4. Activity 2. System Context Analysis (v. 0.9)

4.1. Overview of the Activity

This activity is to acquire a high-level context of the target system and represent the acquired system context in diagrams and textual descriptions. The system context is an initial understanding of the target system in terms of the system boundary, functionality to deliver, information manipulated, and the runtime behavior of the system. Therefore, the system context is not intended to serve as a design model; rather, it is utilized as the basis for making specific and detailed design decisions in subsequent activities.

This activity is especially useful when the complexity of a target system is considerably high, or the architect has no prior knowledge or experiences in the target domain.

The steps in this activity are shown in Figure 4-1.

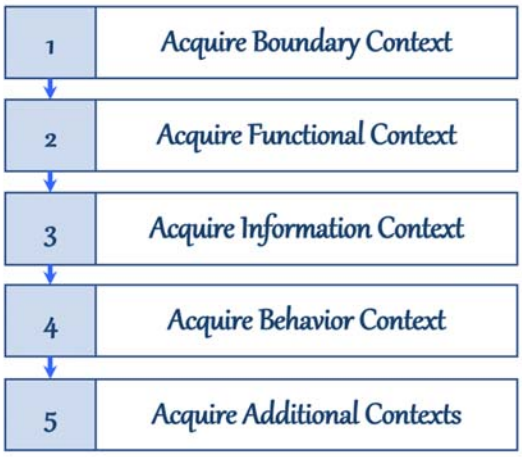

Figure 4-1. Steps in Activity of System Context Analysis

Step 1 is to acquire the context of the system boundary, which specifies the computing nodes, users, external systems, connected devices, and persistent datasets. The boundary context provides the architect with the high-level view of the target system and its interacting elements.

Step 2 is to acquire the functional context, which specifies the list of system functionalities to deliver and the actors invoking the functionalities. The functional context provides the architect with the whole functionality of the system and various subjects interacting with the functionalities.

Step 3 is to acquire the information context, which specifies the persistent datasets and their relationships. The information context provides the architect with the whole datasets and their relationships that should be maintained in secondary storages or databases.

Step 4 is to acquire the behavior context, which specifies the overall control flow of the target system. The behavior context provides the architect with the overall runtime behavior of the target system.

Step 5 is to acquire any additional system contexts if applicable. The additional contexts should be determined by considering the requirements and the characteristics of the target system.

The input artifacts to performing this activity are the followings.

- **Software Requirement Specification (SRS)**

  SRS specifies the functional and non-functional requirements of a target system. Therefore, the context analysis should be performed around the given SRS.

- **Domain Knowledge**

  The initial SRS provides the description of essential requirements. However, the given SRS may not be complete, consistent, and/or precise. Hence, architects should also utilize his or her domain knowledge in modeling the system contexts.

The output artifacts from this activity are the followings.

- **Boundary Context**

  The boundary context specifies the target system, elements in its boundary, and their interactions in terms of dataflow. The elements in the boundary can be users, external systems, and hardware devices interacting with the system. Data Flow Diagram can be utilized to model the boundary context.

- **Functional Context**

  Functional Context specifies the functionality provided by the target system, and the system interaction relationships with users, connected hardware devices, and external systems. Use Case Diagram can be utilized to represent the functional context.

- **Information Context Model**

  Information Context specifies the persistent datasets manipulated by the target system and their relationships. A persistent dataset is modeled as a persistent object class in object-oriented development. Class Diagram can be utilized to specify the persistent information of the system.

✦ Behavior Context Model

Behavior Context specifies the overall runtime behavior of the target system, i.e., the set of all valid control flows in the system. Activity Diagram can be utilized to specify the behavior of the system.

✦ Other Contexts

The set of 4 context models is generally sufficient for acquiring the system context. However, a target system may have additional types of contexts that should be comprehended by architects. For examples, *Presentation Context* could be useful for systems with intensive user interactions such as online games, and *Deployment Context* could be useful for platform-as-a-service systems.

4.2. STEP 1. ACQUIRE BOUNDARY CONTEXT

This step is to acquire the boundary context of the target system and represent the context. The boundary context is mainly to show the target system, the elements in the system boundary, and the interaction between them.

The boundary context can well be specified using Data Flow Diagram (DFD). It 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.

✦ Process

A process in DFD represents a specific functionality, and it is represented as an oval shape. A process in a DFD can be decomposed into multiple sub-processes in its lower-level DFD, and hence the system functionality can be refined in incrementally in lower-level DFDs, i.e., level 0 DFD, level 1 DFDs, level 2 DFDs, and more. We utilize only the level 0 DFD, also called *Context Diagram*, to represent the boundary context of the system.

✦ Terminal

A terminal in DFD represents an external entity that provides input to the system and/or consumes an output from the system. It is represented as a rectangle shape. The common types

of terminals are user, external system interacting with the system, and hardware device connected.

✦ **Data Store**

A data store in DFD represents a persistent dataset and it is represented as a double-lined shape. A data store specifies an information storage that can be implemented as file, database table, or a main memory buffer.

✦ **Data Flow**

A data flow in DFD represents the transfer of information from one part of the system to another. It is represented as a textual label over directional arrows. A data flow can be defined between a terminal and a process, between two processes, or between a process and a data store.

By using DFD, we can effectively represent the boundary context of the target system. That is, the elements of boundary context are well mapped to the elements of DFD.

- **Functionality of the System → Process in DFD**
- **Elements in Boundary → Terminal in DFD**
- **Persistent Data → Data Store in DFD**
- **Flow of Information → Data Flow in DFD**

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

4.2.1. TASK 1. DEFINE PROCESSES IN DFD

A process in DFD is to represent a specific functionality. At the level 0 DFD, a process represents a system node, i.e., a computer system or a sub-system. If a target system runs on a single computer system, i.e., a single node, there will be just one process representing the whole systema the level 0 DFD. Often, the name of the process becomes the name of the target system.

For Car Rental Management System, the SRS specifies four system nodes as shown in Figure 4-3.

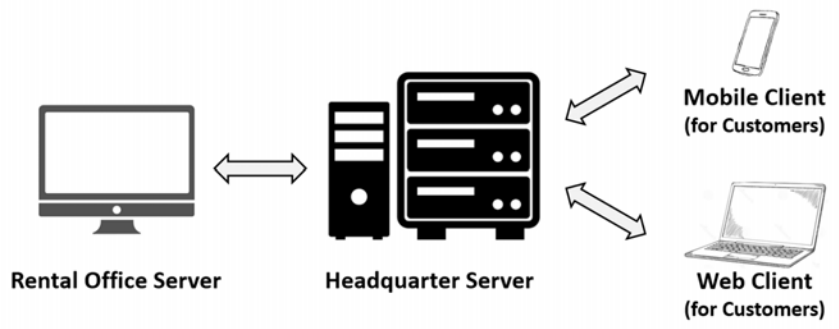

Figure 4-3. System Nodes of Car Rental Management System

Accordingly, we define four processes of level 0 DFD as shown in Figure 4-4.

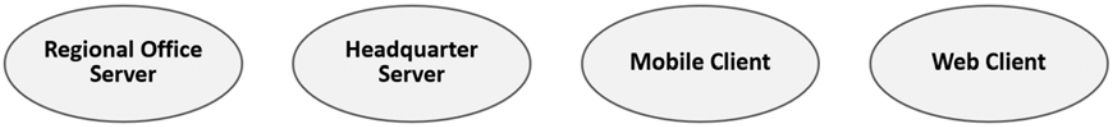

Figure 4-4. Car Rental Management System with 4 Nodes

The process of *Mobile Client* models a mobile app that is used by customers. It provides all the functionality required for the customers. The process of *Web Client* models a web application that is used by customers. It provides the same functionality as the mobile app, but its interface is web browser based. The process of *Regional Office Server* models a system node that is used by staffs of regional centers of the company. It provides all the functionality required for the staffs. The process of *Headquarter Server* models a system node that is used by headquarter staffs. It provides all the functionality required for the headquarter staffs.

Another observation of mobile app is how the mobile app is implemented. A mobile app can be a native app, web app, or hybrid. The web app form of mobile apps is not physically deployed on mobile devices; rather, it represents a mobile web browser-based interface and hence its web service is deployed on its server. This is considered as a virtual node.

Accordingly, the web client is not physically deployed on the clients' devices; rather, it represents a web-browser-based interface and hence its web service is deployed on its server. This is also considered as a virtual node. If we develop a web app for mobile device users, then the level-0 DFD should be represented as a two-tier system as shown in Figure 4-5.

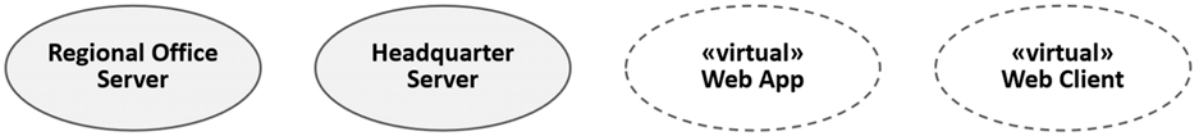

Figure 4-5. Car Rental Management System with 2 Nodes

The remaining parts of the architecture design will be based on this two node-architecture.

4.2.2. TASK 2. DEFINE TERMINALS IN DFD

This task is to model the elements in the system boundary and represent them as terminals of DFD. The common types of elements in the system boundary are listed below.

- **Terminal of User type**

  A system typically interacts with users. A user of the system is not a part of the system; rather, the user resides in the system boundary. For example, customers of eCommerce system enter various requests to the system, and the system provides the results of processing the requests. Hence, users reside in the system boundary.

  Since a user provides input to the system to invoke the system functionality and/or consumes output of the system, it is modeled as a terminal.

- **Terminal of External System type**

  A system may interact with external systems. Often, an external system does not invoke the functionality of the target system; rather, the functionality of the external system gets invoked by the target system. For example, e-commerce systems interact with an external system, *Credit Card Payment Authorizer,* system to acquire authorizations for product purchases. Another example is *Car Rental Management System* that interact with an external system, *Driver License Validation System,* to validate the driver license of a customer.

  Since an external system may provide an input to the target system and/or receives an output from the target system, it is modeled as a terminal.

- **Terminal of Hardware Device type**

  A system may be installed with hardware devices, and it gathers information from the devices and/or control the devices. For example, a smart factory system is configured with various hardware sensors to acquire the state information of a factory and various actuators to control the factory through the actuators.

  Since a connected hardware device provides its state value to the target system or receives control commands from the target system, it is modeled as a terminal.

For Car Rental Management System, the SRS specifies four system nodes as shown in Figure 4-6.

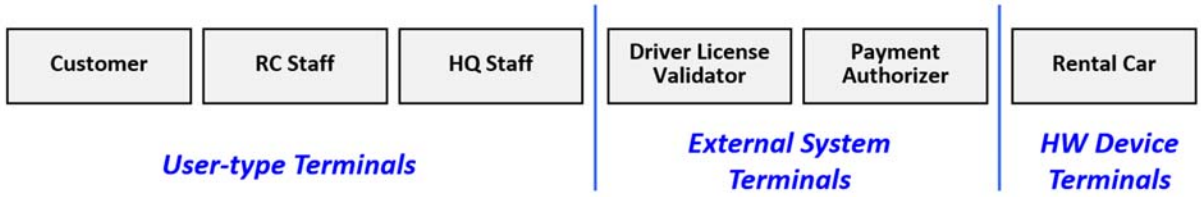

Figure 4-6. Terminals of DFD for Car Rental Management System

The DFD for Car Rental Management System includes 3 terminals of user-type, 2 terminals of external system-type, and 1 terminal for hardware device-type. According to the SRS of the system, the rental car is equipped with an embedded software that communicates with the Headquarter server to exchange information including the location and the status of a rental car, and hence it is treated as a terminal.

4.2.3. TASK 3. DEFINE DATA STORES IN DFD

This task is to identify the persistent information managed by the target system and represent the information as data stores. A system often manages datasets that should be stored persistently or permanently in the form of a database. The persistent information managed by the system is modeled as data stores in DFD. The data store of DFD can be implemented as a file on secondary storage, database table, or a long-lasting main memory buffer.

For Car Rental Management System, we can derive the persistent information from the SRS as shown in Figure 4-7.

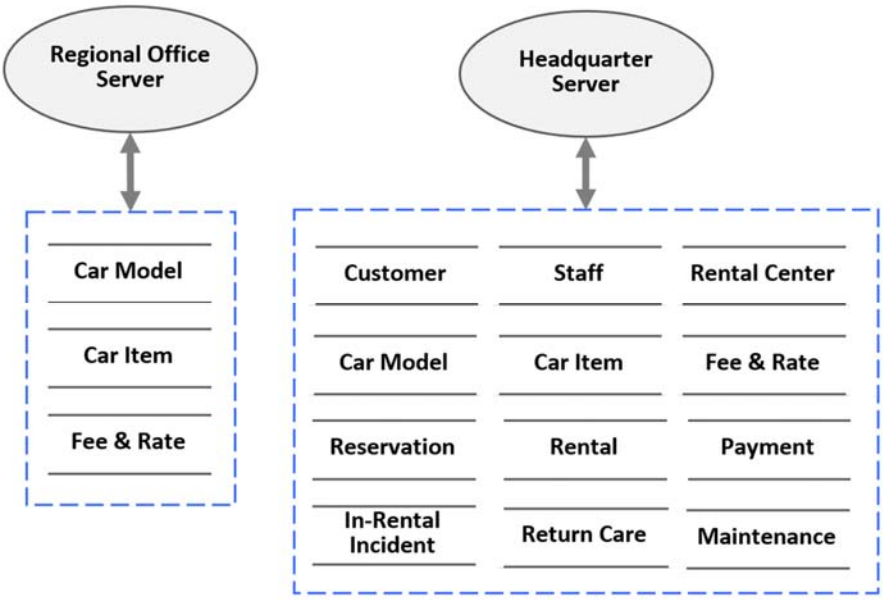

Figure 4-7. Data Stores of DFD for Car Rental Management System

The system is modeled with 4 processes, i.e., physical nodes. Only 2 of the system nodes are to maintain the persistent datasets as shown in the figure. The node of *Headquarter Server* maintains the master repository, that stores all the persistent datasets. The node of *Regional Center Server* maintains a subset of the master repository to maintain the information of the car inventory and the fees and rates in each regional center.

The other nodes, *Mobile App* and *Web Service* do not maintain any persistent datasets and hence they retrieve the necessary data from the master repository.

4.2.4. TASK 4. DEFINE DATA FLOWS IN DFD

This task is to identify the flow of data items among the processes, terminals, and data stores. Each data flow is represented as an arrow-headed line with a textual label between elements. The following questionnaire can be utilized to derive the data flows of the system.

- **What are the key inputs from users?**
- **What are the responses of the target system to the given input?**
- **What are the data items exchanged by the target system and its external systems?**
- **What are the data items exchanged between two interacting processes?**
- **What are the data items that are stored in data stores?**

Once the flow of data from an element to another is identified, it is represented as a directed arrow and a name of the data element transmitted. Figure 4-8 show data flows among selected elements of DFD for car rental management system.

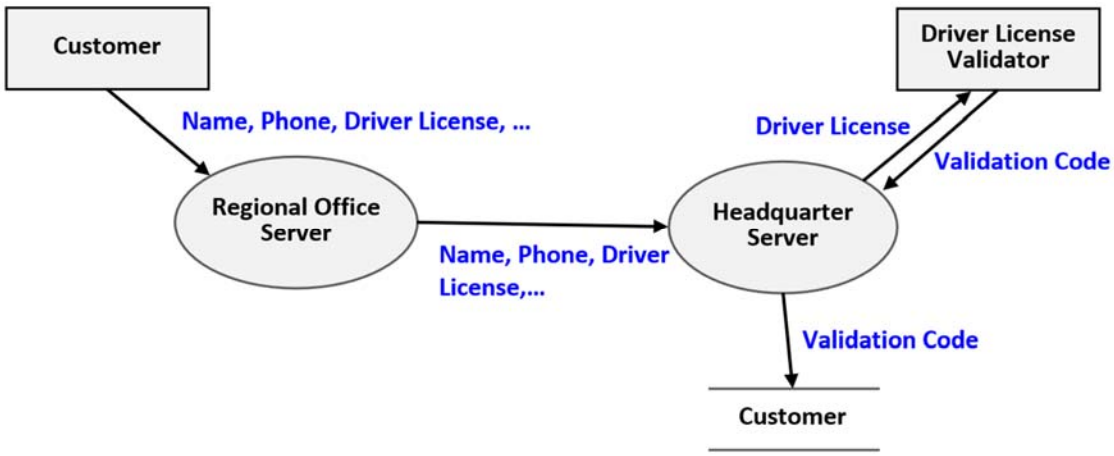

Figure 4-8. Data Flow of DFD for Car Rental Management System

A terminal, *Customer*, provides his or her identification information and a driver's license to the system node, *Regional Center Server*. Then, the server transmits the customer information to *Headquarter Server*, which sends the customer's driver license information to an external system *Driver License Validator*. And the external system sends the Validity Code to *Headquarter Server*, which stores the customer's information and the validation code in the data store *Customer*.

The context diagram with a level 0 DFD for Car Rental Management System is shown in Figure 4-9.

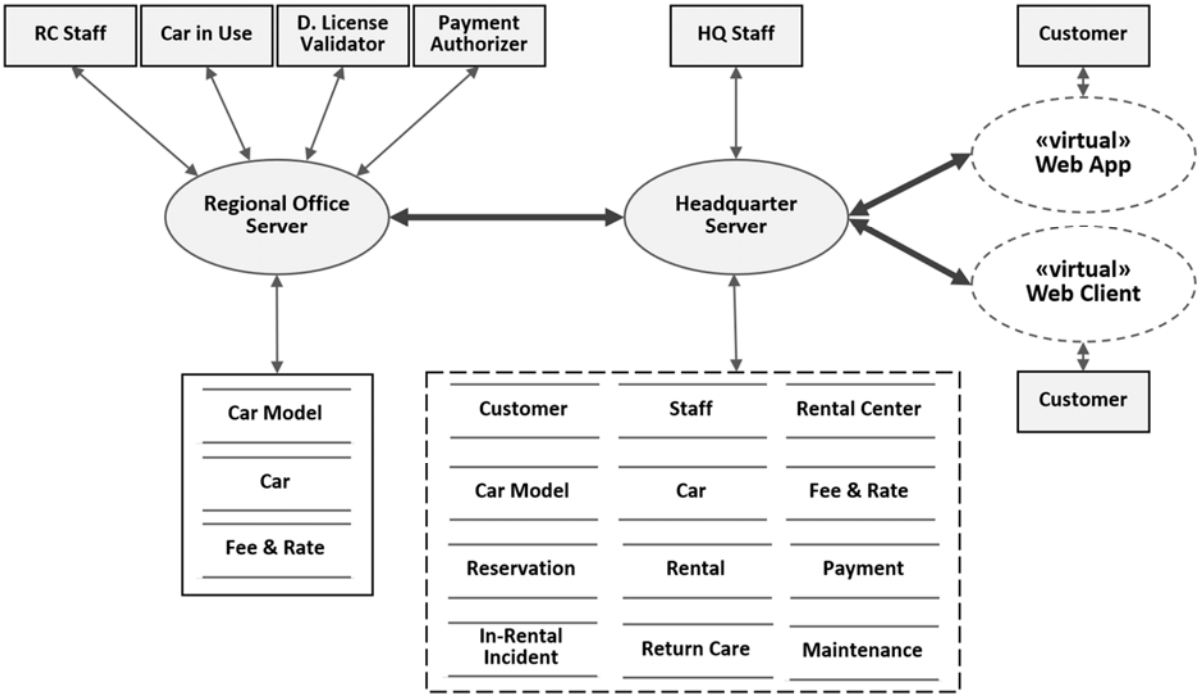

Figure 4-9. Resulting DFD for Car Rental Management System

The DFD shows the two system nodes, i.e., tiers, terminals of user types and external systems, data stores, and data flows for Car Rental Management. As shown in this example, the level 0 DFD effectively provides architects with the high understanding of the target system in terms of system nodes, elements in its boundary, persistent datasets, and the data flows among the elements. Note that the names of input and output on the data flow is omitted in this context diagram.

4.3. STEP 2. ACQUIRE FUNCTIONAL CONTEXT

Every system provides some functionality, which should be comprehended by architects. That is, an architect wishes to understand what functionality is delivered by the system and how the system interacts with users and external systems.

The functional context of the system can be well modeled using Use Case Diagram. The key elements of Use Case Diagram are actor, use case, and their relationships.

✦ Actor

An actor in use case diagram represents a type of role played by an entity that interacts with the system. An actor describes a role played by a human user, a connected hardware device, or an external system interacting with the system. Examples of actor for Car Rental System are shown in Figure 4-10.

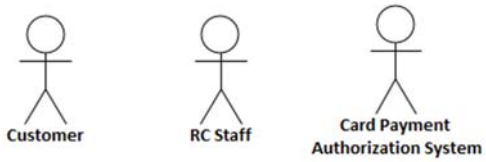

Figure 4-10. Examples of Actor

An actor can be active or passive. An active actor is to invoke use cases, and a passive actor is requested to perform a certain task by a use case. In Figure 4-10. *Customer* and *RC* Staff are active actors, whereas *Card Payment Authorization System* is a passive actor that is invoked by a use case.

✦ Use Case

A use case is a unit of functionality in use case diagram. It specifies a cohesive functionality provided by a target system. The name of use cases should be in verb form to reveal some functionality such as *Register Customer* and *Make Payment*. Some use cases for Car Rental Systems are shown in Figure 4-11.

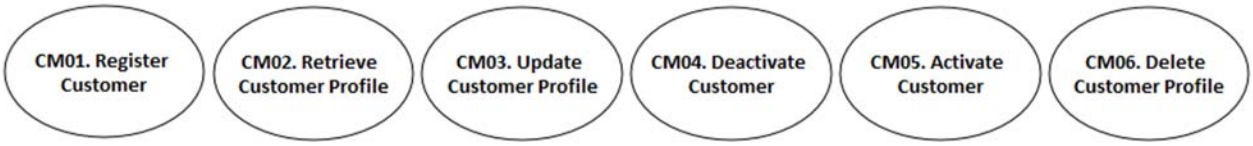

Figure 4-11. Examples of Use Case

✦ Relationship

A relationship in use case diagram can be an invocation, generalization, include, and extend.

◆ Invocation Relationship

An invocation relationship occurs between an actor and a use case. An active actor invokes the functionality of a use case, and a passive actor is invoked by a use case. Examples of invocation relationship are shown in Figure 4-12.

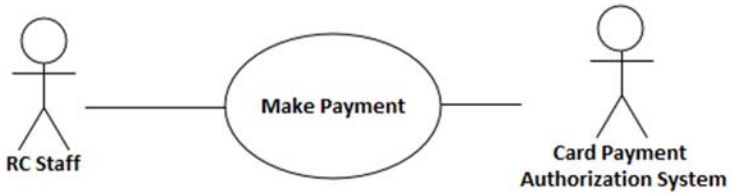

Figure 4-12. Examples of Invication Relationship

In the figure, an active actor *RC Staff* invokes the use case *Make Payment*, and a passive actor *Card Payment Authorization System* is invoked by the use case for an authorization of a credit card payment.

- **Generalization Relationship**

  Generalization in UML diagrams is a relationship in which an element is based on another element. It is used as a modeling construct in use case diagram and class diagram in UML. And it is depicted as a closed arrow-headed line.

  In use case diagram, generalization can occur in two forms: between actors and between use cases.

  - Generalization between Actors

    A generalization may occur between a base actor and its derived actors. A derived actor can invoke all the use cases that its base actor can invoke. In Figure 4-15 (a). *Base Actor* can invoke the use cases A and B whereas *Derived Actor* can invoke the use cases A and B as well as the use case C.

  - Generalization between Use Cases

    A generalization may occur between a base use case and its derived use cases. The base case is a functionality placeholder which can be specialized by derived use cases. In Figure 4-15 (b), the base use case *Make Payment* represent a functionality of making a payment but does not specify how the payment is made. Its derived use cases provide specific ways of making the payments: *Pay with Case*, *Pay with Credit Card*, and *Pay by Wire Transfer*. At runtime, the base use case itself is not executed; rather, one of its derived use case gets executed.

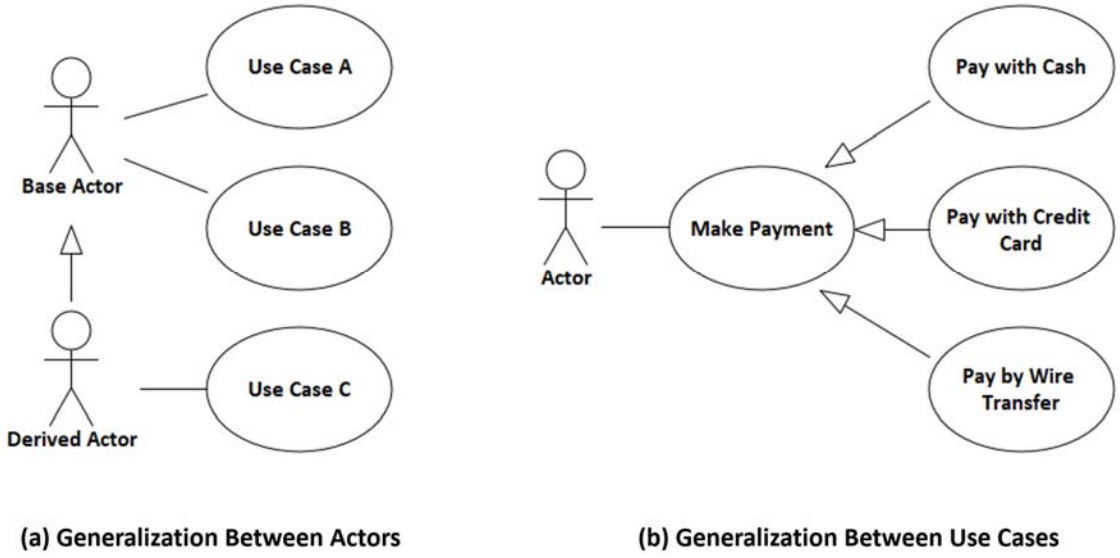

Figure 4-13. Generalization Relationships in Use Case Diagram

- **Include Relationship**

  Include Relationship is defined between a base use case and its included use case. When the base use case is invoked, it always invokes its included use case. In Figure 4-14, the base use *'RT01. Return Car'* always invokes the four included use cases: *Get Mileage, Get Gas Level, Check Car Condition,* and *Compute Additional Fee*. The order of invoking the 4 included use cases is not specified in use case diagram; rather, it is shown in a behavior diagram such as sequence diagram or activity diagram.

- **Extend Relationship**

  Extend Relationship is defined between a base use case and its extended use case. When the base use case is invoked, it may optionally invoke its included use case. In Figure 4-14, the base use *'RT05. Compute Additional Fee'* may or may not invoke its extended use case *'RT06. Make Additional Payment'*. If a rental car is returned later than the due date or the gas level is not full as specified in a contract, then the base use invokes the extended use case to make a payment for the addition fee.

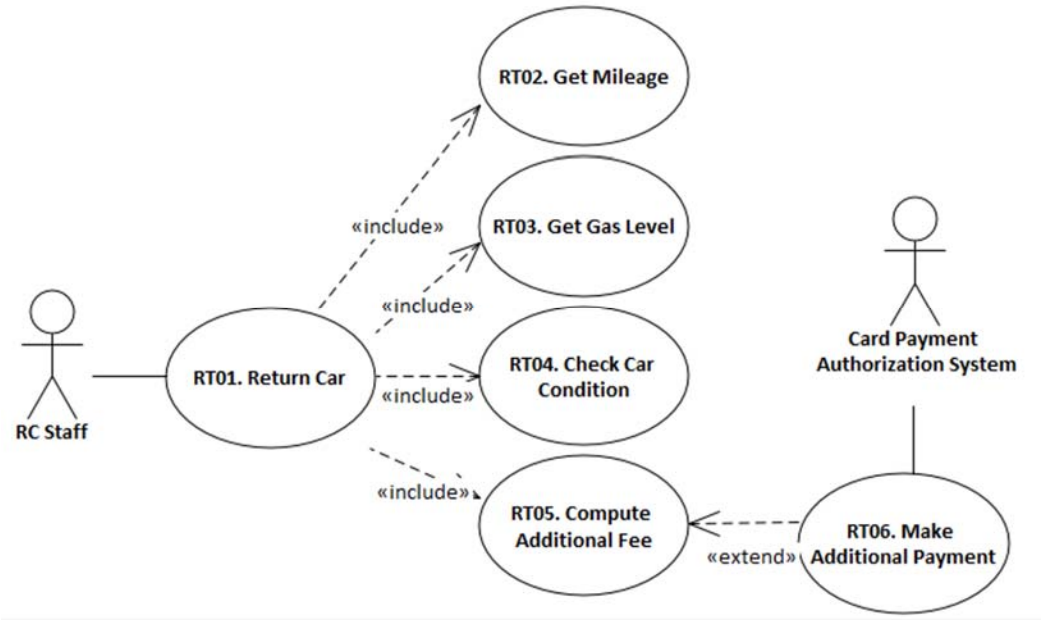

Figure 4-14. Include and Extend Relationships in Use Case Diagram

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

4.3.1. TASK 1. DEFINE ACTORS

This task is to identify the actors of the target system. An actor in use case diagram denotes a specific role played by users, external systems, or connected hardware devices. Identify both active and passive actors of the system.

4.3.2. TASK 2. DEFINE USE CASES

This task is to model the functionality provided by a target system as a number of use cases. Each use case represents a cohesive unit of functionality, and it interacts with actors.

✦ **Defining Functional Groups**

A functional group represents a cohesive set of functions in a target system, and the whole system functionality can be seen as a collection of functional groups. The decomposition of the system functionality into functional groups is useful in modeling the functionality of complex systems.

The functional groups of a system can effectively be derived from the functional requirements of a given SRS. Upon defining the functional groups, we then assign a 2 or 3 character-long identifier to each functional group. For Car Rental Management System, we can define the following set of functional groups.

- Customer management → CM
- Staff Management → SM
- Rental Center Registration → RC
- Inventory Management → IM
- Car Maintenance → CA
- Rental Fee Management → RF
- Reservation Management → RS
- Checkout Management → CH
- In-Rental Management → IR
- Return Management → RT
- Business Analytics → BA

✦ **Defining Use Cases for each Functional Group**

We now identify use cases for each functional group according to the given SRS. Each use case is given an identifier that consists of a functional group identifier and a sequence number. For the functional group of Reservation Management, we can use ‘RM’ as the identify.

An effective way of identifying use cases is to apply CRUD operations on the target dataset manipulated in each functional group. For Reservation Management, we can define the four use cases by considering the CRUD operations.

- RM01. Make Reservation
- RM02. Retrieve Reservation
- RM03. Modify Reservation
- RM04. Cancel Reservation

4.3.3. TASK 3. DEFINE INVOCATIONS

This task is to define the invocation relationships between actors and use cases. Active actors invoke their relevant use cases, and passive actors are invoked by use cases. The invocation relation is denoted as a solid line between an actor and its use case. A directed arrow can be used as an alternative to the solid line in order to indicate the direction of invocation.

Figure 4-15 shows the two types of actors for Smart Mirror System. *Calendar Agent* is a software agent-type actor, and it invokes the two use cases. Hence, it is an active actor. During the invocation of 'CR01. Acquire Calendar', it accesses a passive actor, *Calendar System*, to retrieve the calendar content of the current user.

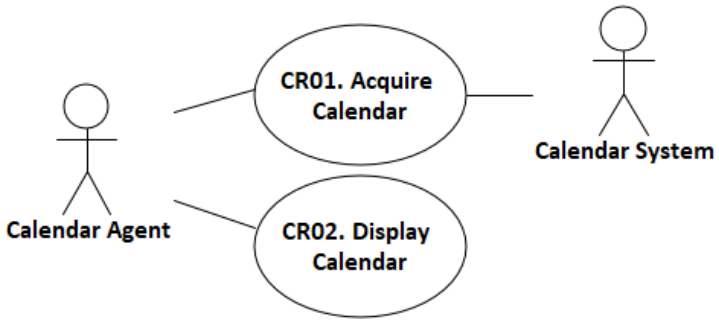

Figure 4-15. Invocation Relationships in Use Case Diagram

Note that all the actors identified in task 1 are connected with their relevant use case(s). The diagram shows that many of the use cases are invoked by software agent-type actors.

4.3.4. TASK 4. DEFINE RELATIONSHIPS

This task is to define the relationships between use cases and between actors. There are three types of relationships in UML.

✦ Define *Include* Relationship

Define an «include» relationship between a base use case and its included use case. When the base use case runs, it always invokes the included use case. For Smart Mirror System, the base use case ‘DM01. Acquire Image Frame’ is to a stream of image frames from a camera, and it invokes its included use case ‘DM02. Display Image’ displaying each frame in order to provide the same effect of conventional mirrors.

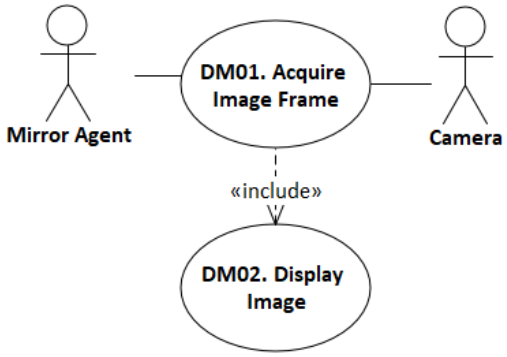

Figure 4-16. Include Relationship in Use Case Diagram

✦ Define *Extend* Relationship

Define an «extend» relationship between a base use case and its extended use case. When the base use case runs, it optionally invokes the included use case. For Smart Mirror System, the base use case ‘SM01. Detect People’ is to detect the presence of people on each image frame. If multiple people are found, then it invokes its extended use case ‘SM02. Handle Multiple People. The base use case invokes another extend use case ‘SM03. Recognize Face’ only when one target person is identified. If people are not found on the image, it does not invoke the use case ‘SM03.’

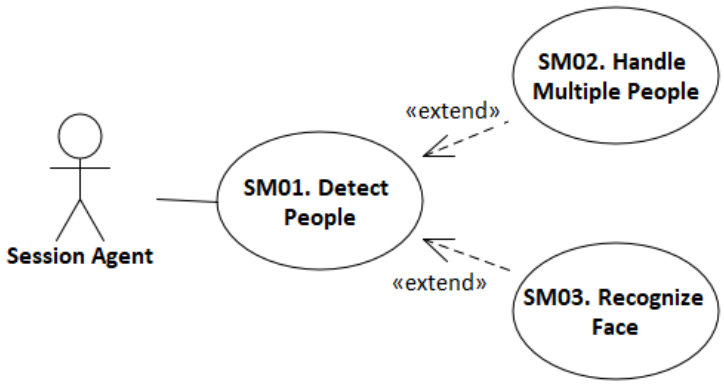

Figure 4-17. Extend Relationship in Use Case Diagram

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

4.4. STEP 3. ACQUIRE INFORMATION CONTEXT

This step is to acquire the information context of the target system by modeling the persistent datasets managed by the system. A persistent dataset must have some persistent data attributes as well as transient attributes. For example, the persistent data attributes of a dataset *Customer* can be customer name, address, identification number, phone, and etc.

The information context of the system can be well modeled using Class Diagram. The key elements of Class Diagram are class, relationships, and cardinalities on the relationship. A class is further defined with attributes and relevant methods. A relationship in Class Diagram can be *Dependency, Association, Aggregation, Composition,* and *Inheritance*.

- **Class**

  A class is a specification of attributes and permissible operations for a family of objects. A class can be seen as a group of objects. For Car Rental Management System, *Customer* is a class that models a group of individual customers. The name of a class is given in a noun form and should signify a group of objects. The classes for Car Rental Management System can be *Customer, Staff, Car, Reservation and Rental.*

- **Relationship → Dependency**

  Dependency establishes a temporal link between instances from two classes. As the weakest form of relationships between two classes, the dependency is often omitted in class diagrams.

- **Relationship → Association**

  Association establishes a persistent link between instances from two classes. This is the most common type of relationships in class diagram.

- **Relationship → Aggregation**

  Aggregation is a whole-part relationship between instances from two classes. That is, an instance of one class contains the instances of the other class as its attributes.

- **Relationship → Composition**

  Composition is a strong form of aggregation where the lifetime of part objects belongs to the whole object. When the whole object is decarded, its part objects are also discarded.

✦ **Relationship → Inheritance**

Inheritance is a generalization relationship between a general class and a specific class, i.e., superclass and subclass. A subclass inherits the attributes and methods of its superclass.

4.4.1. TASK 1. DEFINE PERSISTENT CLASSES

This task is to define classes for persistent datasets. In object-oriented paradigm, a persistent dataset is modeled as class, which is stored in a permanent storage or a database. A persistent object class contains one or more persistent attributes.

✦ **Derive Classes from SRS**

The main source for deriving persistent object classes is the given SRS, which specifies the system functionality and the information manipulated by the functionality. The SRS may not specify the whole datasets of the system, and hence architects may need to infer the persistent datasets from each functional requirement item. For example, the SRS of Car Rental Management System specifies a use case of 'Reserve Car'. Then, architects may infer persistent datasets of *Rental Car* as a physical object class and *Reservation* as a conceptual object class.

For Car Rental Management System, we can identify the persistent object classes as shown in Figure 4-18.

Figure 4-18. Classes for Car Rental Management System

Note that each class captures a persistent dataset that should be stored on a secondary storage typically in a form of database. For example, *Rental* contains persistent attributes of customer ID, car item ID, rental fee, deposit amount, insurance option, date checked, rental period, and date returned.

4.4.2. TASK 2. DEFINE RELATIONSHIPS BETWEEN CLASSES

This task is to define relationships between classes. There are 5 types of relationships in class diagram.

✦ Dependency

Dependency is a temporal relationship between two classes, and it establishes an interaction path for a short-term interaction between two objects. As the weakest form of relationships between two classes, the dependency is often omitted in class diagrams.

✦ Association

Association establishes a persistent link between instances from two classes. The links between instances are stored for persistency. This is the most common type of relationships in class diagram. As in Figure 4-19, the two classes are associated each other, meaning that an instance of *Rental* has a persistent link to an instance of *Customer*.

Figure 4-19. Association between Classes

✦ Aggregation

Aggregation is a whole-part relationship between instances from two classes. That is, an instance of one class contains the instances of the other class as its attributes. In Figure 4-20, an instance of *Rental* contains a persistent link to an instance of *Car Item*. That is, *Rental* cannot be instantiated without an insurance of *Car Item*.

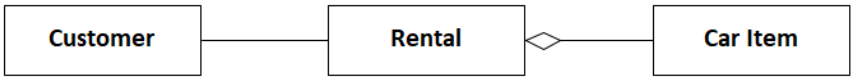

Figure 4-20. Aggregation between Classes

✦ Composition

Composition is a stronger form of aggregation where the lifetime of part objects belongs to the lifetime of the whole object. The destructor of a *Whole Object* class must include a logic to delete its part objects.

✦ Inheritance

Inheritance is a generalization relationship between a general class and a specific class, i.e., superclass and subclass. In Figure 4-21, the superclass *Person* generalizes its subclasses *Customer* and *Staff*, and each subclass inherits the property of the superclass *Person*.

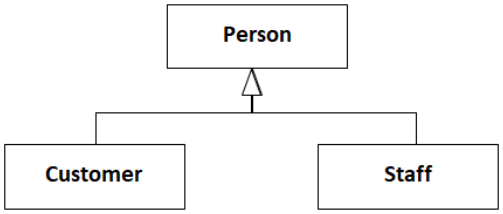

Figure 4-21. Inheritance between Classes

4.4.3. TASK 3. DEFINE CARDINALITY ON RELATIONSHIPS

This task is to define cardinalities on relationships between classes. UML defines a convention to denote various types of cardinalities. A cardinality between two classes specifies how many instances of another class can be linked to each instance of one class. Figure 4-22 shows a class diagram with relationships and their cardinalities for Car Rental Management System.

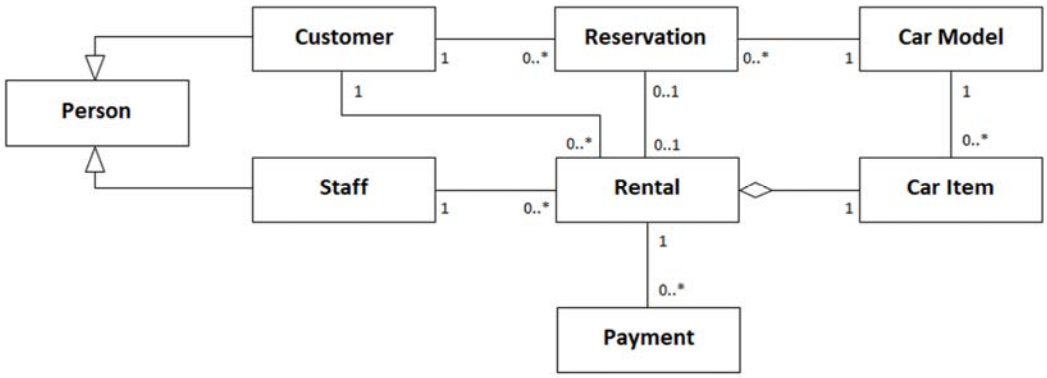

Figure 4-22. Cardinality on Relationships in Class Diagram

Customer has an association with *Reservation* and a cardinality is defined for each direction of the association. For the direction from *Customer* to *Reservation*, an instance of *Customer* may have zero or more instances of *Reservation*. For the other direction, an instance of *Reservation* must have exactly one instance of *Customer*. The cardinalities as well as relationships between classes should correctly be specified because they effectively define the structure of the implementation program codes.

The information context in a class diagram provides architects with a sufficient level of comprehension on the persistent datasets managed by the target system.

4.5. STEP 4. ACQUIRE BEHAVIORAL CONTEXT

This step is to acquire the behavioral context of the target system and represent the context. The behavior context is a high-level description of the system behavior at runtime, and the runtime behavior is defined by the valid control flows of the system.

✦ Functionality vs. Behavior

The *functionality* of a system is the service provided by the system, i.e., the responsibility of the system. The functionality can be modeled with a use case diagram.

The *behavior* of a system is the dynamic behavior of the system, i.e., the sequences of execution at runtime. The behavior can be specified in behavioral diagrams such as activity diagram, sequence diagram, state machine Diagram, and timing diagram.

The behavior of a whole system or a sub-system can be modeled with Activity Diagram. The key elements of Activity Diagram include action, activity, decision, control flow, fork and join, partition, object node, and data store.

✦ Action

An action represents a single atomic operation, i.e., either completed or aborted. The action represents a fine-grained operation.

✦ Activity

An operation represents a sequence of behavior, which is specified as a workflow among actions and other activities. Hence, an activity is a larger-grained unit than an action.

✦ Initial Node

The initial node represents where the system flow starts.

✦ Final Node

A final node represents an end of the system flow. It can be an Activity Final Node or a Flow Final Node.

✦ Control Flow

A Control flow represents the flow of control from one action to the next. It is not to specify object flows or data flows.

✦ **Decision**

A decision node represents a choice among multiple outgoing flows. Each outgoing flow has a guard condition, and the guard conditions must be mutually exclusive.

✦ **Merge**

A merge node represents that multiple incoming flows are merged into an outgoing flow.

✦ **Fork and Join**

Fork and Join nodes represent the parallel processing of multiple reads. The fork node represents the start of multiple threads, and a join node represents the end of running the parallel threads.

4.5.1. TASK 1. ALLOCATE FUNCTIONAL GROUPS ONTO TIERS

This task is to allocate the functional groups of a system onto tiers if the system has multiple tiers. That is, we determine which tier of the system should be responsible for providing each functional group. If a functional group is allocated to more than one tier, the functionality of a functional group on one tier may slightly be different from the functionality on another tier.

We specify the allocation of functional group in a table of 'Functional Group Allocation' as shown in Table 4-1.

Table 4-1. Functionality Group Allocation

	Tier #1	*Tier #2*	*Tier #3*
Functional Group #1	✓		
Functional Group #2		✓	✓
Functional Group #3	✓	✓	
…	…	…	….

The tiers of the system are entered on the first row, and the functional groups are entered on the first column. Place a check mark, ✓, in an entry of the table if the functional group is allocated to the tier.

For Car Rental Management System, the boundary context in the DFD specifies 2 physical nodes as shown in Figure 4-5.

As discussed earlier, *Web App* and *Web Client* are virtual tiers, invoking the functionality of *Headquarter Server* using a web browser. Accordingly, there is no need to allocate the functionality on virtual nodes. Table 4-2 shows the allocation of functional groups onto the tiers.

Table 4-2. Allocation of Functionality Groups for Car Rental Management System

	Regional Office Server	*Headquarters Server*
Customer management	✓	✓ (For Clients)
Staff Management	✓	✓
Rental Center Registration	✓	✓
Inventory Management	✓	✓
Car Maintenance	✓	
Rental Fee Management	✓	✓
Reservation Management	✓	✓ (For Clients)
Checkout Management	✓	✓ (For Clients)
In-Rental Management	✓	✓ (For Clients)
Return Management	✓	
Business Analytics		✓

As shown in the table, each functional group of the system is allocated to its relevant node(s). Some functional groups such as *Customer Management* are allocated to both nodes. The functionality of *Business Analytics* is allocated only to *Headquarter Server* node.

4.5.2. TASK 2. DEFINE INVOCATION PATTERNS

An invocation pattern is a specific type of invoking a given function such as 'invoking a functionality explicitly by a user', 'invoking a functionality upon arrival of an event, and 'repeat invoking a functionality in a closed loop'. The invocation patterns become the basis for defining the runtime behavior of the system using behavior diagrams such as activity diagram.

This task is to define the patterns of invoking the allocated functional groups. That is, we determine how each functional group should be invoked, and reflect the specified invocation patterns in drawing an activity diagram representing the whole system behavior.

The common patterns of invoking system functionalities are shown in Figure 4-23.

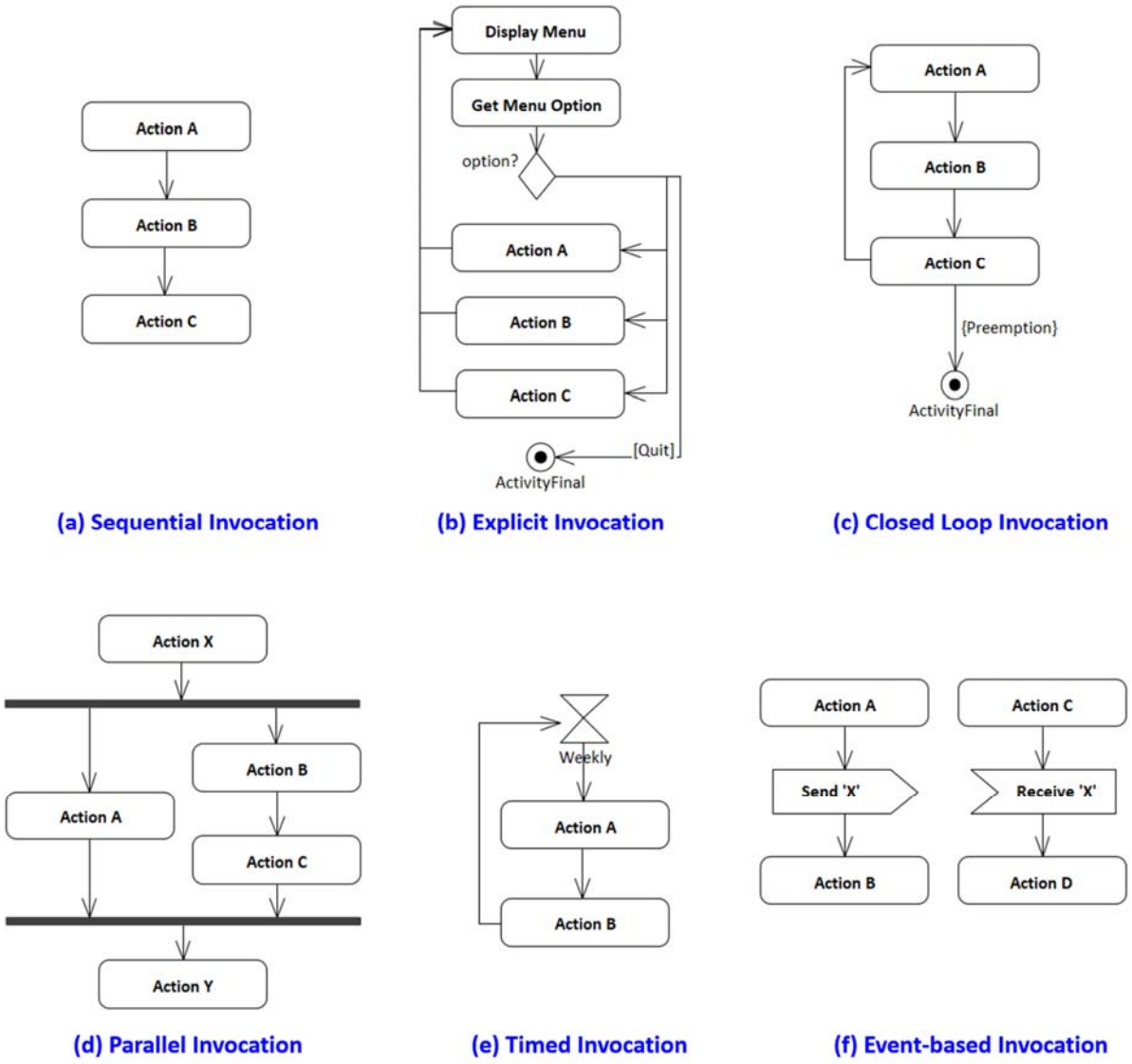

Figure 4-23. Invocation Patterns shown in Activity Diagram

✦ Sequential Invocation

This invocation pattern is applied when a system needs to invoke actions or activities in sequence without any branching, as shown in (a). A sequential invocation may appear in the

main process of a system, but it often occurs within other invocation patterns such as closed loop pattern or parallel invocation.

✦ Explicit Invocation

This invocation pattern is applied when a user chooses an invocation option from a given set of choices, as shown in (b). With this pattern, a system menu is given, a user chooses an option from the menu, and the system runs the chosen action or activity. Note that the system runs the selected functionality in a synchronous manner. That is, when a system runs a selected option, all other options in the menu become unavailable until the current option finishes running.

✦ Closed Loop Invocation

This invocation pattern is applied when a system needs to repeat running some functionality without being interrupted or preempted. This pattern is often applied to developing embedded systems that include self-managing capability. For example, autonomous vehicles have limited interventions from drivers; instead, the vehicles operated in a self-management manner. Hence, closed loop invocation pattern be used to model this behavior.

✦ Parallel Invocation

This invocation pattern is applied when a system needs to run multiple threads of control in parallel. Each thread in the parallel processing runs independently from others. Once all the threads finish their tasks, they are merged.

✦ Timed Invocation

This invocation pattern is applied to model the functionality that should be invocated with some time constrains. The typical types of time constraints are the followings.

- **Periodical Invocation**

  A functionality needs to start running every week.

- **During a Time Period**

  A functionality needs between 9am to 10am.

- **After Elapsed Time Period**

  A functionality needs to be started after the given elapsed time, such as “After 10 Minutes.”

✦ Event-based Invocation

This invocation pattern is applied to model a functionality that is invoked with events. This type of systems consists of event emitters and event handlers, and the systems runs an appropriate event hander upon the arrival of an event. The source of an event and the handler

of the event should be different parties, such as different threads, processes, or tiers. The event emitters and event handlers run asynchronously.

We now specify the invocation patterns of the identified functional groups. That is, each function group is associated with one or more invocation patterns. The association of invocation patterns can be specified in a Table of Invocation Patterns. Table 4-3 shows the invocation patterns of functional groups for Car Rental Management System.

Table 4-3. Invocation Patterns of Functionality for Car Rental Management System

	Regional Office Server	*Headquarter Server*
Customer management	Explicit	Explicit
Staff Management	Explicit	Explicit
Rental Center Registration	Explicit	Explicit
Inventory Management	Explicit	Explicit
Car Maintenance	Explicit	
Rental Fee Management	Explicit	Explicit
Reservation Management	Explicit	Explicit
Checkout Management	Explicit	Explicit
In-Rental Management	Event-based, Explicit	Event-based, Explicit
Return Management	Explicit	
Business Analytics		Explicit

Each functional group is specified with appropriate invocation patterns. All the functional groups except 'In-Rental Management' reveal the explicit invocation pattern. For 'In-Rental Management', the system may monitor the locations of active rental cars using GPS. When the location of a rental car is found to be outside of the allowed driving regions, an event is generated and sent to its event hander. Therefore, its invocation is made in event-driven pattern.

4.5.3. TASK 3. DEFINE THE CONTROL FLOWS OF THE SYSTEM

This task is to define the control flow of the target system by drawing Activity Diagrams. An activity diagram must be defined for each tier of the system because the software systems on different tiers provide different functionalities and behaviors.

Once the invocation patterns are defined for all the functional groups, then we can systematically define their control flows in an activity diagram. The typical order of drawing an activity diagram is given here.

✦ **Defining Start and End Nodes**

Define a start node and one or more end nodes.

✦ **Defining Initialization Behavior**

Software systems often require an initialization of the main screen, the network connection, devices connected and data attributes. Define the initialization behavior with actions and activities at the beginning part of the activity diagram.

✦ **Defining Threads**

If the system has a parallel processing pattern, define threads using fork and join constructs.

✦ **Defining Control Flow with Actions and Activities According to Behavior Patterns**

Define control flows with actions and activities by considering the behavior pattens of the target system.

The activity diagrams showing the control flows of the two tiers are shown in Figure 4-24.

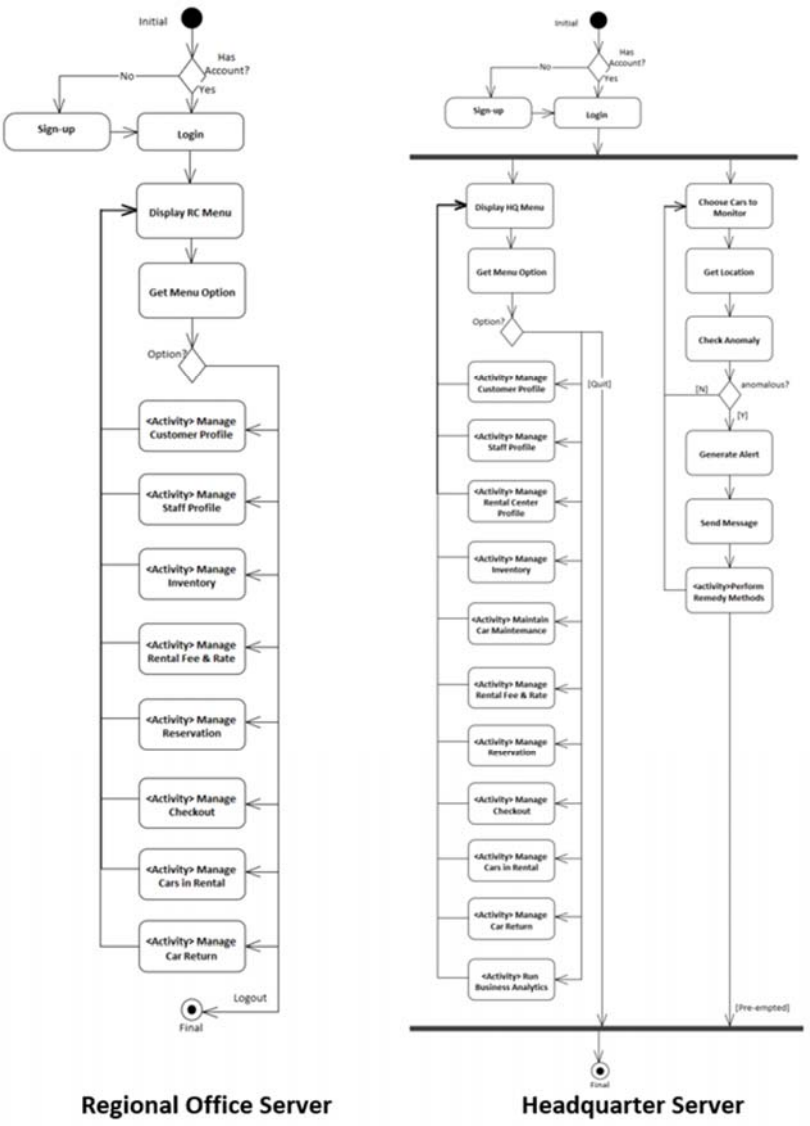

Figure 4-24. Control Flows of the Two Tiers

The control flow of Regional Office Server is mainly invocation-based. The control of Headquarter Server consists of two parallel threads. The left thread depicts the explicit invocation-based control flow, and the right thread depicts a closed loop control flow for monitoring the checked cars and checking any anomaly on the cars.

4.6. CHECKLIST FOR SYSTEM CONTEXT ANALYSIS

The checklist for software design is a list of criteria and guidelines for evaluating the quality of the design. Therefore, a checklist is specific to the type of software design artifact. The system context model can be validated with the following checklists.

4.6.1. CHECKLIST FOR BOUNDARY CONTEXT

The following checklist can be used to validate the boundary context of the system.

✦ **For Process**

- Do the processes in the DFD correspond to the nodes, i.e., tiers, specified or implied in the SRS of the target system?
- Does each tier correspond to the sub-system, which is highly independent and self-contained?
- Is the runtime overhead of network-based interaction among tiers minimal? If not, is the runtime overhead inevitable for its benefits?
- Is each tier deployed and maintained independently from other tiers?
- Are the development and operation costs for deploying each node specified by a process justifiable with the benefits provided?

✦ **For Terminal**

- For each process, are all the terminals interacting with the process identified?
- Does the DFD include terminals of all user types?
- Does the DFD include terminals of connected hardware devices, sensors and actuators?
- Does the DFD include terminals of all interacting external systems?
- Does the DFD include terminals of all interacting cloud services and microservices?

✦ **For Data Store**

- For each tier specified by a process, does the DFD identify and specify data stores for all the relevant persistent data maintained by the tier?
- Does each data store represent data elements that are persistent, not transient?
- Does each data store that is redundantly specified in multiple processes deliver significant benefits such as increased data availability?
- Check if each data store that is redundantly specified in multiple processes does not result in un-justifiable cost and risk.

✦ **For Data Flow**

- Check if each data flow is specified with one or more data items transferred?
- Check if the data flows are only specified between terminal and process, between process

and data store, and between processes.

- Check if the DFD does not include invalid data flows such as between two terminals, between two datasets, and between terminal and datastore.

4.6.2. CHECKLIST FOR FUNCTIONAL CONTEXT

✦ **For Functional Groups**

- Do the functional groups correspond to the functional categories specified in the SRS?
- If not completely corresponding, does the set of functional groups represent a logical and valid refinement over the functional categories specified in the SRS?
- Is each functional group defined to be mutually exclusive from any other functional group? That is, there is no functional redundancy between every pair of two functional groups.

✦ **For Actors**

- Does use case diagram include all its relevant active actors.
- Does use case diagram include all its relevant passive actors of connected hardware devices, external systems, and microservices?
- Does use case diagram include all its relevant actors of software agent type? Actors of software agent type are often utilized to enable automatic and autonomous control.

✦ **For Use Cases**

- Does the use case diagram include use cases of CRUD-related operations for manipulating relevant data elements?
- Does the use case diagram include the system intrinsic use cases beyond CRUD-type use cases?
- Is each use case specified with a use case ID number and a meaningful name? Often, the use case ID number starts with a prefix that represents a functional group such as CP ‘for Customer Profile Management’.
- Is the name of each use case specified in a verb form and in a meaning way?
- Are the granularities of use cases sufficiently consistent?
- Are the use cases mutually exclusive for their functionality? That is, check if the functionality specified in a use case is unique and distinct from those of other use cases.

✦ **For Invocations**

- Does each active actor such as user and customer invoke its relevant and appropriate use

cases?

- Does each passive actor such as an external system or a microservice is invoked by some use case(s)?

✦ **For Relationships**

- Check if the only relationship allowed between actors is generalization.
- For generalization, check if a generalized use case is specialized into multiple derived use cases.
- For «include» relationship, check if the base use case must always invoke its included use cases?
- For «extend» relationship, check if the base use case may or may not invoke its extended use cases?
- Check if the «include» relationship is misused to specify the invocation order, i.e., control flow, among use cases. The «include» relationship is not intended to specify a control flow: rather, it specifies the inclusion of a sub-functionality in a whole functionality.

4.6.3. CHECKLIST FOR INFORMATION CONTEXT

✦ **For Persistency of Classes**

- Check if each class in the class diagram corresponds to some persistent data manipulated by the system. That is, a class should not represent a transient and temporal dataset.
- Check if the granularity of each class is defined logically based on the principle of data cohesiveness and single responsibility.

✦ **For Physical and Logical Object Classes**

- Does the class diagram capture all the physical data objects as classes, such as Customer and Vehicle for Car Rental Management System?
- Does the class diagram capture all the logical data objects as classes that capture the results of business transactions, such as Reservation and Rental.
- Check if the class diagram includes logical objects that capture the session-related information? A session can be user session, system operation-intrinsic session, and system session.

✦ **For Relationships**

- Does the class diagram include all the necessary relationships of association, aggregation, composition, and inheritance? The dependency relationship can be omitted in the diagram.

- For two classes A and B with association, does each instance of A need to maintain links, i.e., permanent dependencies, with instances of B? And, vice versa?
- For aggregation relationship, check if every instance of the whole object class can only be created with its part objects. That is, a whole object cannot be created without part objects.
- For composition relationship, check if every instance of the whole object class can only be created with its part objects and the part objects must be deleted when the whole object gets deleted. That is, check if the lifetime of the whole object is shared with part objects.
- For Inheritance relationship, is the ISA or AKO relationship maintained between a superclass and its subclass? That is, check if a subclass is defined to be a subtype of its superclass.

✦ **For Cardinalities**

- Does the class diagram include valid cardinalities on every relationship except inheritance?
- Check if the use of the value 1 as cardinality can be 0..1 to allow the optional link.
- When two classes are defined with multiple occurrences of cardinality on both ends, check if an association class should better be defined between them.

4.6.4. CHECKLIST FOR BEHAVIOR CONTEXT

✦ **Number of Activity Diagrams**

- Is the number of Activity Diagrams same as the number of tiers?

✦ **Partitioning of Functionality over Tiers**

- Is each functional group allocated to one or more tiers?

✦ **Consistency with Functional Context in Use Case Diagram**

- Do the actions and activities in the Activity Diagram correspond to the use cases in the use case diagrams?
- Are there any use cases that are not reflected in the Activity Diagram?
- Are there any actions and activities that have not corresponding use case(s)?

✦ **For Invocation Patterns**

- Check if each functional group is assigned with one or more interaction patterns.

✦ **Drawing Activity Diagram**

- Is the number of outgoing threads at fork same as the number of incoming threads at join?

- Are there any interactions among parallel threads? There must be no interactions between threads since they should run independently from other threads to truly be parallel.
- Is a ‘Receive Event’ defined for each ‘Send Event’?
- Are the interactions between tiers defined with appropriate interaction schemes such as event-based interaction?
- For the control flow of closed loops, is the termination of the flow defined with a preemption?