Section 3.4
Refined System Decomposition

1. Overview
2. Components and nodes
3. Activities
4. MyTrip example

3.4.1 Overview

- Learning outcomes
  - ➢ group subsystems into components and nodes
  - ➢ identify persistent data, access control and global control flow mechanisms, boundary conditions
  - ➢ identify subsystem services and interfaces

Overview (cont.)

- Refined system decomposition strategy
  - iterate over initial subsystem decomposition
    - do this until all design goals are addressed
  - establish system strategies for:
    - hardware/software mapping
    - data management
    - access control
    - control flow
    - boundary conditions

Overview (cont.)

- System strategies:
  - hardware/software mapping
    - off-the-shelf and legacy software components
    - hardware configuration
    - inter-node communications strategy
  - data management
    - identification of persistent data, storage location, access mechanisms
  - access control
    - authentication, authorization, confidentiality

Overview (cont.)

- System strategies (cont.):
  - control flow
    - mechanism for control, concurrency
  - boundary conditions
    - system initialization, shutdown
    - exceptional cases

3.4.2 Components and Nodes

- Definitions
  - *component*:
    - consists of a large subsystem or a group of subsystems
    - provides services to actors or other components
  - *runtime component*:
    - a component that is a unique *process*
  - *node*
    - physical device or execution environment where components run

Components and Nodes (cont.)

- UML notation
  - deployment diagrams are used for components and nodes
  - they show relationships between runtime components and nodes
- A system:
  - is composed of interacting runtime components
  - these components can be distributed over multiple nodes

Components and Nodes (cont.)

- UML deployment diagram

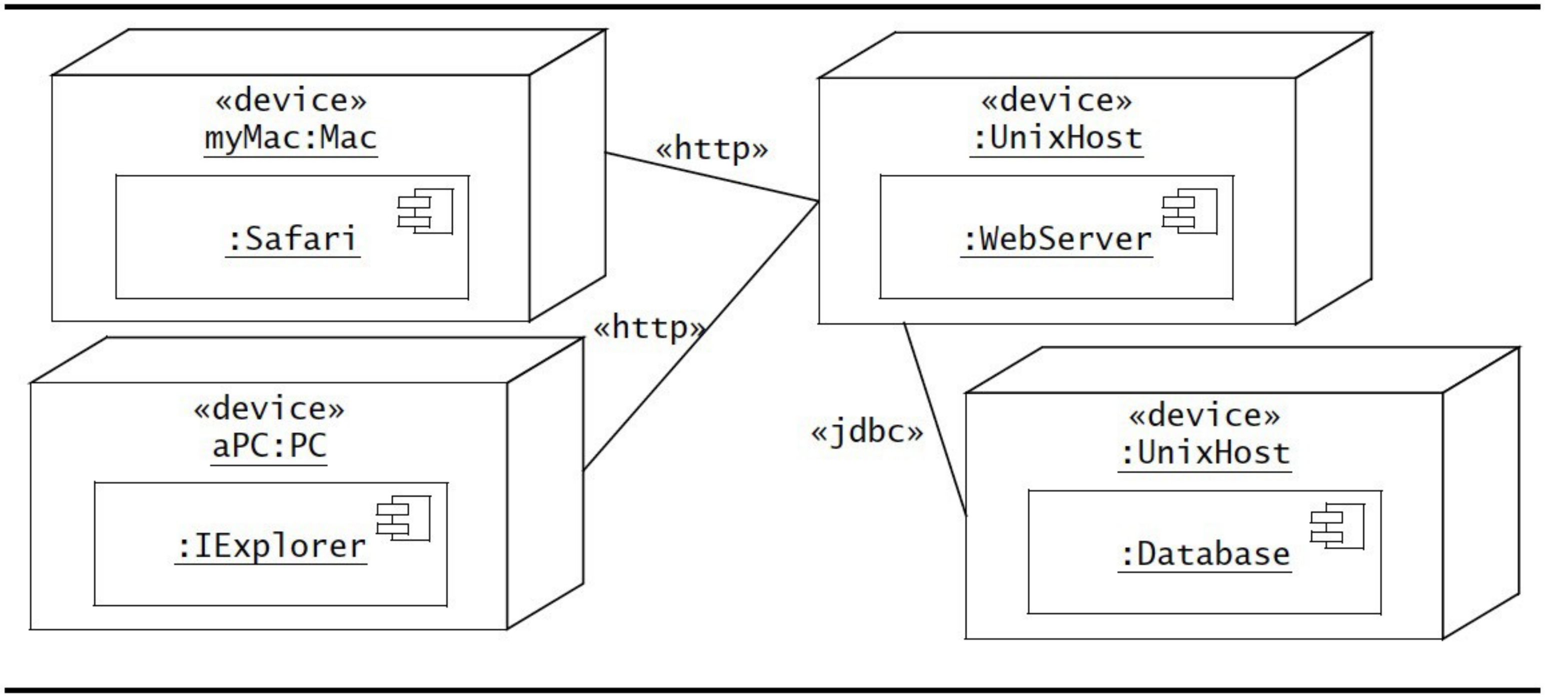


Figure 7-2 A UML deployment diagram representing the allocation of components to different nodes. Web browsers on PCs and Macs can access a WebServer that provides information from a Database.

3.4.3 Refined Decomposition Activities

- Mapping subsystems to components
- Storing persistent data
- Providing access control
- Designing global control flow
- Identifying services
- Identifying boundary conditions

Mapping Subsystems to Components

- Select hardware configuration and platform
  - decide on the required nodes and select the hardware
  - determine the mechanism to support communications
  - select the virtual machine:
    - operating system
    - required software components
      - legacy system
      - COTS (e.g. database, communications packages, UI libraries)

- Allocate objects and subsystems to nodes
  - enables equitable distribution of:
    - functionality
    - processing power

Mapping Subsystems to Components (cont.)

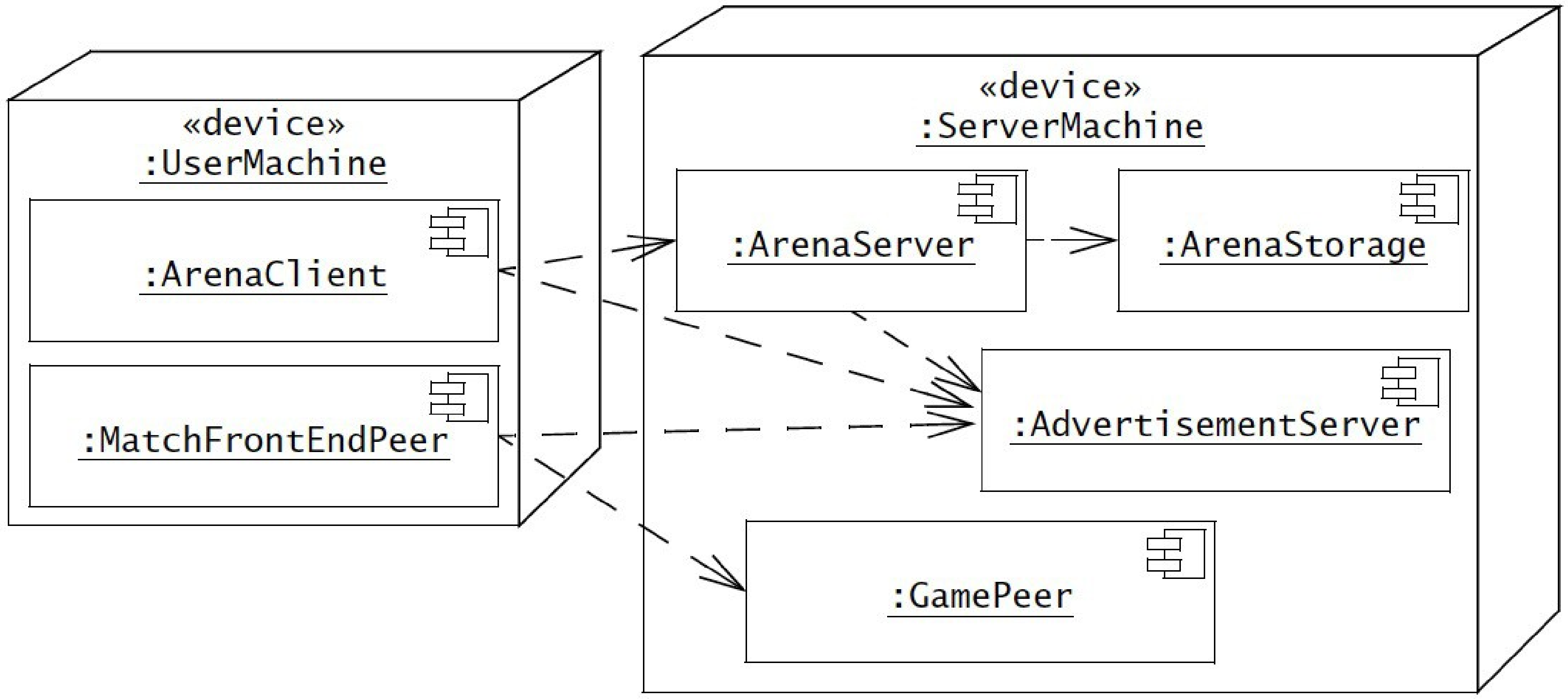


Figure 7-21 ARENA hardware/software mapping (UML deployment diagram). Note that each run-time component may support several subsystems.

Storing Persistent Data

- What is *persistent data*?
  - data that outlives a single execution of the system

- Issue
  - impact of storage method
    - control strategy
    - concurrency management

- Candidates for persistent objects
  - entity objects
  - user information
  - some aspects of boundary objects
  - classes that must survive system shutdown

Storing Persistent Data (cont.)

- Selecting a storage management strategy
  - dictated by non-functional requirements
  - options:
    - flat files
      - low-level sequence of bytes
      - application can optimize for size and speed
      - application must deal with serious issues (concurrent access, data loss)
    - relational database
      - tables of a predefined type (schema)
      - mapping objects to schema can be complex
      - provides concurrency management, access control, crash recovery
    - OO database
      - supports objects and their associations
      - slower than relational database

Storing Persistent Data (cont.)

- Trade-offs of storage management strategy
  - advantages of flat files
    - good for large data, temporary data
    - support low information density
  - advantages of databases
    - allow concurrent access, detailed access
    - support multiple platforms/applications for the same data
  - advantages of relational databases
    - support complex queries
    - support large amount of data
  - advantages of OO databases
    - support association-based queries
    - support irregular associations

Providing Access Control

- What is access control?
  - determines what actors have access to what information
- Types of access control
  - authentication
  - confidentiality
  - authorization

Providing Access Control (cont.)

- Authentication
  - ➢ whether an actor is who it says it is
  - ➢ achieved with login credentials, or smart card, or biometrics
- Confidentiality
  - ➢ whether data can be understood only by intended actors
  - ➢ achieved with encryption
- Authorization
  - ➢ whether data/operations can be accessed only by intended actors
- Always use COTS software for authentication and confidentiality

Providing Access Control (cont.)

- Strategy for ensuring authorization at object level
  - determine which objects are shared by multiple actors
  - identify which actors are authorized to perform which operations on which shared objects
  - the purpose is to **document** which actors have access
  - the implementation has other ways of regulating access
- Access to classes modelled with an *access matrix*

Providing Access Control (cont.)

- Three ways to implement an access matrix
  - global access table
    - list of (actor, class, operation) tuples
      - a *tuple* is an ordered list of elements
    - if the tuple exists, the operation is allowed, otherwise it is denied
  - access control list
    - list of (actor, operation) tuples associated with each class
    - each time an object is accessed, the list is checked
  - capability
    - list of (class, operation) tuples associated with each actor

Providing Access Control (cont.)

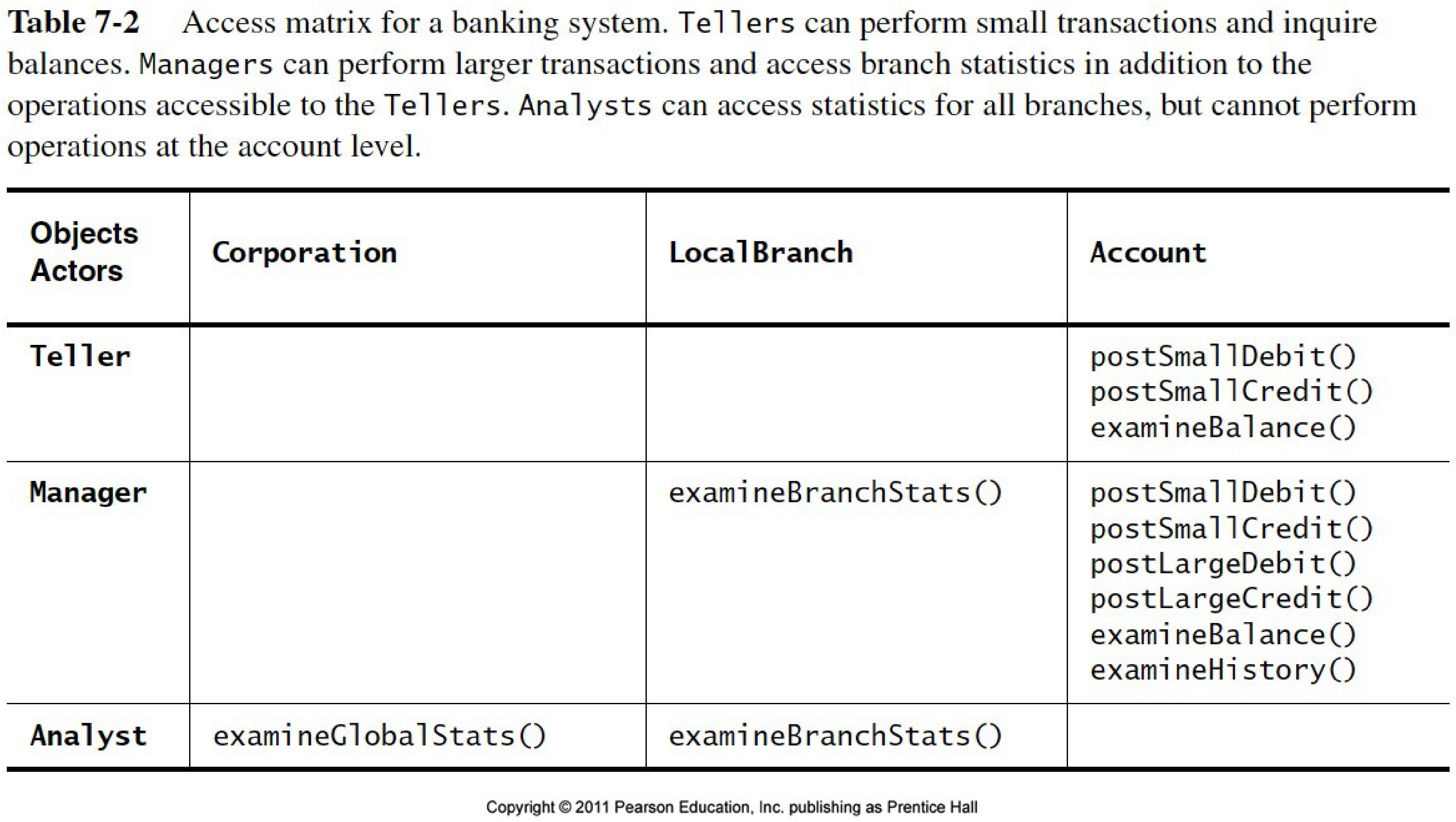


Table 7-2 Access matrix for a banking system. `Tellers` can perform small transactions and inquire balances. `Managers` can perform larger transactions and access branch statistics in addition to the operations accessible to the `Tellers`. `Analysts` can access statistics for all branches, but cannot perform operations at the account level.

Objects Actors	Corporation	LocalBranch	Account

Teller			postSmallDebit()
			postSmallCredit()
			examineBalance()

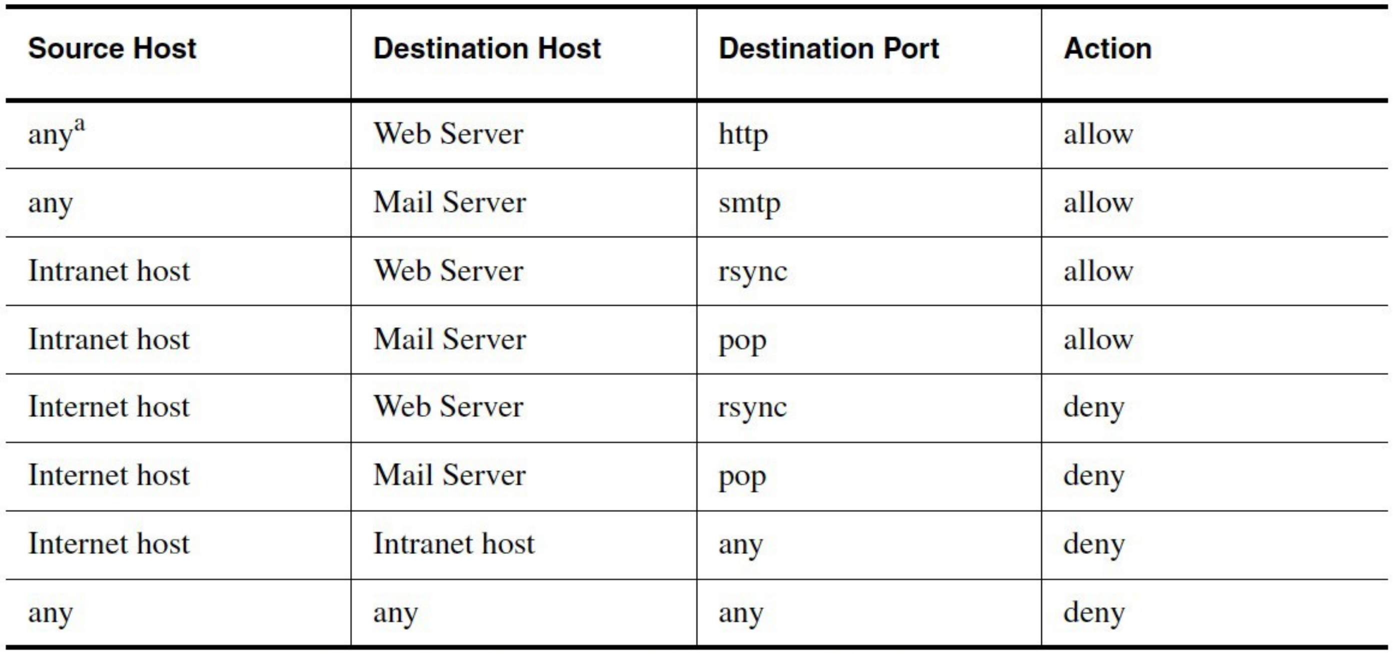
Manager		examineBranchStats()	postSmallDebit()
			postSmallCredit()
			postLargeDebit()
			postLargeCredit()
			examineBalance()
			examineHistory()

Analyst	examineGlobalStats()	examineBranchStats()	

Providing Access Control (cont.)

- Rule-based representation of access matrix
  - good for large numbers of actors and classes
  - provide compact representation
  - allows access rules between broad types of actors and classes

Table 7-3 Simplified example of packet filtering rules for firewall of Figure 7-8.



Source Host	Destination Host	Destination Port	Action
any[a]	Web Server	http	allow
any	Mail Server	smtp	allow
Intranet host	Web Server	rsync	allow
Intranet host	Mail Server	pop	allow
Internet host	Web Server	rsync	deny
Internet host	Mail Server	pop	deny
Internet host	Intranet host	any	deny
any	any	any	deny

a. **any** means any one of Intranet host, Internet host, Web Server, or Mail Server.

Providing Access Control (cont.)

- Static access control
  - access rights are known at compile time
  - can be modelled as attributes of system objects
  - implementation: access matrix

- Dynamic access control
  - access rights are only known at runtime
  - implementation: Proxy design pattern

Providing Access Control (cont.)

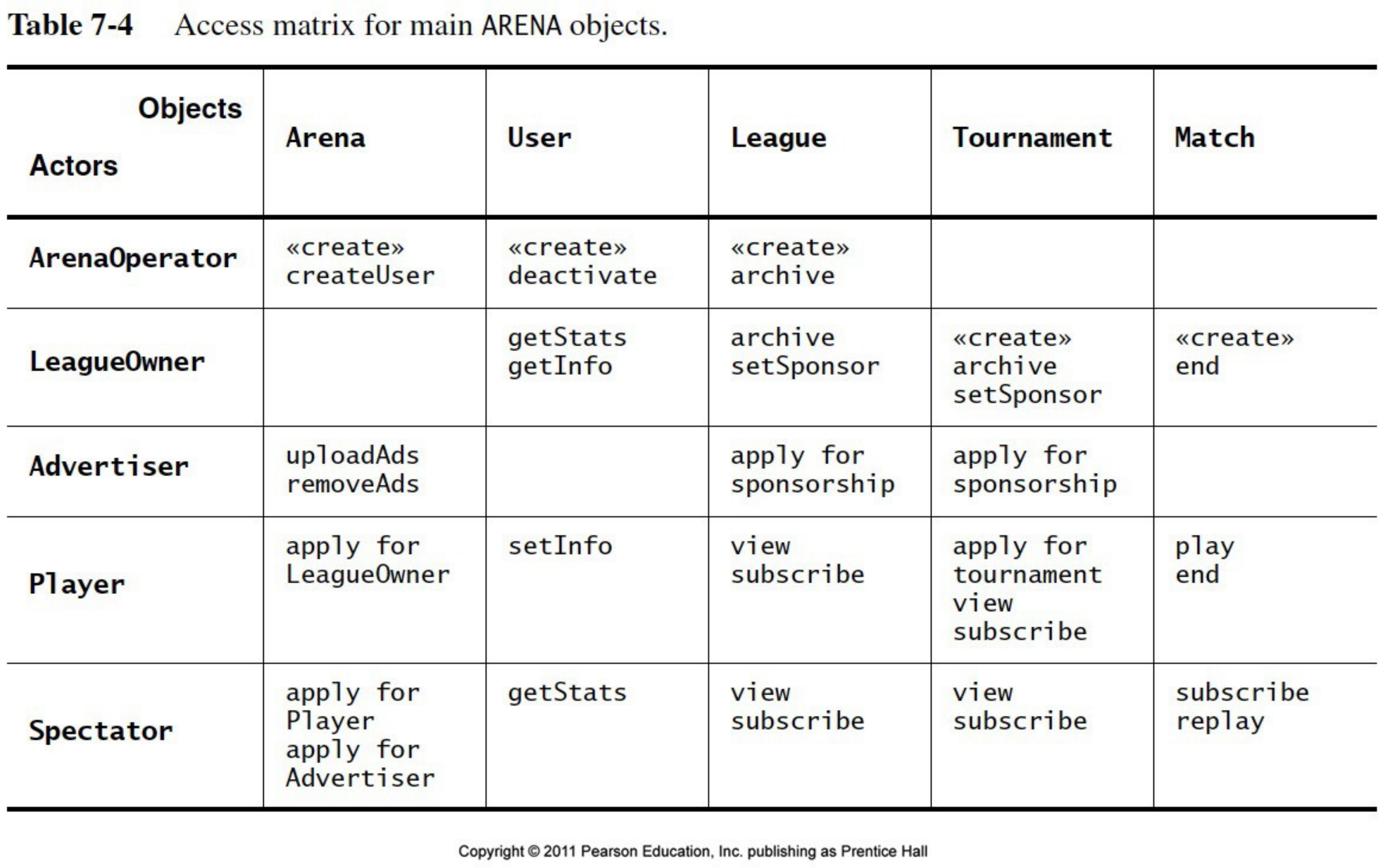


Table 7-4 Access matrix for main ARENA objects.

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

Designing Global Control Flow

- What is control flow?
  - which operations execute in what order

- Control flow mechanisms
  - procedure-driven
    - single flow of control, blocks when waiting for input
    - not natural for OO languages
  - event-driven
    - main loop waits for external event
    - events are *dispatched* to the appropriate objects
  - threads
    - concurrent variation of procedure-driven
    - each thread responds to a different event and gets input from actor
    - intuitive, but difficult to debug and test

Designing Global Control Flow (cont.)

- Role of control objects
  - they are set up to implement the selected control flow mechanism
  - they contain the control flow for a single use case
  - for each external event received:
    - they record the event
    - they store the temporary event states
    - they issue sequence of operation calls on boundary and entity objects

Identifying Services

- For each subsystem, identify the service(s) it provides

- Strategy
  - review the dependencies between subsystems
  - define an interface for each service identified
    - services are named with noun phrases
    - operations provided by interface are named with verb phrases

Identifying Services (cont.)

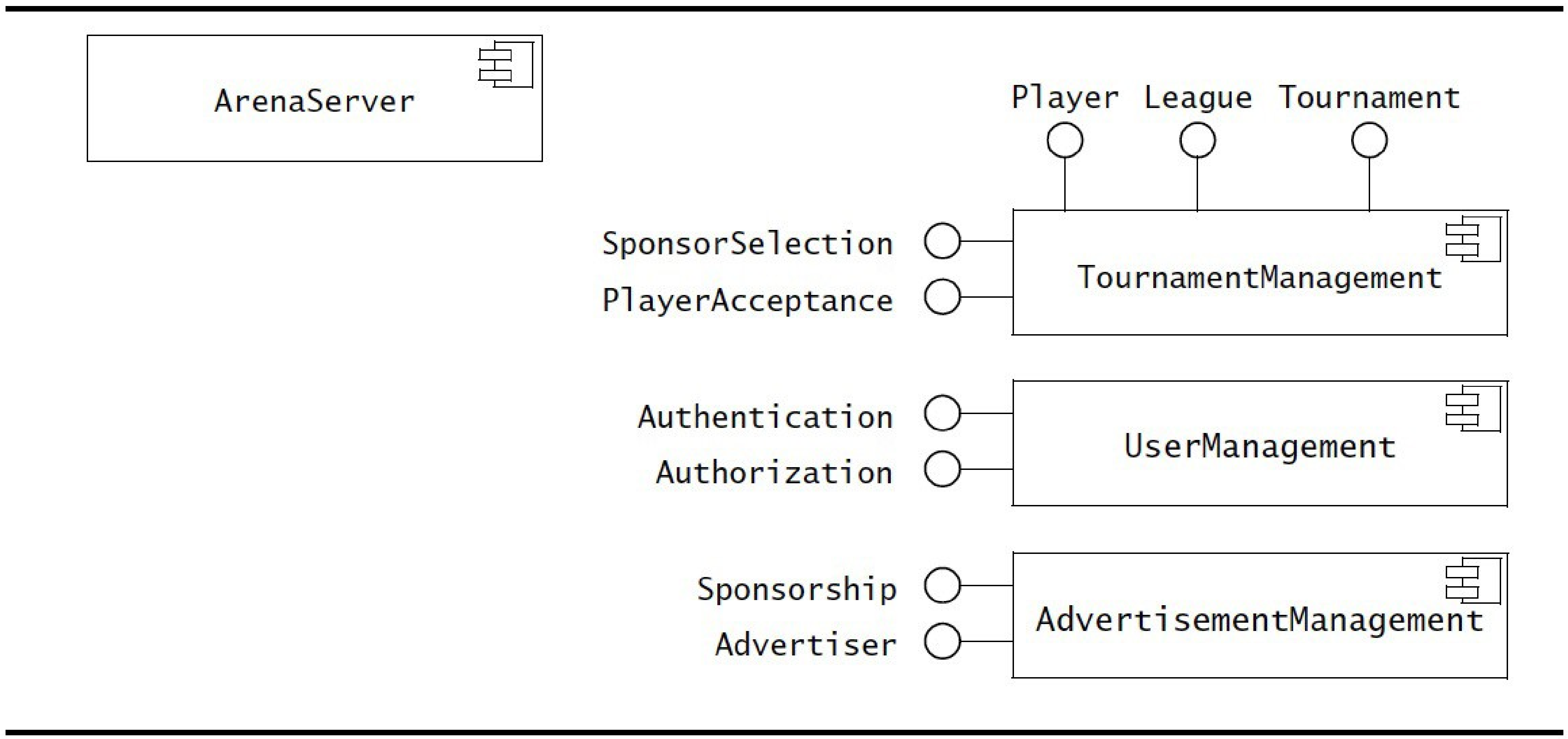


Figure 7-22 ARENA subsystem decomposition, game organization part with services identified (UML component diagram, ball-and-socket notation, dependencies omitted for clarity).

Identifying Boundary Conditions

- What is a *steady state*?
  - the normal operation of the system

- What are *boundary conditions*?
  - system behaviour outside the normal operation
  - captured in boundary use cases

- Examples
  - system startup, initialization, shutdown (normal and abnormal)
  - data corruption
  - network outages
  - administrative tasks (user management, data configuration)

Identifying Boundary Conditions (cont.)

- What is an *exception*?
  - an event or error during system execution
  - can be caused by:
    - hardware failure (e.g. hard disk, network link)
    - changes in operating environment (e.g. loss of connectivity)
    - software fault (e.g. software design error)

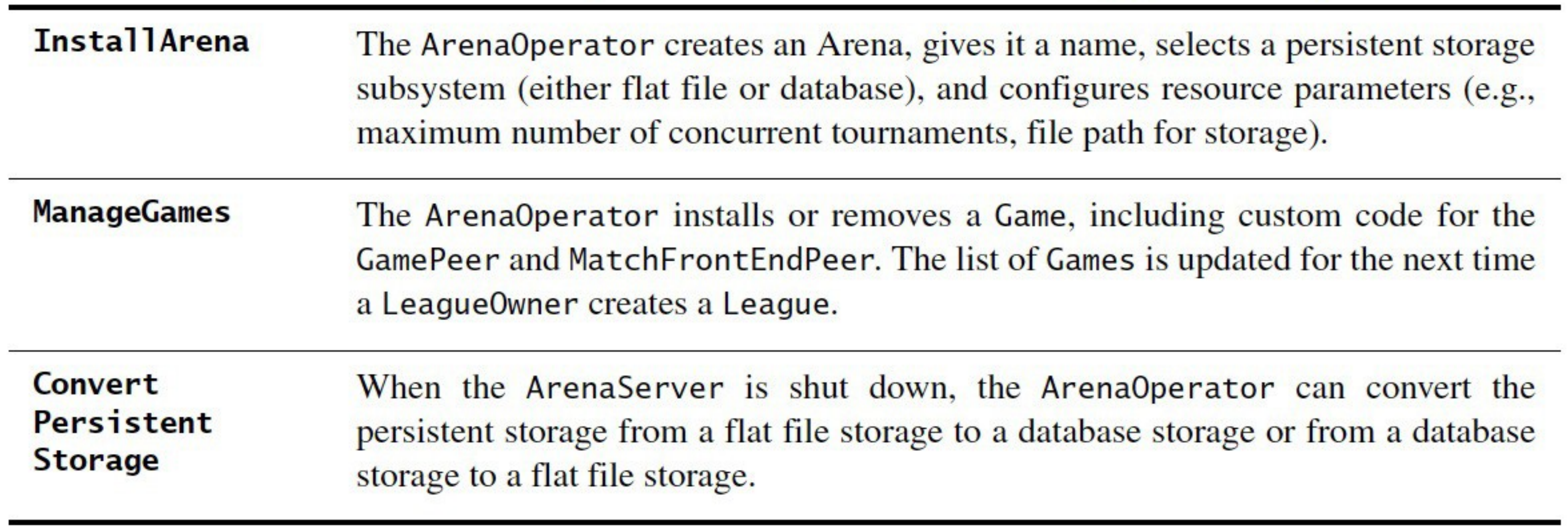
- Role of exception handling
  - mechanism with which the system deals with exceptions
    - example: error messages, logs, recovery
  - failure may be:
    - tolerated by the system
    - communicated to the user, as documented in boundary use case

Identifying Boundary Conditions (cont.)

- Strategy for identifying boundary use cases
  - configuration of persistent objects
    - examine use cases where persistent objects are created/destroyed
    - for objects never created/destroyed, add new administration use case
  - startup and shutdown of each component
    - add a use case for component startup
    - add a use case for component shutdown
    - add a use case for component configuration
  - exception handling for each type of component failure
    - decide on system action
    - add extending use case

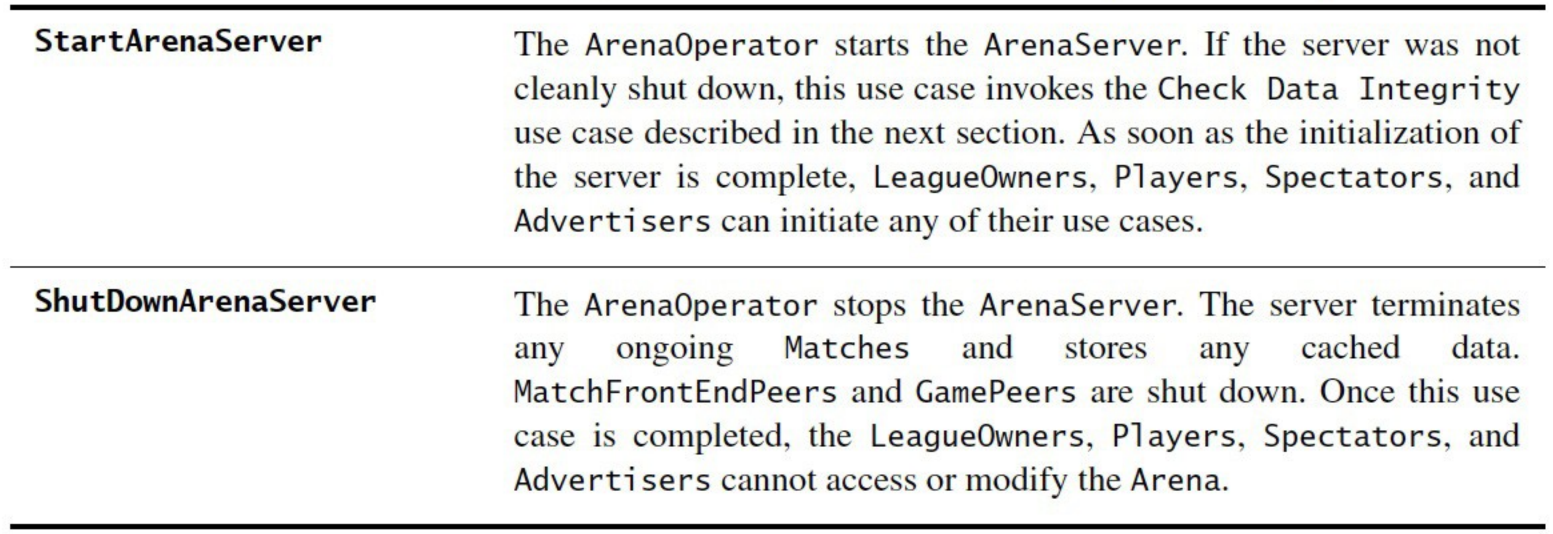
Identifying Boundary Conditions (cont.)

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.

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.

Identifying Boundary Conditions (cont.)

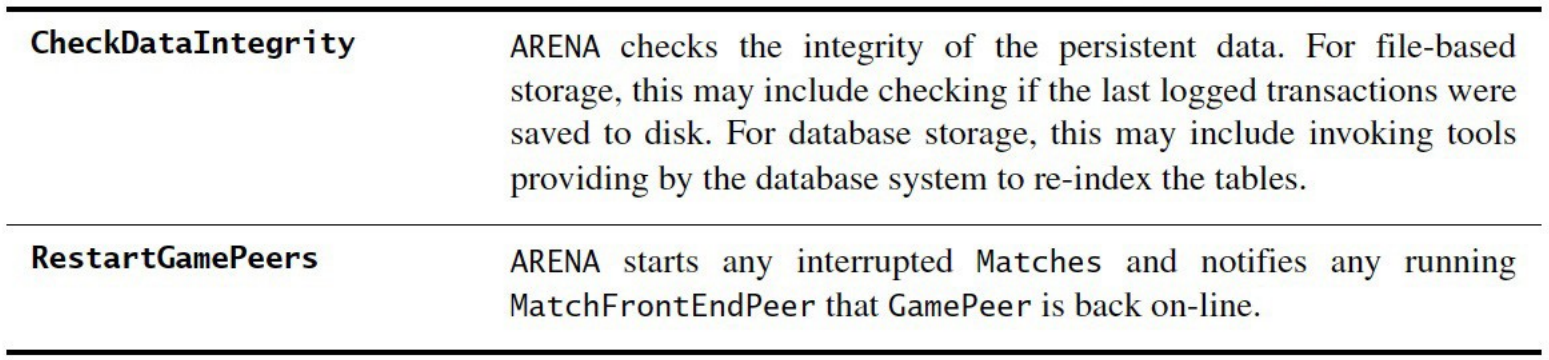
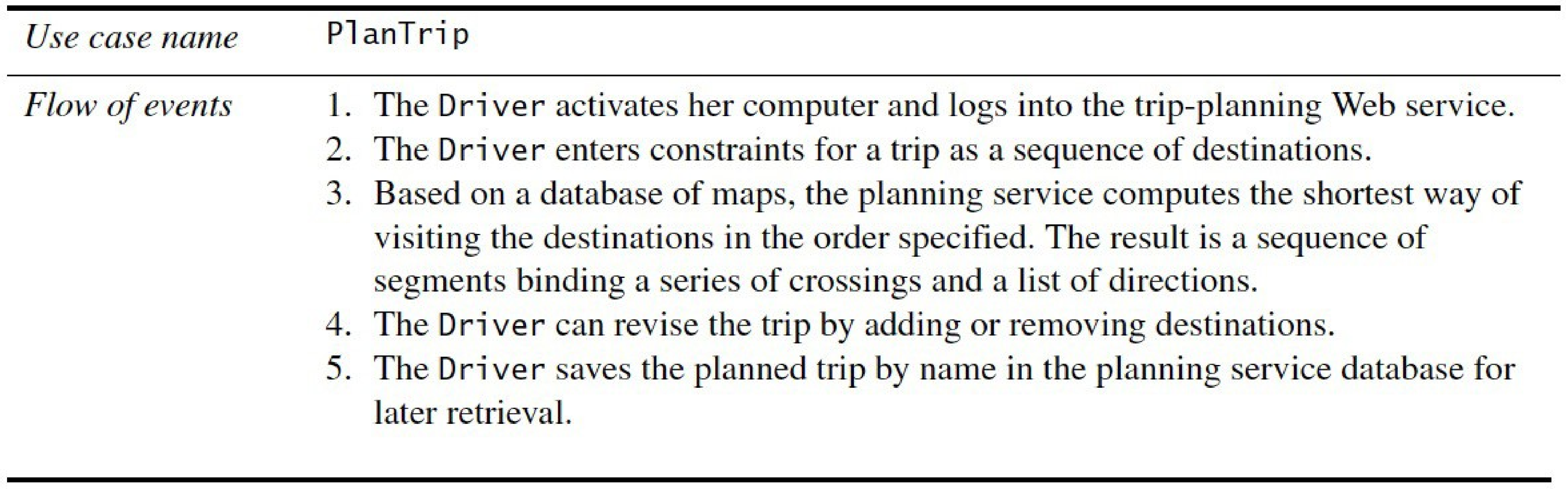


Table 7-7 Additional ARENA boundary use cases identified when reviewing persistent objects.

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.

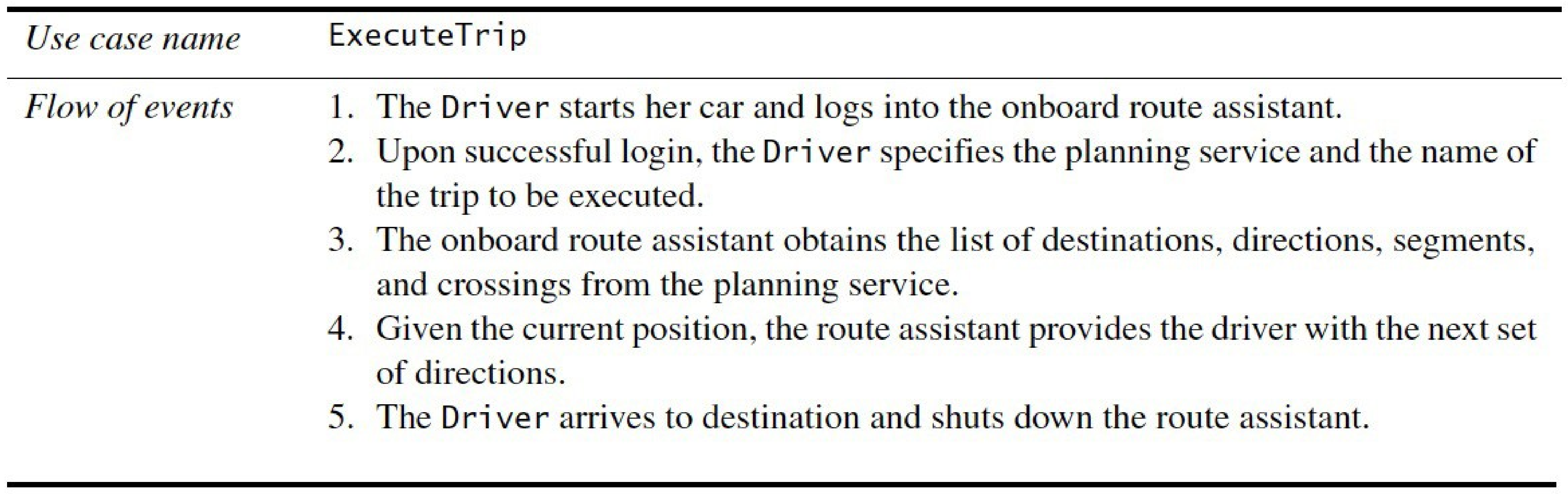
3.4.4 MyTrip Example



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.

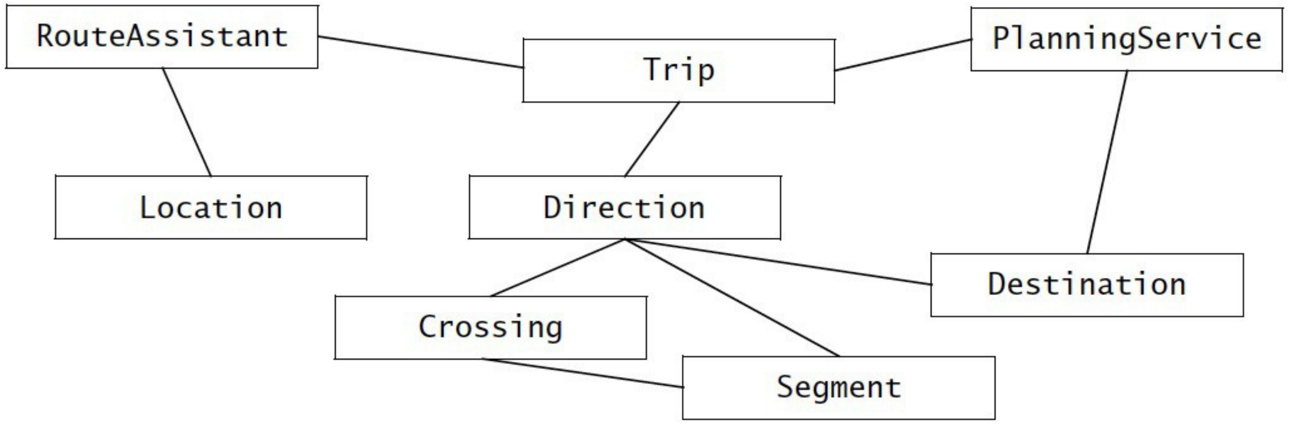
Figure 6-26 `PlanTrip` use case of the `MyTrip` system.

MyTrip Example (cont.)



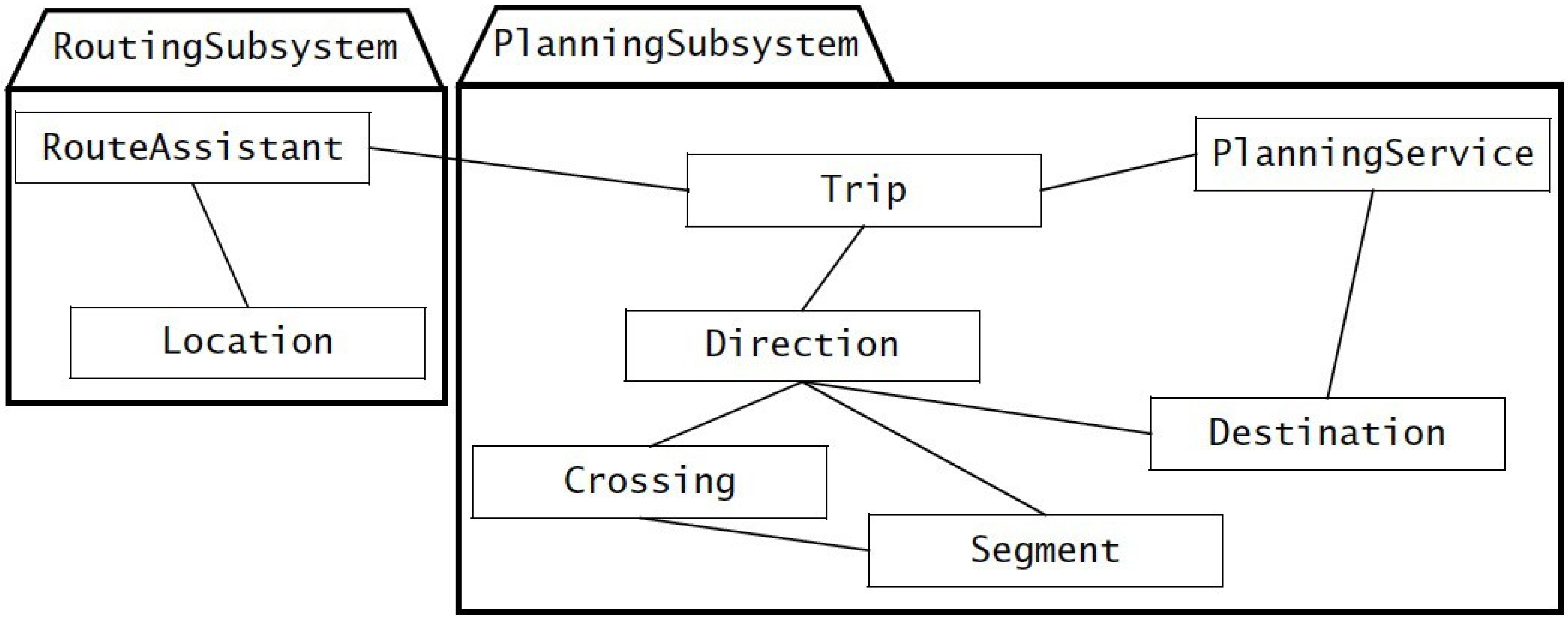
Use case name	`ExecuteTrip`
Flow of events	1. The `Driver` starts her car and logs into the onboard route assistant. 2. Upon successful login, the `Driver` specifies the planning service and the name of the trip to be executed. 3. The onboard route assistant obtains the list of destinations, directions, 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).

MyTrip Example (cont.)

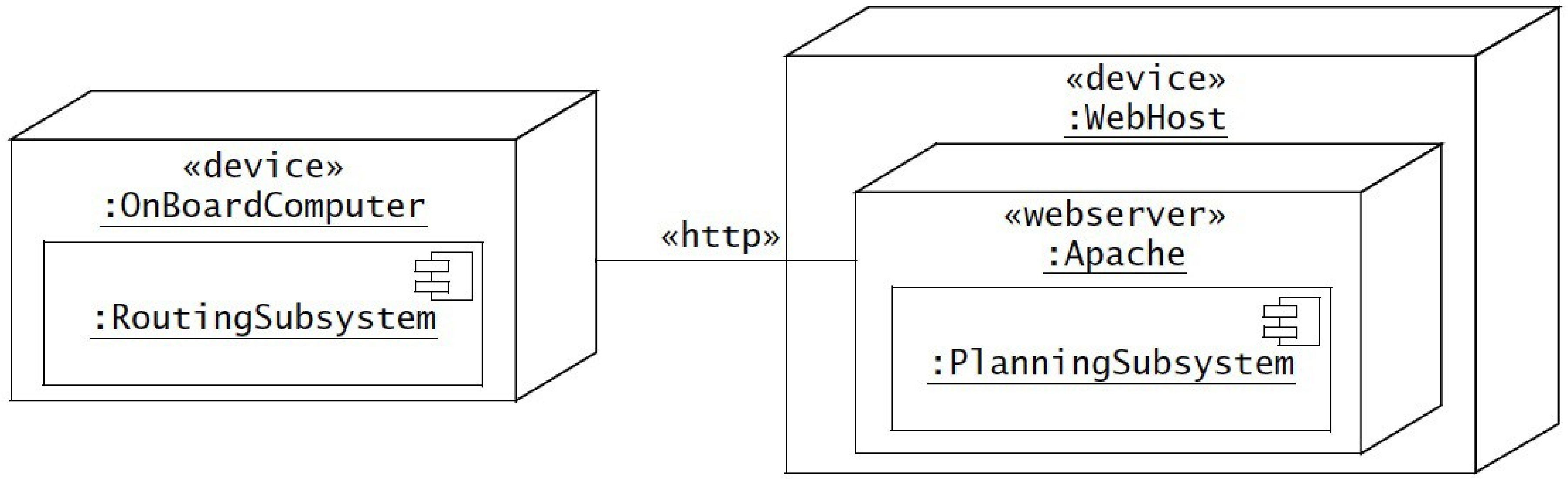
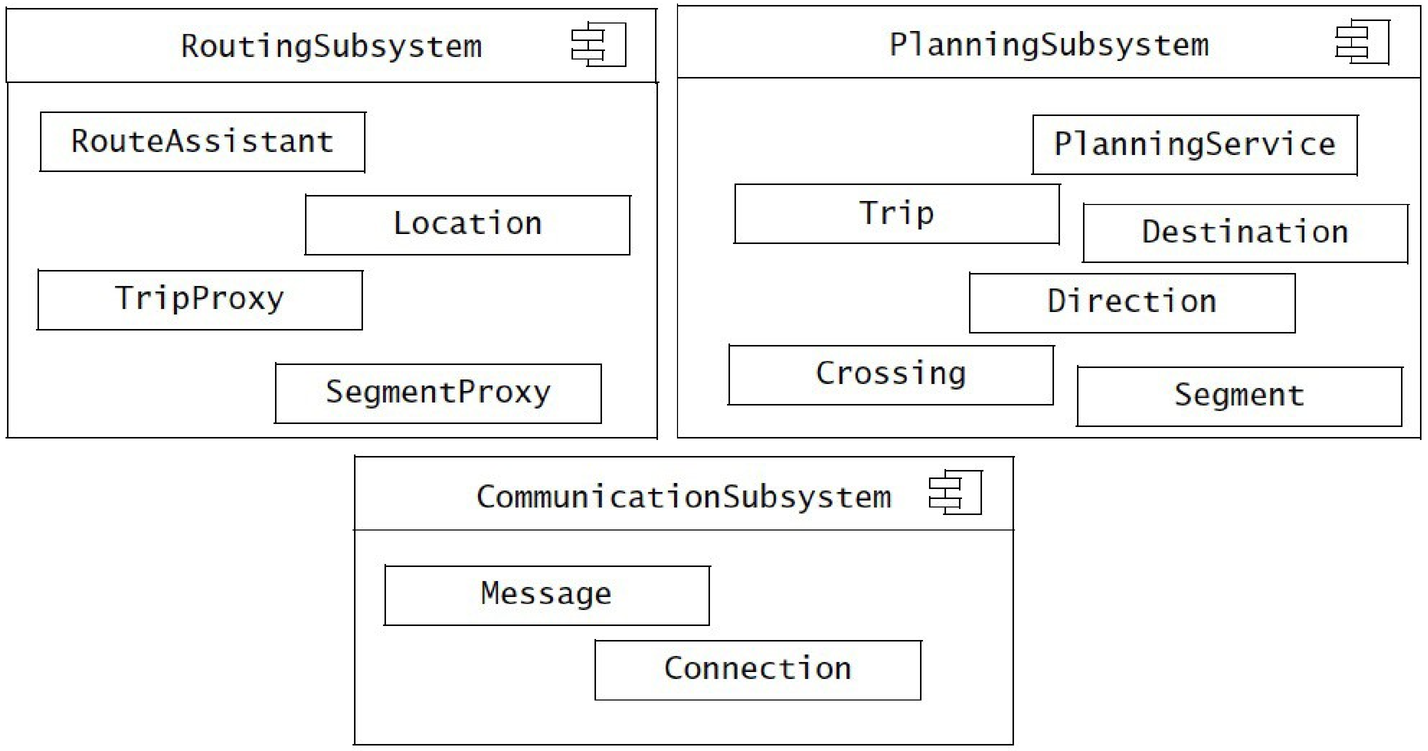
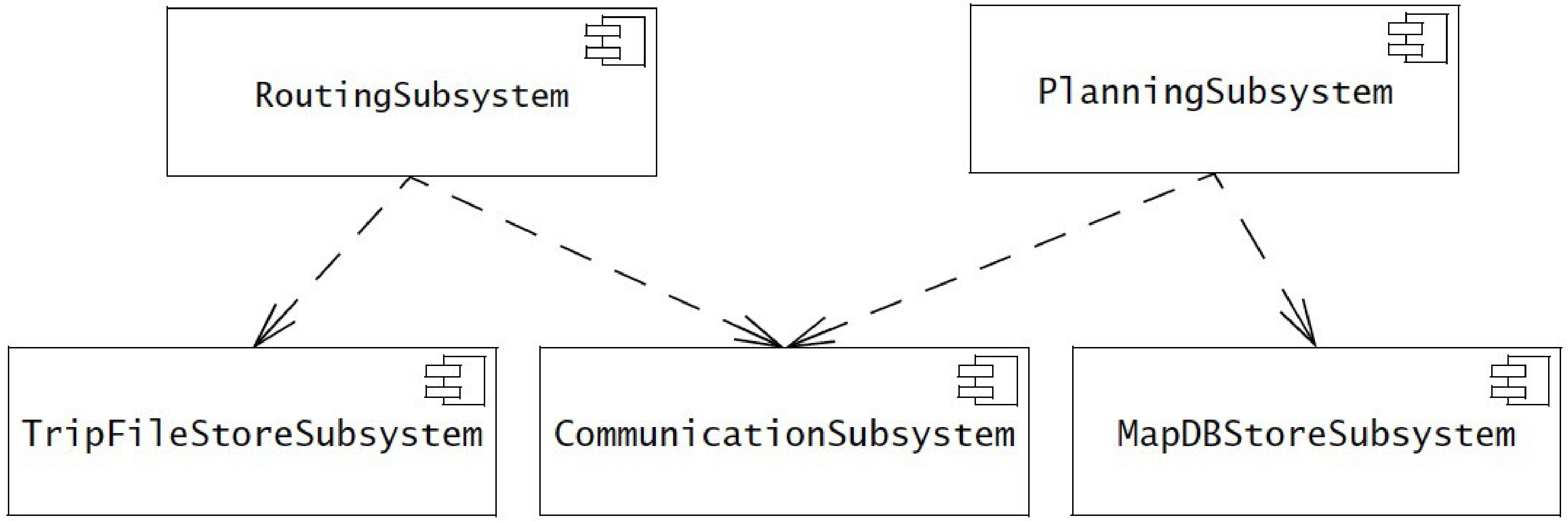


Figure 7-4 Allocation of MyTrip subsystems to devices and execution environments (UML deployment diagram). RoutingSubsystem runs on the OnBoardComputer; PlanningSubsystem runs on an Apache server.



CommunicationSubsystem	The CommunicationSubsystem is responsible for transporting objects from the PlanningSubsystem to the RoutingSubsystem.
Connection	A Connection represents an active link between the Planning-Subsystem and the RoutingSubsystem. A Connection object handles exceptional cases associated with loss of network services.
Message	A Message represents a Trip and its related Destinations, Segments, Crossings, and Directions, encoded for transport.

Figure 7-5 Revised design model for MyTrip (UML component diagram).



TripFileStoreSubsystem	The TripFileStoreSubsystem is responsible for storing trips in files on the onboard computer. Because this functionality is only used for storing trips when the car shuts down, this subsystem only supports the fast storage and loading of whole trips.
MapDBStoreSubsystem	The MapDBStoreSubsystem is responsible for storing maps and trips in a database for the PlanningSubsystem. This subsystem supports multiple concurrent Drivers and planning agents.

Figure 7-6 Subsystem decomposition of MyTrip after deciding on the issue of data stores (UML component diagram).

MyTrip Example (cont.)

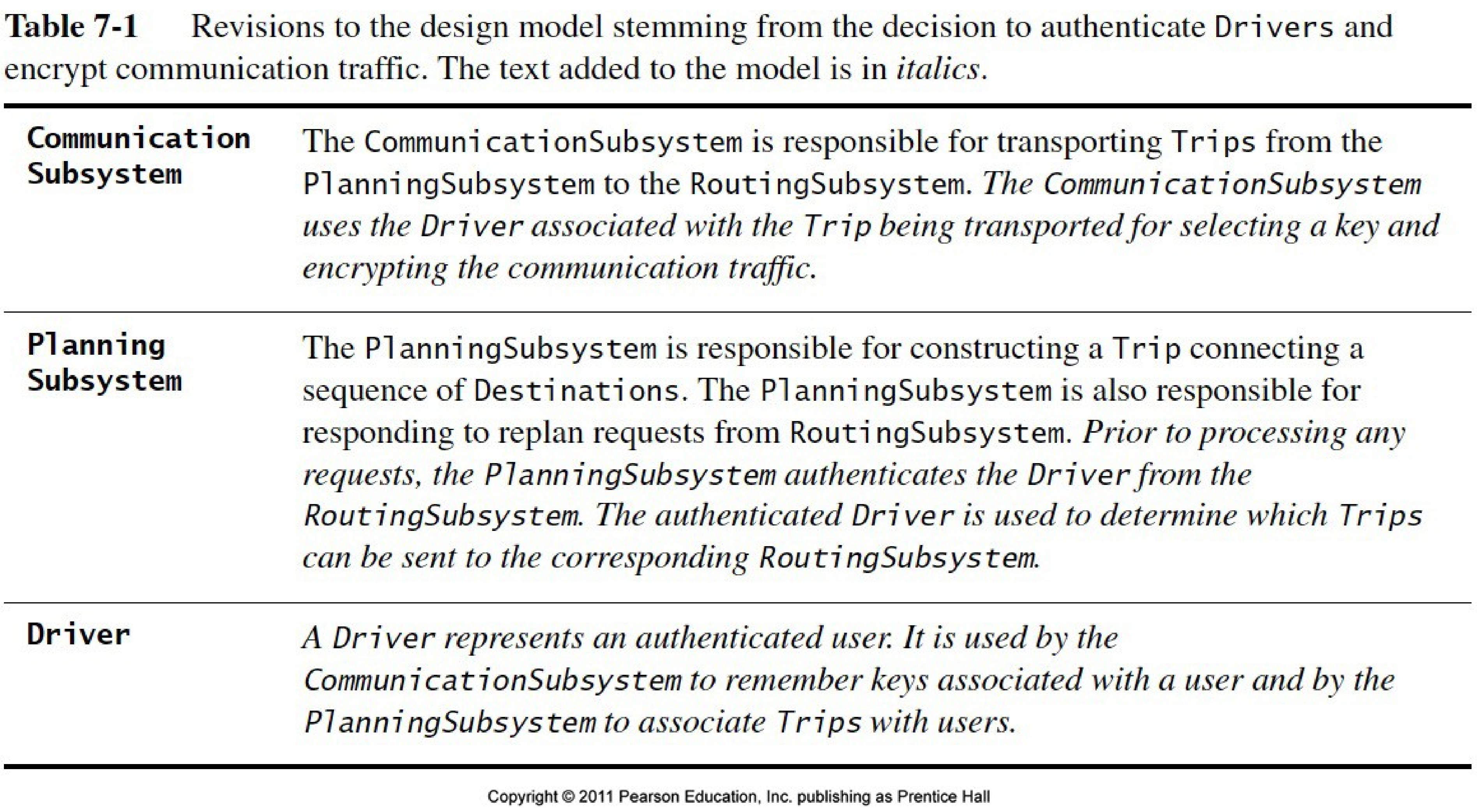


Table 7-1 Revisions to the design model stemming from the decision to authenticate Drivers and encrypt communication traffic. The text added to the model is in *italics*.

Communication Subsystem	The CommunicationSubsystem is responsible for transporting Trips from the PlanningSubsystem to the RoutingSubsystem. *The CommunicationSubsystem uses the Driver associated with the Trip being transported for selecting a key and encrypting the communication traffic.*
Planning Subsystem	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. *Prior to processing any requests, the PlanningSubsystem authenticates the Driver from the RoutingSubsystem. The authenticated Driver is used to determine which Trips can be sent to the corresponding RoutingSubsystem.*
Driver	*A Driver represents an authenticated user. It is used by the CommunicationSubsystem to remember keys associated with a user and by the PlanningSubsystem to associate Trips with users.*

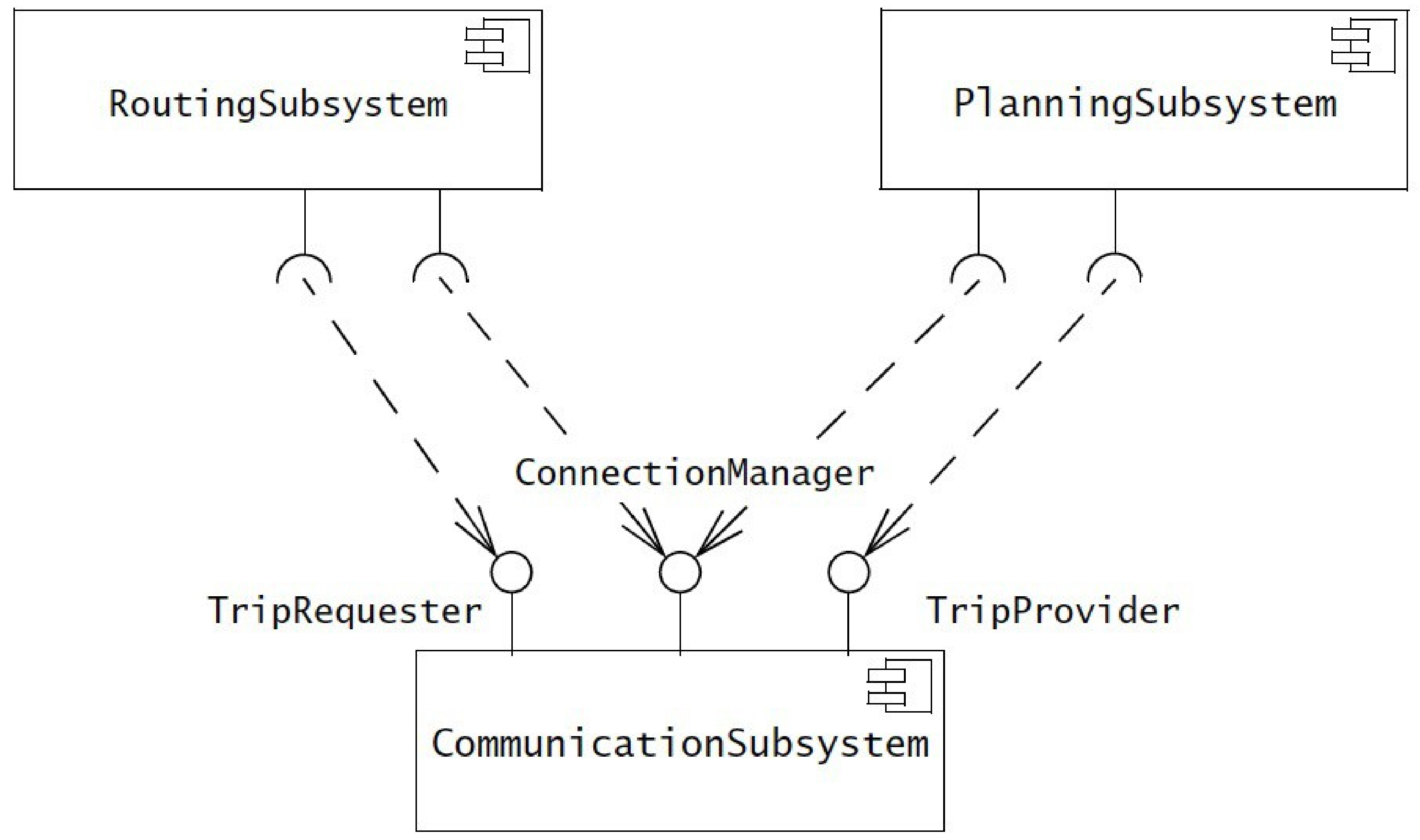


Figure 7-14 Refining the subsystem decompositions by identifying subsystem services (UML component diagram). The CommunicationSubsystem provides three services for managing connections, uploading trips, and downloading trips.

MyTrip Example (cont.)

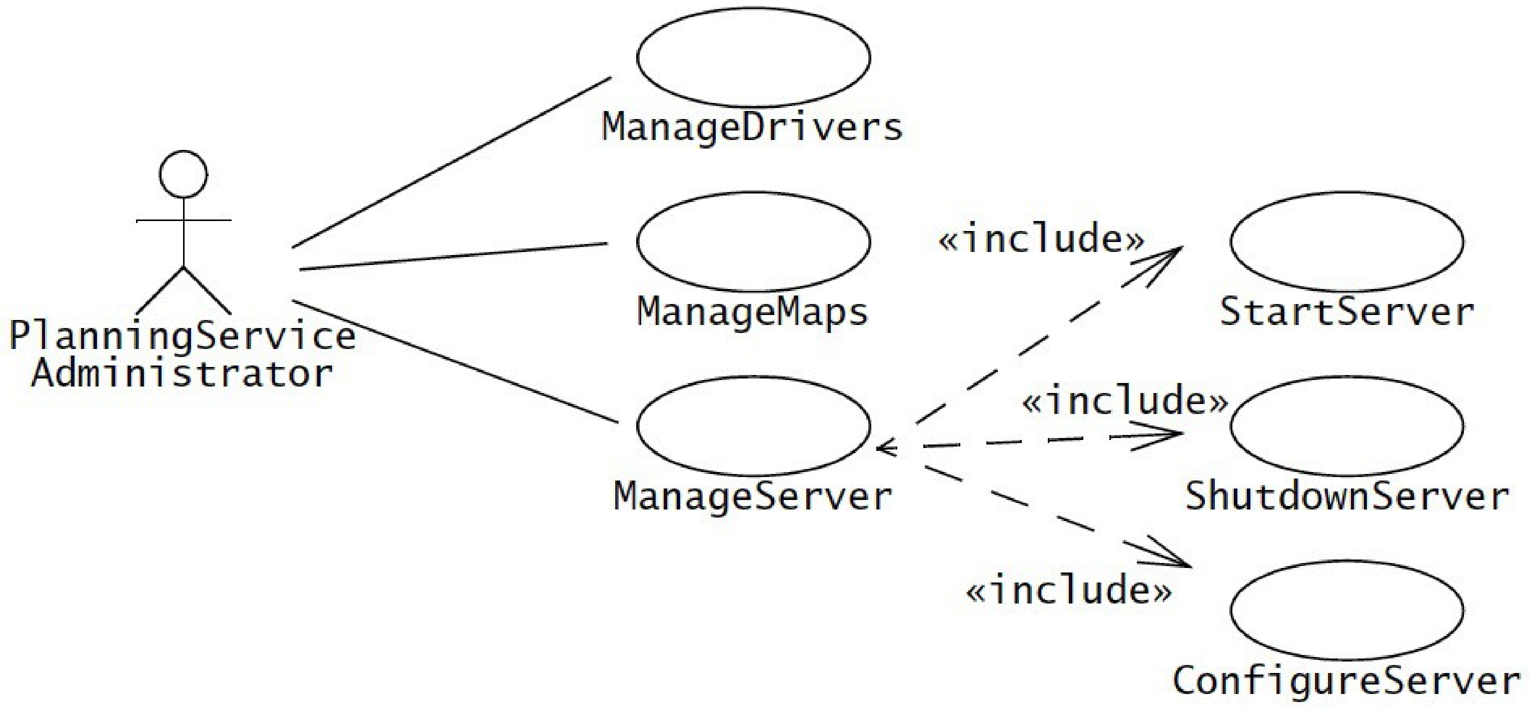
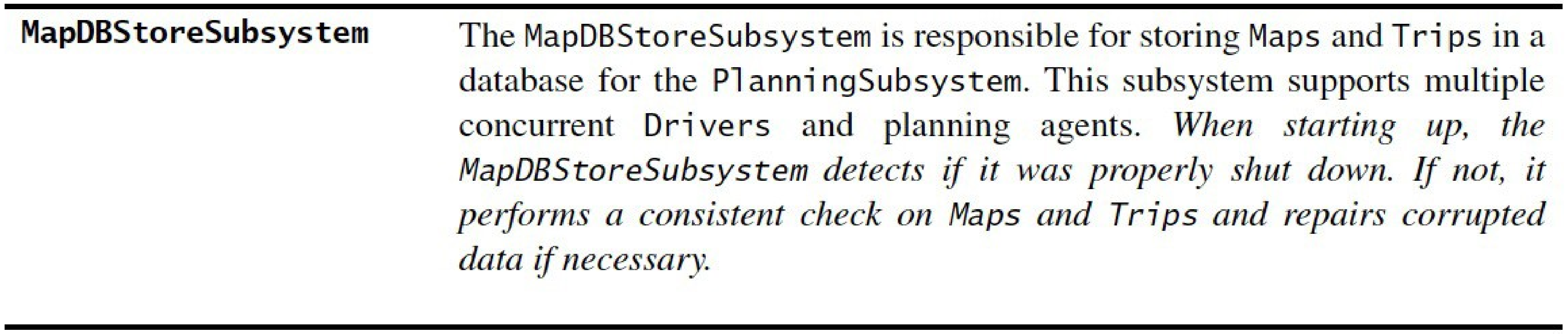


Figure 7-15 Administration use cases for MyTrip (UML use case diagram). ManageDrivers is invoked to add, remove, modify, or read data about drivers (e.g., user name and password, usage log, encryption key generation). ManageMaps is invoked to add, remove, or update maps that are used to generate trips. ManageServer includes all the functions necessary to start up and shutdown the server.

MyTrip Example (cont.)



MapDBStoreSubsystem	The MapDBStoreSubsystem is responsible for storing Maps and Trips in a database for the PlanningSubsystem. This subsystem supports multiple concurrent Drivers and planning agents. *When starting up, the MapDBStoreSubsystem detects if it was properly shut down. If not, it performs a consistent check on Maps and Trips and repairs corrupted data if necessary.*

Figure 7-17 Revised description for MapDBStoreSubsystem based on the additional StartServer use case of Figure 7-16. (Changes indicated in *italics*.)

Refined Decomposition Recap

- What we learned:
  - group subsystems into components and nodes
  - identify persistent data, access control and global control flow mechanisms, boundary conditions
  - identify subsystem services and interfaces