ICT - 4231
OBJECT ORIENTED SOFTWARE ENGINEERING

Lecture - 1

Chapter – 1 : Introduction to Software Engineering

CONTENTS

INTRODUCTION : SOFTWARE ENGINEERING FAILURES

INTRODUCTION

“The Amateur Software Engineer Is Always In Search Of Magic, Some Sensational Method Or Tool Whose Application Promises To Render Software Development Trivial. It Is The Mark Of The Professional Software Engineer To Know That No Such Panacea Exists.”

-----Grady Booch, in Object-Oriented Analysis and Design

INTRODUCTION: SOFTWARE ENGINEERING FAILURES

- Software systems are complex creations. They perform many functions; they are built to achieve many different, and often conflicting, objectives. They comprise many components; many of their components are custom-made and complex themselves. Many participants from different disciplines take part in the development of these components. The development process and the software life cycle often span many years. Finally, complex systems are difficult for any single person to understand completely.
- Many systems are so hard to understand, even during their development phase, that they are never finished: these are called **vaporware**.

INTRODUCTION: SOFTWARE ENGINEERING FAILURES (CONT.)

- Software development projects are subject to constant **change**. Because requirements are complex, they need to be updated when errors are discovered and when the developers have a better understanding of the application. If the project lasts many years, the staff turn-around is high, requiring constant training. The time between technological changes is often shorter than the duration of the project. The widespread assumptions of a software project manager that all changes have been dealt with and that the requirements can be frozen will lead to the deployment of an irrelevant system.

WHAT IS SOFTWARE ENGINEERING?

MODELING ACTIVITY

- Software engineers deal with complexity through modeling, by focusing at any one time on only the relevant details and ignoring everything else.
- In the course of development, software engineers build many different models of the system and of the application domain.

PROBLEM-SOLVING ACTIVITY

- Models are used to search for an acceptable solution.
- This search is driven by experimentation.
- Software engineers do not have infinite resources and are constrained by budget and deadlines.

KNOWLEDGE ACQUISITION ACTIVITY

- In modeling the application and solution domain, software engineers collect data, organize it into information, and formalize it into knowledge.
- Knowledge acquisition is not sequential, as a single piece of additional data can invalidate complete models.

RATIONALE-DRIVEN ACTIVITY

- Rationale information, represented as a set of issue models, enables software engineers to understand the implication of a proposed change when revisiting a decision.

MODELING

- The purpose of science is to describe and understand complex systems, such as a system of atoms, a society of human beings, or a solar system.
- One of the basic methods of science is modeling.
- A model is an abstract representation of a system that enables us to answer questions about the system.
- Models are useful when dealing with systems that are too large, too small, too complicated, or too expensive to experience firsthand.
- Models also allow us to visualize and understand systems that either no longer exist or that are only claimed to exist.

MODELING (CONT.)

- **First, software engineers need to understand the environment in which the system has to operate.** For a train traffic control system, software engineers need to know train signaling procedures. For a stock trading system, software engineers need to know trading rules. The software engineer does not need to become a fully certified train dispatcher or a stock broker; they only need to learn the application domain concepts that are relevant to the system. In other terms, they need to build a model of the application domain.
- **Second, software engineers need to understand the systems they could build, to evaluate different solutions and trade-offs.** Most systems are too complex to be understood by any one person, and most systems are expensive to build. To address these challenges, software engineers describe important aspects of the alternative systems they investigate. In other terms, they need to build a model of the solution domain.

MODELING (CONT.)

- Object-oriented methods combine the application domain and solution domain modeling activities into one.
- The application domain is first modeled as a set of objects and relationships.
- This model is then used by the system to represent the real-world concepts it manipulates.
- The idea of object-oriented methods is that the solution domain model is a transformation of the application domain model.
- Developing software translates into the activities necessary to identify and describe a system as a set of models that addresses the end user's problem.
- Detail modeling and the concepts of objects are described in Chapter 2, Modeling with UML.

PROBLEM SOLVING

- Engineering is a problem-solving activity. Engineers search for an appropriate solution, often by trial and error, evaluating alternatives empirically, with limited resources and incomplete knowledge.
- In its simplest form, the engineering method includes five steps:
    1. Formulate the problem.
    2. Analyze the problem.
    3. Search for solutions.
    4. Decide on the appropriate solution.
    5. Specify the solution.

PROBLEM SOLVING (CONT.)

- Object-oriented software development typically includes six development activities: requirements elicitation, analysis, system design, object design, implementation, and testing.
- During requirements elicitation and analysis, software engineers formulate the problem with the client and build the application domain model.

PROBLEM SOLVING (CONT.)

- Requirements elicitation and analysis correspond to **steps 1 and 2** of the engineering method. During system design, software engineers analyze the problem, break it down into smaller pieces, and select general strategies for designing the system. During object design, they select detail solutions for each piece and decide on the most appropriate solution. System design and object design result in the solution domain model.
- System and object design correspond to **steps 3 and 4** of the engineering method. During implementation, software engineers realize the system by translating the solution domain model into an executable representation.
- Implementation corresponds to **step 5** of the engineering method. What makes software engineering different from problem solving in other sciences is that change occurs in the application and the solution domain while the problem is being solved.

KNOWLEDGE ACQUISITION

- A common mistake that software engineers and managers make is to assume that the acquisition of knowledge needed to develop a system is linear.
- Rather, it is a nonlinear process.
- The addition of a new piece of information may invalidate all the knowledge we have acquired for the understanding of a system.
- Even if we had already documented this understanding in documents and code (“The system is 90% coded, we will be done next week”), we must be mentally prepared to start from scratch. This has important implications on the set of activities and their interactions we define to develop the software system.

KNOWLEDGE ACQUISITION (CONT.)

- There are several software processes that deal with this problem by avoiding the sequential dependencies inherent in the waterfall model.
- Risk-based development attempts to anticipate surprises late in a project by identifying the high-risk components. Issue-based development attempts to remove the linearity altogether.
- Any development activity—analysis, system design, object design, implementation, testing, or delivery—can influence any other activity.
- In issue-based development, all these activities are executed in parallel. The difficulty with nonsequential development models, however, is that they are difficult to manage.

RATIONALE

- Assumptions that developers make about a system change constantly. Even though the application domain models eventually stabilize once developers acquire an adequate understanding of the problem, the solution domain models are in constant flux.
- Design and implementation faults discovered during testing and usability problems discovered during user evaluation trigger changes to the solution models.
- Changes can also be caused by new technology.
- Change introduced by new technology often allows the formulation of new functional or nonfunctional requirements.
- To change the system, it is not enough to understand its current components and behavior. It is also necessary to capture and understand the context in which each design decision was made. This additional knowledge is called the rationale of the system.
- In order to deal with changing systems, however, software engineers must address the challenges of capturing and accessing rationale.

SOFTWARE ENGINEERING CONCEPTS

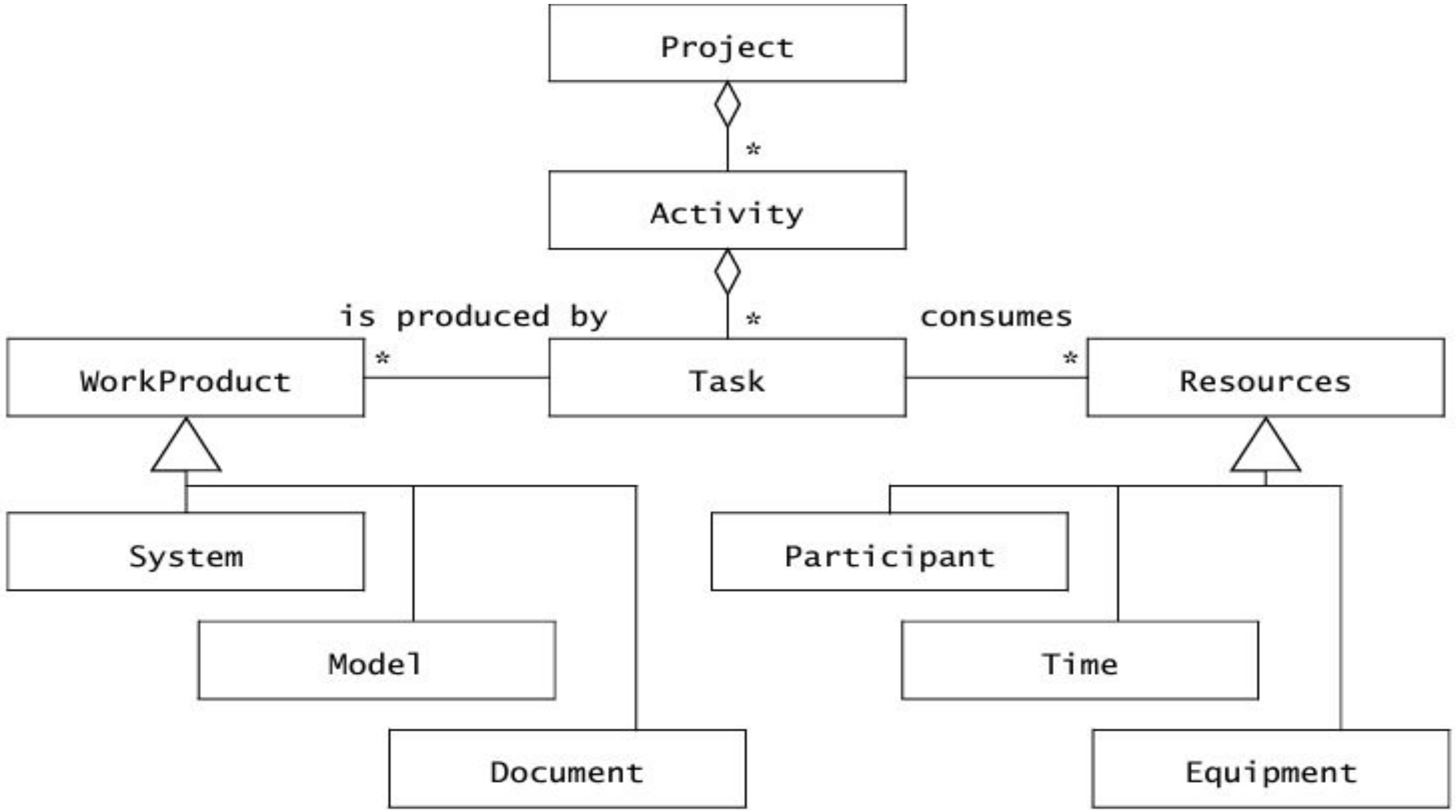


Figure 1-1 Software engineering concepts depicted as a UML class diagram [OMG, 2009].

PARTICIPANTS AND ROLES

- **Developing a software system requires the collaboration of many people with different backgrounds and interests.**
- **The client orders and pays for the system.**
- **The developers construct the system.**
- **The project manager plans and budgets the project and coordinates the developers and the client. The end users are supported by the system.**
- **All the persons involved in the project are referred as participants.**
- **A set of responsibilities in the project or the system are referred as a role.**
- **A role is associated with a set of tasks and is assigned to a participant. The same participant can fill multiple roles.**

PARTICIPANTS AND ROLES (CONT.)

Consider a TicketDistributor system:

TicketDistributor is a machine that distributes tickets for trains. Travelers have the option of selecting a ticket for a single trip or for multiple trips, or selecting a time card for a day or a week. The TicketDistributor computes the price of the requested ticket based on the area in which the trip will take place and whether the traveler is a child or an adult. The TicketDistributor must be able to handle several exceptions, such as travelers who do not complete the transaction, travelers who attempt to pay with large bills, and resource outages, such as running out of tickets, change, or power.

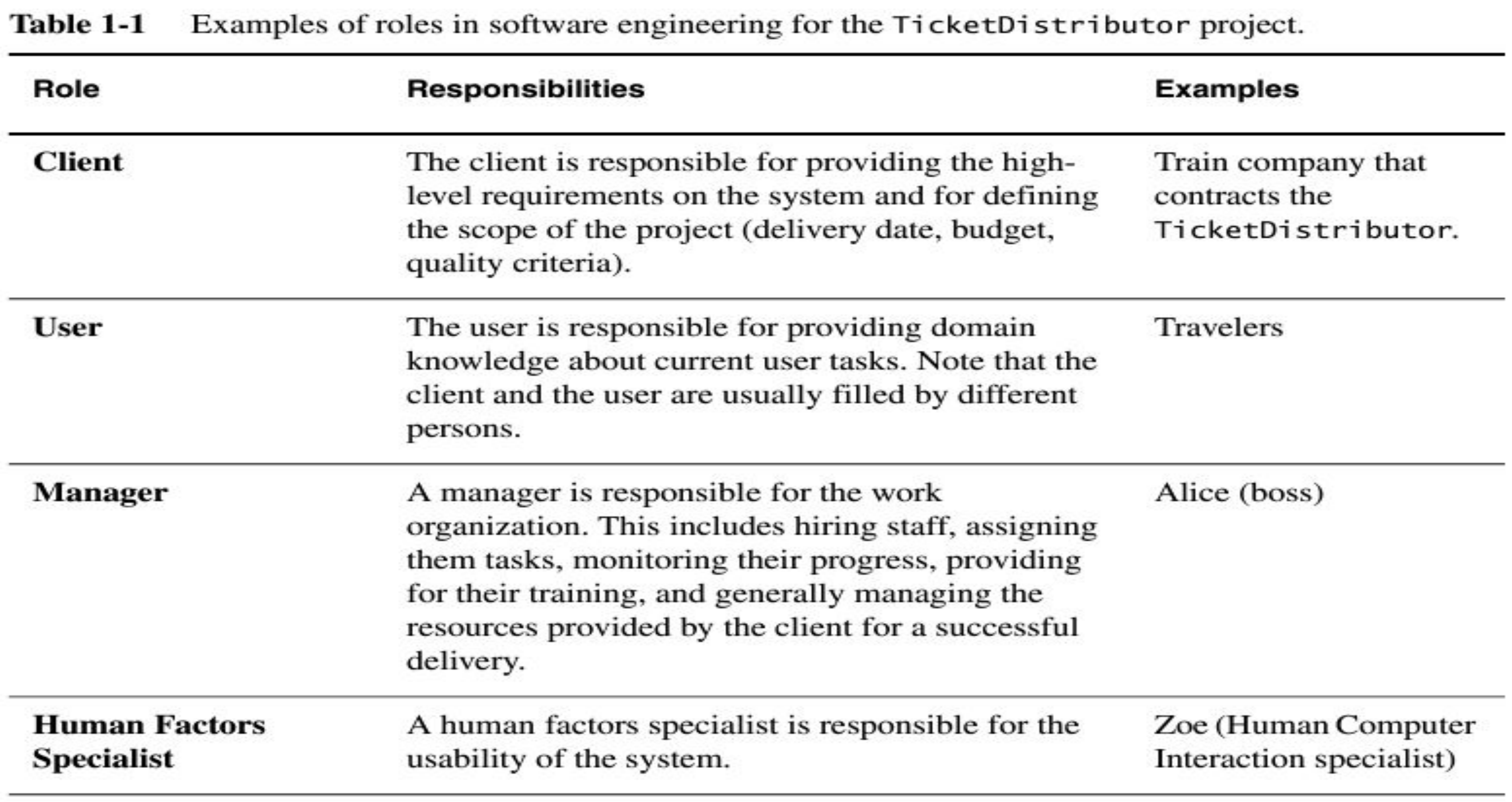
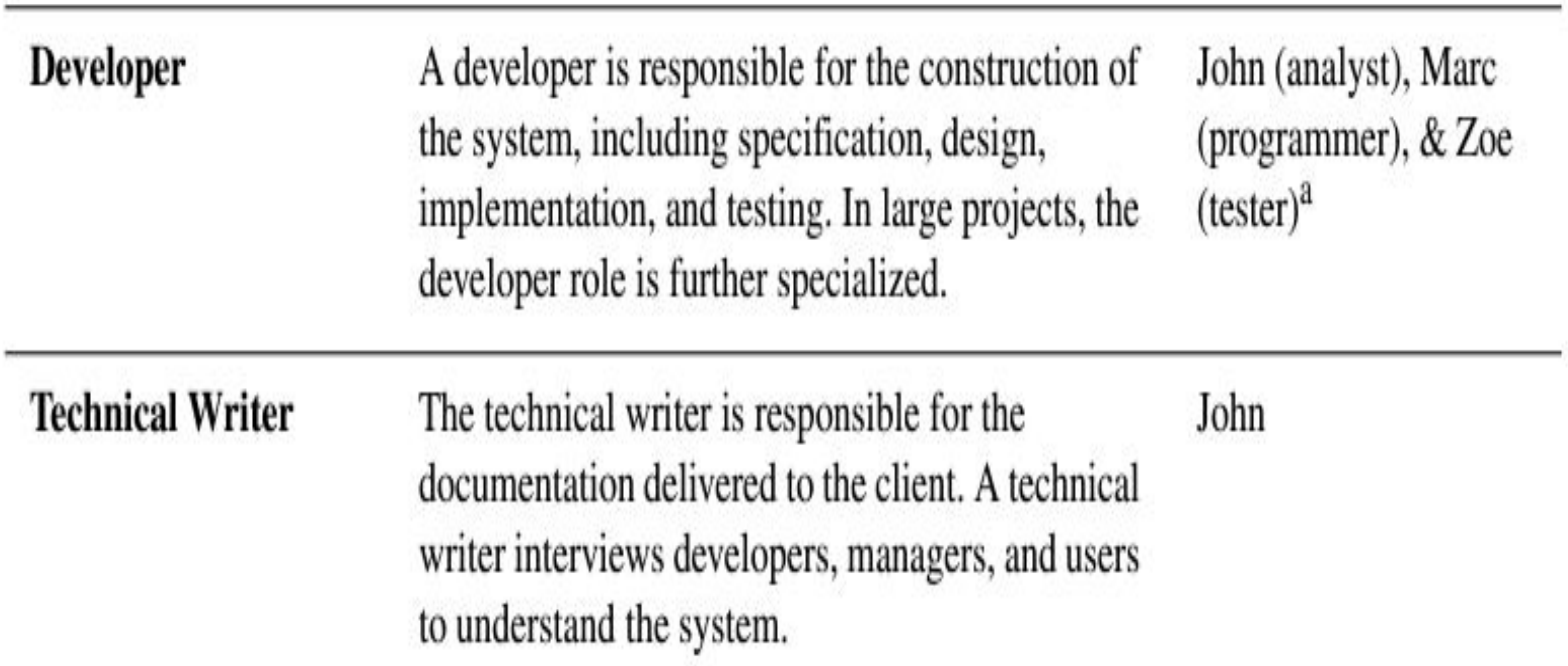


Table 1-1 Examples of roles in software engineering for the `TicketDistributor` project.

Role	Responsibilities	Examples
Client	The client is responsible for providing the high-level requirements on the system and for defining the scope of the project (delivery date, budget, quality criteria).	Train company that contracts the `TicketDistributor`.
User	The user is responsible for providing domain knowledge about current user tasks. Note that the client and the user are usually filled by different persons.	Travelers
Manager	A manager is responsible for the work organization. This includes hiring staff, assigning them tasks, monitoring their progress, providing for their training, and generally managing the resources provided by the client for a successful delivery.	Alice (boss)
Human Factors Specialist	A human factors specialist is responsible for the usability of the system.	Zoe (Human Computer Interaction specialist)

Cont.



Developer	A developer is responsible for the construction of the system, including specification, design, implementation, and testing. In large projects, the developer role is further specialized.	John (analyst), Marc (programmer), & Zoe (tester)[a]
Technical Writer	The technical writer is responsible for the documentation delivered to the client. A technical writer interviews developers, managers, and users to understand the system.	John

SYSTEMS AND MODELS

- The term system is used as a collection of interconnected parts.
- Modeling is a way to deal with complexity by ignoring irrelevant details.
- The term model is used to refer to any abstraction of the system.
- A TicketDistributor for an underground train is a system.
- Blueprints for the TicketDistributor, schematics of its electrical wiring, and object models of its software are models of the TicketDistributor.
- Note that a development project is itself a system that can be modeled.
- The project schedule, its budget, and its planned deadlines are models of the development project.

WORK PRODUCTS

- A work product is an artifact that is produced during the development, such as a document or a piece of software for other developers or for the client.
- Internally-focused artifacts are termed "internal work products," catering to the project team's needs.
- Work products destined for the client are referred to as "deliverables."
- Deliverables are predetermined before the project begins, specified in a contract binding developers to client expectations.
- The distinction between internal work products and deliverables ensures clarity in project development and client satisfaction.

Table 1-2 Examples of work products for the `TicketDistributor` project.

Work product	Type	Description
Specification	**Deliverable**	The specification describes the system from the user's point of view. It is used as a contractual document between the project and the client. The `TicketDistributor` specification describes in detail how the system should appear to the traveler.
Operation manual	**Deliverable**	The operation manual for the `TicketDistributor` is used by the staff of the train company responsible for installing and configuring the `TicketDistributor`. Such a manual describes, for example, how to change the price of tickets and the structure of the network into zones.
Status report	**Internal work product**	A status report describes at a given time the tasks that have been completed and the tasks that are still in progress. The status report is produced for the manager, Alice, and is usually not seen by the train company.
Test manual	**Internal work product**	The test plans and results are produced by the tester, Zoe. These documents track the known defects in the prototype `TicketDistributor` and their state of repair. These documents are usually not shared with the client.

ACTIVITIES, TASKS, AND RESOURCES

- **Activity:** An activity is a set of tasks that is performed toward a specific purpose. For example,
  - requirements elicitation is an activity whose purpose is to define with the client what the system will do.
  - Delivery is an activity whose purpose is to install the system at an operational location.
  - Management is an activity whose purpose is to monitor and control the project such that it meets its goals (e.g., deadline, quality, budget).

  Activities, also known as phases, can be constructed from other activities; for instance, the delivery activity encompasses tasks like software installation and operator training.
- **Tasks:** A task is a atomic unit of work management: Managers assign tasks to developers, who execute them, while managers oversee progress and completion. Tasks utilize resources, yield work products, and rely on work products generated by other tasks.
- **Resources:** Resources are assets that are used to accomplish work. Resources include time, equipment, and labor. When planning a project, a manager breaks down the work into tasks and assigns them to resources.

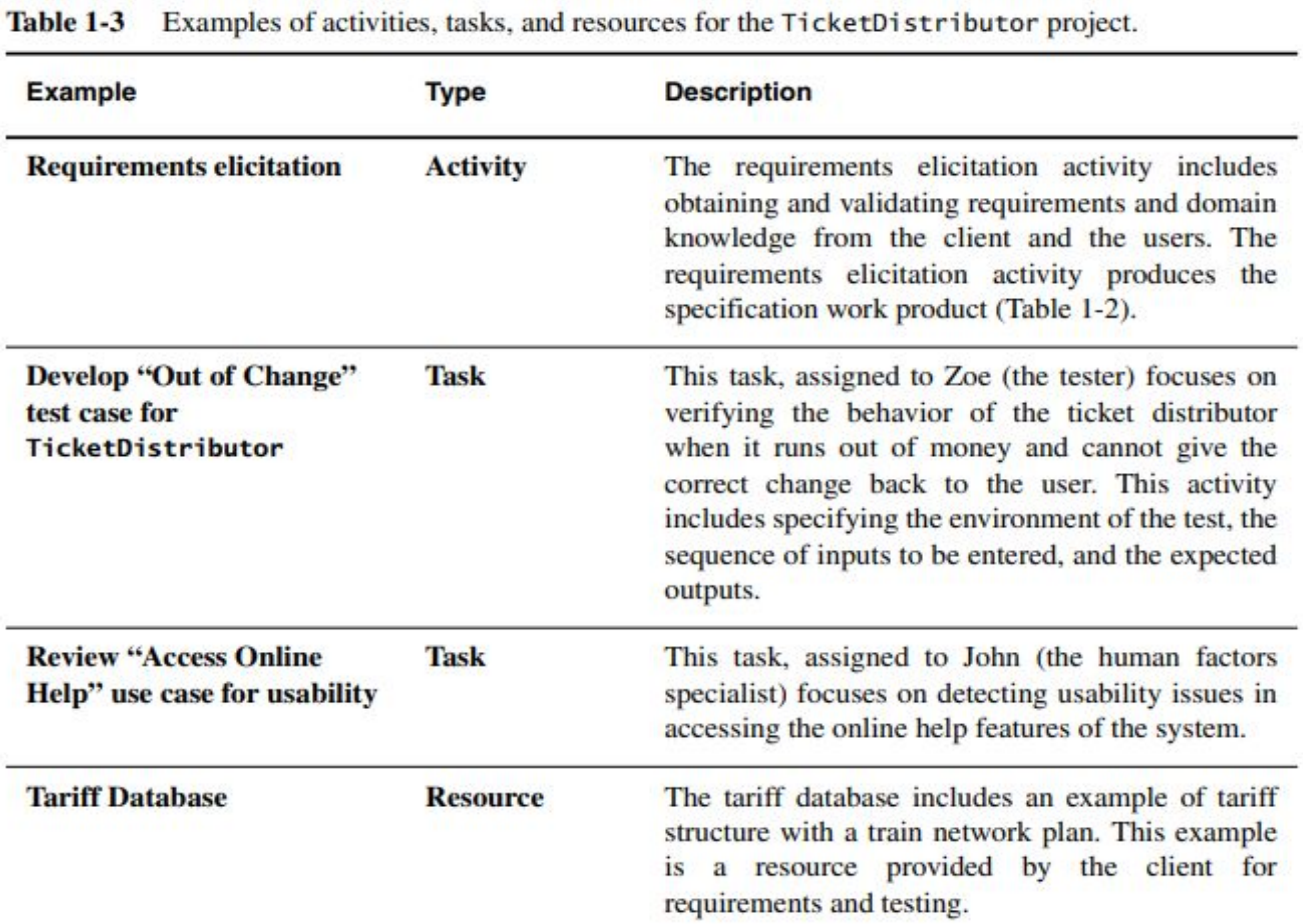


Table 1-3 Examples of activities, tasks, and resources for the `TicketDistributor` project.

Example	Type	Description
Requirements elicitation	**Activity**	The requirements elicitation activity includes obtaining and validating requirements and domain knowledge from the client and the users. The requirements elicitation activity produces the specification work product (Table 1-2).
Develop "Out of Change" test case for `TicketDistributor`	**Task**	This task, assigned to Zoe (the tester) focuses on verifying the behavior of the ticket distributor when it runs out of money and cannot give the correct change back to the user. This activity includes specifying the environment of the test, the sequence of inputs to be entered, and the expected outputs.
Review "Access Online Help" use case for usability	**Task**	This task, assigned to John (the human factors specialist) focuses on detecting usability issues in accessing the online help features of the system.
Tariff Database	**Resource**	The tariff database includes an example of tariff structure with a train network plan. This example is a resource provided by the client for requirements and testing.

FUNCTIONAL AND NONFUNCTIONAL REQUIREMENTS

➔ Requirements specify a set of features that the system must have.

❖ **Functional Requirements**

- A functional requirement is a specification of a function that the system must support.
- For example, The user must be able to purchase tickets and The user must be able to access tariff information are functional requirements.

❖ **Nonfunctional Requirements**

- A nonfunctional requirement is a constraint on the operation of the system that is not related directly to a function of the system.
- The user must be provided feedback in less than one second and The colors used in the interface should be consistent with the company colors are nonfunctional requirements.

NOTATIONS, METHODS, AND METHODOLOGIES

- **Notation:** A notation is a graphical or textual set of rules for representing a model. The use of notations in software engineering is common and predates object-oriented concepts. For example,
  - UML, the notation we use throughout this book, is a notation for representing object-oriented models.
  - Data flow diagrams is a notation for representing systems in terms of data sources, data sinks, and data transformations.

- **Method:** A method is a repeatable technique that specifies the steps involved in solving a specific problem.
  - A sorting algorithm is a method for ordering elements of a list.
  - Rationale management is a method for justifying change.
  - Configuration management is a method for tracking change.

NOTATIONS, METHODS, AND METHODOLOGIES

- **Methodology:** A methodology is a collection of methods for solving a class of problems and specifies how and when each method should be used. Software development methodologies decompose the process into activities. For Example,
  - A seafood cookbook with a collection of recipes is a methodology for preparing seafood if it also contains advice on how ingredients should be used and what to do if not all ingredients are available.
  - Royce's methodology, the Object Modeling Technique (OMT), the Booch methodology, and Catalysis are object-oriented methodologies for developing software.

- The OMT methodology assumes that requirements have already been defined and does not provide methods for eliciting requirements. OMT provides methods for three activities:
  - Analysis, which focuses on formalizing the system requirements into an object model,
  - System Design, which focuses on strategic decisions, and
  - Object Design, which transforms the analysis model into an object model that can be implemented.

SOFTWARE ENGINEERING DEVELOPMENT ACTIVITIES

➔ Development activities deal with the complexity by constructing and validating models of the application domain or the system. Development activities include:

- Requirements Elicitation
- Analysis
- System Design
- Object Design
- Implementation
- Testing

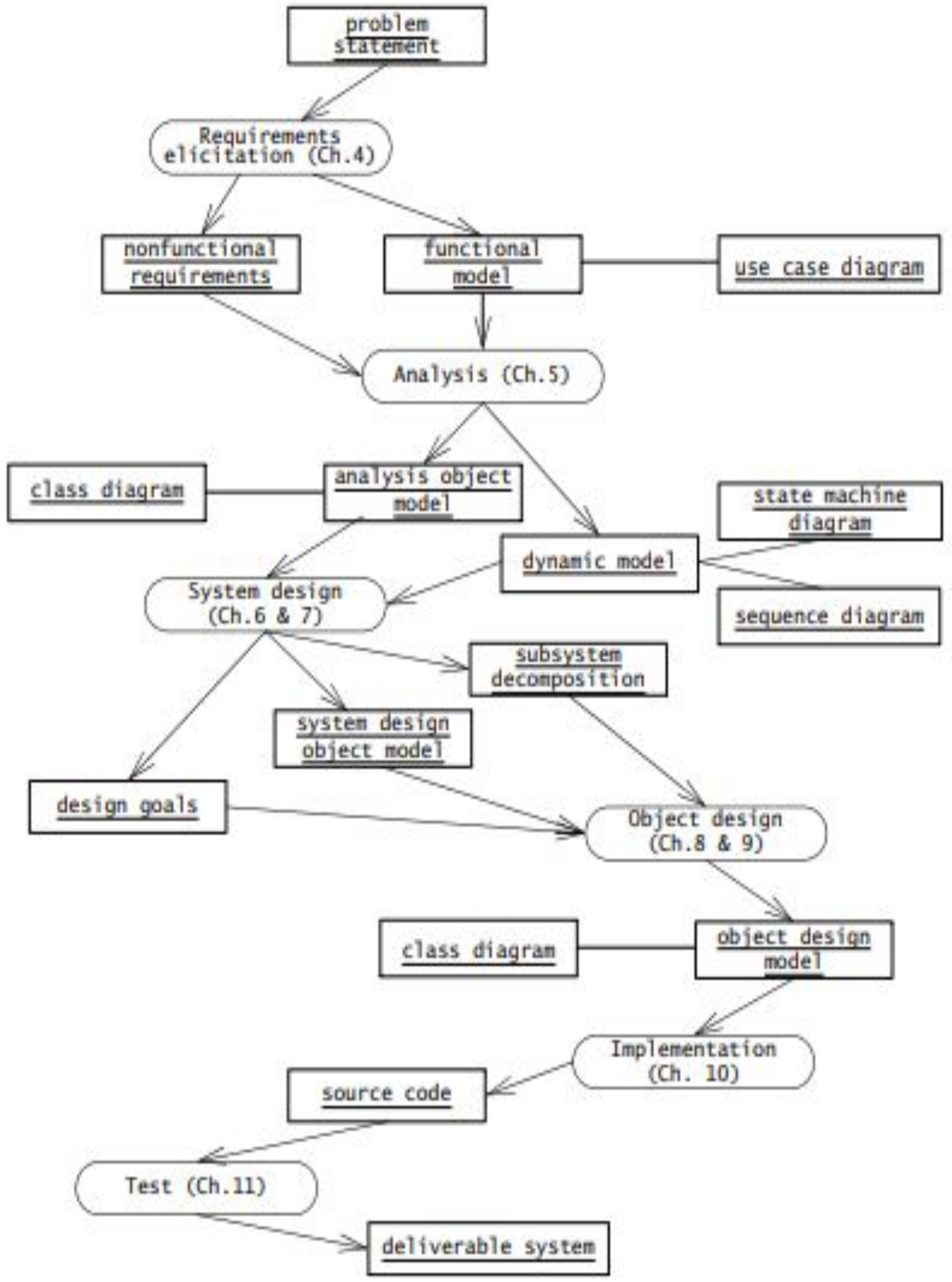
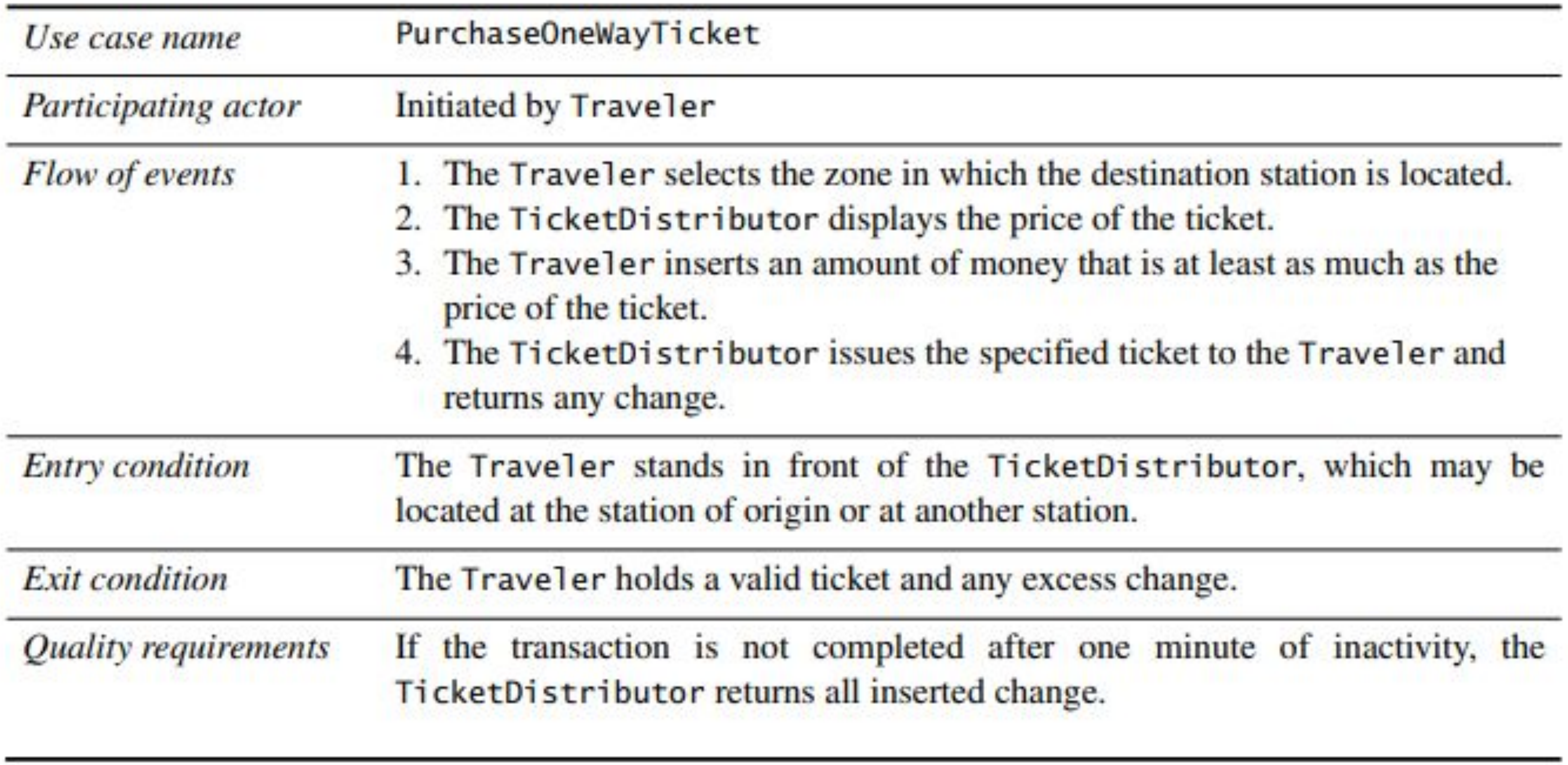


Figure 1-2 An overview of object-oriented software engineering development activities and their products. This diagram depicts only logical dependencies among work products. Object-oriented software engineering is iterative; that is, activities can occur in parallel and more than once.

REQUIREMENTS ELICITATION

- ❖ During **requirements elicitation**, the client and developers define the purpose of the system. The result of this activity is a description of the system in terms of **actors** and **use cases.**

  - ➢ **Actors** represent the external entities that interact with the system. Actors include roles such as end users, other computers the system needs to deal with (e.g., a central bank computer, a network), and the environment (e.g., a chemical process).
  - ➢ **Use cases** are general sequences of events that describe all the possible actions between an actor and the system for a given piece of functionality.



Use case name	PurchaseOneWayTicket
Participating actor	Initiated by Traveler
Flow of events	1. The Traveler selects the zone in which the destination station is located. 2. The TicketDistributor displays the price of the ticket. 3. The Traveler inserts an amount of money that is at least as much as the price of the ticket. 4. The TicketDistributor issues the specified ticket to the Traveler and returns any change.
Entry condition	The Traveler stands in front of the TicketDistributor, which may be located at the station of origin or at another station.
Exit condition	The Traveler holds a valid ticket and any excess change.
Quality requirements	If the transaction is not completed after one minute of inactivity, the TicketDistributor returns all inserted change.

Figure 1-3 An example of use case, PurchaseOneWayTicket.

ANALYSIS

- During **analysis**, developers aim to produce a model of the system that is correct, complete, consistent, and unambiguous.

  - Developers transform the use cases produced during requirements elicitation into an object model that completely describes the system.
  - During this activity, developers discover ambiguities and inconsistencies in the use case model that they resolve with the client.
  - The result of analysis is a system model annotated with attributes, operations, and associations. The system model can be described in terms of its structure and its dynamic interoperation.

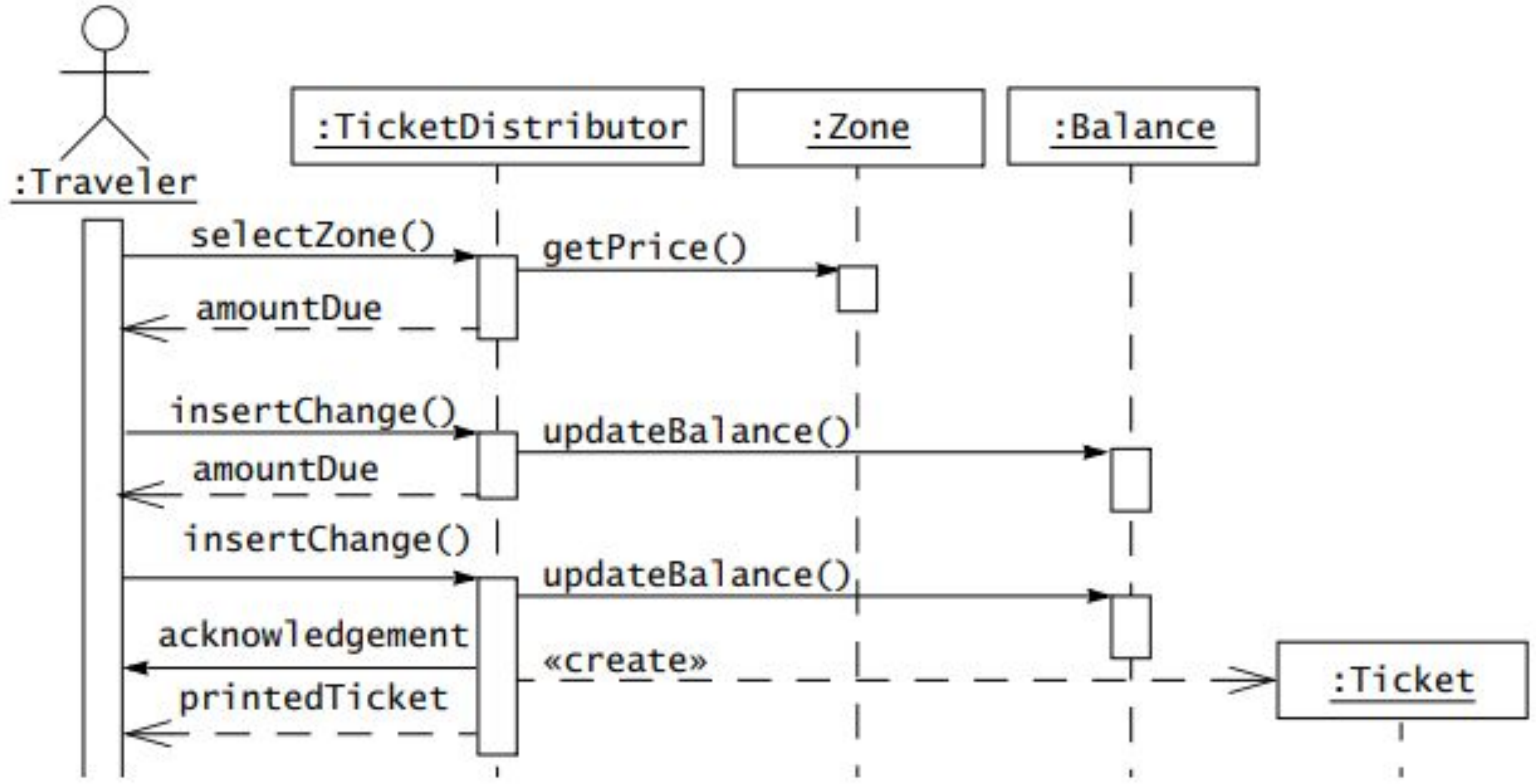


Figure 1-4 A dynamic model for the `TicketDistributor` (UML sequence diagram). This diagram depicts the interactions between the actor and the system during the `PurchaseOneWayTicket` use case and the objects that participate in the use case.

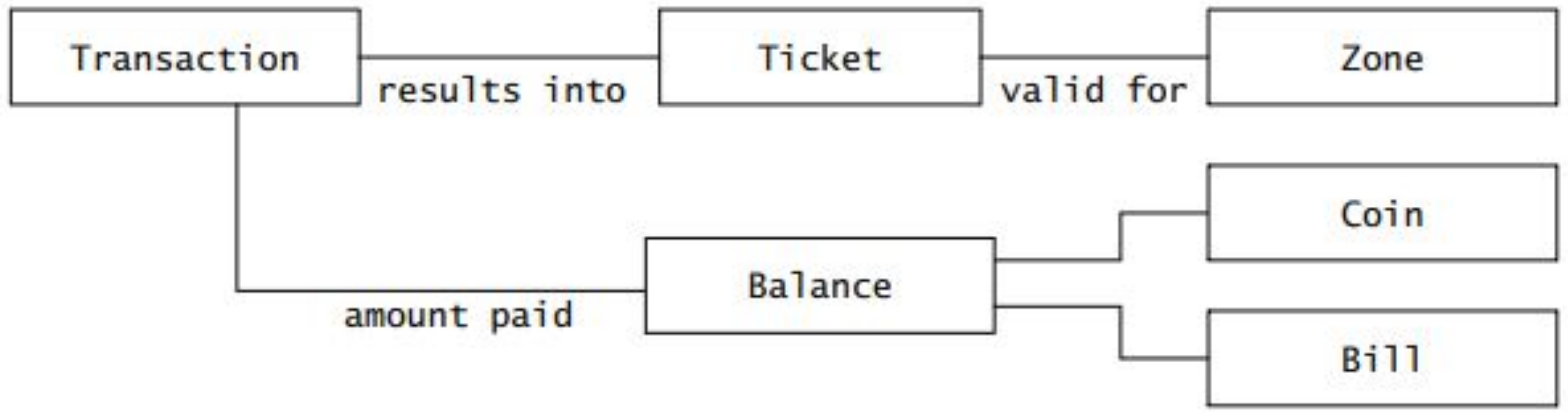


Figure 1-5 An object model for the TicketDistributor (UML class diagram). In the PurchaseOneWayTicket use case, a Traveler initiates a transaction that will result in a Ticket. A Ticket is valid only for a specified Zone. During the Transaction, the system tracks the Balance due by counting the Coins and Bills inserted.

SYSTEM DESIGN

- ❖ During **system design**, developers define the design goals of the project and decompose the system into smaller subsystems that can be realized by individual teams.
- ❖ The result of system design is a clear description of each of these strategies, a subsystem decomposition, and a deployment diagram representing the hardware/software mapping of the system.
- ❖ Whereas both analysis and system design produce models of the system under construction, only analysis deals with entities that the client can understand. System design deals with a much more refined model that includes many entities that are beyond the comprehension (and interest) of the client.

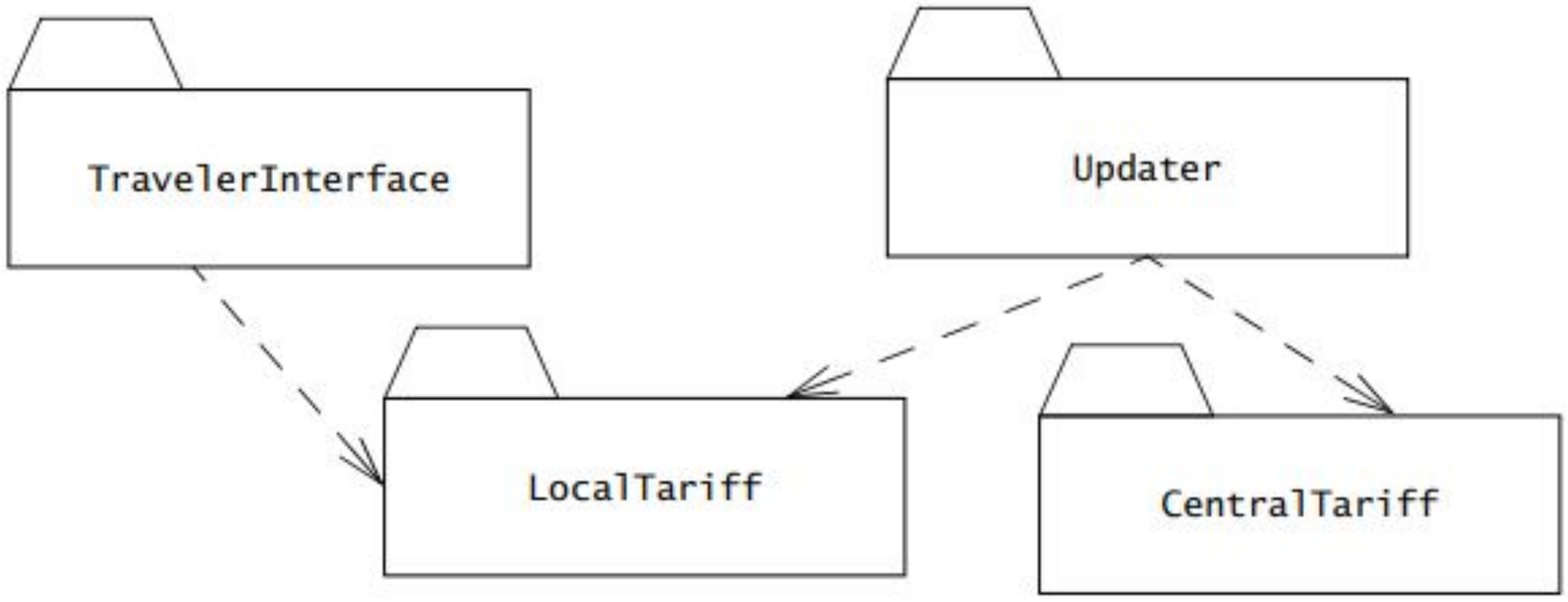


Figure 1-6 A subsystem decomposition for the `TicketDistributor` (UML class diagram, packages represent subsystems, dashed lines represent dependencies). The `TravelerInterface` subsystem is responsible for collecting input from the `Traveler` and providing feedback (e.g., display ticket price, returning change). The `LocalTariff` subsystem computes the price of different tickets based on a local database. The `CentralTariff` subsystem, located on a central computer, maintains a reference copy of the tariff database. An `Updater` subsystem is responsible for updating the local databases at each `TicketDistributor` through a network when ticket prices change.

OBJECT DESIGN

- ❖ During **object design**, developers define solution domain objects to bridge the gap between the analysis model and the hardware/software platform.
- ❖ This includes precisely describing object and subsystem interfaces, selecting off-the-shelf components, restructuring the object model to attain design goals such as extensibility or understandability, and optimizing the object model for performance.
- ❖ The result of the object design activity is a detailed object model annotated with constraints and precise descriptions for each element.

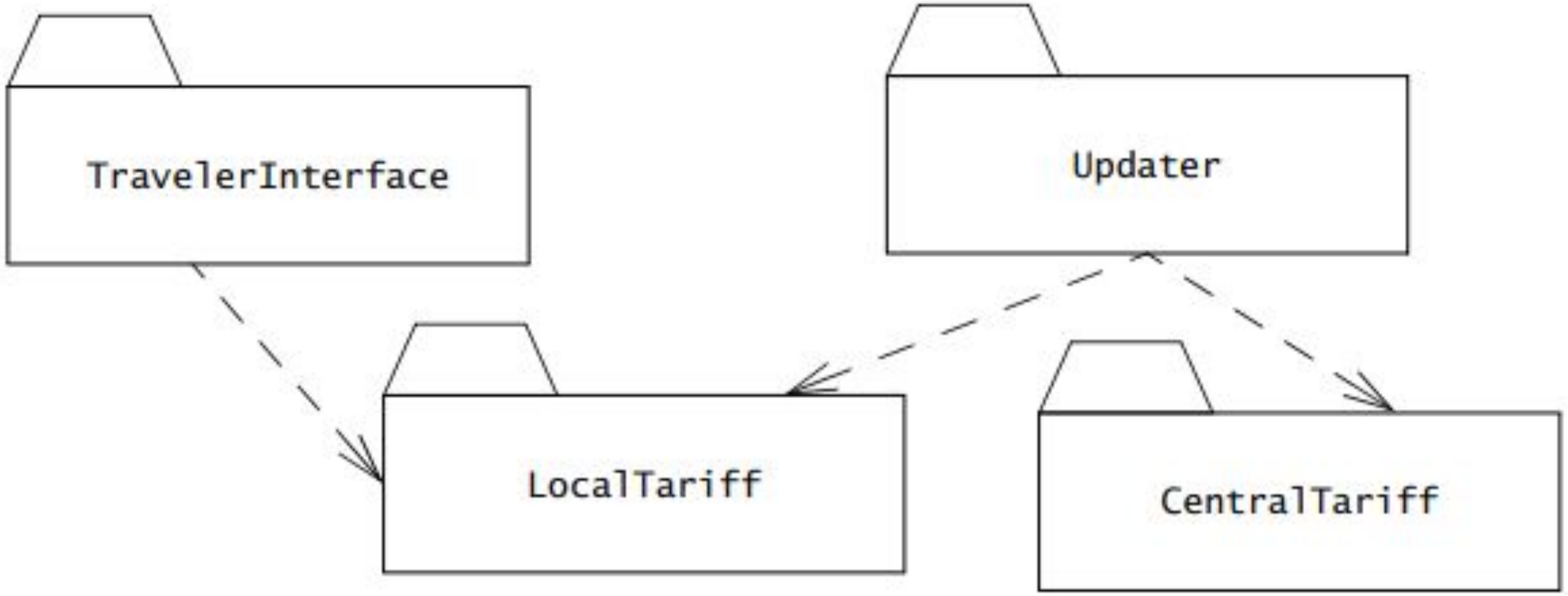


Figure 1-6 A subsystem decomposition for the `TicketDistributor` (UML class diagram, packages represent subsystems, dashed lines represent dependencies). The `TravelerInterface` subsystem is responsible for collecting input from the `Traveler` and providing feedback (e.g., display ticket price, returning change). The `LocalTariff` subsystem computes the price of different tickets based on a local database. The `CentralTariff` subsystem, located on a central computer, maintains a reference copy of the tariff database. An `Updater` subsystem is responsible for updating the local databases at each `TicketDistributor` through a network when ticket prices change.

IMPLEMENTATION

- ❖ During implementation, developers translate the solution domain model into source code.
- ❖ This includes implementing the attributes and methods of each object and integrating all the objects such that they function as a single system.
- ❖ The implementation activity spans the gap between the detailed object design model and a complete set of source code files that can be compiled.

TESTING

- During testing, developers find differences between the system and its models by executing the system (or parts of it) with sample input data sets.
  - During unit testing, developers compare the object design model with each object and subsystem.
  - During integration testing, combinations of subsystems are integrated together and compared with the system design model.
  - During system testing, typical and exception cases are run through the system and compared with the requirements model.
- The goal of testing is to discover as many faults as possible such that they can be repaired before the delivery of the system.

MANAGING SOFTWARE DEVELOPMENT

- ➢ Management activities focus on planning the project, monitoring its status, tracking changes, and coordinating resources such that a high-quality product is delivered on time and within budget. Management activities include:
  - Communication
  - Rationale Management
  - Software Configuration Management
  - Project Management
  - Software Life Cycle

COMMUNICATION

- ❖ Communication is a vital yet time-consuming aspect in software engineering, with misunderstandings leading to costly errors.
- ❖ It involves exchanging models and documents, reporting work product status, providing feedback, addressing issues, and communicating decisions.
- ❖ Diverse participant backgrounds, geographic distribution, and the complexity of information contribute to communication challenges.
- ❖ Conventions, such as agreed-upon notations, tools, and procedures are effective in mitigating these challenges. Examples include UML diagrams, document templates, CASE tools, word processors, and meeting procedures.
- ❖ Shared conventions, rather than being the best, are crucial for eliminating misunderstandings.

RATIONALE MANAGEMENT

- ❖ Rationale is the justification for decisions, encompassing the addressed problem, considered alternatives, evaluation criteria, consensus-building debates, and the final decision.
- ❖ It is crucial information for system changes, allowing developers to reevaluate decisions based on changing criteria or new alternatives.
- ❖ Despite its importance, rationale is complex to handle, making it challenging to update and maintain.
- ❖ Developers address this by capturing rationale during meetings, online discussions, representing it with issue models, and accessing it during changes.

SOFTWARE CONFIGURATION MANAGEMENT

- ❖ Software Configuration Management is the process overseeing and controlling changes in work products throughout software development.
- ❖ It allows tracking system changes by representing it as independently revised configuration items, each with an evolution tracked through versions.
- ❖ Configuration management facilitates rollback to well-defined system states in case of change failures and ensures controlled change implementation.
- ❖ After defining a baseline, all changes undergo assessment and approval, enabling management to align system evolution with project goals and limit the introduction of problems.

PROJECT MANAGEMENT

- ❖ Project management doesn't create its own artifacts but involves oversight to ensure timely, budgeted delivery of a high-quality system.
- ❖ This includes planning and budgeting during client negotiations, team organization, monitoring project status, and intervention when deviations arise.
- ❖ The project management activities that are visible to the developers and techniques that make the development–management communication more effective.

SOFTWARE LIFE CYCLE

- ❖ Software engineering is characterized as a modeling activity where developers construct models of the application and solution domains to address complexity.
- ❖ This involves ignoring irrelevant details and concentrating on what is pertinent to a particular issue, enhancing issue resolution and question answering.
- ❖ This results in the utilization of similar modeling techniques for both software artifacts and software processes.
- ❖ A comprehensive model of the software development process is known as a software life cycle.

THANK YOU

moinul39.iit@gmail.com