Section 2.2
Requirements Elicitation

1. Overview
2. Concepts
3. Activities
4. ARENA case study

2.2.1 Overview

- Learning outcomes
  - extract functional and non-functional requirements from a problem description
  - construct the corresponding functional model, using use cases and scenarios
  - incorporate traceability into every work product

Overview (cont.)

- What is elicitation?
  - the acquisition of information
  - the information flows:
    - from the client
    - to the development team

- What are *requirements*?
  - what the client wants the system to do
    - the features that the system must provide
    - the tasks that the system must perform
  - the constraints on the system

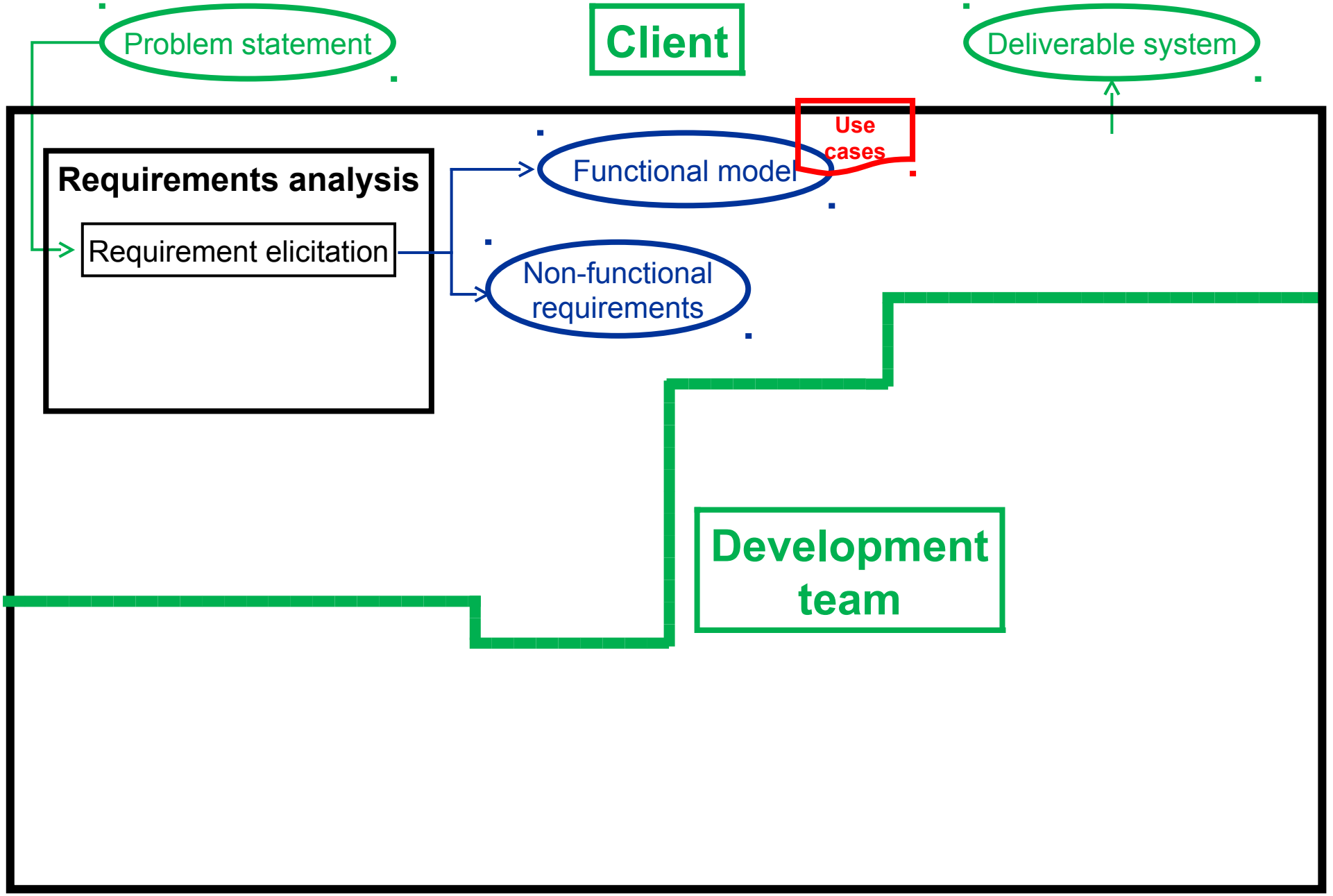
Overview (cont.)

- Two types of requirements
  - functional requirements
  - non-functional requirements

- Characteristics of requirements elicitation activity
  - highly iterative
  - incremental

- Iterative steps
  - functionality is:
    - mapped
    - proposed to clients/users
    - requirements are added and refined

Requirements Elicitation Work Products



2.2.2 Requirements Elicitation Concepts

- Functional requirements
- Non-functional requirements
- Requirements specification
- Traceability
- Requirements validation

Functional Requirements

- Functional requirements (FRs) describe:
  - **what** the system will do, not **how** it will do it
  - the features that the system must provide
  - the tasks that the system must perform
  - the interactions between the system and its environment
- Characteristics
  - must be design-independent and implementation-independent

Non-Functional Requirements

- Non-functional requirements (NFRs) describe:
  - user-visible constraints on the system
  - aspects of the system not related to functional behaviour
  - must be very specific
    - must be quantifiable and testable
- Feasibility?
  - investigated through:
    - technology studies
    - prototyping

Non-Functional Requirements (cont.)

- Main NFR categories:
  - usability
    - ease of use, colour schemes
    - online help, documentation
  - reliability
    - ability to perform under certain conditions
    - dependability (includes fault-tolerance and security)
  - performance
    - response time, throughput
    - availability
  - supportability
    - maintainability
    - portability

Non-Functional Requirements (cont.)

- Main NFR categories (cont.):
  - implementation
    - programming languages, platform, environment
  - interface
    - details of interfaces with external systems
  - operations
    - installation, maintenance
  - packaging
    - software delivery
  - legal
    - copyright, regional laws

Requirements Specification

- What is it?
  - a system definition that the client understands
- Purpose
  - serves as a contract between:
    - the client
    - the development team

Requirements Specification (cont.)

- Includes:
  - functional requirements
    - embodied in the functional model
  - non-functional requirements

- Characteristics
  - complete: nothing is missing
  - consistent: there are no contradictions
  - unambiguous: only one way of interpreting anything
  - correct: reflects what the client truly wants

Traceability

- What is traceability?
  - the system's "cradle-to-grave" accountability
  - every requirement can be tracked through development life cycle
    - use cases
    - objects
    - programming language functions
    - test cases
  - all dependencies are documented

Traceability (cont.)

- Purpose
  - to facilitate maintenance activity
  - impact of changes can be assessed through all work products
- How do we achieve traceability?
  - numbering scheme and cross-references
  - requirements traceability matrix

Requirements Validation

- Purpose
  - requirement specification must be validated by:
    - client
    - development team

- Characteristics
  - completeness
  - consistency
  - clarity
  - correctness
  - realism
  - verifiability
  - traceability

Dorc Slayer Case Study

- Read the problem statement
- Group exercise:
  - list the functional requirements
  - list the non-functional requirements

2.2.3 Requirements Elicitation Activities

- Identifying actors
- Identifying scenarios
- Identifying and refining use cases
- Identifying relationships among actors and use cases
- Identifying initial analysis objects
- Identifying non-functional requirements

Identifying Actors

- What are *actors*?
  - external entities that will interact with the system
    - people (users)
    - existing, external systems
  - actors are *role abstractions*
- Why do we need to do this?
  - it serves to define the *system boundaries*

Identifying Actors (cont.)

- How?
  - identify the user groups that will:
    - need access to the system
    - execute its main functionality
  - identify the existing, external systems we need:
    - for input data
    - for output data
    - these are high-level concepts, not implementation ones
      - for example, the OS and libraries are **not** external systems

Identifying Scenarios

- What are *scenarios*?
  - they describe the system in terms of interactions with users
  - they are *instances of a use case*
  - they are concrete descriptions of a single feature
    - from the user's point of view

Identifying Scenarios (cont.)

- Why do we need to do this?
  - we figure out the main system functionality through scenarios
  - clients/users and developers gain a shared understanding of:
    - the application domain
    - the system to be built
  - scenarios can be used as input to create:
    - acceptance test plan
    - system test plan

Identifying Scenarios (cont.)

- How do we identify scenarios?
  - interviews with clients and users
  - procedure manuals, application domain documentation
  - UI mock-ups
  - we need to identify:
    - the tasks that the actors need the system to perform
    - the information that actors create, access, modify
    - the events and exchanges between the actors and the system

Identifying Scenarios (cont.)

- How do we represent scenarios?
  - table-based descriptions
  - these contain:
    - a unique scenario name
    - the participating actors
    - the flow of events in the scenario

Identifying Scenarios (cont.)

- UniCentral example

Scenario name	**registerForCOMP3004**
Participating actor instances	**matilda:Student** **:RegistrarSystem** **:FinanceSystem**
Flow of events	1. From a list of departments, Matilda selects COMP, and the list of offered COMP courses is displayed. 2. Matilda selects COMP 3004 and confirms her input; she waits for confirmation of her registration. 3. The **FinanceSystem** receives the registration request and validates that Matilda's account has no holds on it. 4. The **RegistrarSystem** receives the registration request and validates that Matilda has the necessary prerequisites and that she has no schedule conflicts. The **RegistrarSystem** registers Matilda for the course. 5. Matilda receives confirmation of her registration.

Identifying Use Cases

- What is a use case?
  - an abstraction of a set of scenarios
  - initiated by an actor
  - describes interactions resulting from use case initiation
  - distinguishes steps performed by the actor vs. the system

- Why do need to do this?
  - to facilitate communication between the client and developers
  - to help developers establish the scope of the system

Identifying Use Cases (cont.)

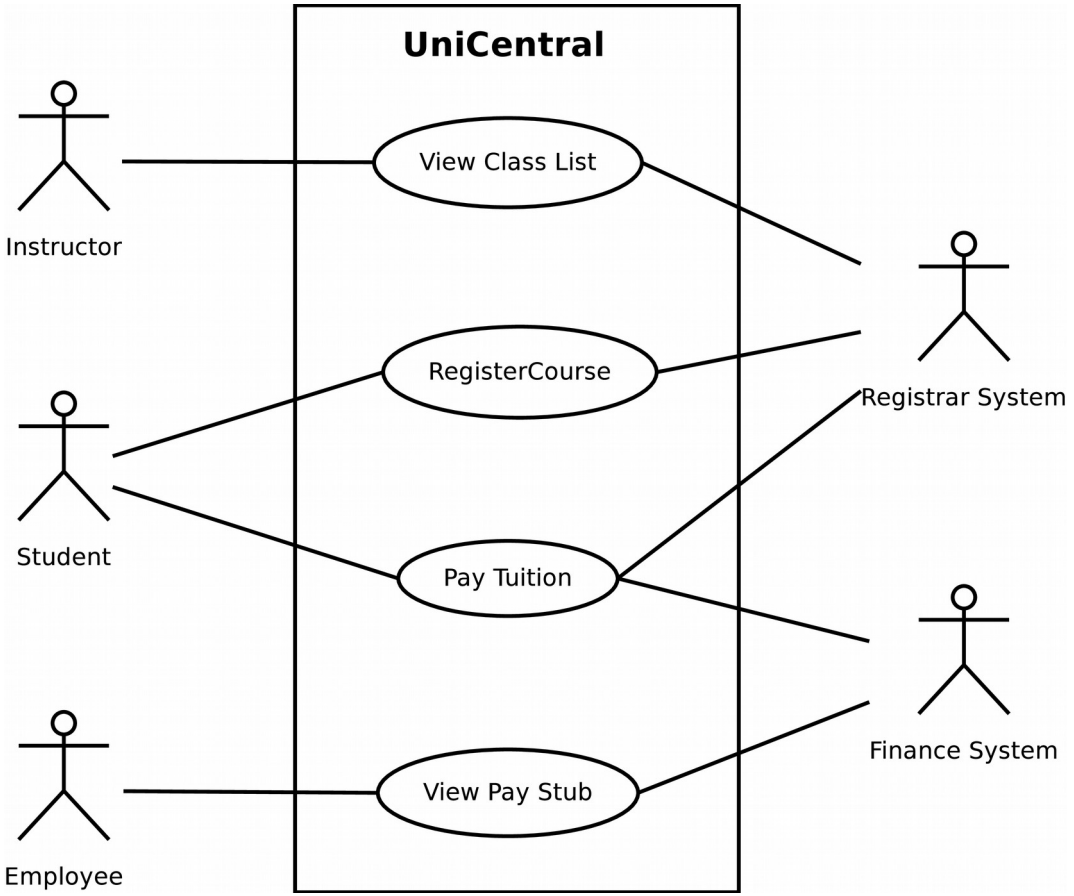
- How?
  - we generalize from identified scenarios
  - we prompt the client/users for:
    - different alternatives
    - system feature details
    - constraints
  - start with high-level use cases
  - break down each high-level use case into detailed use cases

Identifying Use Cases (cont.)

- How do we represent use cases?
  - table-based descriptions that contain:
    - a unique use case identifier
    - a unique use case name
    - the participating actors
    - the flow of events in the use case
    - the entry conditions
    - the exit conditions
    - the quality requirements
    - traceability back to FRs and NFRs
  - UML use case diagrams

Identifying Use Cases (cont.)

- Simplistic example (does **not** show high-level use cases)



ViewClassList	The `Instructor` can view the list of `Students` registered for a particular course.
RegisterCourse	The `Student` can view the courses offered for a department, select one and register in it.
PayTuition	The `Student` can see the current balance owed for tuition, and pay it.
ViewPayStub	The `Employee` can view his/her pay stubs for the past year, select one and view its details.

Identifying Use Cases (cont.)

- Example of use case table-based description

Use case name	**`ViewClassList`**
Participating actors	Initiated by **`Instructor`** Communicates with **`RegistrarSystem`**
Flow of events	1. The **`Instructor`** selects the View Class List option. 2. The **`RegistrarSystem`** provides a list of all courses taught by the **`Instructor`**. 3. The **`Instructor`** selects the desired course. 4. The **`RegistrarSystem`** provides a list of **`Students`** registered in the selected course. 
Entry condition	• The **`Instructor`** must be logged in to UniCentral.
Exit condition	
Quality requirements	• While the system is processing the **`Instructor`**'s request, a wait icon should be displayed. • Response time between the **`Instructor`** selecting the View Class List option and receiving confirmation should be no more than 20 seconds.
Traceability	• FR-17, NFR-23, NFR-28

Identifying Relationships

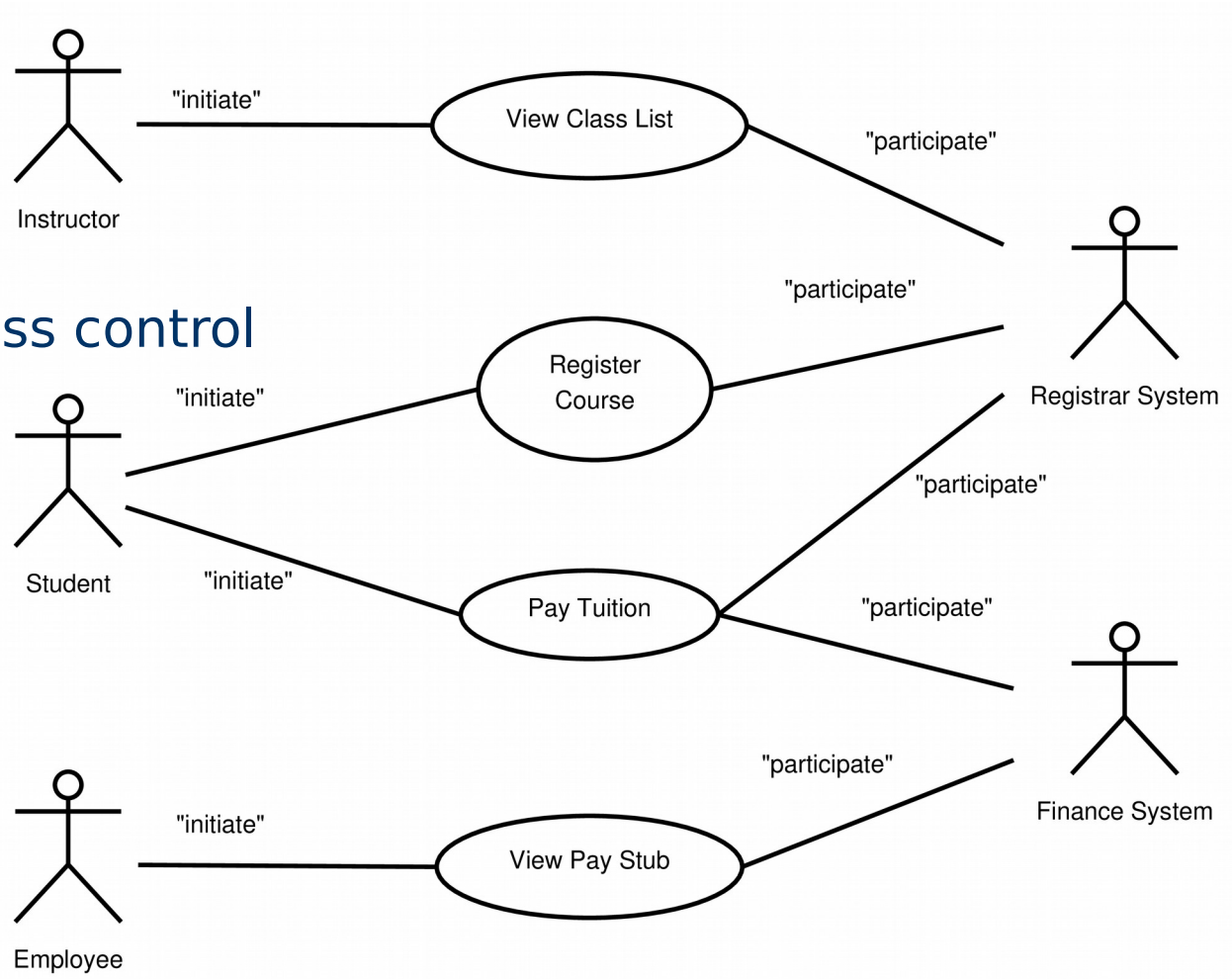
- Types of use case relationships:
  - between actors and use cases
    - communication relationships
  - between use cases
    - extend
    - include
    - inheritance

Identifying Relationships (cont.)

- How do we represent use case relationships?
  - use case diagrams
    - arrows
    - labels
  - use case table-based descriptions
    - using relationship terms, e.g. “extends”, “includes”, “inherits”
    - highlighting text in **bold** font

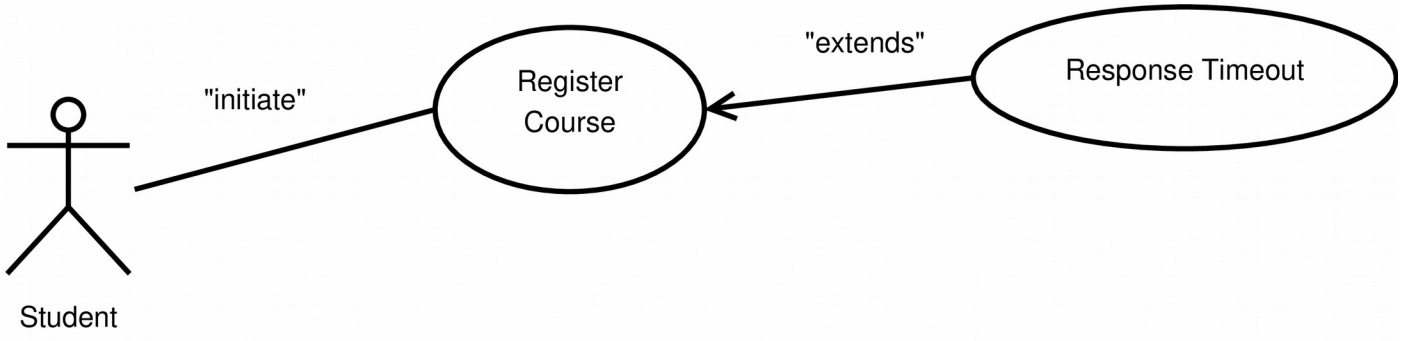
Identifying Relationships (cont.)

- Communication relationships
  - relationships between actors and use cases
    - initiation
    - participation
  - help to specify access control
    - permissions



Identifying Relationships (cont.)

- Extend relationship
  - relationship between use cases
  - one use case extends functionality of another in certain conditions
  - **only** used for exceptional flow of events
    - errors and exceptions
  - who knows what:
    - the originating use case does **not** know about the extending use case
    - the extending use case has a specific event as its entry condition



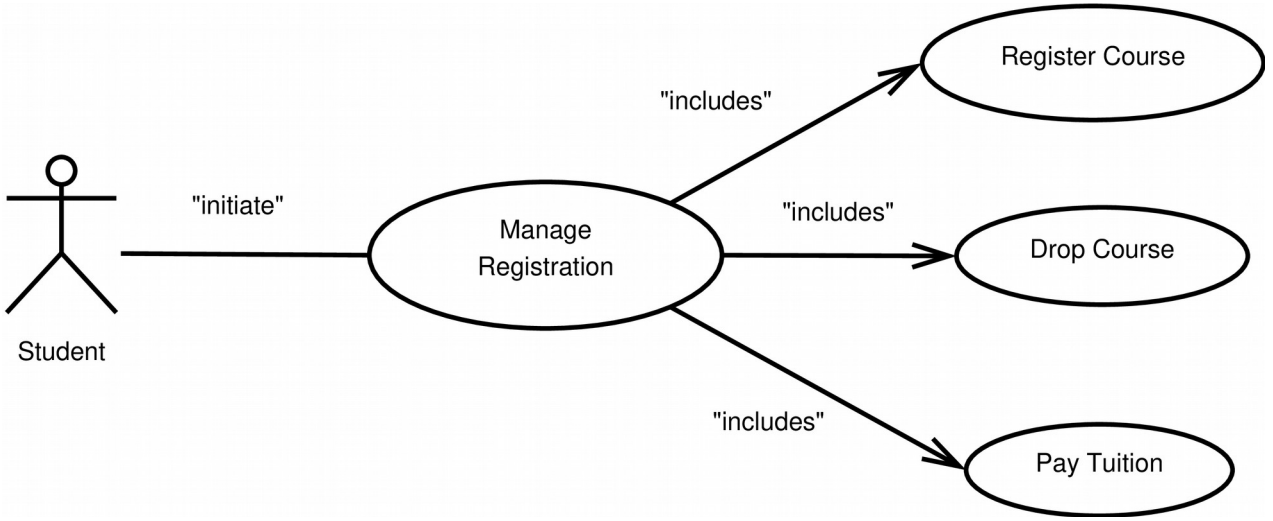
Identifying Relationships (cont.)

- Example of extending use case

Use case name	**`ResponseTimeout`**
Participating actors	Communicates with **`Student`**
Flow of events	1. A timeout messages is displayed to the **`Student`**. 2. The registration request is cancelled, and all actions taken to serve the pending request are rolled back.
Entry condition	• This use case **extends** the **`RegisterCourse`** use case. It is initiated by the system whenever the **`RegistrarSystem`** or the **`FinanceSystem`** has failed to respond within 20 seconds.
Exit condition	• The **`Student`**'s information is reset to its status from before the registration request was initiated.
Traceability	• NFR-28

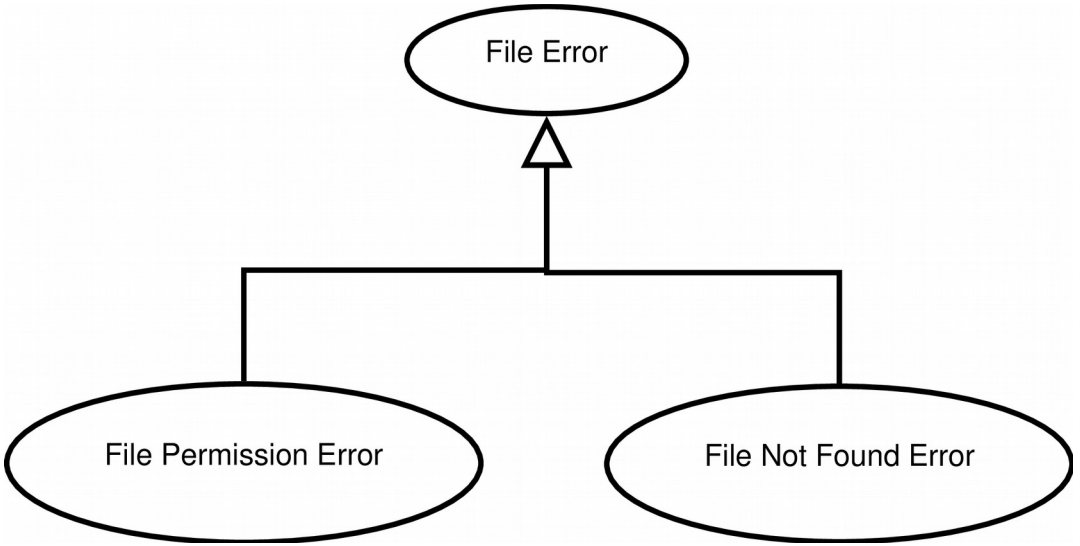
Identifying Relationships (cont.)

- Include relationship
  - relationship between use cases
  - **only** used for **two** reasons:
    - to break down the system use cases to reduce complexity
    - to factor out redundant functionality into a reusable use case
  - who knows what:
    - the originating use case triggers the included use case
    - the included use case doesn't know which use case triggers it



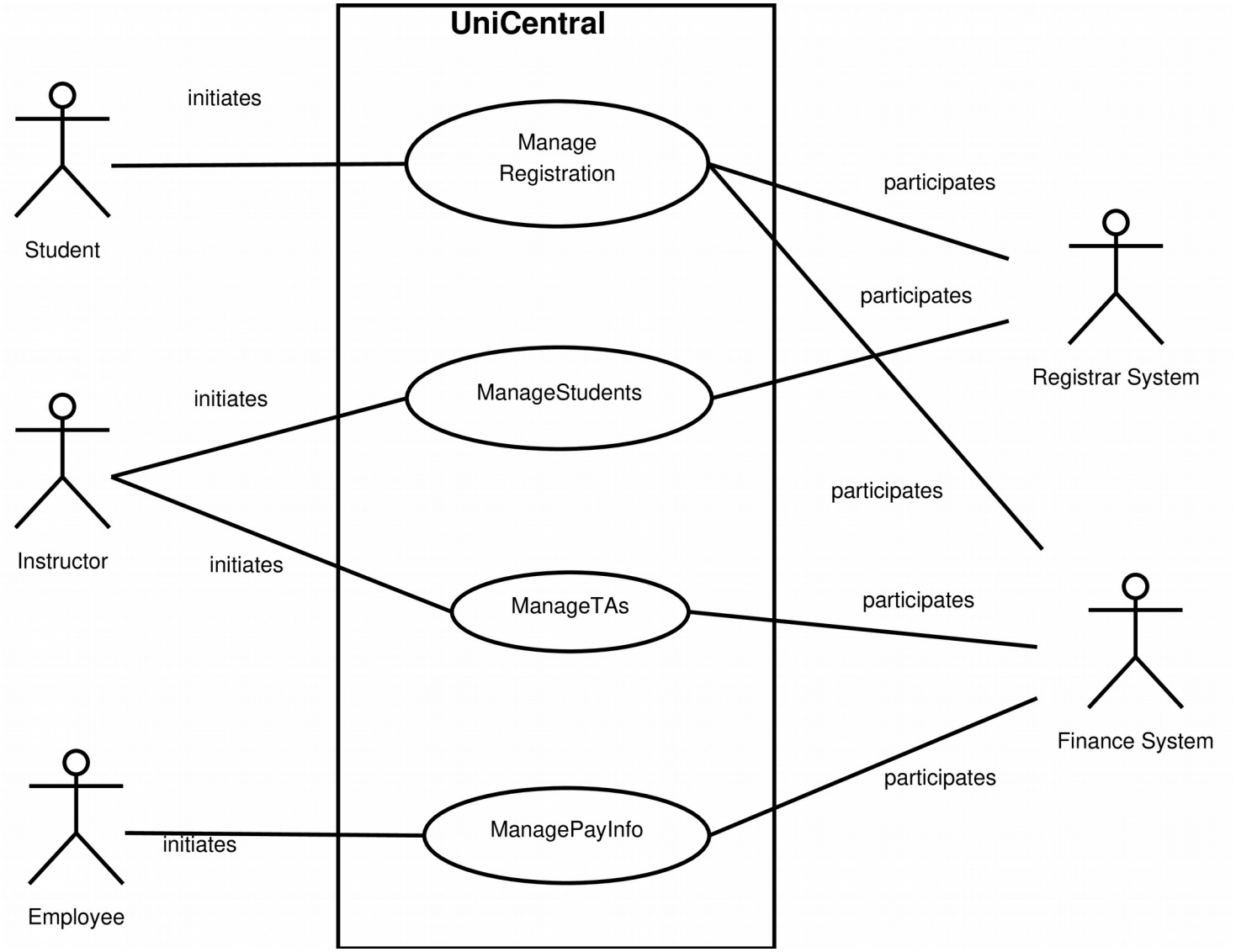
Identifying Relationships (cont.)

- Inheritance relationship
  - "is-a" relationship between use cases
  - indicates a sub-type of use cases
  - this is **not** interchangeable with includes relationship
  - applicable in very few cases



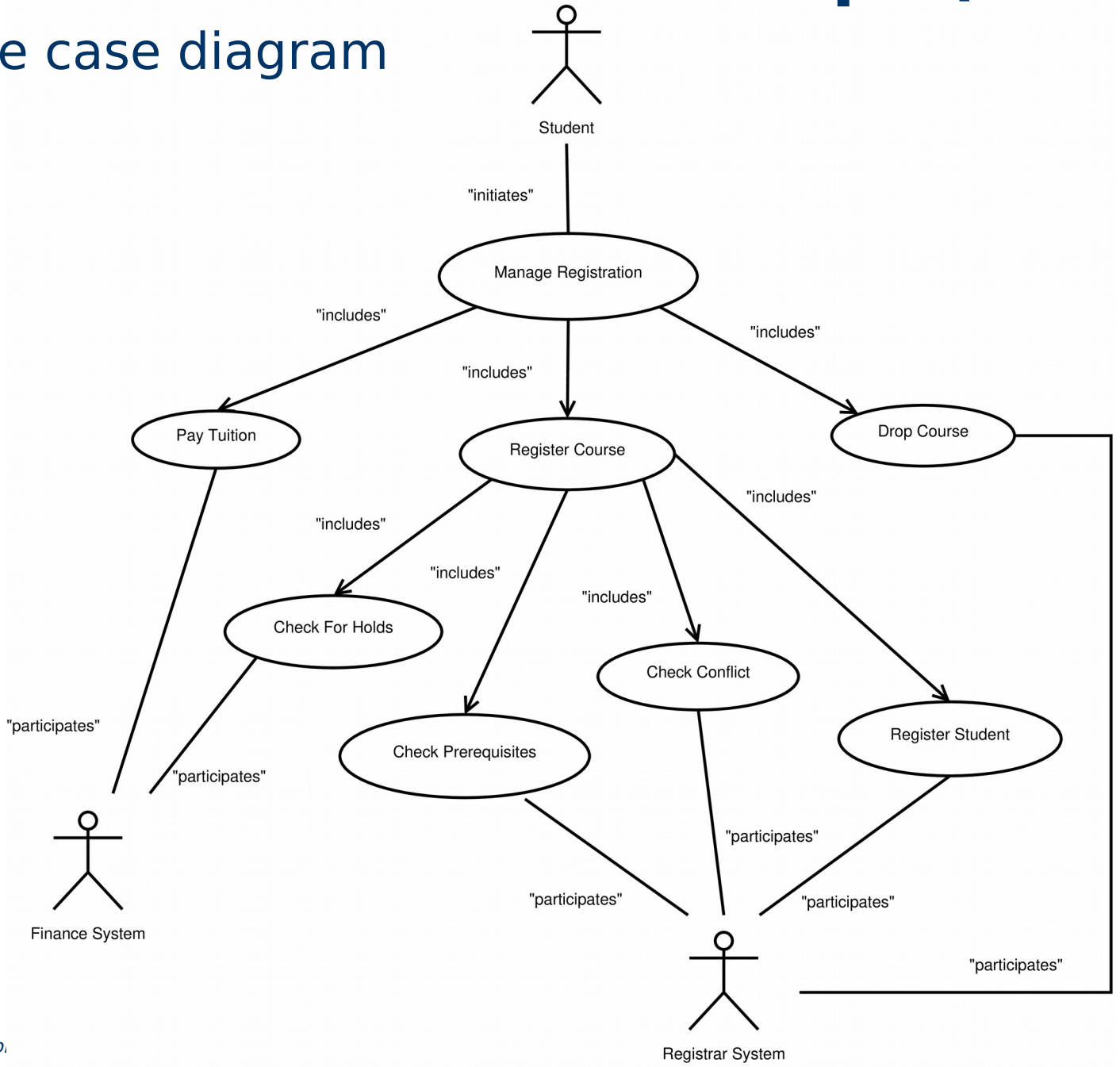
Example of Use Case Relationships

- High-level use case diagram



Example of Use Case Relationships (cont.)

- Detailed use case diagram



Example of Use Case Relationships (cont.)

- High-level use case table-based description

Use case name	**`RegisterCourse`**
Participating actors	Initiated by **`Student`** Communicates with **`RegistrarSystem`** and **`FinanceSystem`**
Flow of events	1. The **`Student`** selects the Register option. 2. The **`RegistrarSystem`** displays a list of all departments in the university. 3. The **`Student`** selects the department in which the course is offered. 4. The **`RegistrarSystem`** displays a list of courses offered within the selected department. 5. The **`Student`** selects the desired course and confirms the input. 6. The **`FinanceSystem`** receives the registration request and validates that the **`Student`**'s account has no holds on it (**include** use case **`CheckForHolds`**). 7. The **`RegistrarSystem`** receives the registration request and validates that the **`Student`** has the necessary prerequisites for the selected course (**include** use case **`CheckPrerequisites`**), and that the **`Student`** has no schedule conflicts (**include** use case **`CheckConflict`**). The **`RegistrarSystem`** registers the **`Student`** for the course (**include** use case **`RegisterStudent`**). 8. The **`Student`** receives confirmation of registration in the selected course.
Entry condition	• The **`Student`** must be logged in to UniCentral.
Exit condition	• The **`Student`** is registered in the selected course. • The tuition fee is added to the **`Student`**'s balance owing.
Quality requirements	• While the system is processing the **`Student`**'s request, a wait icon should be displayed. • Response time between the **`Student`** selecting the Register option and receiving confirmation should be no more than 20 seconds.
Traceability	• FR-12, NFR-23, NFR-28

Identifying Initial Analysis Objects

- What are the initial analysis objects?
  - the main *participating objects* in the system
  - the objects that participate in the use cases
- Why do we need to do this?
  - this activity is a precursor to the analysis phase

Identifying Initial Analysis Objects (cont.)

- How do we identify these objects?
  - we examine *every use case* to find the participating objects
  - we find:
    - the recurring nouns
    - the real-world entities and processes
    - the data sources and data sinks
  - we should use the application domain terminology
  - thinking of object lifetime helps to ensure use case completeness

- How do we represent the initial analysis objects?
  - we collect them into a glossary

Identifying Non-Functional Requirements

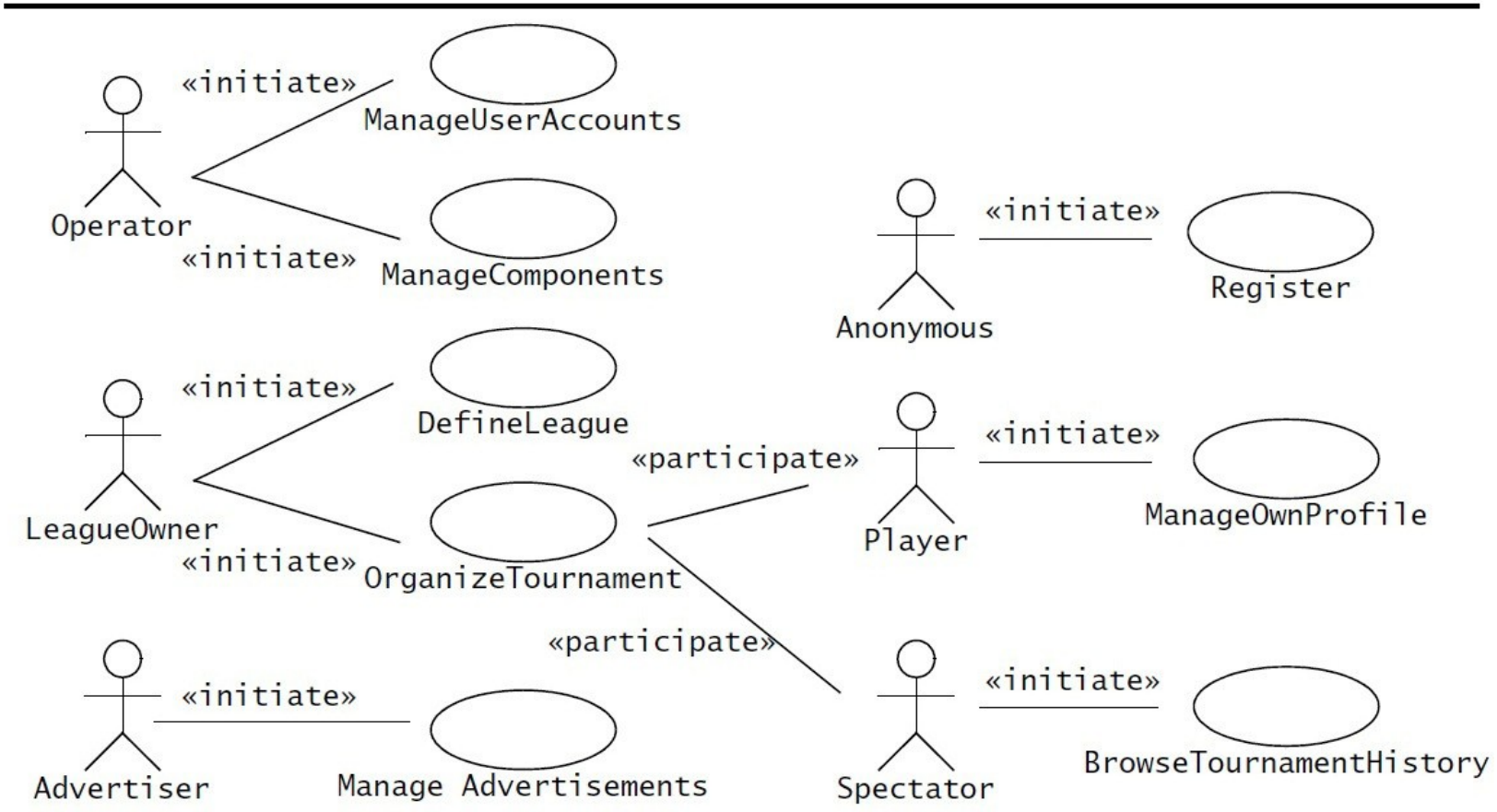
- What are non-functional requirements?
  - user-visible constraints on the system
  - aspects of the system not related to functional behaviour

- Why do we need to do this?
  - to document the client's needs
  - to prioritize conflicting requirements

- How?
  - examine the possible categories:
    - usability
    - reliability
    - performance
    - supportability
    - implementation
    - interface
    - operations
    - packaging
    - legal

2.2.4 ARENA Case Study

- Read the problem statement in textbook Figure 4-17



ARENA Case Study (cont.)

Register	Anonymous users register with an Arena for a Player or a League-Owner 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.

ARENA Case Study (cont.)

Use case name	`OrganizeTournament`
Participating actors	Initiated by `LeagueOwner` 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`). 2. The `Players` apply for the `Tournament` (include use case `ApplyForTournament`). 3. The `LeagueOwner` processes the `Player` applications and assigns them to matches (include use case `ProcessApplications`). 4. The `LeagueOwner` kicks off the `Tournament` (include use case `KickoffTournament`). 5. The `Players` compete in the matches as scheduled and `Spectators` view the matches (include use case `PlayMatch`). 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 • 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`.

ARENA Case Study (cont.)

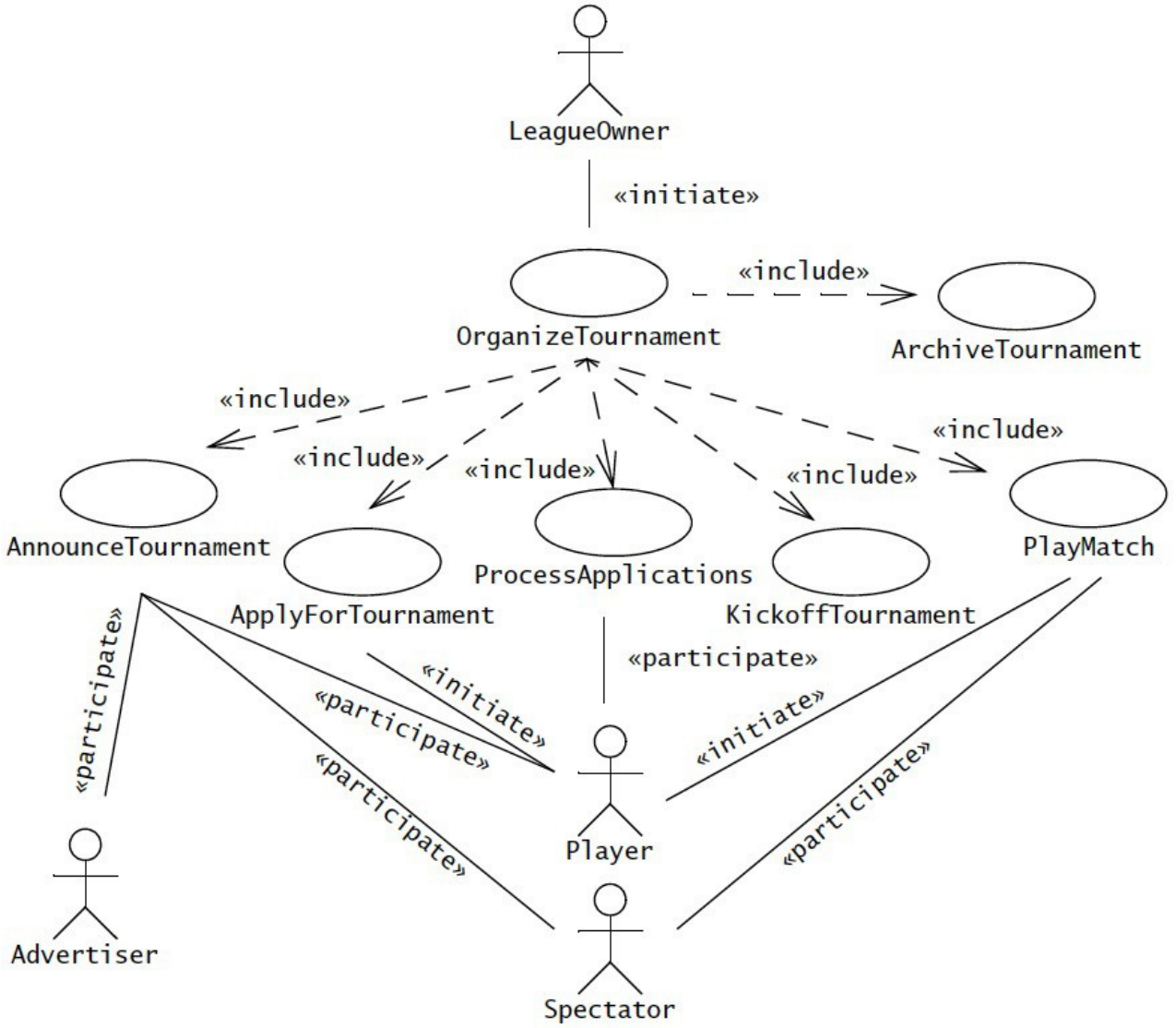


Figure 4-23 Detailed use cases refining the OrganizeTournament high-level use case.

Name	AnnounceTournament
Participating actors	Initiated by LeagueOwner Communicates with Player, Advertiser, Spectator
Flow of events	1. The LeagueOwner requests the creation of a tournament. 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. 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. 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. 5. If the LeagueOwner decides to seek an exclusive sponsor, he selects a subset of the names of the proposed sponsors. 6. The system notifies the selected sponsors about the upcoming tournament and the flat fee for exclusive sponsorships. 7. The system communicates their answers to the LeagueOwner. 8. If there are interested sponsors, the LeagueOwner selects one of them. 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. 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. 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. 12. The LeagueOwner selects which groups to notify. 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). 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.

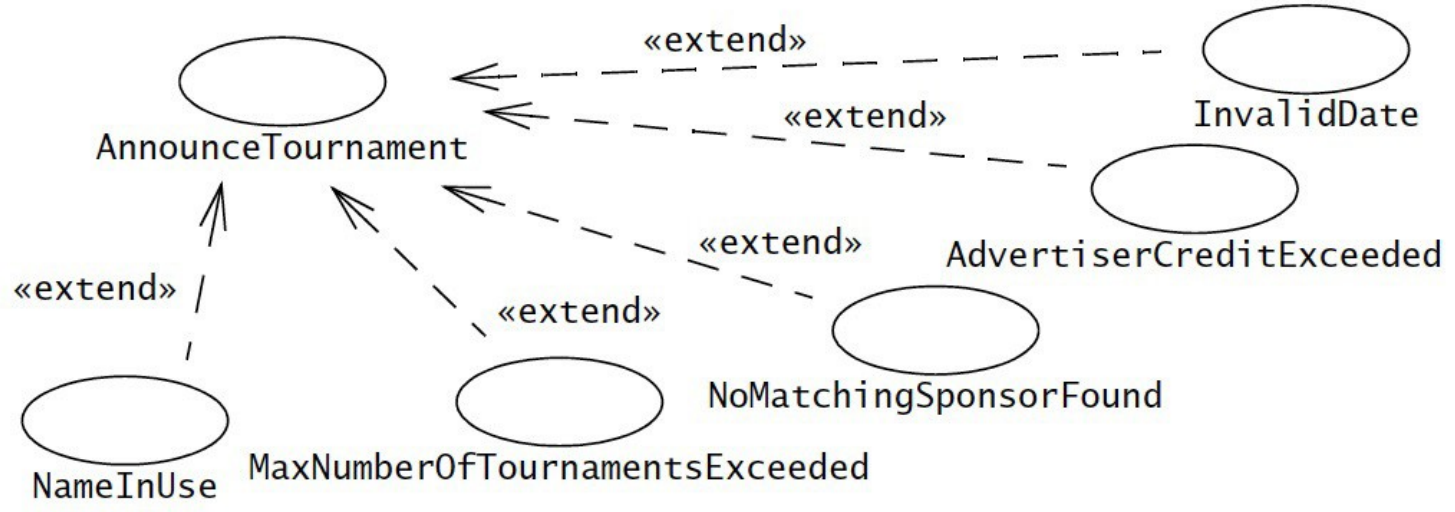
Figure 4-24 An example of a detailed use case, AnnounceTournament.

ARENA Case Study (cont.)

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`. • Potential `Players` received a notice concerning the upcoming tournament and can apply for participation. • Potential `Spectators` received a notice concerning the upcoming tournament and know when the tournament is about to start. • 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. • `Advertisers` should be able to cancel sponsorship agreements within a fixed period, as required by local laws.

Figure 4-24 *Continued.*

ARENA Case Study (cont.)



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).

Dorc Slayer Case Study

- Group exercise:
  - identify actors and system boundaries
  - define high-level use cases
    - draw use case diagrams, including all relationships
  - defined detailed use cases
    - draw use case diagrams, including all relationships

Requirements Elicitation Recap

- What we learned:
  - extract functional and non-functional requirements from a problem description
  - construct the corresponding functional model, using use cases and scenarios
  - incorporate traceability into every work product