Section 3.2
Initial System Decomposition

1. Overview
2. Concepts
3. Activities
4. MyTrip example
5. ARENA case study

3.2.1 Overview

- Learning outcomes
  - understand different types of architectural styles
    - repository, client-server, peer-to-peer, MVC, 3-tier, 4-tier
  - break down analysis object model into very high-level subsystems, using UML component diagrams

Overview (cont.)

- Initial system decomposition strategy
  - establish design goals
    - based on non-functional requirements
  - prioritize design goals
    - minimize trade-offs
  - determine the high-level subsystems, using:
    - design goals
    - analysis object model
    - dynamic model
    - selected system architectural style
    - our main goal: minimal inter-subsystem dependencies

3.2.2 Subsystem Decomposition Concepts

- Subsystems and classes
- Services and subsystem interfaces
- Coupling and cohesion
- Layers and partitions
- System architectural styles

Subsystems and Classes

- What is a *subsystem*?
  - a group of related classes
  - one part of the system
  - part of the solution domain

- Characteristics of subsystems
  - they have well-defined interfaces
  - they are relatively independent from each other
  - they encapsulate the state and behaviour of their classes
  - they can be assigned to one developer, or one team of developers

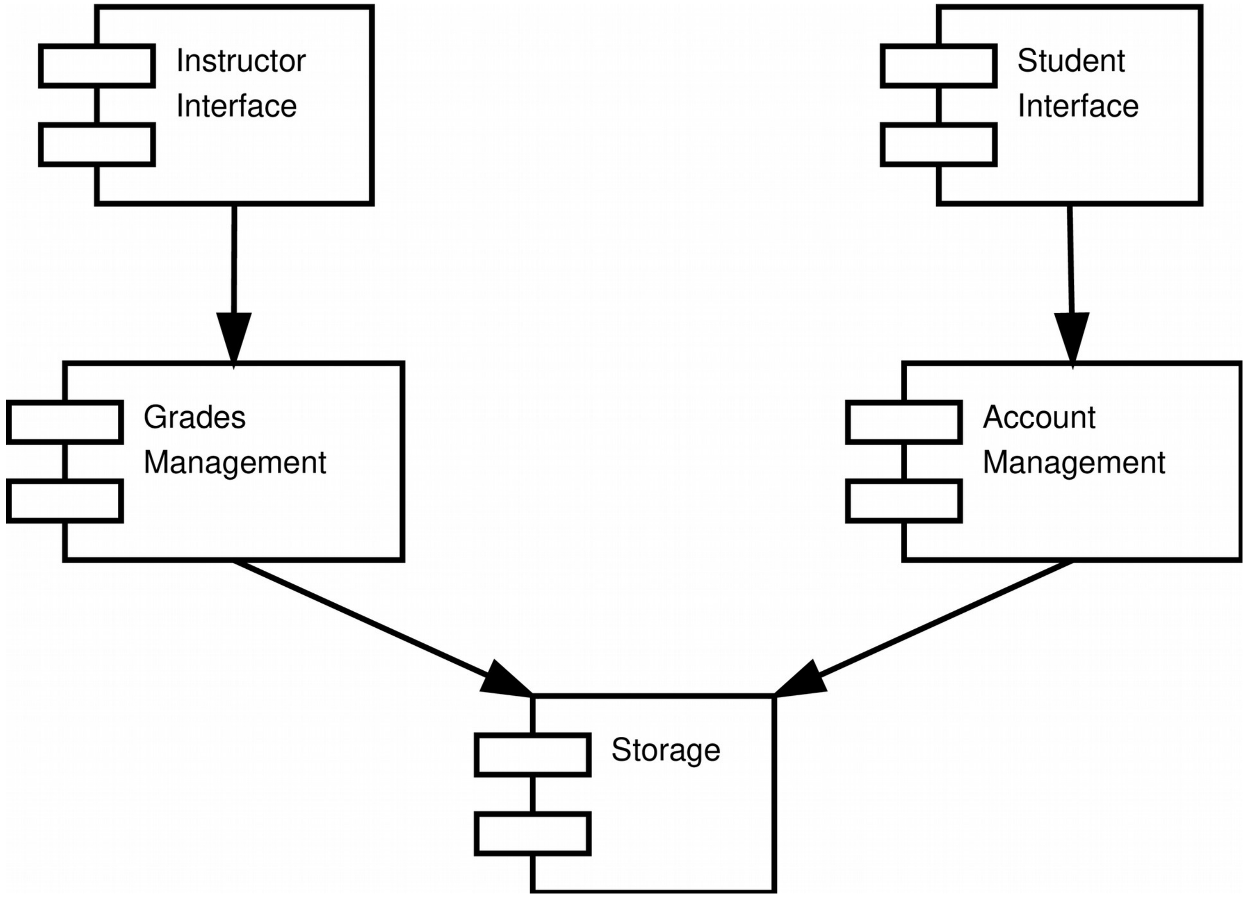
Subsystems and Classes (cont.)

- Types of subsystems
  - logical:
    - is conceptual, part of design only
    - has no runtime equivalent
  - physical:
    - is part of the code
    - has an explicit runtime equivalent

- Implementation
  - Java has packages
  - C/C++ has no explicit grouping constructs, but conventions exist
    - for example, related classes can be grouped by sub-directory

Subsystems and Classes (cont.)

- Example of subsystems:



Services and Subsystem Interfaces

- Class interface
  - the set of public operations provided by the class
- Subsystem interface
  - the set of public operations provided by all classes in the subsystem
- Service
  - a name given to a subset of related operations in the subsystem interface
    - name must must be a noun phrase
  - a subsystem can provide multiple services to other subsystems

Services and Subsystem Interfaces (cont.)

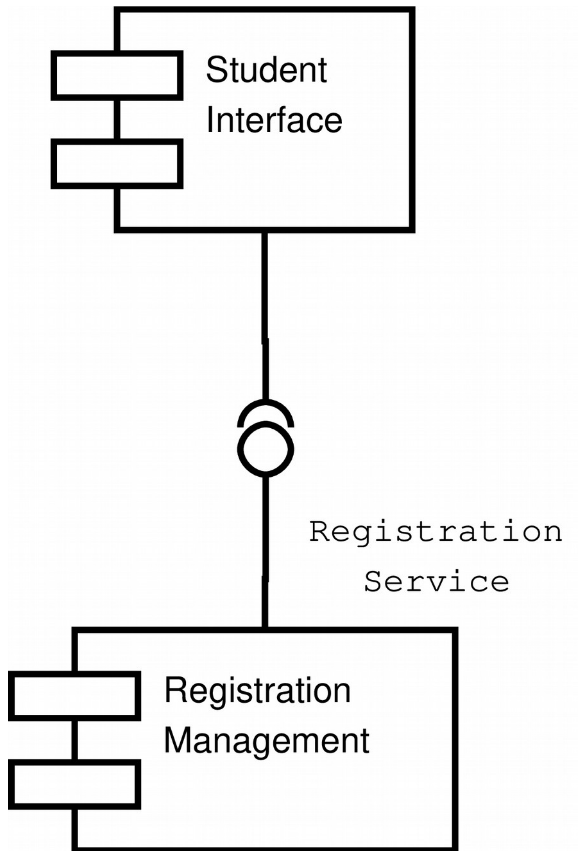
- What is a *service*?
  - a set of related operations that share a common purpose
  - they are operations provided by the classes inside a subsystem

- Characteristics of services
  - each subsystem offers services to other subsystems
  - services make up the subsystem interface
  - subsystem interface includes operation:
    - names, parameters, types, return values

- You must minimize implementation details at this stage
  - we are still far from the code

Services and Subsystem Interfaces (cont.)

- Example of a service



Coupling and Cohesion

- Coupling
  - the number of dependencies (associations) between subsystems
  - *loose coupling*
    - the subsystems are relatively independent
    - modifications to one subsystem have little impact on the other
  - *strong coupling*
    - many dependencies exist between the subsystems
    - modifications to one subsystem have strong impact on the other

- Subsystems should be as *loosely coupled* as possible
  - example: an interface to a Storage subsystem should not be dependent on the type of database used

Coupling and Cohesion (cont.)

- Cohesion
  - the number of dependencies (associations) within a subsystem
  - *high cohesion*
    - many objects in the subsystem are related to each other
    - objects perform similar tasks
  - *low cohesion*
    - the subsystem contains many unrelated objects

- Subsystems should be as *highly cohesive* as possible

- Trade-off between coupling and cohesion

Layers and Partitions

- Layer:
  - a grouping of subsystems that provide related services
  - characteristics
    - subsystem decomposition results in an ordered set of layers
    - each layer:
      - depends on (uses the services of) the lower-level layers
      - knows nothing of high-level layers

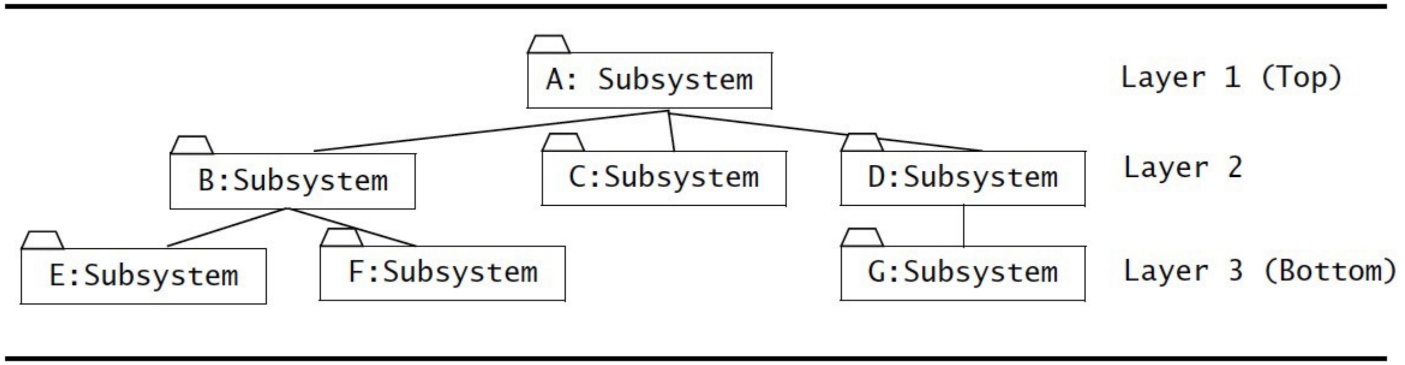


Figure 6-9 Subsystem decomposition of a system into three layers (UML object diagram, layers depicted as packages). A subset from a layered decomposition that includes at least one subsystem from each layer is called a vertical slice. For example, the subsystems A, B, and E constitute a vertical slice, whereas the subsystems D and G do not.

Layers and Partitions (cont.)

- What is a *closed architecture*?
  - each layer can **only** use the layer immediately below
  - has loose coupling
  - introduces overhead
- What is an *open architecture*?
  - each layer can use **any** lower layer

Layers and Partitions (cont.)

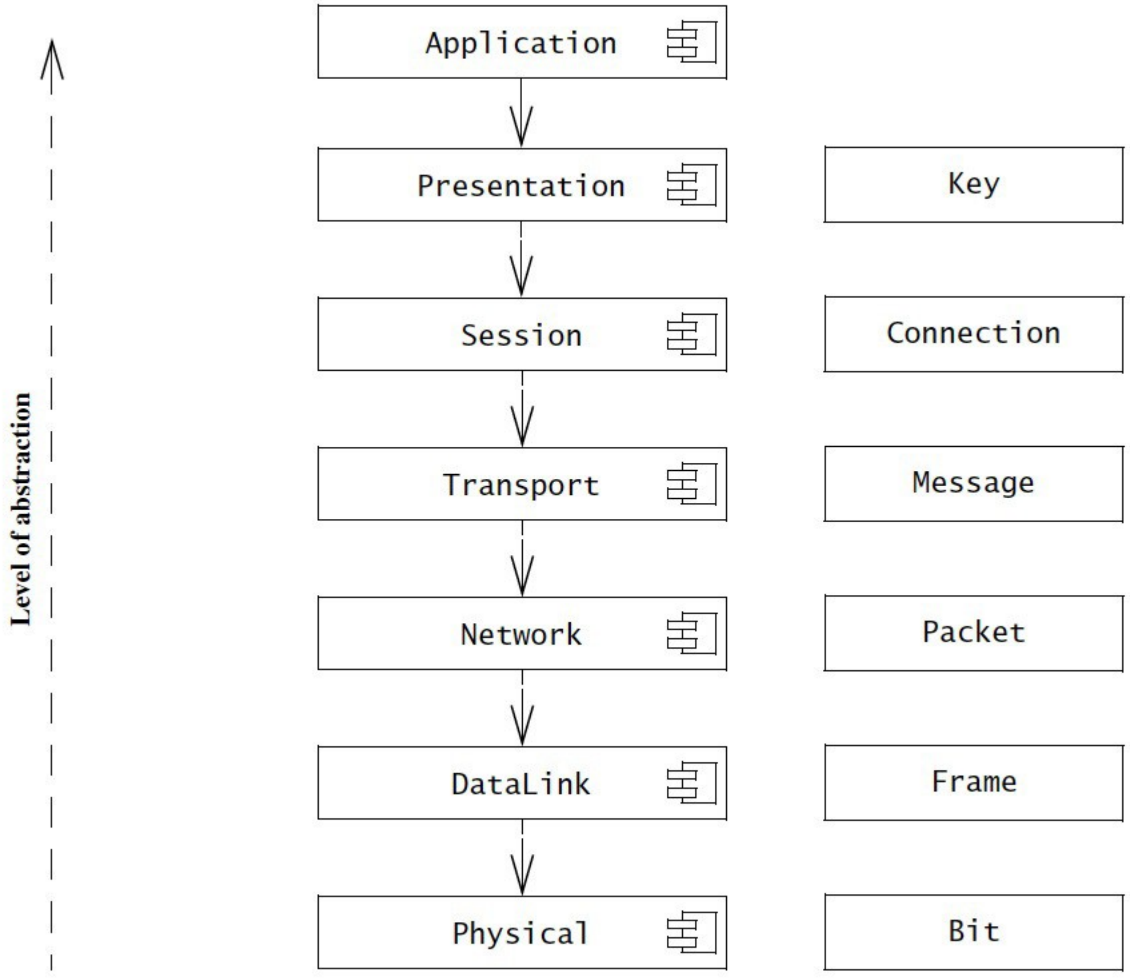


Figure 6-10 An example of closed architecture: the OSI model (UML component diagram). The OSI model decomposes network services into seven layers, each responsible for a different level of abstraction.

Layers and Partitions (cont.)

- Partition
  - a group of peer subsystems
    - each group is responsible for a different set of services
    - each group provides different functionality
  - characteristics:
    - partitions are very loosely coupled
    - they can operate in isolation from each other
  - example:
    - in *cuLearn*, the functionality for assignment submission and discussion forums can be partitioned

- Subsystem decomposition == partitioning + layering

System Architecture Styles

- What are software architecture styles?
  - a way of organizing and grouping subsystems at the highest levels
  - they can be used as a basis for a new system

- Examples:
  - repository
  - model/view/controller (MVC)
  - client-server
  - peer-to-peer
  - three-tier
  - four-tier
  - pipe and filter

Repository

- Characteristics
  - subsystems:
    - access and modify a single data structure (the repository)
    - are *independent* from each other
    - communicate only through the repository
  - control flow can be dictated by:
    - the repository, as triggers on data
    - the subsystems, synchronized through repository locks
- Examples
  - database management systems

Repository (cont.)

- Example

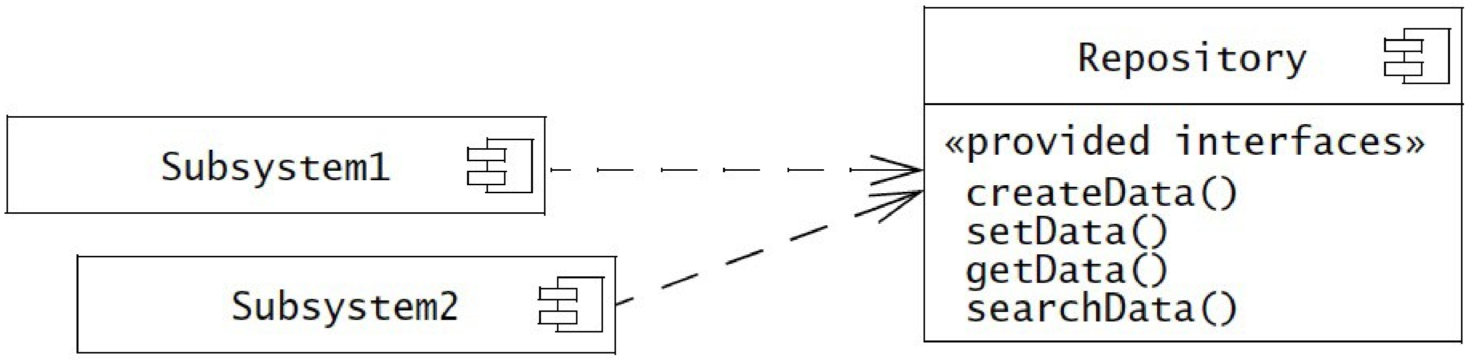


Figure 6-13 Repository architectural style (UML component diagram). Every Subsystem depends only on a central data structure called the Repository. The Repository has no knowledge of the other Subsystems.

Repository (cont.)

- Advantages
  - good for applications with:
    - complex data processing
    - changing data processing tasks
  - new services are easily added
- Disadvantages
  - central repository is a bottleneck
  - high coupling between repository and subsystems
    - changes to repository has impact across all subsystems

Model/View/Controller (MVC)

- Characteristics
  - model subsystems maintain the application domain knowledge
  - view subsystems display knowledge to the user
  - controller subsystems manage the sequence of interactions
  - model subsystems are independent of view or controller ones
  - Observer design pattern propagates model changes to the view
  - MVC is a special case of the repository architectural style

- Example
  - multi-player online games

MVC (cont.)

- Example

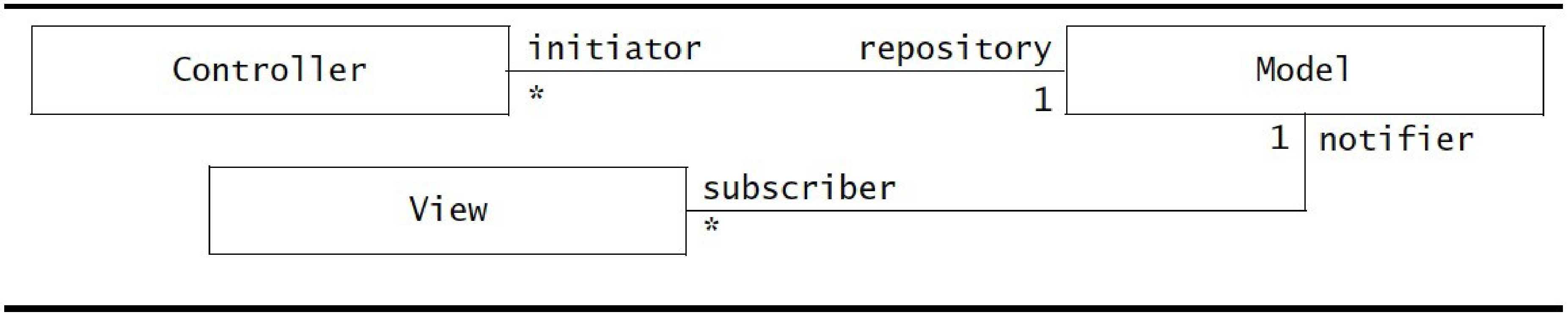


Figure 6-15 Model/View/Controller architectural style (UML class diagram). The Controller gathers input from the user and sends messages to the Model. The Model maintains the central data structure. The Views display the Model and are notified (via a subscribe/notify protocol) whenever the Model is changed.

MVC (cont.)

- Advantages
  - loose coupling between view and model
    - view can change without impact on the model
    - different views can be linked to the same model
  - maps well to entity-boundary-control object categories
  - based on the Observer design pattern

Client-Server

- Characteristics
  - server subsystems provide services to client subsystems
  - client subsystems interact with users
  - requests to the server are usually handled with:
    - remote procedure calls (RPC)
    - sockets
  - client and server:
    - are usually independent processes (executables)
    - have independent control flow
    - synchronize only on requests/replies
  - client-server can be a special case of repository style

Client-Server (cont.)

- Example
  - information systems with a central database
- Advantages
  - well-suited to distributed systems
  - can support multiple clients and multiple servers

Client-Server (cont.)

- Example

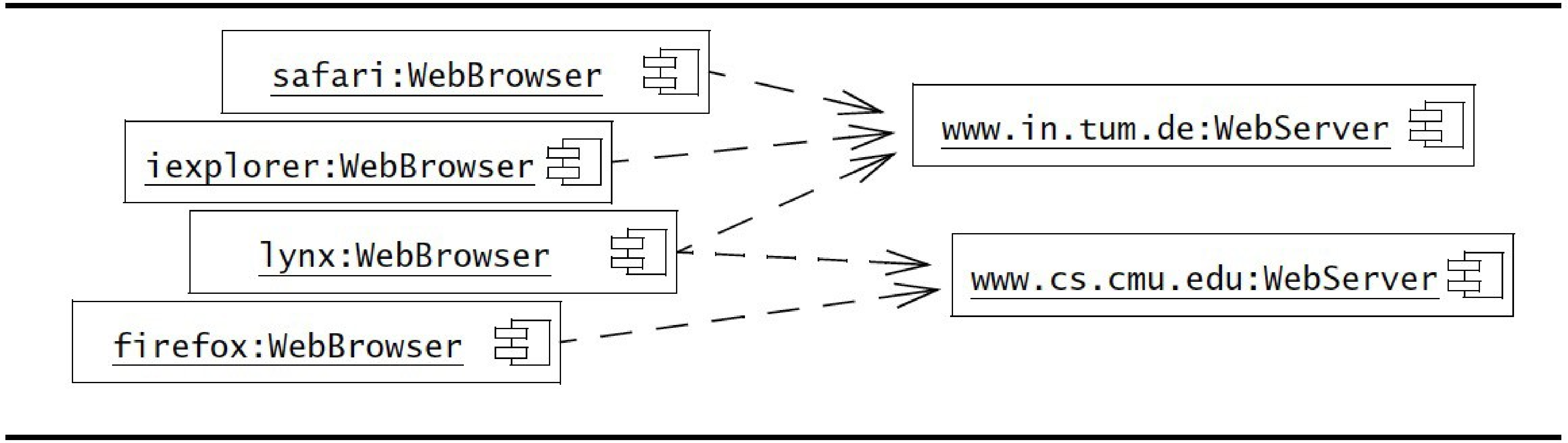


Figure 6-19 The Web as an instance of the client/server architectural style (UML deployment diagram).

Peer-to-Peer

- Characteristics
  - generalization of client-server
  - a subsystem can be both a client and a server
  - subsystems:
    - have independent control flows
    - synchronize only on requests/replies
- Example
  - a database that accepts requests and notifies others of changes

Peer-to-Peer (cont.)

- Advantage
  - same as client-server
- Disadvantage
  - risk of deadlocks

Peer-to-Peer (cont.)

- Example

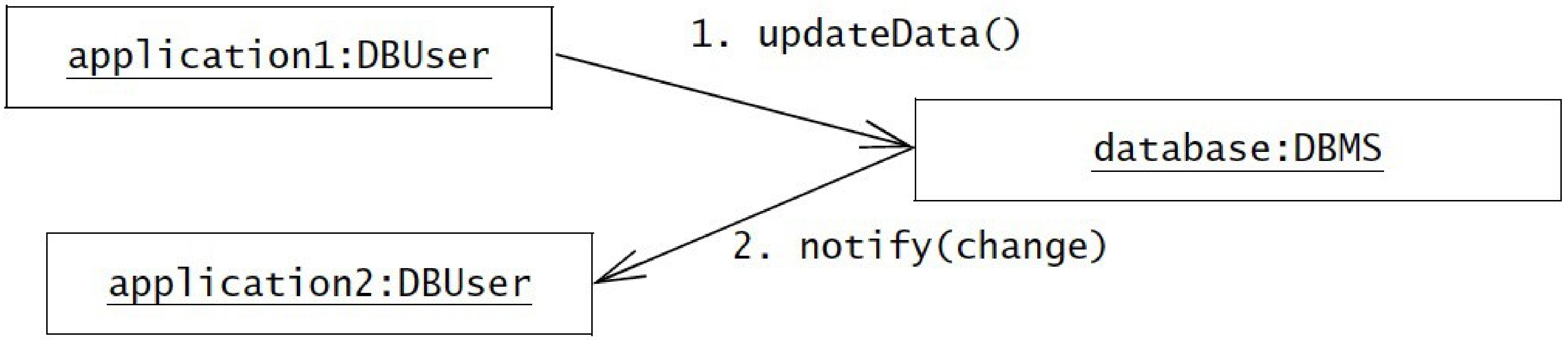


Figure 6-21 An example of peer-to-peer architectural style (UML communication diagram). The database server can both process requests from and send notifications to applications.

Three-Tier

- Subsystems are organized in three layers
  - interface layer
    - includes all boundary objects
  - application logic layer
    - includes all control and entity objects
    - responsible for processing and notification
  - storage layer
    - implements storage and retrieval of persistent objects
    - can be shared by different applications

Three-Tier (cont.)

- Example

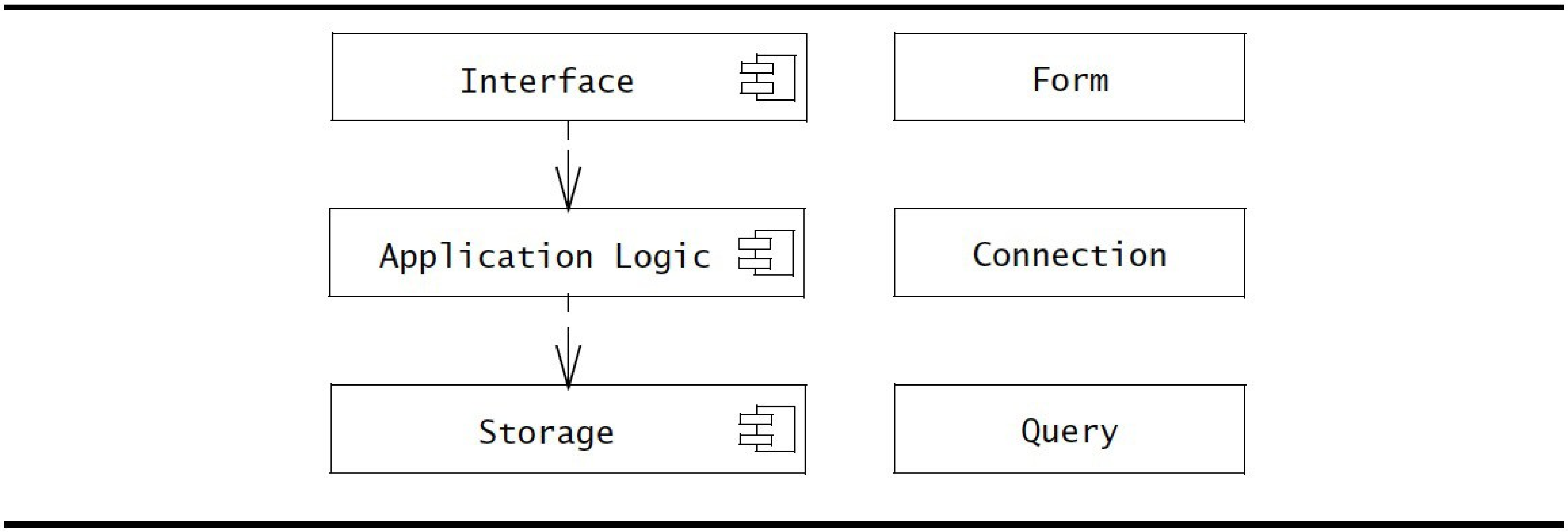


Figure 6-22 Three-tier architectural style (UML component diagram). Objects are organized into three layers realizing the user interface, the processing, and the storage.

Four-Tier

- Characteristics
  - similar to three-tier
  - interface layer is separated into:
    - presentation client layer
      - located on client hosts
    - presentation server layer
      - located on server hosts
  - different presentation clients == different UIs
  - common data across UIs is processed in presentation server layer

Four-Tier (cont.)

- Example

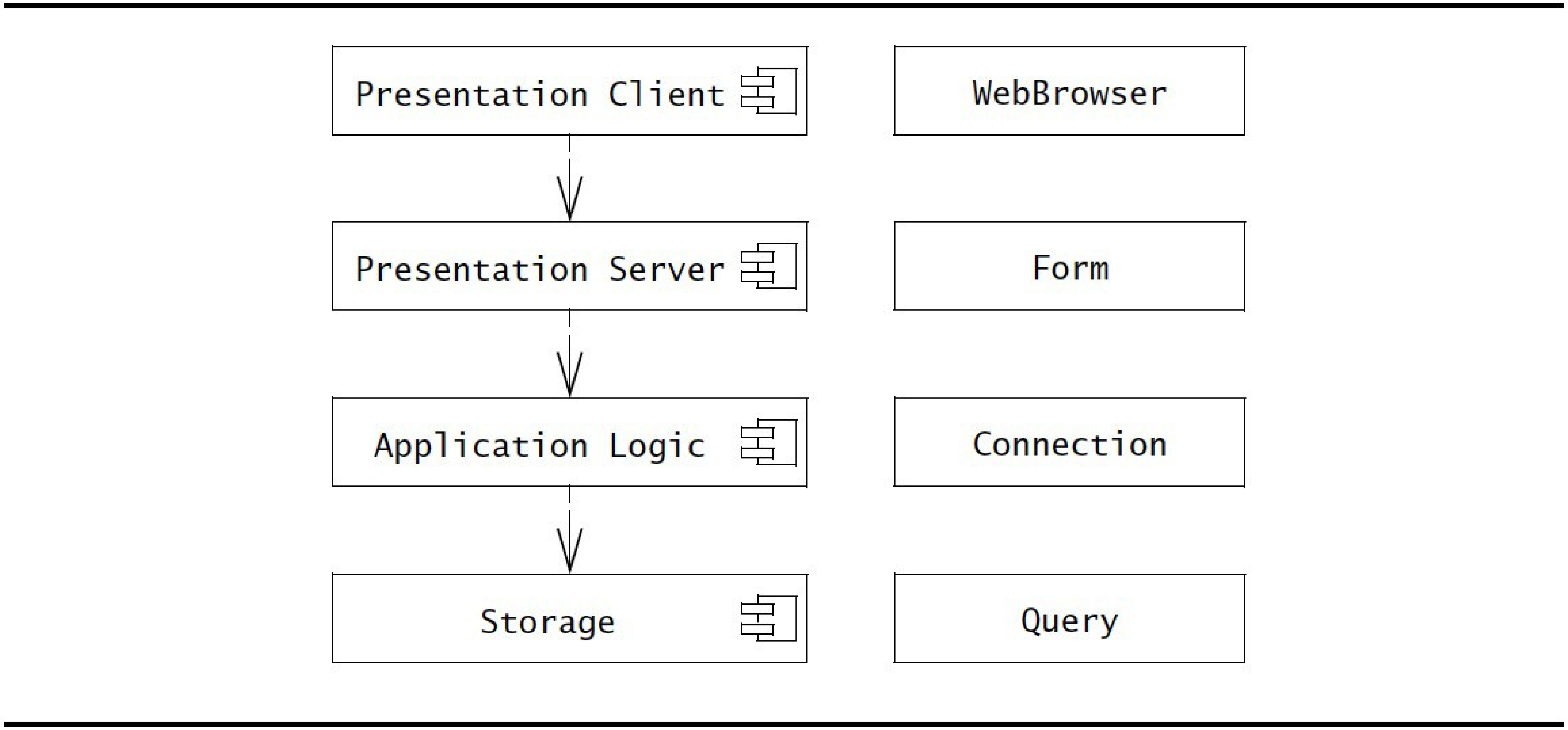


Figure 6-23 Four-tier architectural style (UML component diagram). The `Interface` layer of the three-tier style is split into two layers to enable more variability on the user interface style.

Pipe and Filter

- Characteristics
  - definitions:
    - filters == subsystems
    - pipes == association between subsystems
  - each filter:
    - processes a set of inputs, and sends results as a set of outputs
    - only knows format of input data, independent of filter that produced it
  - filters are independent of each other
  - filters are synchronized by pipes

- Example
  - Unix shell

Pipe and Filter (cont.)

- Example

```
% ps auxwww | grep dutoit | sort | more

dutoit   19737  0.2  1.6 1908 1500 pts/6    O 15:24:36  0:00 -tcsh
dutoit   19858  0.2  0.7  816  580 pts/6    S 15:38:46  0:00 grep dutoit
dutoit   19859  0.2  0.6  812  540 pts/6    O 15:38:47  0:00 sort
```

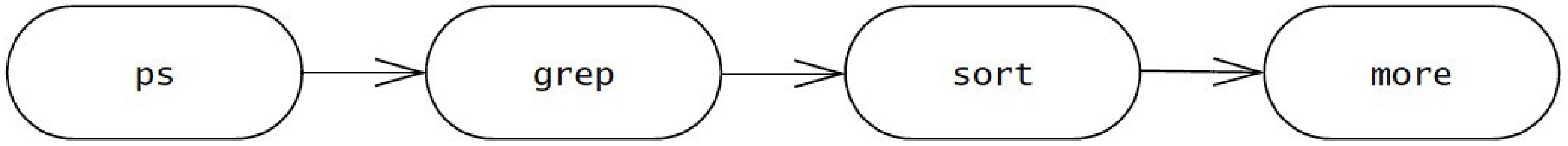


Figure 6-25 Unix command line as an instance of the pipe and filter style (UML activity diagram).

Pipe and Filter (cont.)

- Advantage
  - good for systems that transform streams of data
- Disadvantage
  - not good for complex interactions between filters

3.2.3 Subsystem Decomposition Activities

- Identifying design goals
- Identifying subsystems

Identifying Design Goals

- What are design goals?
  - qualities that the system must focus on
- Source
  - inferred from the non-functional requirements
  - inferred from the application domain
  - elicited from the client

Identifying Design Goals (cont.)

- Five design criteria to consider
  - performance
  - dependability
  - cost
  - maintenance
  - end user

Identifying Design Goals (cont.)

- Performance criteria

response time:	time to acknowledge user request
throughput:	# of tasks the system can do in fixed time
memory:	space required for system to run

- Dependability criteria

robustness:	ability to survive invalid user input
reliability:	diff. between specified and observed behaviour
availability:	% of time that system can be used
fault tolerance:	ability to work under erroneous conditions
security:	ability to withstand attacks
safety:	ability to avoid endangering lives

Identifying Design Goals (cont.)

- Maintenance criteria

➢ extensibility:	ease of adding functionality
➢ modifiability:	ease of changing functionality
➢ adaptability:	ease of porting system to different app. domains
➢ portability:	ease of porting system to different platforms
➢ readability:	ease of understanding the system from the code
➢ traceability:	ease of mapping the code back to requirements

Identifying Design Goals (cont.)

- Cost criteria

development:	cost of developing initial system
deployment:	cost of installing system and training users
upgrade:	cost of porting data from previous system
maintenance:	cost of bug fixes and enhancements
administration:	cost of administrating the system

- End user criteria

utility:	how well the system supports user tasks
usability:	how easy the system is to use

Identifying Design Goals (cont.)

- Strategy to identify design goals
  - examine design criteria to find equivalent in requirements
    - performance, dependability, end user criteria:
      - usually found in requirements or application domain
    - cost and maintenance criteria:
      - dictated by client
  - prioritize design goals and establish trade-offs
    - space vs. speed
    - delivery time vs. functionality
    - delivery time vs. quality
    - delivery time vs. staffing

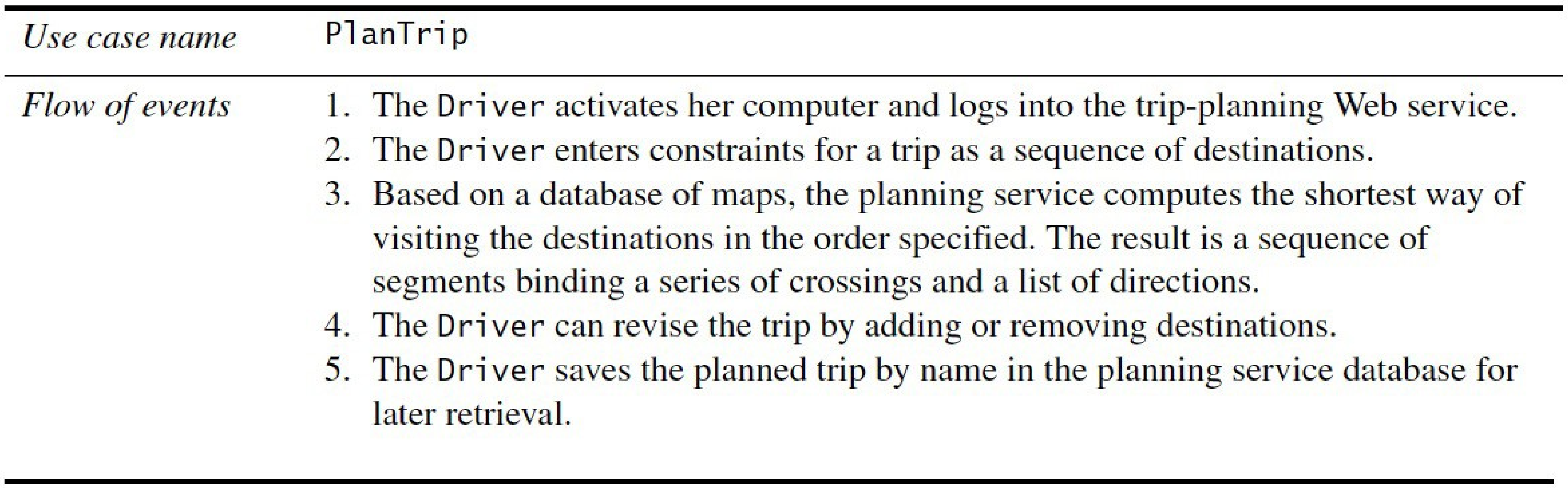
Identifying Subsystems

- Initial decomposition
  - similar to finding objects in the Analysis phase
  - derived from the requirements
  - goal is to group subsystems with:
    - low coupling
    - high cohesion

Identifying Subsystems (cont.)

- Strategy
  - start by assigning the objects in one use case to same subsystem
  - create dedicated subsystems to move data between subsystems
  - minimize number of associations crossing subsystem boundaries
  - all objects within a subsystem should be functionally related

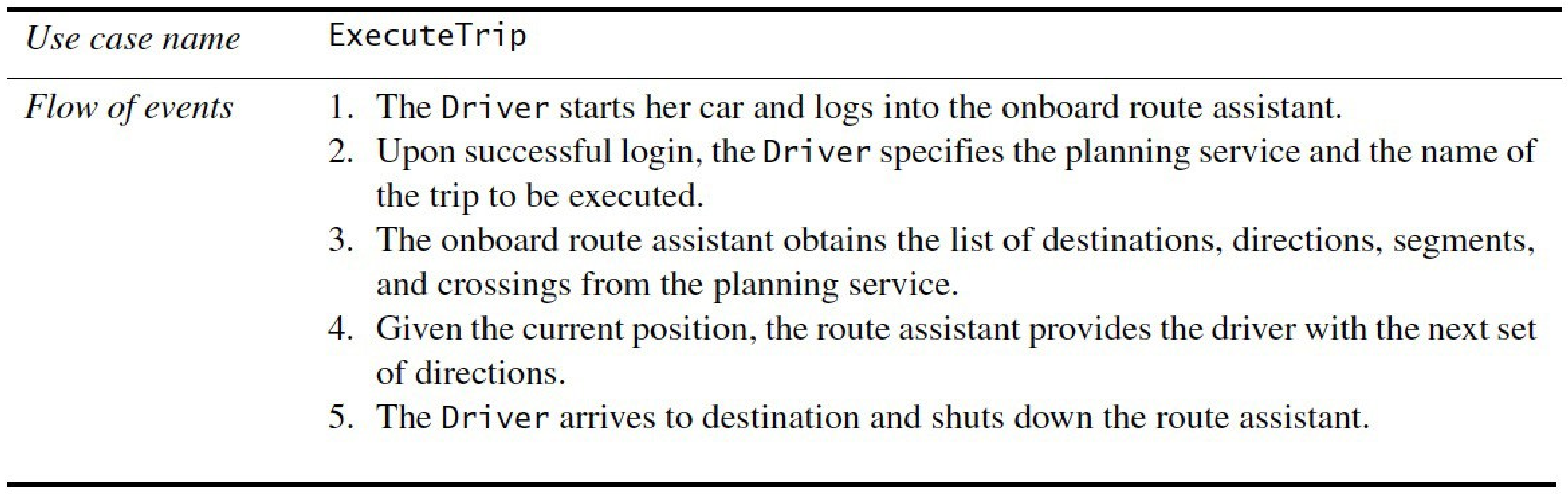
3.2.4 MyTrip Example



Use case name	`PlanTrip`
Flow of events	1. The `Driver` activates her computer and logs into the trip-planning Web service. 2. The `Driver` enters constraints for a trip as a sequence of destinations. 3. Based on a database of maps, the planning service computes the shortest way of visiting the destinations in the order specified. The result is a sequence of segments binding a series of crossings and a list of directions. 4. The `Driver` can revise the trip by adding or removing destinations. 5. The `Driver` saves the planned trip by name in the planning service database for later retrieval.

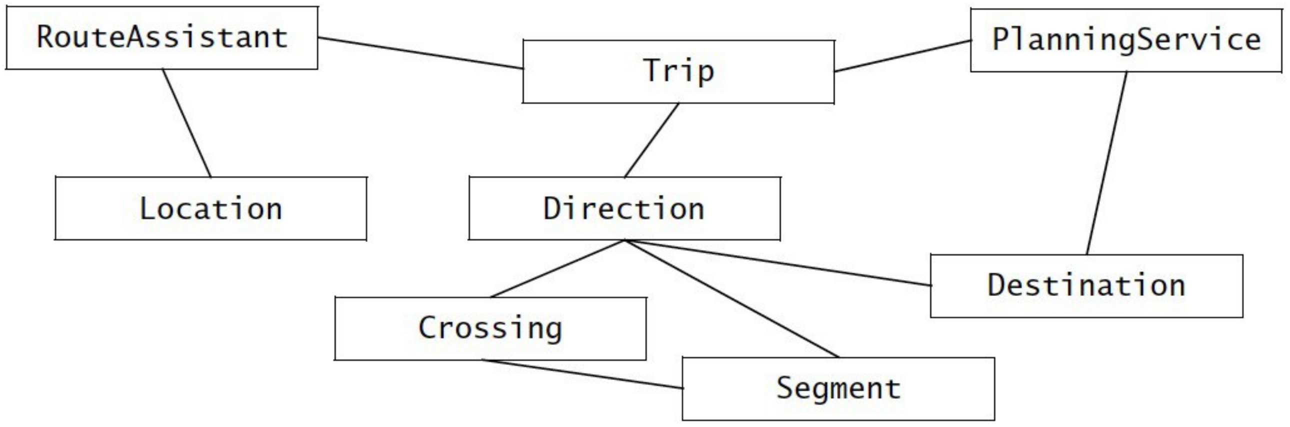
Figure 6-26 `PlanTrip` use case of the `MyTrip` system.

MyTrip Example (cont.)



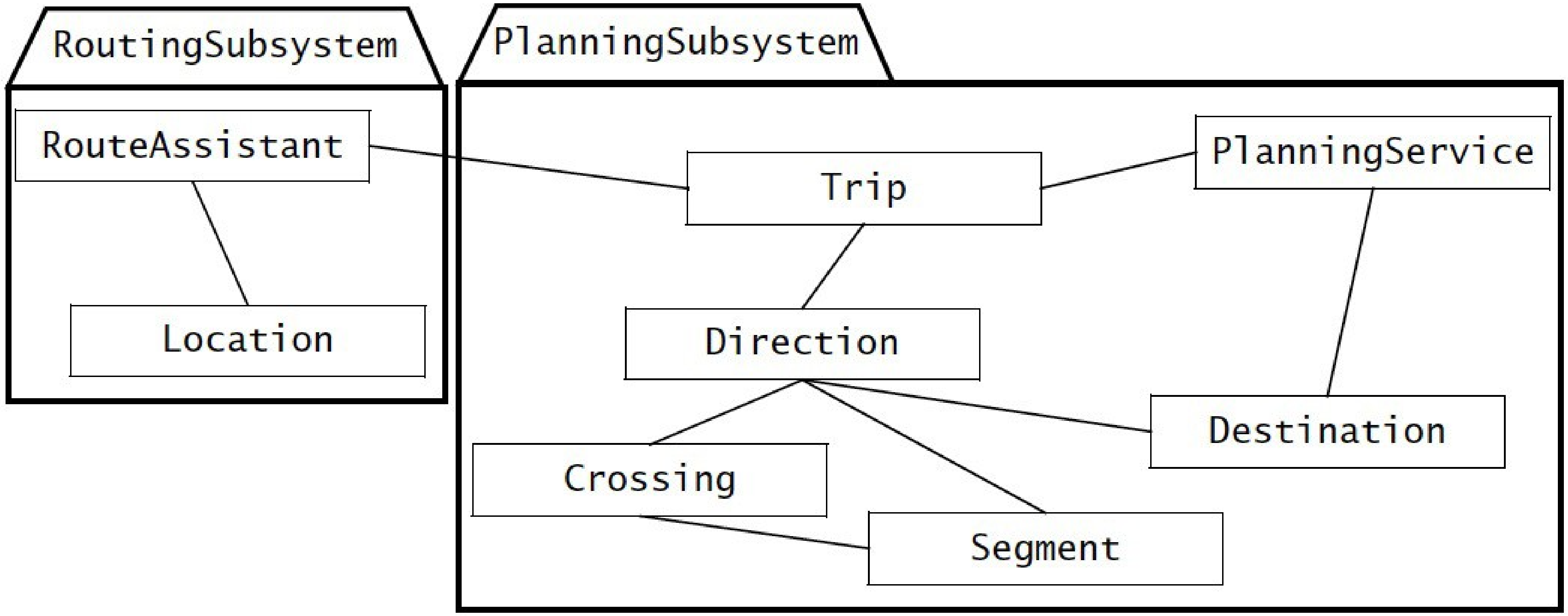
Use case name	`ExecuteTrip`
Flow of events	1. The `Driver` starts her car and logs into the onboard route assistant. 2. Upon successful login, the `Driver` specifies the planning service and the name of the trip to be executed. 3. The onboard route assistant obtains the list of destinations, directions, segments, and crossings from the planning service. 4. Given the current position, the route assistant provides the driver with the next set of directions. 5. The `Driver` arrives to destination and shuts down the route assistant.

Figure 6-27 `ExecuteTrip` use case of the `MyTrip` system.



Crossing	A Crossing is a geographical point where several Segments meet.
Destination	A Destination represents a location where the driver wishes to go.
Direction	Given a Crossing and an adjacent Segment, a Direction describes in natural language how to steer the car onto the given Segment.
Location	A Location is the position of the car as known by the onboard GPS system or the number of turns of the wheels.
PlanningService	A PlanningService is a Web server that can supply a trip, linking a number of destinations in the form of a sequence of Crossings and Segments.
RouteAssistant	A RouteAssistant gives Directions to the driver, given the current Location and upcoming Crossing.
Segment	A Segment represents the road between two Crossings.
Trip	A Trip is a sequence of Directions between two Destinations.

Figure 6-28 Analysis model for the MyTrip route planning and execution.



PlanningSubsystem	The PlanningSubsystem is responsible for constructing a Trip connecting a sequence of Destinations. The PlanningSubsystem is also responsible for responding to replan requests from RoutingSubsystem.
RoutingSubsystem	The RoutingSubsystem is responsible for downloading a Trip from the PlanningService and executing it by giving Directions to the driver based on its Location.

Figure 6-29 Initial subsystem decomposition for MyTrip (UML class diagram).

3.2.5 ARENA Case Study

- Non-functional requirements:

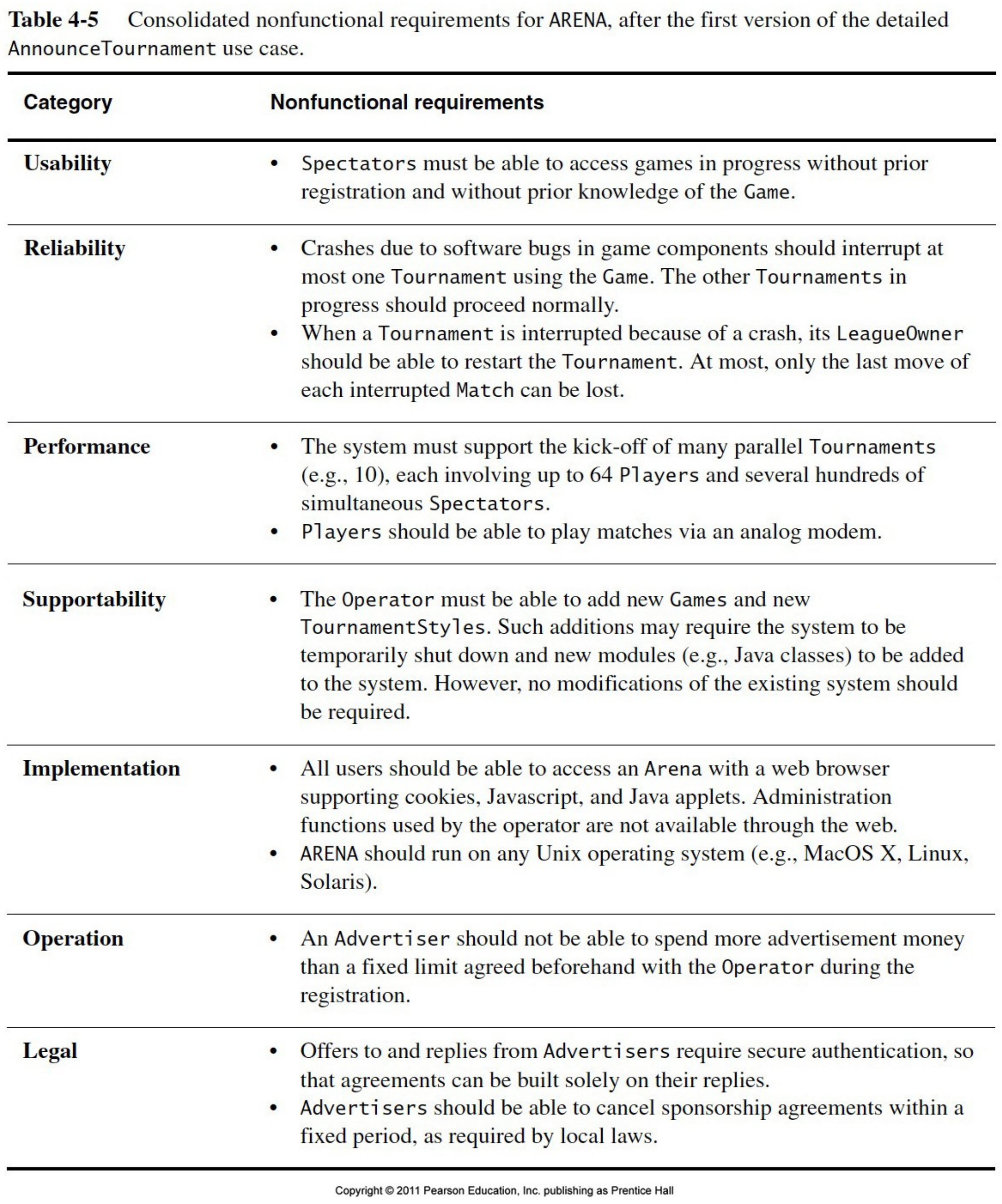


Table 4-5 Consolidated nonfunctional requirements for ARENA, after the first version of the detailed AnnounceTournament use case.

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. • 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. • 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. • 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. • Advertisers should be able to cancel sponsorship agreements within a fixed period, as required by local laws.

ARENA Case Study (cont.)

- Design goals
  - low operating cost
  - high availability
  - scalability in terms of:
    - number of players
    - concurrent tournaments
  - ease of adding new games
  - documentation for open source development

ARENA Subsystems

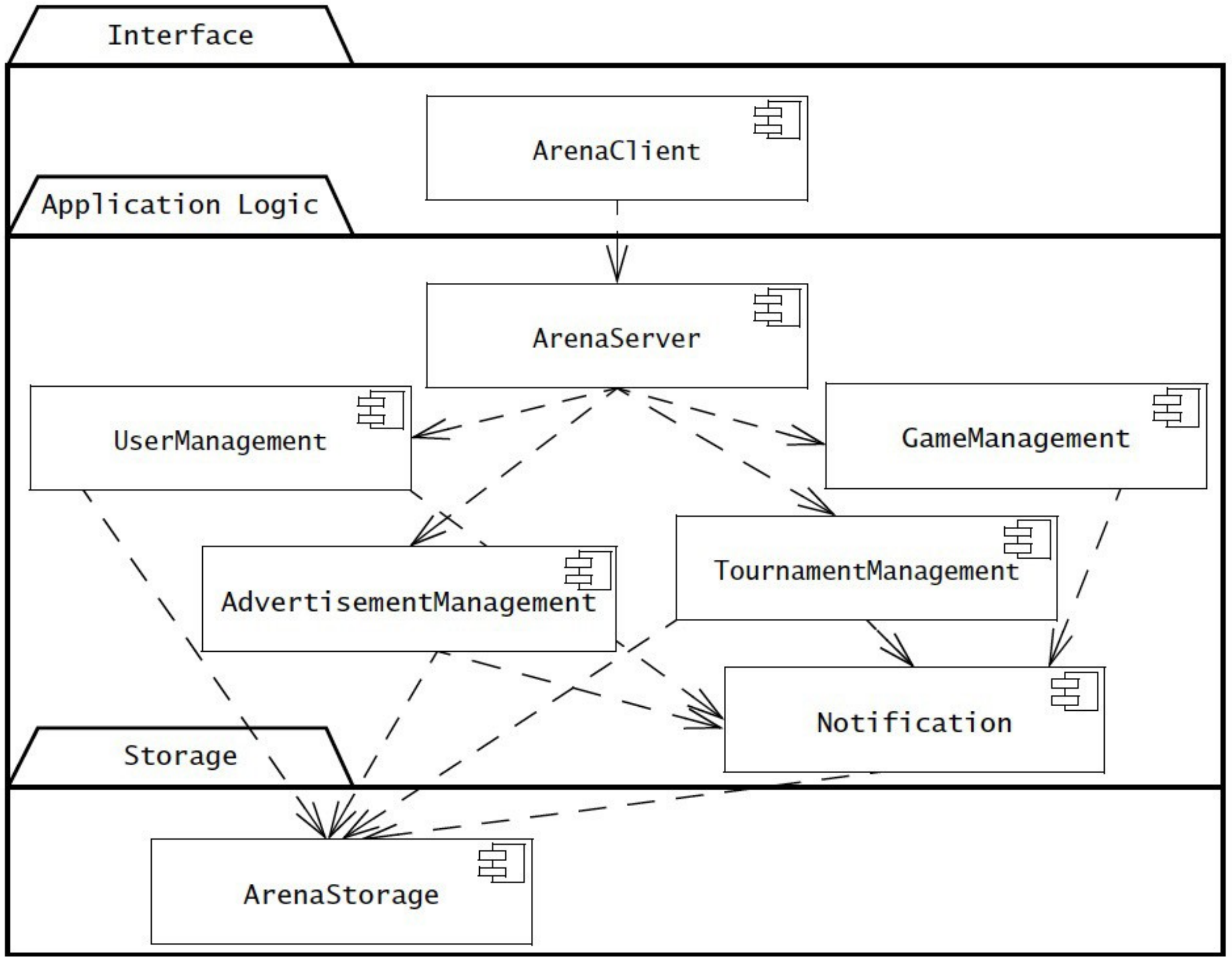


Figure 7-19 ARENA subsystem decomposition, game organization part (UML component diagram, layers shown as UML packages).

ARENA Subsystems (cont.)

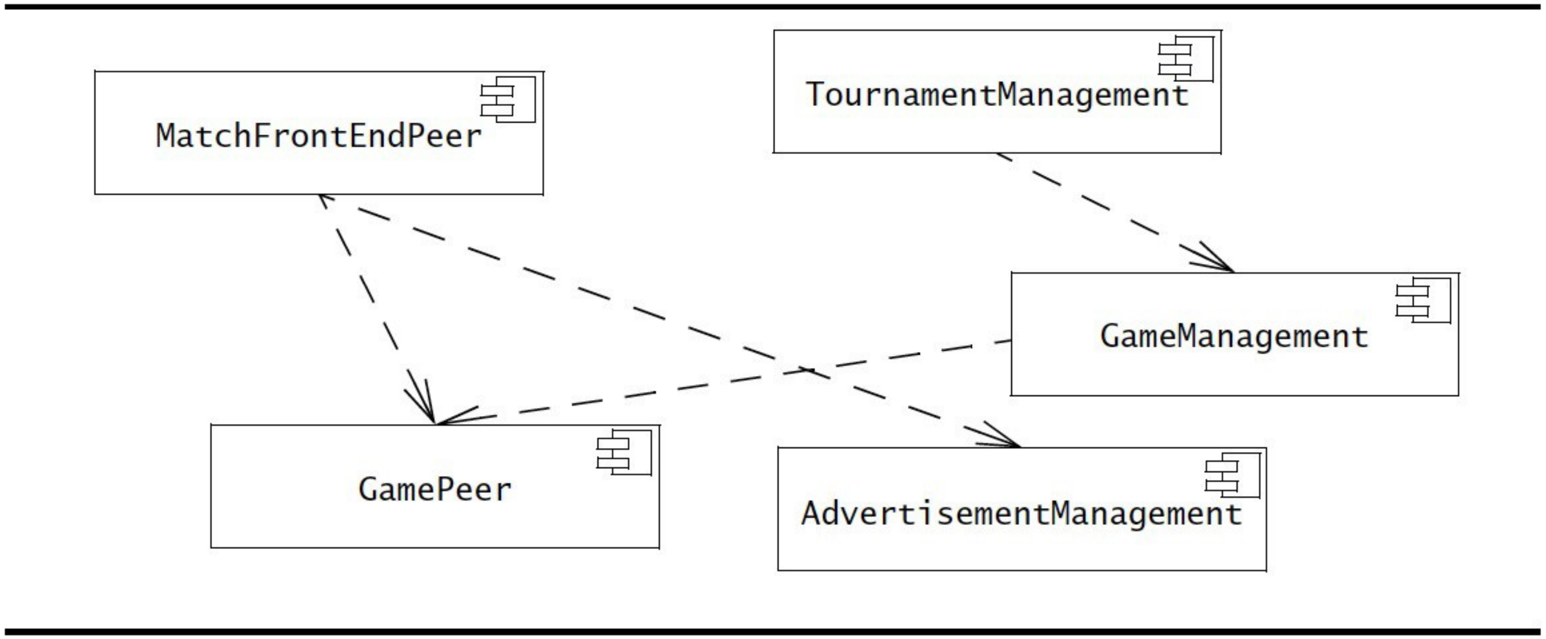


Figure 7-20 ARENA subsystem decomposition, game playing part (UML component diagram).

DorcSlayer Case Study

- Group exercise:
    - identify design goals
    - identify architectural style
        - justify your selection
    - identify high-level subsystems
        - group objects into subsystems

Recap

- What we learned:
  - understand different types of architectural styles
    - repository, client-server, peer-to-peer, MVC, 3-tier, 4-tier
  - break down analysis object model into very high-level subsystems, using UML component diagrams