4. ACTIVITY 2. SYSTEM CONTEXT ANALYSIS

This chapter presents guidelines for Activity 2, which is to analyze the system contexts by considering its boundaries, functionality, manipulated information, and runtime behavior. The purpose of this activity is to gain a comprehensive overview of the target system and express the acquired system context through diagrams and textual descriptions.

✦ What is System Context?

The *System Context*, i.e., the context of a target system, provides an initial understanding of the target system in terms of its boundary, functionality, persistent information, and runtime behavior. It should be noted that the system context is not intended to serve as a design model; rather, it serves as the foundation for making specific and detailed design decisions in subsequent activities.

This activity proves particularly valuable when dealing with highly complex target systems or when the architect lacks prior knowledge or experience in the domain of the target system.

✦ Types of System Contexts

The contexts of a target system can be categorized into the following types.

- **Boundary Context**

  The boundary context of a software system encompasses the system's scope and external interfaces, establishing clear boundaries that determine its internal and external components. This context specifies the system nodes, elements in the system boundary, essential datasets to be stored, and their interactions. The elements in the boundary can be users, external systems, and connected hardware devices.

  The boundary context is further utilized in identifying the components, subsystems, and external entities that interact with the software system.

- **Functional Context**

  The functional context of a software system defines the specific functionality offered by the target system and the system interactions with users, external systems, and hardware devices. This context outlines the core features and capabilities of the software system, highlighting the tasks it can perform and the services it provides to its users.

  By understanding the functional context, stakeholders obtain a comprehensive understanding of the operational aspects of the system, including its capabilities, supported tasks, and interactions with users and external entities.

  This context is further utilized in identifying functional components of the system and their interfaces.

- **Information Context**

  The information context specifies the persistent datasets manipulated by the target system and the relationships among the datasets. The persistent datasets are modeled as persistent object classes in object-oriented development.

  By analyzing the information context, stakeholders gain a comprehensive understanding of the persistent datasets and their relationships. This context is further utilized in deriving data components and designing the persistency of the datasets.

- **Behavior Context**

  The behavior context defines the overall runtime behavior of the target system, encompassing the complete set of valid control flows within the system. While the functional context focuses on modeling the system's functionality, the behavior context focuses on the valid sequences of activities that occur at runtime.

  This context is further utilized in defining both the overall control flow of the target system and the detailed control flows for selected functional elements. Defining the overall control flow involves understanding and mapping the sequence of activities, events, and decisions that occur throughout the system.

- **Other Type of Context**

  In most cases, the set of four context types is typically adequate for capturing the system context. However, certain target systems may possess unique characteristics or complexities that go beyond the scope of these four context types. In such cases, additional context types may be sought and utilized to capture and represent the specific aspects and nuances of the system.

  For instance, a presentation context can be useful and valuable for systems involving extensive user interactions, such as online games. The presentation context focuses on capturing and representing the user interface design, user experience, and user interaction patterns specific to the system.

  Similarly, in the case of platform-as-a-service (PaaS) systems or other complex deployment scenarios, a deployment context becomes crucial. The deployment context focuses on capturing and addressing various considerations related to the deployment architecture, infrastructure requirements, scalability, and system administration aspects.

✦ Steps for System Context Analysis

The *System Context Analysis* can be conducted systematically by following the sequence of steps depicted in Figure 4-1.

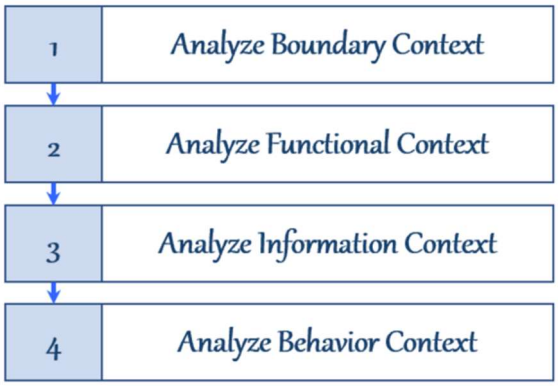

Figure 4-1. Steps for System Context Analysis

Step 1 is to analyze the boundary context of the system in terms of computing nodes, users, external systems, connected devices, and persistent datasets. This context provides the architect with the high-level view of the target system and its interacting elements. Step 2 is to analyze the functional context in terms of the key functional categories and the functionality expressed in use cases. Step 3 is to analyze the information context, which specifies the persistent datasets and their relationships. Step 4 is to analyze the behavior context, which specifies the runtime control flow of a target system.

Additional context types may be sought and utilized, only if applicable as describe earlier.

4.1. Step 1. Analyze Boundary Context

This step is to analyze the boundary context of a target system and visualize it using a data flow diagram. The boundary context specifies the computing nodes of the system, elements in the system boundary, datasets to be stored, and their interactions.

The boundary context can be represented using a Data Flow Diagram (DFD), which visually depicts the processes of the system, external elements, persistent datasets, and the flow of data. To effectively depict the system boundary, the level 0 DFD, also known as Context Diagram, is employed.

A DFD consists of four elements: *Process*, *Terminal*, *Data Store*, and *Data Flow* as shown in Figure 4-2.

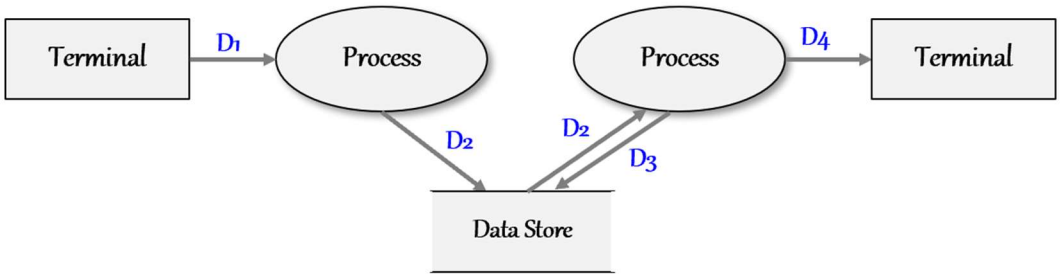

Figure 4-2. Elements of Data Flow Diagram.

A *Data Flow* in DFD represents the movement of information from one part of the system to another. It is represented as a directional arrow with a textual label indicating the content of the data being transferred. A data flow can be defined between a terminal and a process, between two processes, or between a process and a data store.

The elements of a boundary context are mapped to the elements of DFD as follows.

- **Functionality of the System → Process in DFD**
- **Elements in the Boundary → Terminals in DFD**
- **Persistent Dataset → Data Store in DFD**
- **Transfer of Data → Data Flow in DFD**

The DFD for modeling the boundary context can be systematically constructed by applying the following tasks.

4.1.1. TASK 1. DEFINE PROCESSES IN DFD

This task is to identify the high-level computing nodes of a target system, and them as processes. A *Process* in DFS represents a specific functionality within a system, and it is depicted as an oval shape. It encapsulates a set of activities that transform input data into output data. In a lower-level DFD, a process can be further decomposed into smaller sub-processes, enabling a gradual refinement of the functionality.

To accurately depict the boundary context of a system, we utilize the level 0 DFD. In this diagram, a process represents either the entire computer system or a specific sub-system within it. If the target system runs on a single node, the level 0 DFD will consist of a single process, and the name of this process will correspond to the name of the target system itself. However, if the target system operates on multiple sub-systems, i.e., a multi-tier system, the level 0 DFD will contain multiple

processes where each process represents a distinct sub-system. The number of processes in the DFD will correspond to the number of sub-systems within the overall architecture.

✦ **Example) Processes of Car Rental Management System**

The SRS of this system explicitly defines three distinct types of system nodes, and accordingly, they are represented as three processes in a DFD, as illustrated in Figure 4-3.

Figure 4-3. Processes of Car Rental Management System

- **Process of Mobile Client**

  This process represents a native mobile app designed for customers, enabling them to manage reservations and rentals of vehicles.

- **Process of Rental Center Client**

  This process represents a client application specifically designed for staff members working in rental centers. It provides comprehensive functionality for effectively managing car inventories, processing rentals, and handling returns.

- **Process of Headquarter Server**

  This process represents the headquarter server, which is specifically designed to support staff members working at the headquarter office. It provides comprehensive functionality for overseeing rental operations across multiple rental centers, managing car inventories, analyzing business profits and expenses, and generating various business reports.

As seen in this example, the processes of a level 0 DFD explicitly specify the computing nodes of the system and their roles.

4.1.2. TASK 2. DEFINE TERMINALS IN DFD

This task is to model the elements in the system boundary and represent them as terminals of DFD. A *Terminal* in a DFD represents an external entity that interacts with the system by providing input or consuming output. It is depicted as a rectangle shape. An external entity can take various forms, such as a user, an external system, or a connected hardware device.

✦ **Types of Terminals**

- **Terminal for User**

  A system typically interacts with users, who provide input to the system or consume output

from the system. Hence, users are modeled as terminals.

For the Car Rental Management System, customers engage in data exchange with the system for operations such as searching for availability, making reservations, browsing active rentals, and extending the rental period. Hence, customers are effectively represented as a terminal named 'Customer'.

- **Terminal for External System**

  A system may interact with external systems, that can provide data values to the system or process data provided by the system. Hence, such external systems are modeled as terminals.

  For example, the Car Rental Management System interacts with external systems including a system that validates the driver licenses of customers, and a system that grants authorizations for making payments. Hence, these external systems are modeled as terminals named ‘Driver License Validator’ and 'Payment Authorizer'.

- **Terminal for Hardware Device**

  A system may operate with connected hardware devices. For instance, Smart Home applications typically operate with various types of sensors, cameras, IoT devices, and actuators. They gather information from sensors, cameras, and IoT devices, and transmit control commands to the actuators to autonomously manage the home operations. These hardware devices are modeled as terminals.

✦ **Example) Terminals of Car Rental Management System**

The SRS of this system specifies the types of users, external systems to interact and a connected hardware device. These elements in the system environment are modeled as terminals, as shown in Figure 4-4.

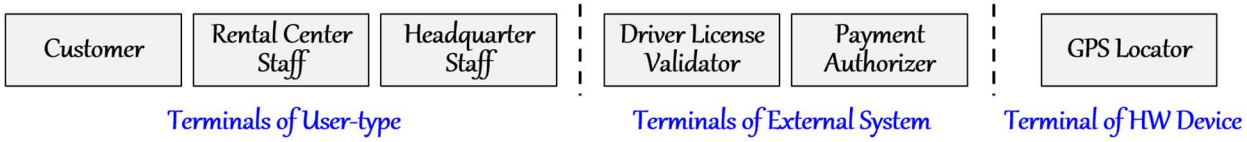

Figure 4-4. Terminals of Car Rental Management System

The figure illustrates seven types of terminals within the system, encompassing three user types, two external system types, and one hardware device. Each terminal plays a role in either providing input to the system or consuming output from it.

The modeling of these terminals accurately aligns with the SRS of the Car Rental Management System.

4.1.3. TASK 3. DEFINE DATA STORES IN DFD

This task is to identify the persistent datasets managed by the target system and represent them as data stores. A *Data Store* in a DFD represents a persistent dataset that is accessed and manipulated by the system. It is depicted as a double-lined shape. A Data Store can manifest in various forms, such as a file, database table, or main memory buffer.

A common representation of a Data Store is a database table, which offers a more structured and organized approach to data storage. Another manifestation of a Data Store is a file, which allows for the storage and retrieval of data on a permanent storage device such as a hard drive. Also, a Data Store can be realized as a main memory buffer, which enables fast access to frequently used or temporary data.

We can infer and derive the persistent datasets manipulated by the target system from the SRS. Typically, the SRS of a target system does not explicitly specify the complete list of persistent datasets. Instead, architects are advised to derive the datasets by posing the question, "What dataset is manipulated by each function of the system?".

✦ **Example) Data Stores of Car Rental Management System**

From the SRS of this system, we can derive a number of datasets as shown in Figure 4-5.

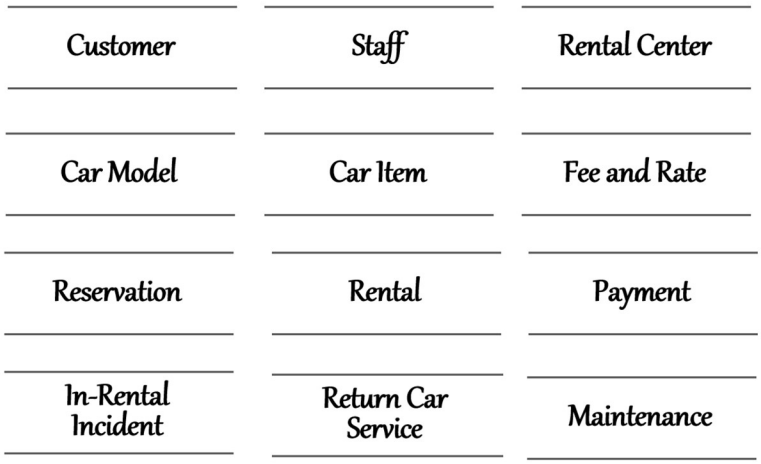

Figure 4-5. Data Stores of DFD for Car Rental Management System

The figure illustrates twelve distinct types of persistent datasets. Each dataset can be realized as a database table.

4.1.4. TASK 4. DEFINE DATA FLOWS IN DFD

This task is to identify data flows among the processes, terminals, and data stores. Each data flow is depicted as an arrow-headed line accompanied by a textual label. The data flows are not typically specified in the SRS. Therefore, architects should infer and specify the data flows by considering the following questions.

- What input is provided by each terminal? And which process manipulates the input?
- What output is generated by each process? And which terminal consumes the output?
- In which data store is the output from a process stored?
- From which data store does a process retrieve its data?
- What are the data items exchanged between two interacting processes?

Once the data flows are identified, they can be represented as directed arrows with accompanying textual labels. However, for the purpose of representing the system boundary, it is acceptable to omit the textual labels, i.e., names of data items.

Once we connect the set of processes, terminals, and data stores with the identified data flows, we can finalize the level 0 of DFD that represents the boundary context of the system.

✦ **Example) Data Flows of Car Rental Management System**

By applying the suggested questions for deriving data flows, we identify a number of data flows. Figure 4-6 shows the result of establishing the connections between the processes, terminals, and data stores with the identified data flows.

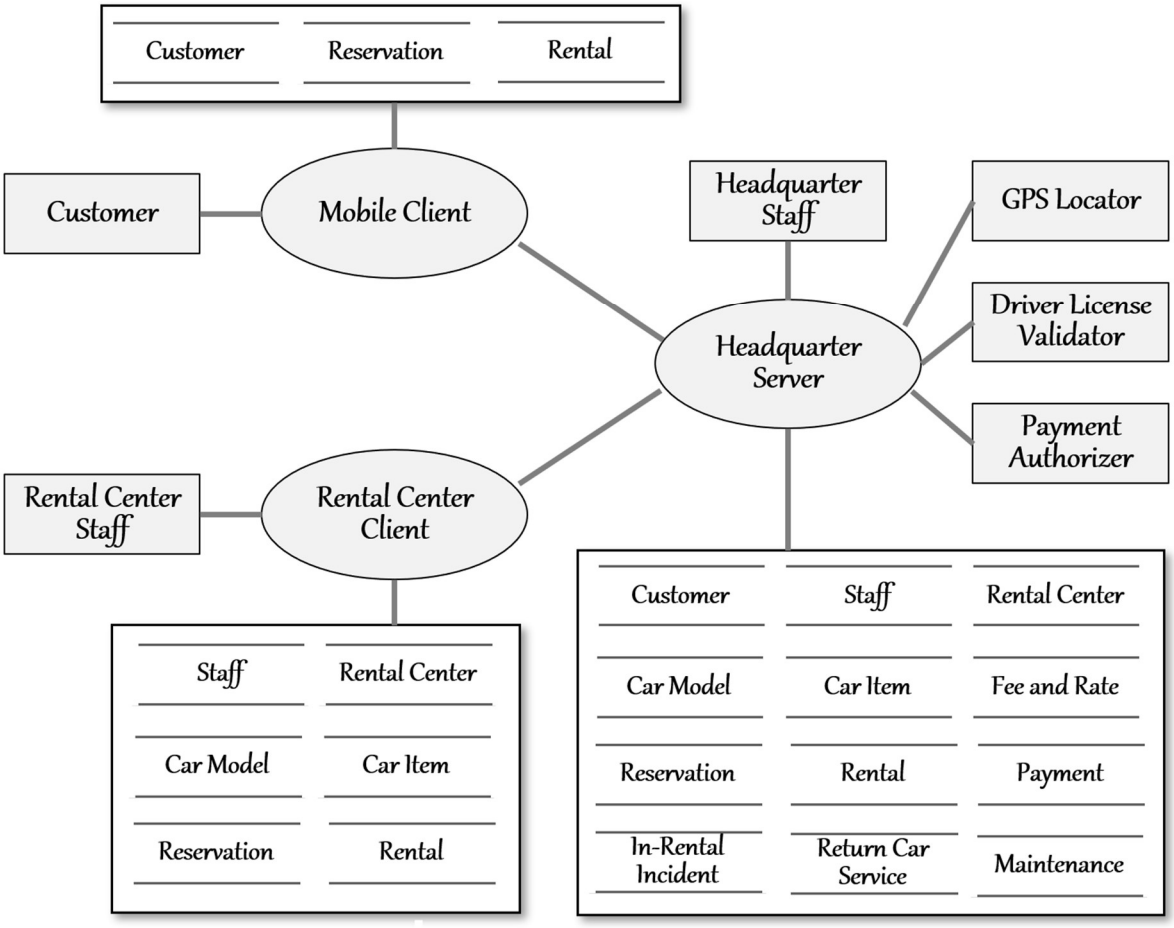

Figure 4-6. Level 0 DFD for Car Rental Management System

This DFD shows all three processes, i.e., sub-systems, of the Car Rental Management System. It also shows all the identified terminals and their connections to the processes.

The Headquarter Server is responsible for managing the master repository of the system, which is why all the data stores are connected to the 'Headquarter Server' process. On the other hand, the Rental Center Client node exclusively handles a subset of the entire dataset within its local database, manipulating it locally. As for the Mobile Client node, it maintains a cache database comprising only three data stores that it frequently accesses.

The resulting Data Flow Diagram (DFD) may not appear complex, but it effectively visualizes all the essential elements of the boundary context for the Car Rental Management System.

4.2. STEP 2. ANALYZE FUNCTIONAL CONTEXT

This step is to analyze the functional context of a target system and visualize it using a use case diagram. Every software system provides some functionality, which should be fully comprehended

by architects. The functional context specifies the set of functions provided by a target system, and actors invoking the functions. This context focuses only on the functionality provided by the system, not considering how the functionality is delivered.

The functional context can be represented using a Use Case Diagram, which consists of actors, use cases, and their relationships as shown in Figure 4-7.

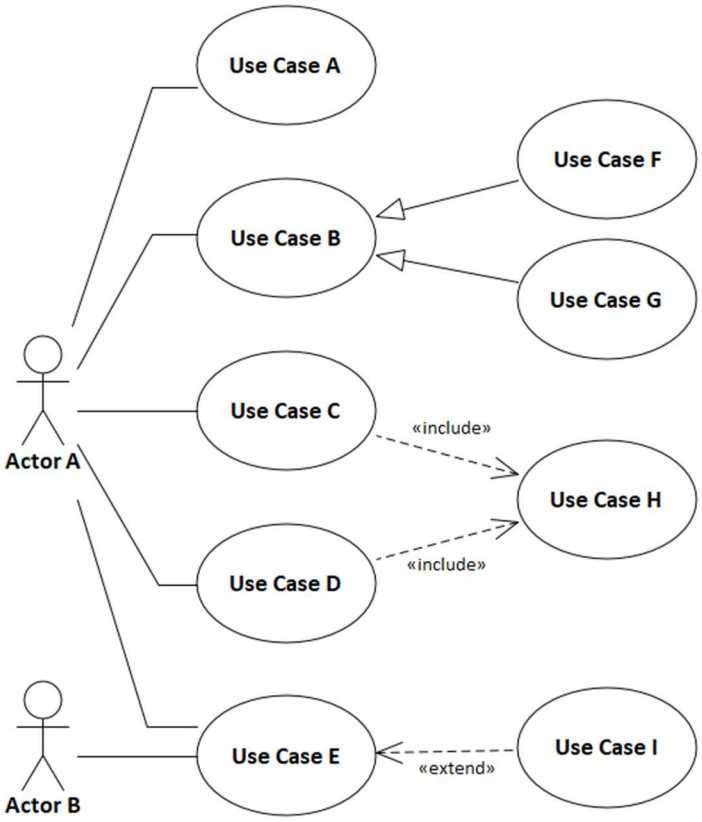

Figure 4-7. Elements of Use Case Diagram

The figure shows two actors A and B, nine use cases A through I, and three types of relationships between use cases: Generalization, Include, and Extend relationships.

The use case diagram for the functional context can systematically be constructed by applying the following sequence of tasks.

4.2.1. TASK 1. DEFINE ACTORS

This task is to identify the actors of the target system. An actor in a use case diagram represents a specific role played by an entity that interacts with the system. An actor can be a human user, an

external system, a hardware device, or a software agent. An actor of software agent type specifies an object that runs in background and invoke its relevant use cases, called a daemon process.

For example, the actors for the Car Rental Management System consists of actors of user type, external system type, and software agent type, as shown in Figure 4-8.

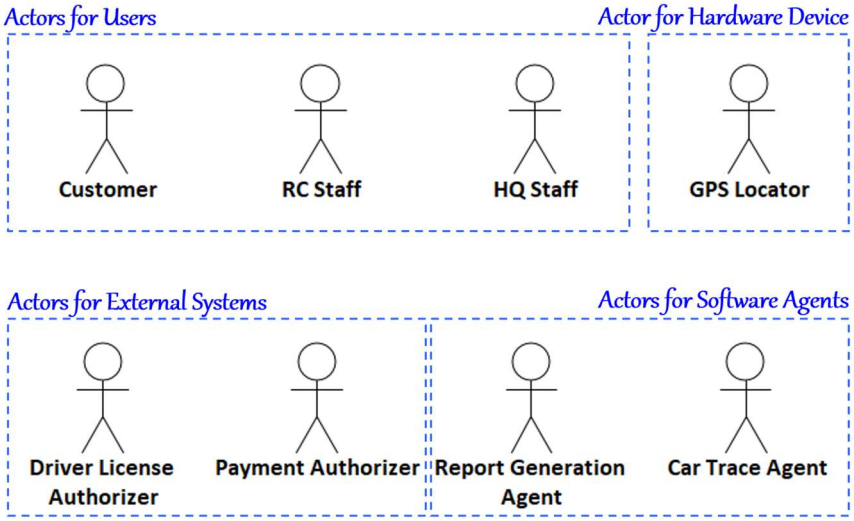

Figure 4-8. Actors for Car Rental Management System

The actors of user type are Customer, RC Staff for staff members at rental centers, and HQ Staff for staff members at Headquarter Office. The actor of hardware device type is GPS Locator which transmits the location of on rental vehicles through cellular networks. The actors of external system type are Driver License Authorizer which validates the customer's driver license and Payment Authorizer which authorizes the credit card payments. The actors of software agent type are Report Generation Agent which generates periodical business reports and Car Trace Agent which traces the location of rental vehicles.

4.2.2. TASK 2. DEFINE USE CASES

This task is to analyze the functionality provided by a target system and represent them as use cases. Each use case represents the cohesive functionality of the system.

The names of use cases are given in verb form, revealing some functionality. Some of the use cases for the Car Rental Management System are shown in Figure 4-9.

Figure 4-9. Use Cases for Car Reservation Management

These use cases represent reservation-related operations of the system. Each use case represents a distinct functionality within the Reservation Management category of the system.

An effective way of defining use cases is to begin by identifying the functional groups within the system and subsequently deriving use cases for each of these groups. This approach allows for a systematic and comprehensive coverage of the system's functionalities.

✦ Defining Functional Groups

A functional group represents a distinct category of the system functionality as the whole system functionality can be viewed as a collection of functional groups.

The functional requirement of SRS is typically logically categorized by domain experts of the system. Therefore, the functional groups can be derived systematically, to a large extent, from the functional requirement of the SRS.

Once functional groups are identified, it is advisable to assign a two or three-character prefix to each functional group. This helps in organizing use cases in a manageable and structured manner.

For example, the functional groups and their corresponding two-character prefixes for the Car Rental Management System can be defined as follows:

- Customer Profile Management → CP
- Staff Profile Management → SP
- Rental Center Profile Registration → RP
- Inventory Management → IM
- Rental Rate Management → RR
- Reservation Management → RS
- Checkout Management → CH
- Rental Incident Management → RI
- Return Management → RE
- Post-Return Maintenance → CA
- Business Report Generation → BR

✦ Defining Use Cases

We now identify use cases for each functional group, assigning an identifier to each use case. The identifier will consist of a prefix for the functional group followed by a sequential number.

Some of the use cases in the Car Rental Management System are shown in Figure 4-10.

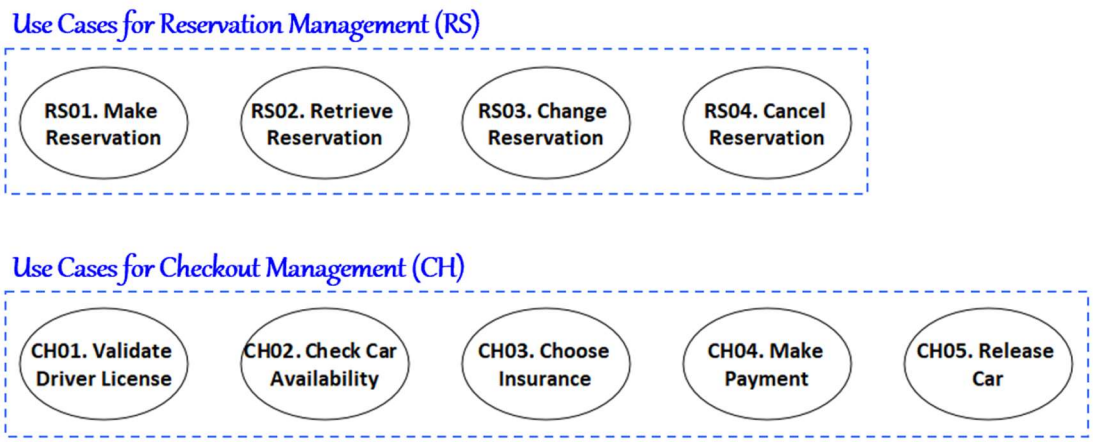

Figure 4-10. Use Cases for Car Rental Management System

Four use cases are defined for the functional group of Reservation Management (RS), and five use cases are defined for the functional group of Checkout Management (CH).

4.2.3. TASK 3. DEFINE RELATIONSHIPS

This task is to define relationships between actors, between actors and use cases, and between use cases. A relationship in a use case diagram can be an invocation, generalization, include, and extend.

Some of the relationships in the Car Renal Management System are shown in Figure 4-11.

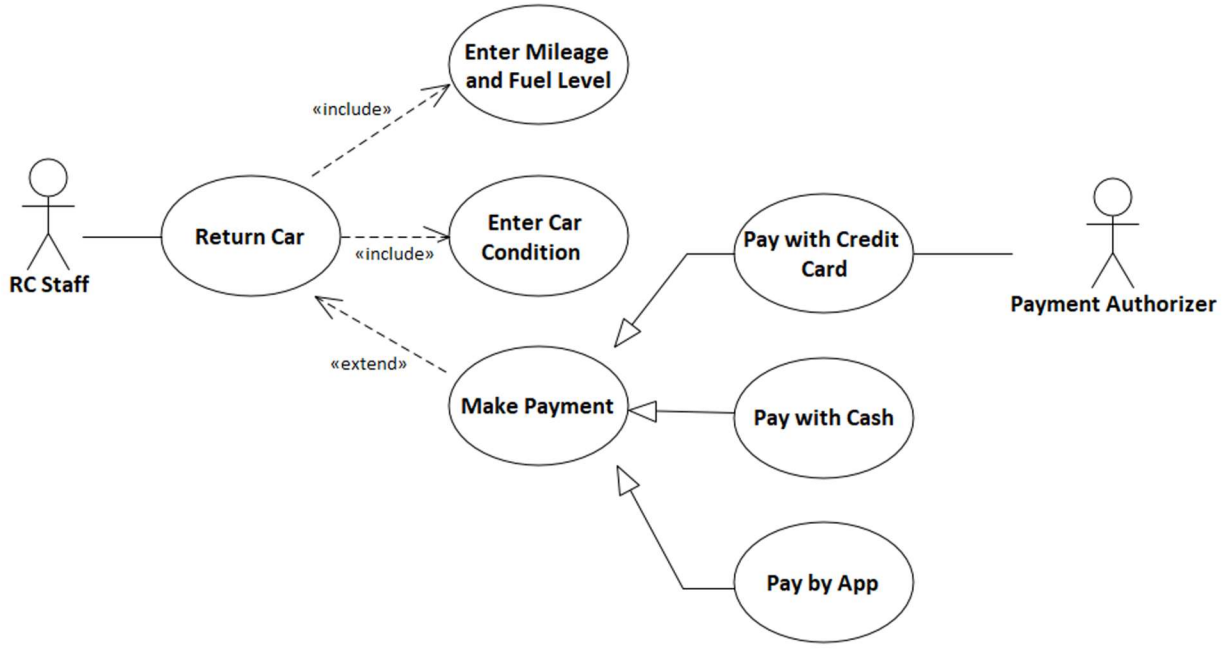

Figure 4-11. Relationships for Car Rental Management System

✦ Invocation Relationship

An invocation occurs between an actor and a use case. An active actor invokes a use case, and a passive actor is invoked by a use case. An invocation in a use case diagram takes place between an actor and a use case. An active actor initiates the invocation of a use case, while a passive actor is invoked by a use case. The invocation is represented by a solid line between the actor and its corresponding use case. Alternatively, a directed arrow can be used to explicitly indicate the direction of the invocation.

In Figure 4-11, an active actor RC Staff invokes the use case Return Car, and a passive actor Payment Authorizer is invoked by the use case Pay with Credit Card.

✦ Generalization Relationship

The generalization in a use case diagram is a relationship where a more general element abstracts and encompasses more specific elements.

A generalization relationship can occur between actors, involving a base actor and its derived actors. A derived actor inherits the ability to invoke all the use cases that are available to the base actor. For example, an actor Customer for a system can be defined as a base actor where another actor VIP Customer is defined as its derived actor. Then, the derived actor VIP Customer can access all the use cases available to its base actor Customer.

A generalization relationship may also occur between use cases: a base use case and its derived use cases. The base case denotes a generic functionality that is specialized by its derived use cases. In Figure 4-11, the use case Make Payment serves as a base use case, with its functionality further specialized by three derived use cases: Pay with Credit Card, Pay with Cash, and Pay with App. When the base use case is invoked, one of the derived use cases is chosen and invoked instead, depending on the specific payment method chosen.

✦ Include Relationship

The *Include* relationship in a use case diagram occurs between use cases: a base use case and its included use case. Whenever a base use case is invoked, its included use case is also invoked.

In Figure 4-11, the invocation of the base use case Return Car will always invoke its included use cases: 'Enter Mileage and Fuel Level' and 'Enter Car Condition.'

✦ Extend Relationship

The *Extend* relationship in a use case diagram occurs between use cases: a base use case and its extended use case. When the base use case is invoked, the extended use case may optionally be invoked depending on the given condition.

In Figure 4-11, the invocation of the base use case Return Car may or may not invoke its extended use case Make Payment. That is, this extended use case is invoked only if a rental car is returned later than the scheduled return date.

The functional context represented in a use case diagram provides architects with a sufficient level of comprehension on the functionality delivered by the target system.

The complete U.C. diagram to be included

Figure 4-12. Functional Context of Car Rental Management System