getStatNames() {...}

    public double getStat(String name, Game g, Player p)
        throws UnknownStatistic, InvalidScope {...}

    public double getStat(String name, League l, Player p)
        throws UnknownStatistic, InvalidScope {...}

    public double getStat(String name, Tournament t, Player p)
        throws UnknownStatistic, InvalidScope {...}
}
```

**Figure 10-27** Public interface of the `StatisticsVault` class (Java).

Note that, with the `InvalidScope` exception, we decided to cover three preconditions with one exception. This results in similar method declarations for all three `getStat()` methods, thereby making it easier for the calling class to handle violations of all three preconditions with the same handling code. In general, a set of overloaded methods should have similar interfaces since they implement the same operation for the different types of parameters.

### 10.6.4 Mapping the Object Model to a Database Schema

In the past two subsections, we transformed the N-ary `Statistics` association class (Figure 10-24) into a set of Java classes and operations. In this section, we start with the same object model and map it into a set of tables. As `Statistics` is an N-ary association class, we cannot use buried keys. We first map the association to a separate table, containing a foreign key for each of the association ends (*Game*, `League`, `Tournament`, `Player`). We then note that any given `Statistic` includes only one link to either a *Game*, a `League`, or a `Tournament`. Consequently, we can collapse the columns for *Game*, `League`, and `Tournament` into a single column denoting the `id` of the scope object and a column denoting the type of object (`scopetype`, Figure 10-28). We decide to encode the type of scope as a long value to save on storage space and on retrieval speed. The mapping between the long values and the actual class is done in the storage subsystem. This decision is a local optimization and does not affect the interface of the `StatisticsVault`. The `id` of the `Statistics` object is the primary key of the table.

As each `Statistic` has a variable number of name/value pairs, we use a separate table for the counter values, including one column for the `name` of the counter, one for the `value`, and a foreign key into the `Statistics` table (i.e., the `Statistics id`). The primary key for the `StatisticsCounter` table is the `Statistics id` and the counter `name`.

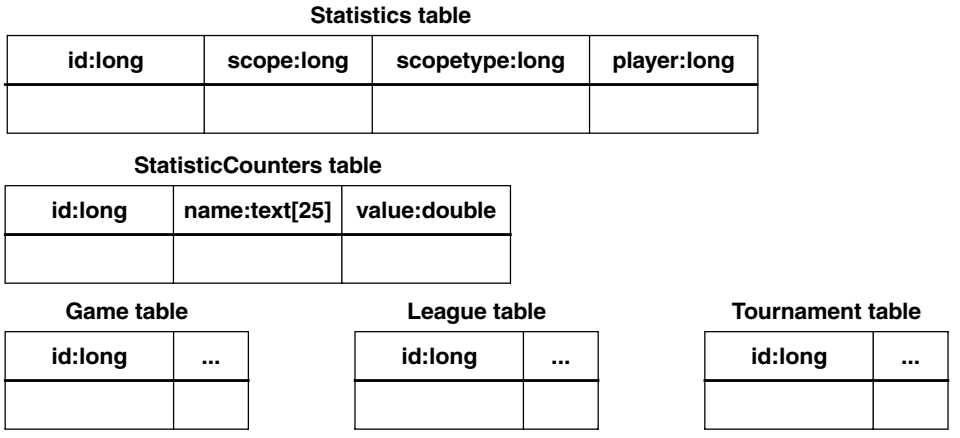



**Figure 10-28** Database schema for the `Statistics` N-ary association of Figure 10-24.

Note that when designing the database schema for the `Statistics` association, we started with the object design model, not the Java classes that we generated in the previous section. There are several reasons for starting with the object design model:

- The object design model represents a view of the application domain and is less likely to change than the source code.
- When starting from the object design model, the database schema uses names that come from the application domain (e.g., `Statistics`) instead of names that denote solution objects (e.g., `StatisticsVault`). This provides better and more direct traceability to the requirements.
- The transformation of the N-ary association into Java classes focused only on runtime concerns, not on data storage concerns.

### 10.6.5 Lessons Learned

In the previous section, we mapped the `Statistics` N-ary association class to source code and to a database schema. In doing so, we learned that

- *Applying one transformation leads to new opportunities for other transformations*. By converting the N-ary `Statistics` association into a separate object, we defined a facade shielding control and boundary objects from the details of the `Statistics` representation.
- *Introducing new interfaces results in forwarding existing exceptions*. When introducing a new interface, we need to make sure exceptions are not masked. When we introduced

a facade in this example, we simply forwarded the lower-level exceptions (`UnknownStatistic`) to the `Client` class.

- *Database schema is based on the object design model, not on the source code*. In general, we need to ensure that the analysis concepts are visible in the implementation, providing traceability back to the requirements. Naming classes, attributes, methods, and tables in terms of names used during analysis is critical in maintaining conceptual integrity.