System Analysis

DR. DRAGUNOV

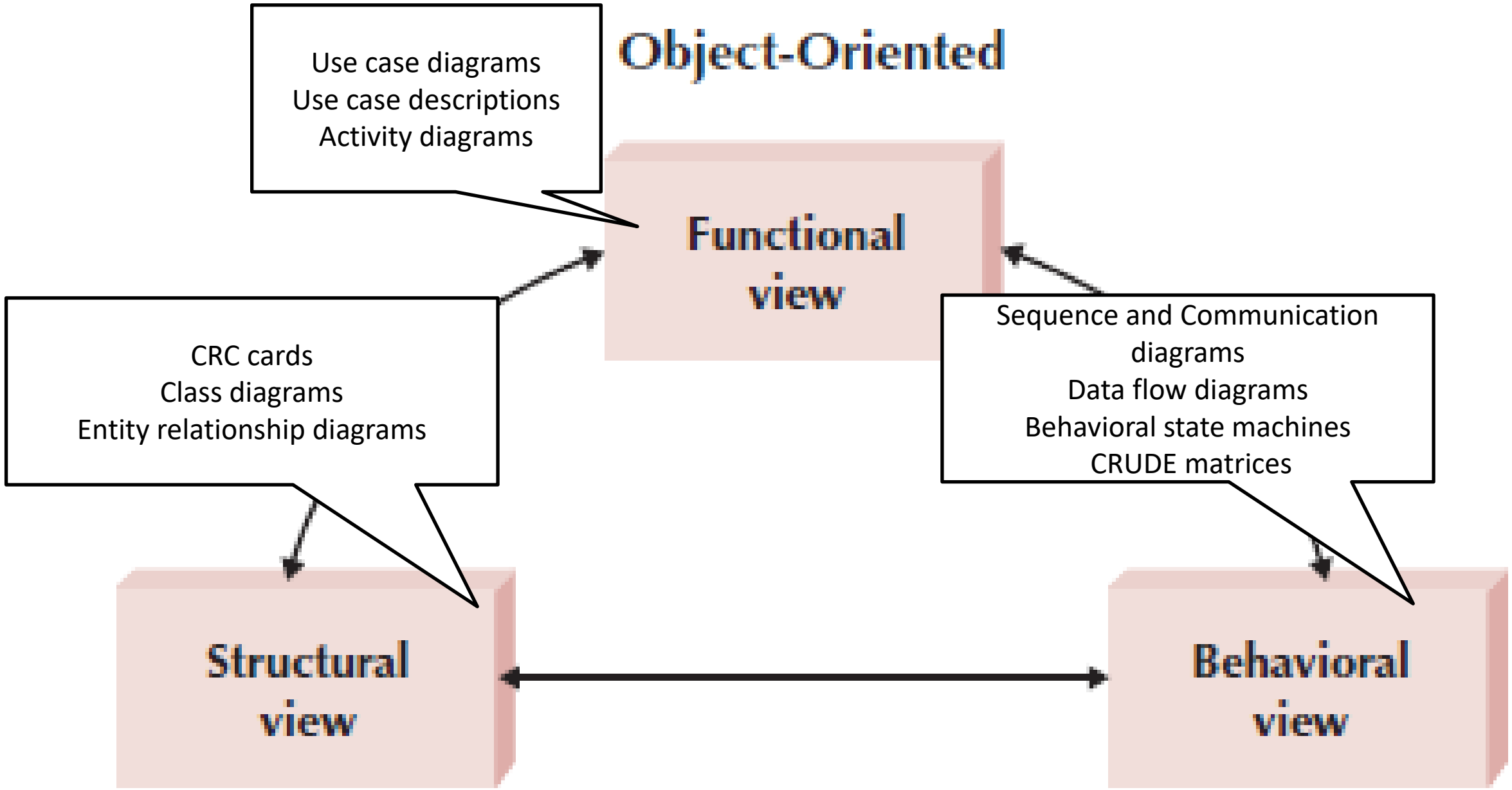
BEHAVIORAL MODELING

5 steps for success (by A. Dragunov)

1. Understand what do they really need (requirements)
2. Understand all business process (use-case modeling)
3. Develop Structure (Structure modeling)
4. Understand behavior (Behavior modeling)
5. Develop it and explain how to use it. (Development, write a program)

OBJECTIVES

- Understand the rules and style guidelines for **sequence and communication diagrams, data flow diagrams and behavioral state machines**
- Understand the processes used to create sequence and communication diagrams, data flow diagrams, behavioral state machines, and CRUDE matrices
- Be able to create sequence and communication diagrams, data flow diagrams, behavioral state machines, and CRUDE matrices
- **Understand the relationship between the behavioral models and the structural and functional models**


Object-Oriented
Use case diagrams
Use case descriptions
Activity diagrams
Functional view
CRC cards
Class diagrams
Entity relationship diagrams
Sequence and Communication diagrams
Data flow diagrams
Behavioral state machines
CRUDE matrices
Structural view
Behavioral view

BEHAVIORAL MODELING

Behavioral models describe the **internal dynamic aspects of an information system** that supports the business processes in an organization.

During analysis, **behavioral models describe what the internal logic of the processes is** without specifying how the processes are to be implemented.

We describe three Unified Modeling Language (UML) diagrams that are used in behavioral modeling:

- **sequence diagrams**
- **communication diagrams**
- **behavioral state machines**
- **data flow diagrams**

and

- **CRUDE** (create, read, update, delete, execute) matrices.

INTRODUCTION

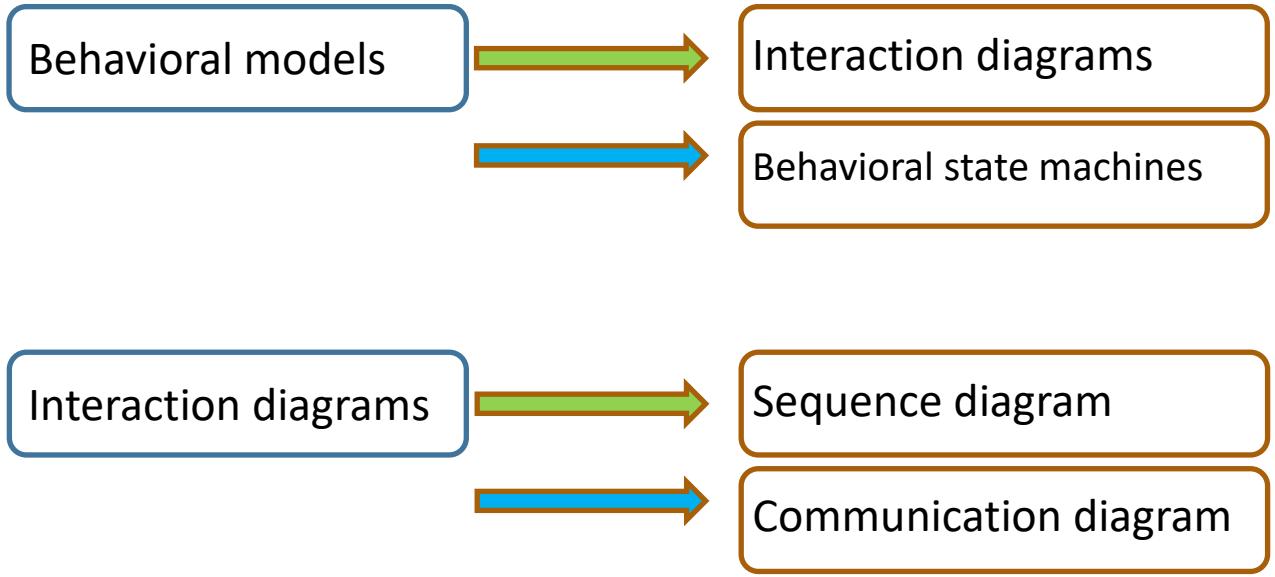
Systems analysts use:

- **Functional models** to describe the external behavioral view of an information system.
- **Structural models** to depict the static view of an information system.

Now, we discuss how analysts use ***behavioral models*** to represent the internal behavior or dynamic view of an information system.

There are two types of behavioral models.

First, there are behavioral models **used to represent the underlying details** of a business process portrayed by a use-case model.



Practically speaking, interaction diagrams allow the analyst to model the distribution of the behavior of the system over the actors and objects in the system.

Second, a behavioral model is **used to represent the changes** that occur in the **underlying data**. UML uses behavioral **state machines** for this.

BEHAVIORAL MODELS

Primary purposes of behavioral models is to show **how the underlying objects in a problem domain will work together to form a *collaboration* to support each of the *use cases*.**

Creating behavioral models is an iterative process that iterates not only over the individual behavioral models but also over the functional and structural models.

INTERACTION DIAGRAMS

One of the primary differences between class diagrams and interaction diagrams, besides the obvious difference that one describes structure and the other behavior, is that the modeling focus on a class diagram is at the class level, whereas the interaction diagrams focus on the object level.

Objects, Operations, and Messages

An object is an instantiation of a *class,* that is, an actual person, place, or thing about which we want to capture information.

Each object has *attributes* that describe information about the object, such as a patient's name, birth date, address, and phone number.

Each object also has *behaviors*. At this point in the development of the evolving system, the behaviors are described by *operations.* An operation is nothing more than an action that an object can perform. For example, an appointment object can probably schedule a new appointment.

Each object also can send and receive messages. *Messages* are information sent to objects to tell an object to execute one of its behaviors. Essentially, a message is a function or procedure call from one object to another object.

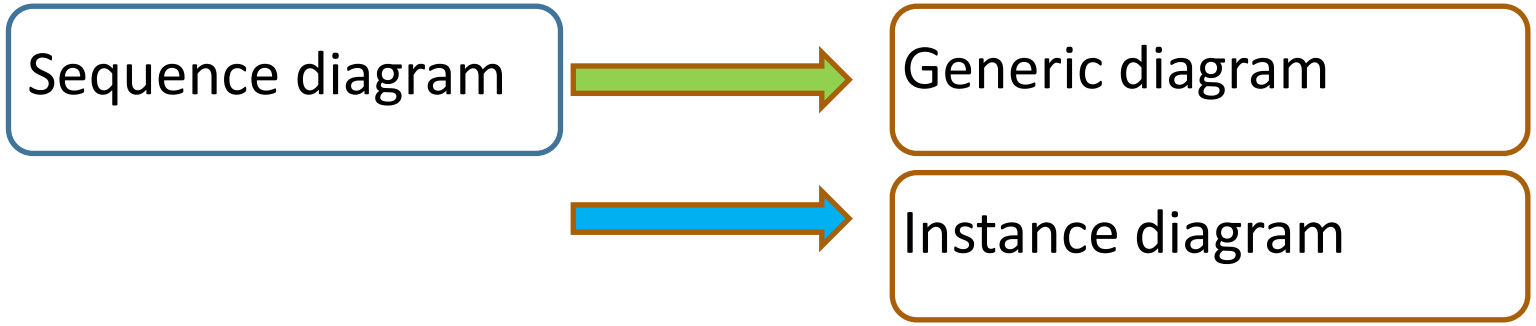
Sequence Diagrams

Sequence diagrams are one of two types of interaction diagrams. They illustrate the objects that participate in a use case and the messages that pass between them over time for *one* use case.

A **sequence diagram is a *dynamic model*** that shows the explicit sequence of messages that are passed between objects in a defined interaction.

Because sequence diagrams emphasize the time-based ordering of the activity that takes place among a set of objects, they are very helpful for understanding real-time specifications and complex use cases.

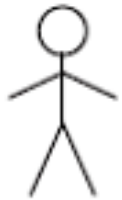
Generic & instance sequence diagrams

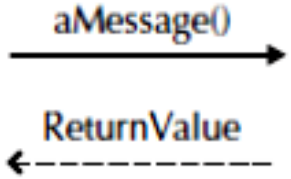


Generic sequence diagrams shows all possible scenarios for a use case

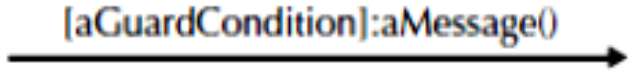
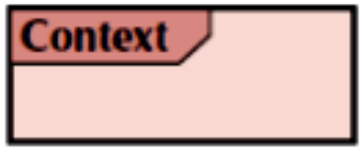
Instance sequence diagrams depicts a single *scenario* within the use case.

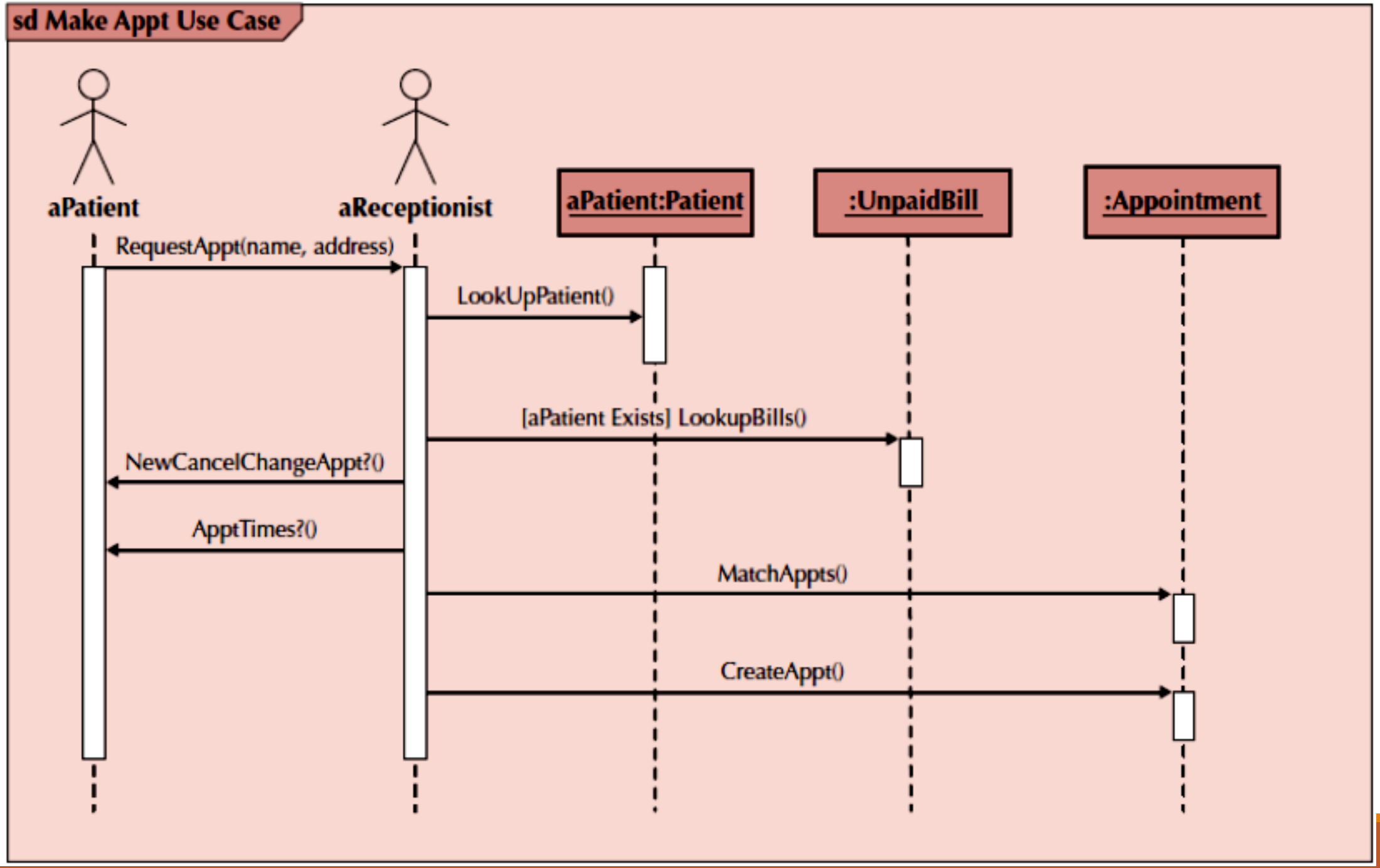
Elements of a Sequence Diagram

Term and Definition	Symbol
An actor: ■ Is a person or system that derives benefit from and is external to the system. ■ Participates in a sequence by sending and/or receiving messages. ■ Is placed across the top of the diagram. ■ Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative).	 **anActor** **<<actor>>** **anActor**
An object: ■ Participates in a sequence by sending and/or receiving messages. ■ Is placed across the top of the diagram.	**anObject : aClass**

A lifeline: ■ Denotes the life of an object during a sequence. ■ Contains an X at the point at which the class no longer interacts.	
An execution occurrence: ■ Is a long narrow rectangle placed atop a lifeline. ■ Denotes when an object is sending or receiving messages.	
A message: ■ Conveys information from one object to another one. ■ A operation call is labeled with the message being sent and a solid arrow, whereas a return is labeled with the value being returned and shown as a dashed arrow.	

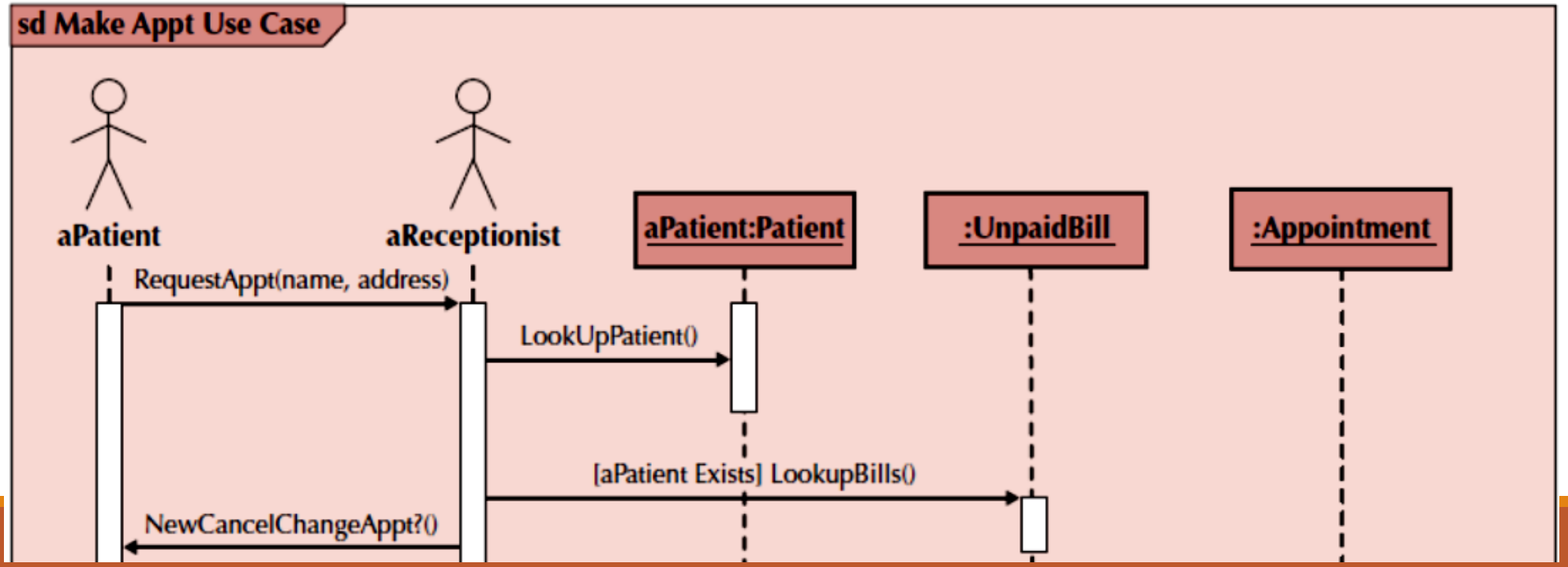
A thin rectangular box, called the *execution occurrence,* is overlaid onto the lifeline to show when the classes are sending and receiving messages. A message is a communication between objects that conveys information with the expectation that activity will ensue. Many different types of messages can be portrayed on a sequence diagram.

A guard condition: ■ Represents a test that must be met for the message to be sent.	
For object destruction: ■ An X is placed at the end of an object's lifeline to show that it is going out of existence.	**X**
A frame: ■ Indicates the context of the sequence diagram.	


sd Make Appt Use Case
aPatient
aReceptionist
aPatient:Patient
:UnpaidBill
:Appointment
RequestAppt(name, address)
LookUpPatient()
[aPatient Exists] LookupBills()
NewCancelChangeAppt?()
ApptTimes?()
MatchAppts()
CreateAppt()

- **Try to have the messages not only in a top to bottom order but also, when possible, in a left to right order.**

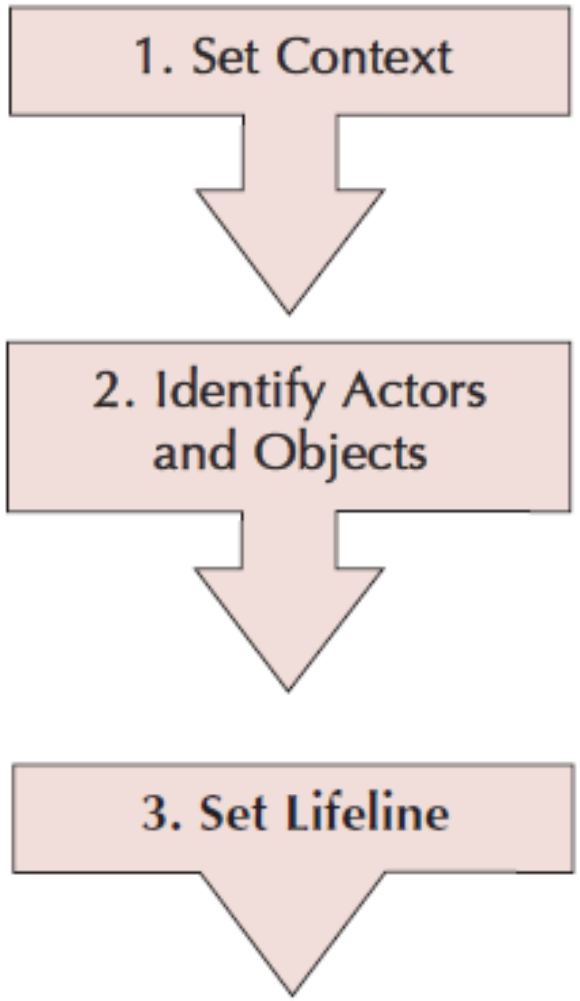
- **If an actor and an object conceptually represent the same idea, one inside of the software and the other outside, label them with the same name.** In fact, this implies that they exist in both the use-case diagram (as an actor) and in the class diagram (as a class).



- **The initiator of the scenario—actor or object—should be the drawn as the farthest left item in the diagram.** This guideline is essentially a specialization of the first guideline. In this case, it relates specifically to the actor or object that triggers the scenario.
- **When there are multiple objects of the same type, be sure to include a name for the object in addition to the class of the object.**

- **Show return values only when they are not obvious.** Showing all of the returns tends to make a sequence diagram more complex and potentially difficult to comprehend. In many cases, less is more. Only show the returns that actually add information for the reader of the diagram.
- **Justify message names and return values near the arrowhead of the message and return arrows, respectively.** This makes it much easier to interpret the messages and their return values.

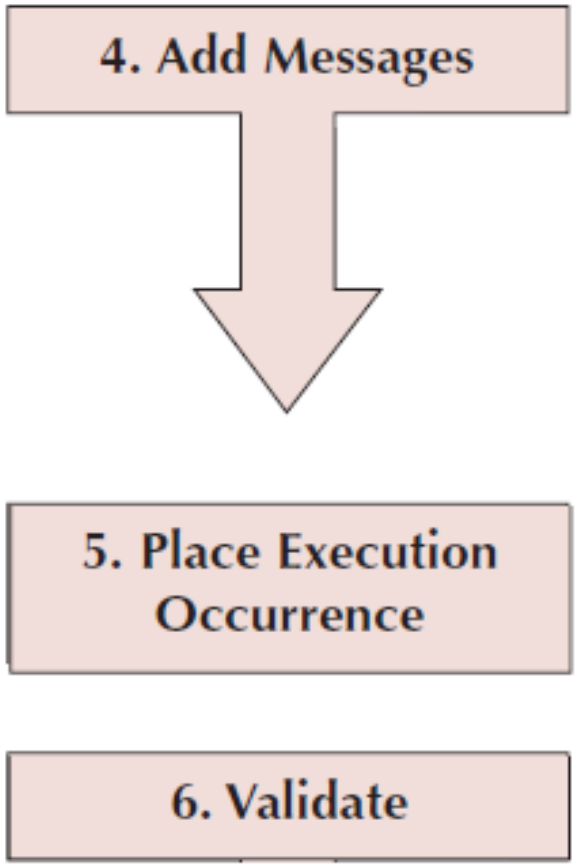
Creating a Sequence Diagram



The first step in the process is to determine the context of the sequence diagram. The context of the diagram can be a system, a use case, or a scenario of a use case.

The second step is to identify the actors and objects that participate in the sequence being modeled—that is, the actors and objects that interact with each other during the use-case scenario.

The third step is to set the lifeline for each object. To do this, you need to draw a vertical dotted line below each class to represent the class's existence during the sequence. An X should be placed below the object at the point on the lifeline where the object goes out of existence.



The fourth step is to add the messages to the diagram. This is done by drawing arrows to represent the messages being passed from object to object, with the arrow pointing in the message's transmission direction.

The fifth step is to place the execution occurrence on each object's lifeline by drawing a narrow rectangle box over the lifelines to represent when the classes are sending and receiving messages.

The sixth and final step is to validate the sequence diagram. The purpose of this step is to guarantee that the sequence diagram completely represents the underlying process. This is done by guaranteeing that the diagram depicts all the steps in the process.

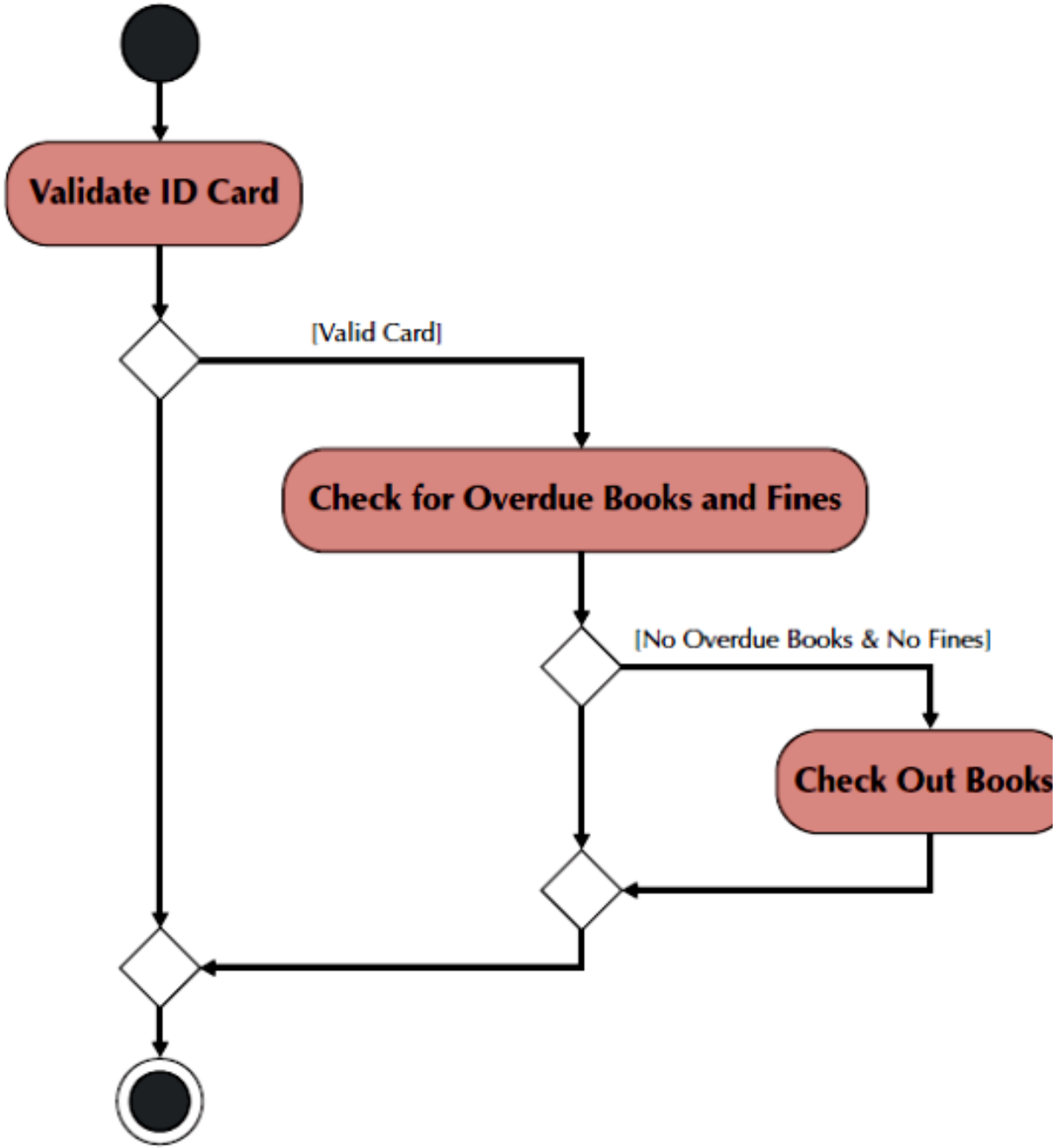
Example

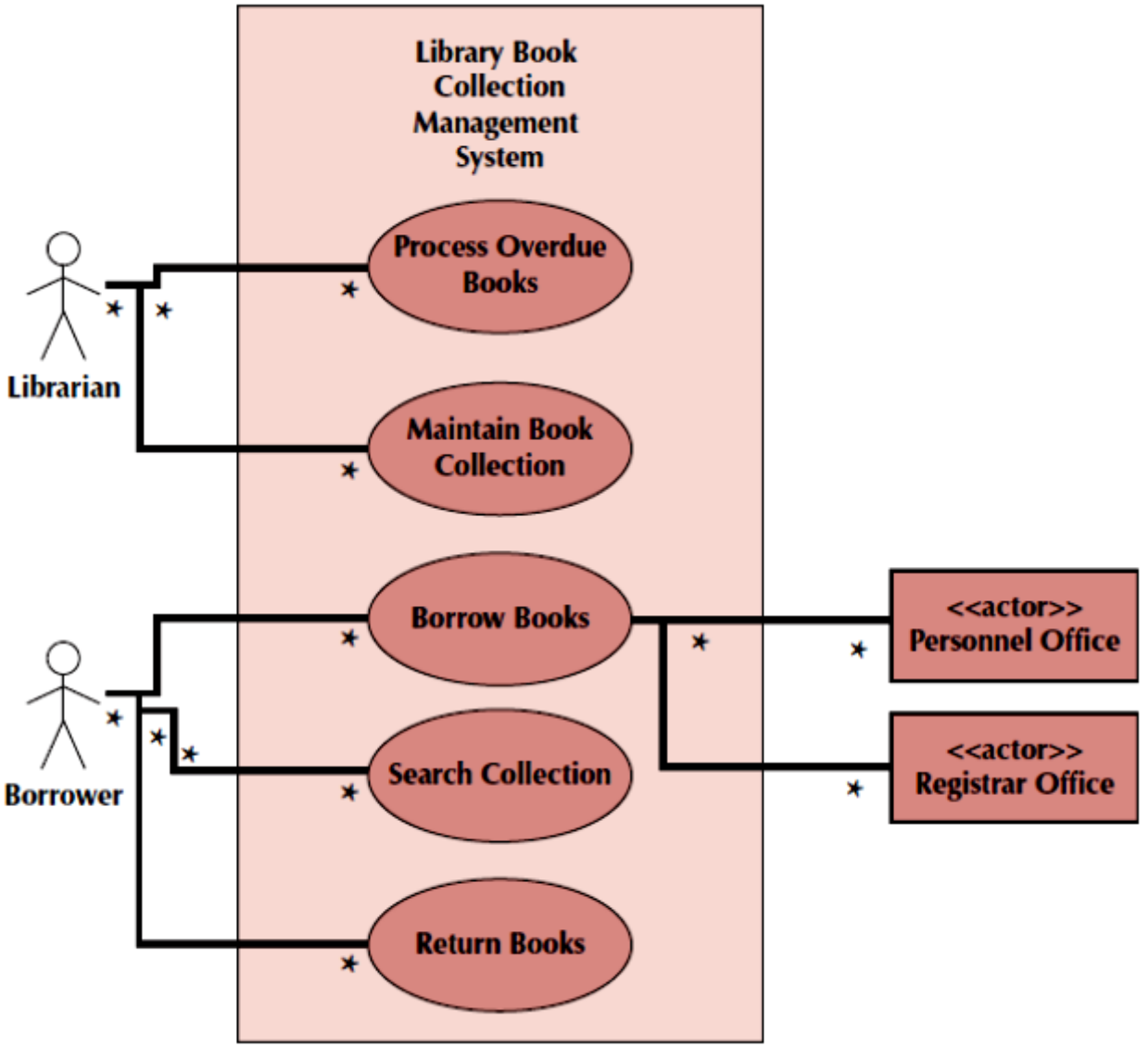
In the previous chapters, we have demonstrated the diagramming and modeling processes using the Borrow Books use case of the Library Book Collection Management System. When considering instance-specific scenario diagrams, we need to draw one sequence diagram per scenario. In the case of the Borrow Books use case, there are nine different scenarios. Therefore, for this one use case, there would be nine separate diagrams.

In this example, we are setting the context of the sequence diagram to only one specific scenario of the Borrow Books use case: Students who have a valid ID and do not have any overdue books or any fines. The other scenarios include Students without a valid ID, Students with a valid ID but who owe fines or have overdue books, and the same three scenarios for the other two types of Borrowers: Faculty/Staff and Guest. In this example, we are only drawing the one sequence diagram for the Students with a valid ID scenario. To begin with, we should review the Flow of Events of the use-case description, the activity diagram, and the use-case diagram.

Normal Flow of Events:

1. The Borrower brings books to the Librarian at the check out desk.
2. The Borrower provides Librarian their ID card.
3. The Librarian checks the validity of the ID Card.
   - If the Borrower is a Student Borrower, Validate ID Card against Registrar's Database.
   - If the Borrower is a Faculty/Staff Borrower, Validate ID Card against Personnel Database.
   - If the Borrower is a Guest Borrower, Validate ID Card against Library's Guest Database.
4. The Librarian checks whether the Borrower has any overdue books and/or fines.
5. The Borrower checks out the books.


Validate ID Card
[Valid Card]
Check for Overdue Books and Fines
[No Overdue Books & No Fines]
Check Out Books

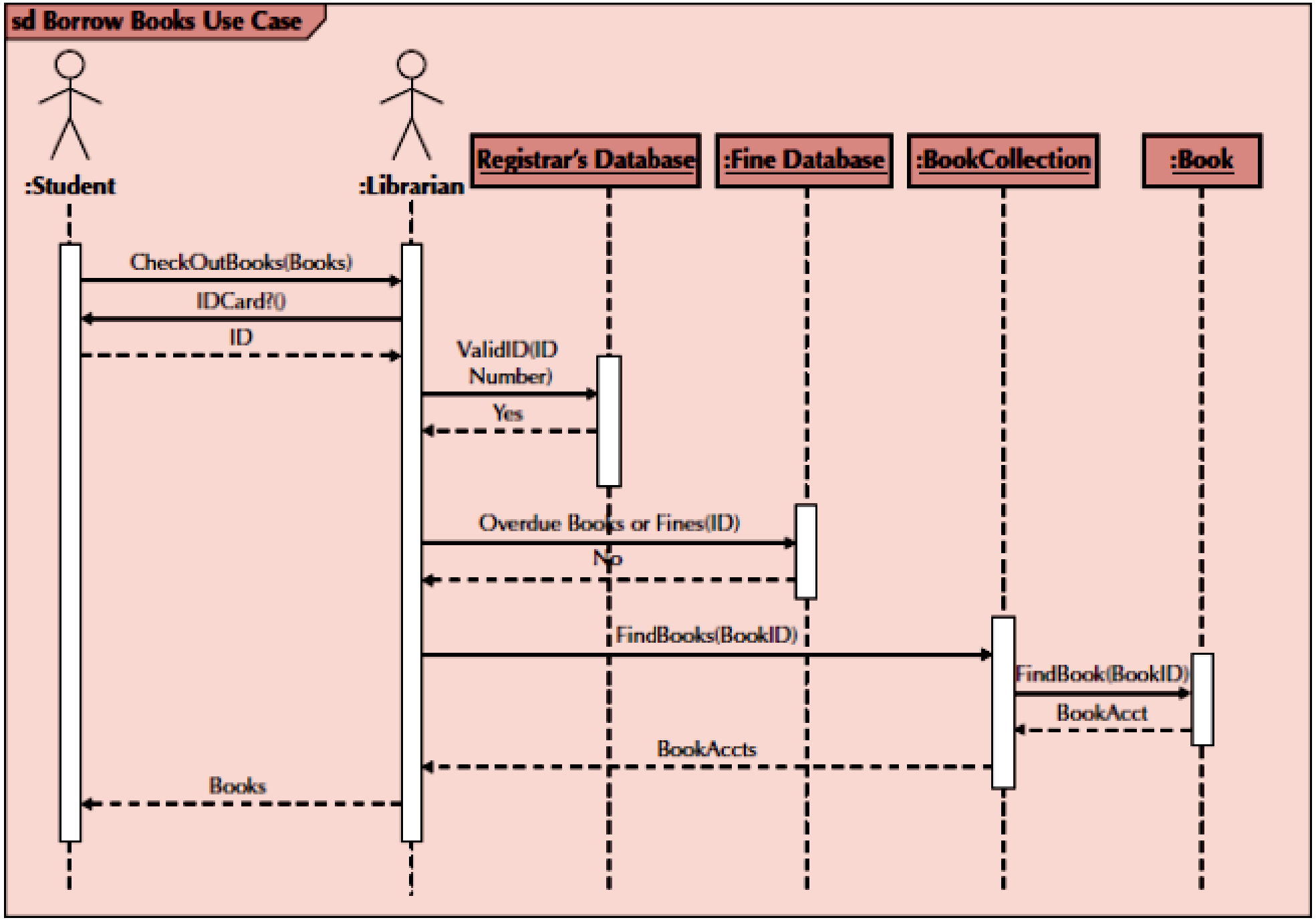

Library Book Collection Management System
Process Overdue Books
Maintain Book Collection
Borrow Books
Search Collection
Return Books
Librarian
Borrower
<<actor>> Personnel Office
<<actor>> Registrar Office

Next,

The next step is to identify the actors and objects involved in the scenario. By studying the flow of events and the use-case diagram we identify students, librarians, and the registrar's database as actors and borrowers, the book collection, and books as the objects. We place the actors and objects across the top of the diagram based on the ordering of their appearance in the normal flow of events. The next step involves simply drawing the lifelines beneath the actors and objects in the scenario. The fourth step is to add the actual messages to the diagram. To do this, we again review the actual steps taken when executing this scenario by reviewing the flow of events and the activity diagram. We also should review any results from the role-playing of the CRC cards .

This will help us to properly portray where the functionality is located.

For example, in next Figure, the Librarian executes the CheckOutBooks() procedure (the Student sends the message CheckOutBooks () to ask the Librarian to execute the Check- OutBooks () procedure) when the student hands the librarian the books to check out. The Librarian in return asks the Student for the ID card. When the student hands the ID Card to the Librarian, the Librarian asks the Registrar's Database to execute the ValidID() procedure when the Librarian passes the student's ID number over to the database system to ask the database system to validate the student's ID number. This continues until the ID Card and Books are returned to the student. Once we have decided from whom the messages are to be sent and to whom they are sent, we can place the messages on the diagram. The fifth step then is to add the execution occurrence to the diagrams to show when each actor or object is in the process of executing one of its operations. Next, we must validate the diagram. Finally, we should replicate this process for the other eight scenarios.


sd Borrow Books Use Case
:Student
:Librarian
Registrar's Database
:Fine Database
:BookCollection
:Book
CheckOutBooks(Books)
IDCard?()
ID
ValidID(ID Number)
Yes
Overdue Books or Fines(ID)
No
FindBooks(BookID)
FindBook(BookID)
BookAcct
BookAccts
Books

Communication Diagrams

Communication diagrams, like sequence diagrams, essentially provide a view of the dynamic aspects of an object-oriented system.

They can show how the members of a set of objects collaborate to implement a use case or a use-case scenario.

They can also be used to model all the interactions among a set of collaborating objects, in other words, a collaboration.

In this case, a communication diagram can portray how dependent the different objects are on one another.

Communication diagrams are equivalent to sequence diagrams, but they emphasize the flow of messages through a set of objects, whereas the sequence diagrams focus on the time ordering of the messages being passed.

Therefore, to understand the flow of control over a set of collaborating objects or to understand which objects collaborate to support business processes, a communication diagram can be used.

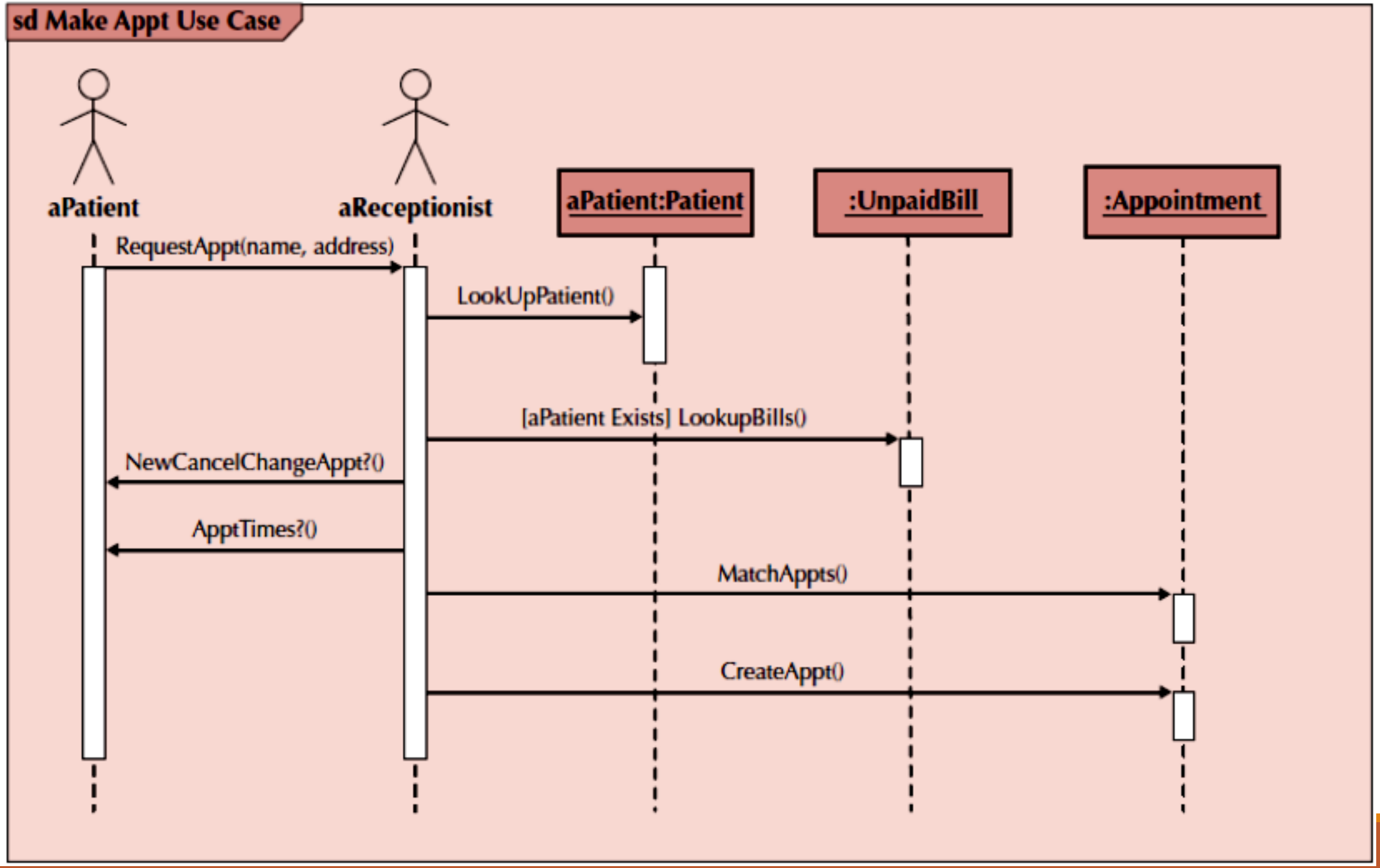
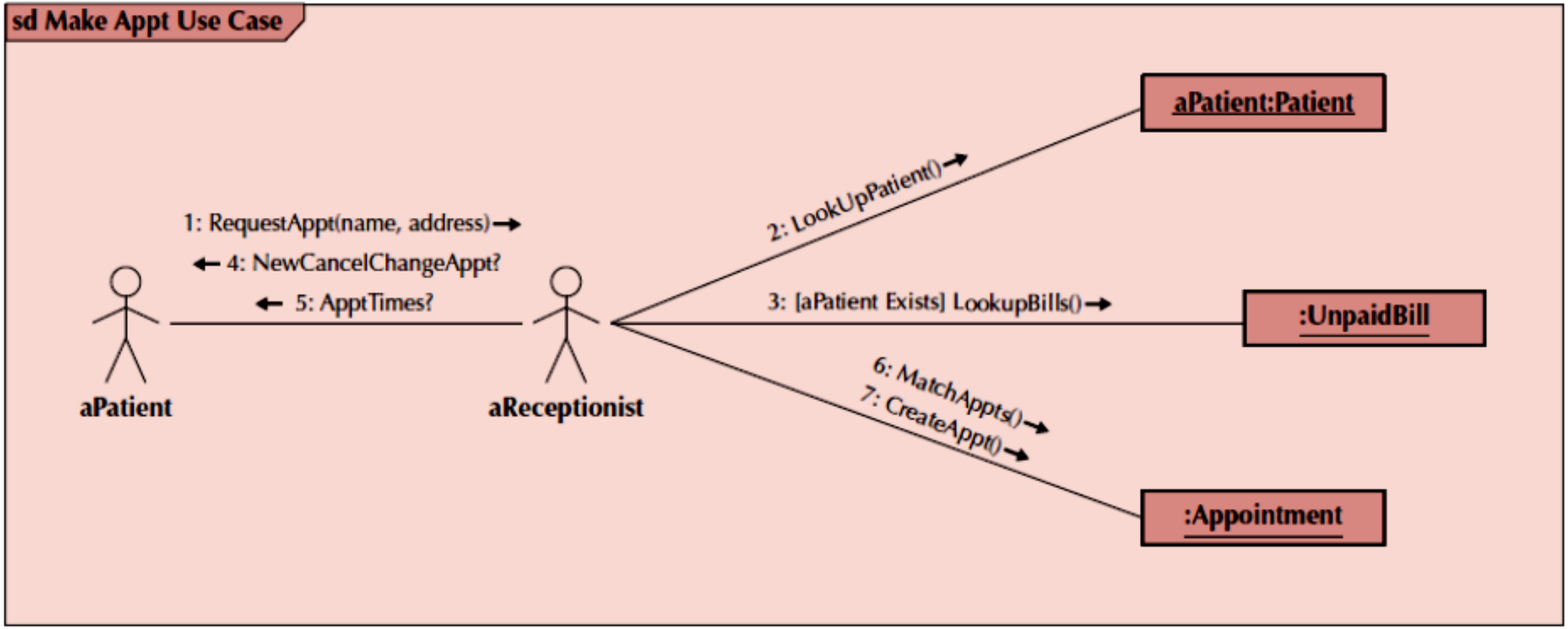
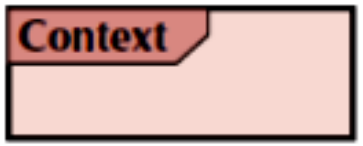

sd Make Appt Use Case
aPatient
aReceptionist
aPatient:Patient
:UnpaidBill
:Appointment
RequestAppt(name, address)
LookUpPatient()
[aPatient Exists] LookupBills()
NewCancelChangeAppt?()
ApptTimes?()
MatchAppts()
CreateAppt()

Figure shows a communication diagram for the Make Old Patient Appt use case.



Actors and objects that collaborate to execute the use case are placed on the communication diagram in a manner to emphasize the message passing that takes place between them.

Term and Definition	Symbol
An actor: ■ Is a person or system that derives benefit from and is external to the system. ■ Participates in a collaboration by sending and/or receiving messages. ■ Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative).	 anActor <<actor>> anActor
An object: ■ Participates in a collaboration by sending and/or receiving messages.	anObject : aClass

An association: ■ Shows an association between actors and/or objects. ■ Is used to send messages.	
A message: ■ Conveys information from one object to another one. ■ Has direction shown using an arrowhead. ■ Has sequence shown by a sequence number.	SeqNumber: aMessage →
A guard condition: ■ Represents a test that must be met for the message to be sent.	SeqNumber: [aGuardCondition]: aMessage →
A frame: ■ Indicates the context of the communication diagram.	

System Analysis

DR. DRAGUNOV

BEHAVIORAL STATE MACHINES

BEHAVIORAL STATE MACHINES

Some of the classes in the *class diagrams* represent a set of objects that are quite dynamic in that they pass through a variety of states over the course of their existence. For example, a patient can change over time from being new to current to former based on his or her status with the doctor's office.

Definition.

A behavioral state machine is a dynamic model that shows the different states through which a single object passes during its life in response to events, along with its responses and actions.

The behavioral state machine shows the different states of the object and what events cause the object to change from one state to another.

The ***state* of an object** is defined by the value of its attributes and its relationships with other objects at a particular point in time.

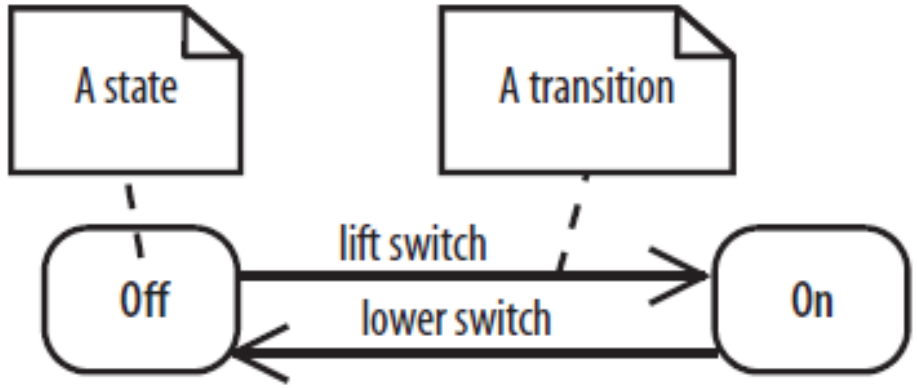
An ***event*** is something that takes place at a certain point in time and changes a value or values that describe an object, which, in turn, changes the object's state.

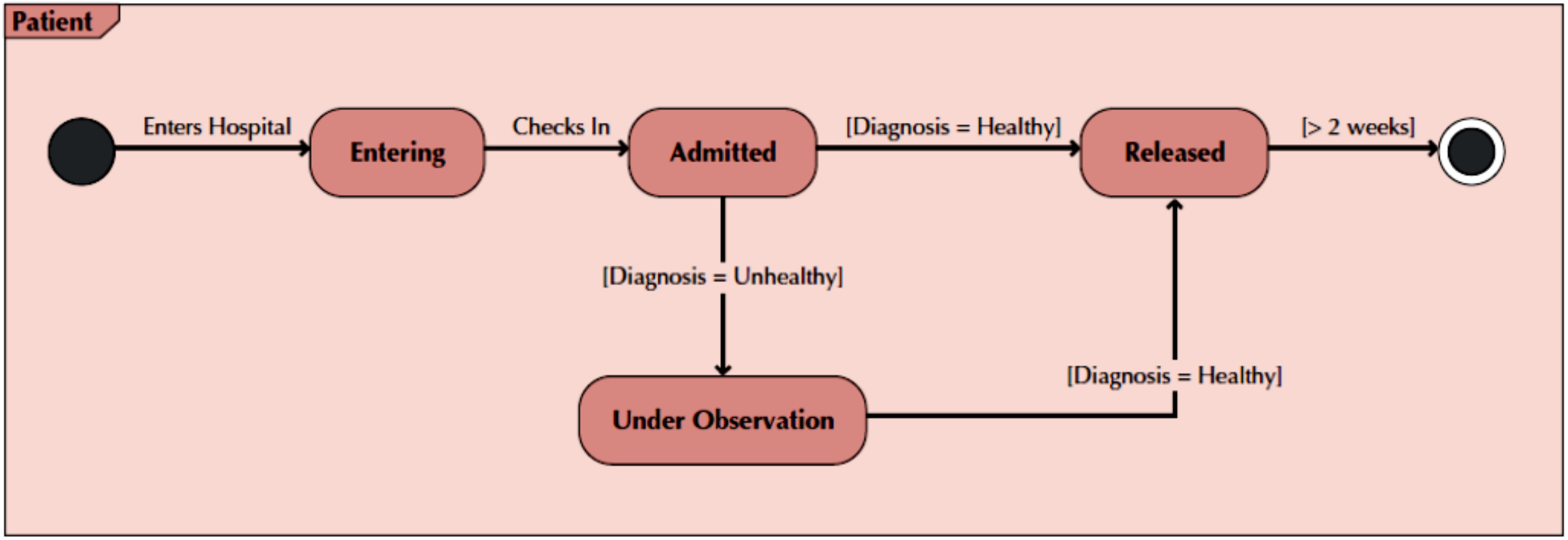
A ***transition*** is a relationship that represents the movement of an object from one state to another state. Some transitions have a guard condition.

A ***guard condition*** is a Boolean expression that includes attribute values, which allows a transition to occur only if the condition is true.

An ***action*** is an atomic, non-decomposable process that cannot be interrupted. From a practical perspective, actions take zero time, and **they are associated with a transition**.

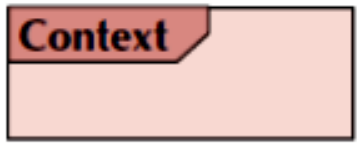
Sample Behavioral State Machine Diagram





Elements of a Behavioral State Machine

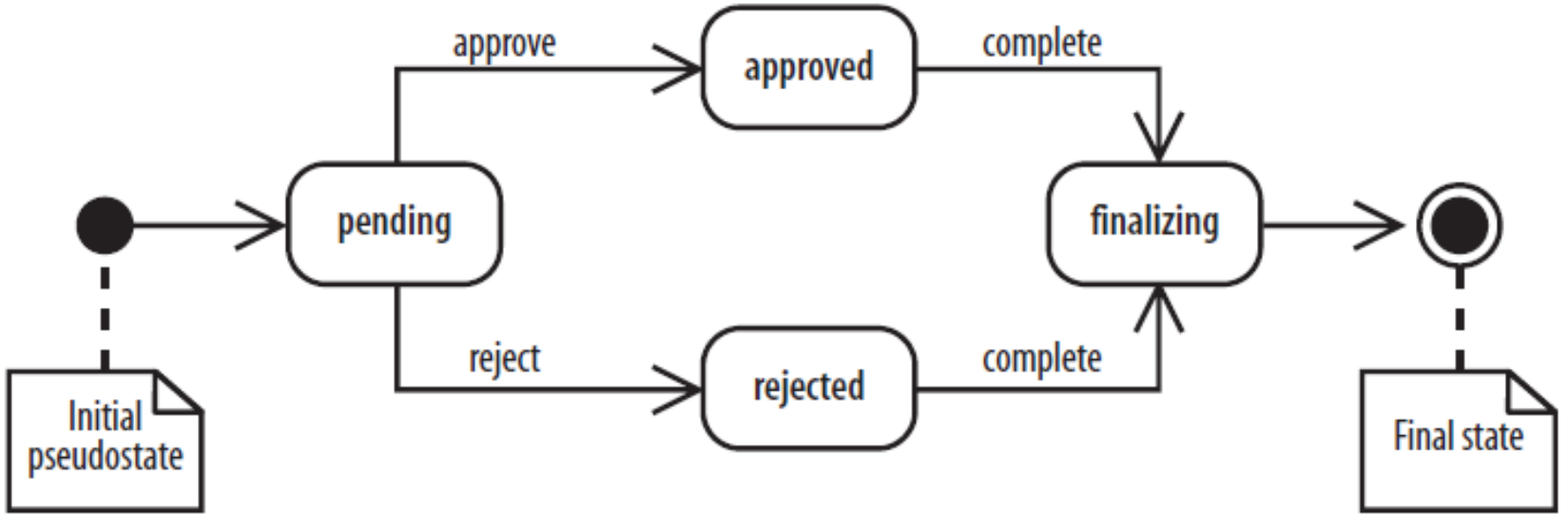
Term and Definition	Symbol
A state: ■ Is shown as a rectangle with rounded corners. ■ Has a name that represents the state of an object.	
An initial state: ■ Is shown as a small, filled-in circle. ■ Represents the point at which an object begins to exist.	
A final state: ■ Is shown as a circle surrounding a small, filled-in circle (bull's-eye). ■ Represents the completion of activity.	

An event: ■ Is a noteworthy occurrence that triggers a change in state. ■ Can be a designated condition becoming true, the receipt of an explicit signal from one object to another, or the passage of a designated period of time. ■ Is used to label a transition.	anEvent
A transition: ■ Indicates that an object in the first state will enter the second state. ■ Is triggered by the occurrence of the event labeling the transition. ■ Is shown as a solid arrow from one state to another, labeled by the event name.	
A frame: ■ Indicates the context of the behavioral state machine.	

States

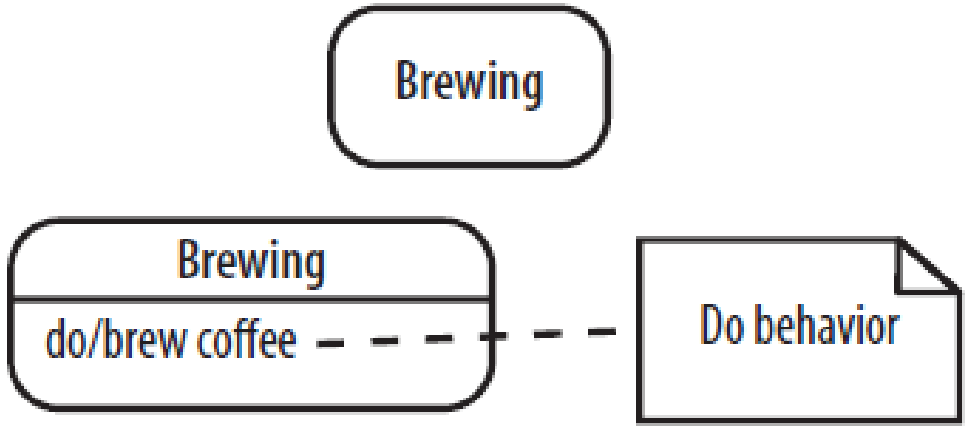
A *state* is a set of values that describes an object at a specific point in time, and it represents a point in an object's life in which it satisfies some condition, performs some action, or waits for something to happen.

State diagrams usually have an ***initial pseudostate*** and a ***final state***, marking the start and end points of the state machine, respectively. An initial pseudostate is drawn with a filled circle, and a final state is drawn with two concentric circles with a filled inner circle. Pseudostates are special markers that direct the flow of traffic in a state diagram.



States

If the state is a “doing” state, you can write the behavior inside the state



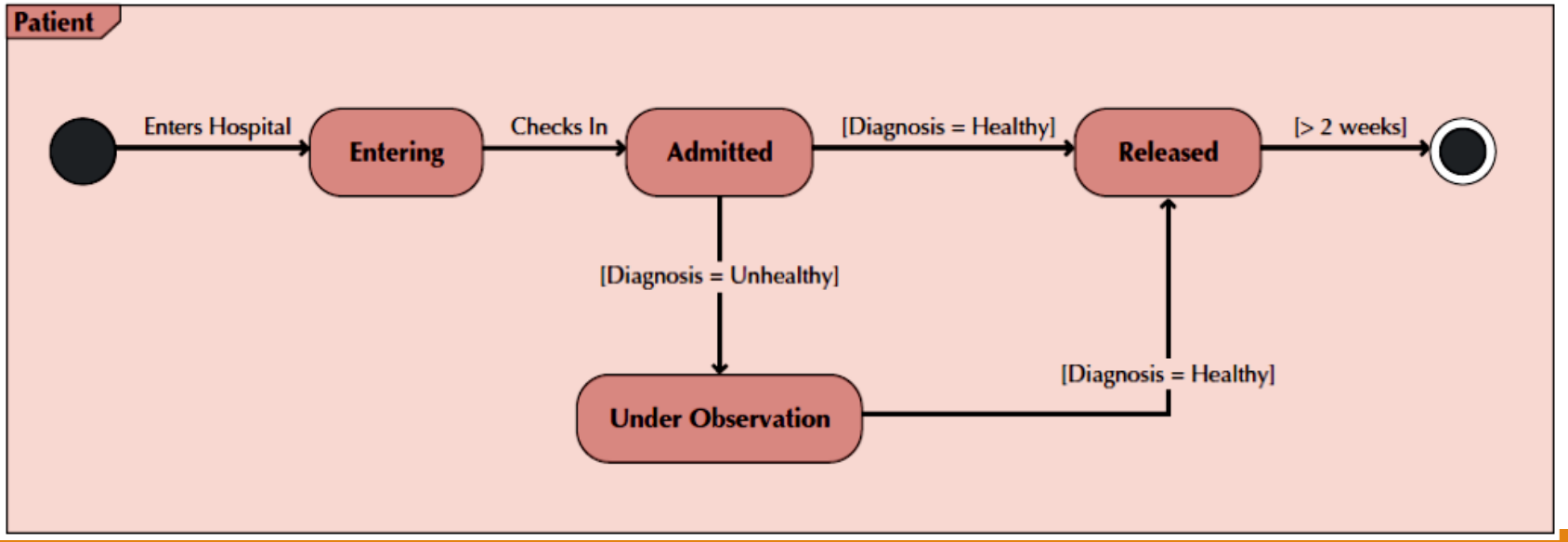
Do behavior, written as do/*behavior*, is behavior that happens as long as the state is active.

Transitions

Arrows are used to connect the state symbols, representing the transitions between states. **Each arrow is labeled with the appropriate event name and any parameters or conditions that may apply.**.

The full notation for transition descriptions is *trigger[guard] / behavior*, where each element is optional

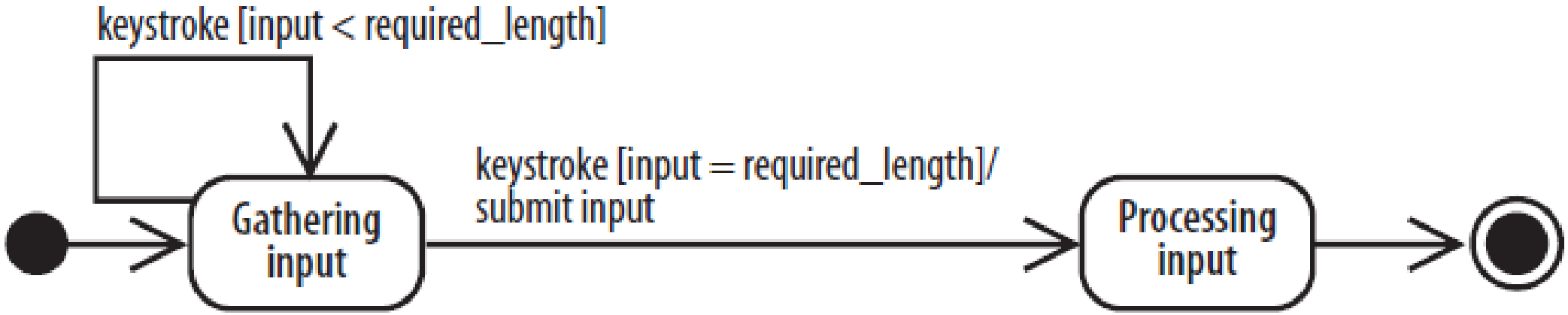
A *trigger* is an event that may cause a transition. In a system that processes user input, a keystroke trigger may cause the system to change states from Gathering input to Processing input.



Transitions

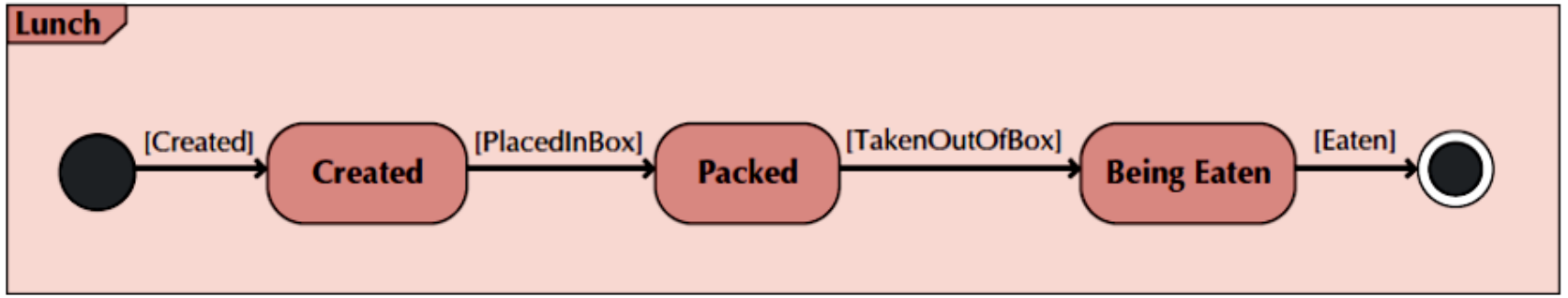
A transition, shown with an arrow, represents a change of states from a *source state* to a *target state*. A ***transition description***, written along the arrow, describes the circumstances causing the state change to occur.

The full notation for transition descriptions is *trigger(event)[guard] / behavior*, where each element is optional

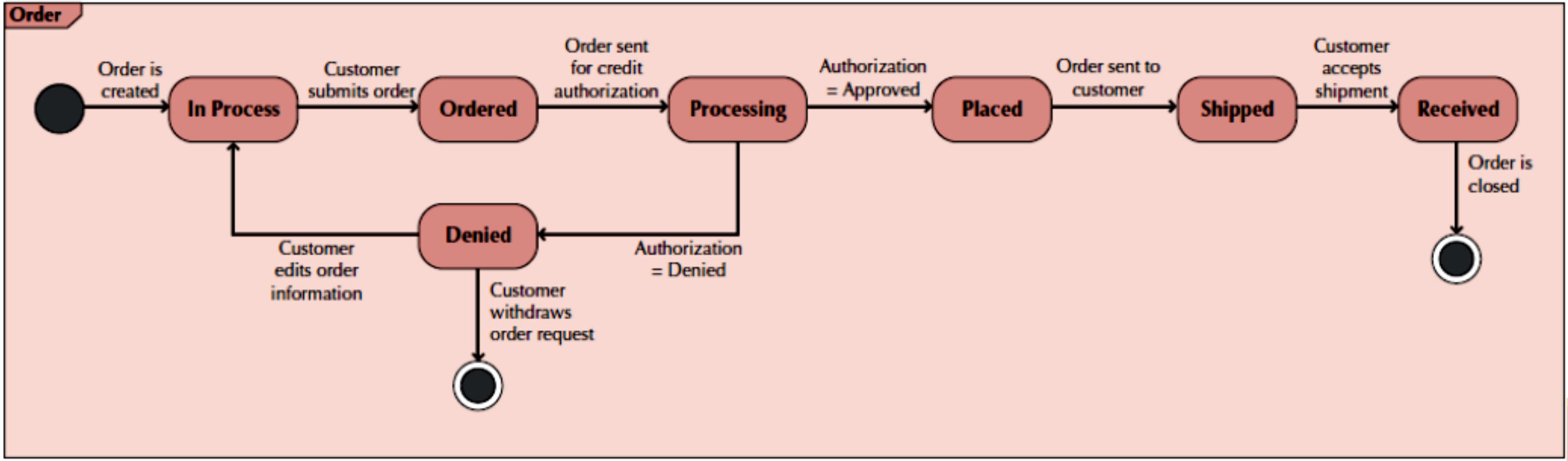


A *trigger* is an event that may cause a transition. In a system that processes user input, a keystroke trigger may cause the system to change states from Gathering input to Processing input.

Figure depicts two additional behavioral state machines. The first one is for the lunch object that was associated with the Make Lunch use-case scenario:

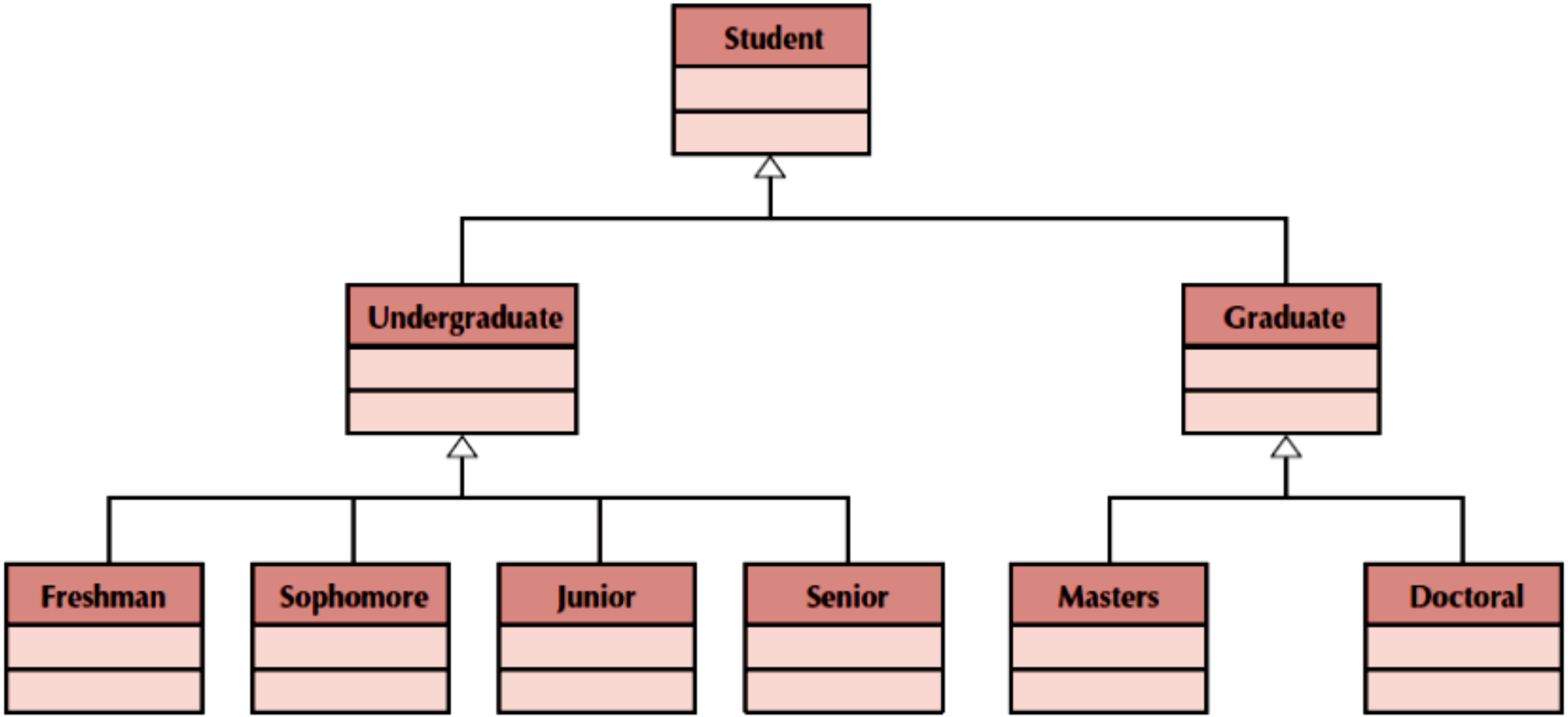


The second behavioral state machine deals with the life cycle of an order.

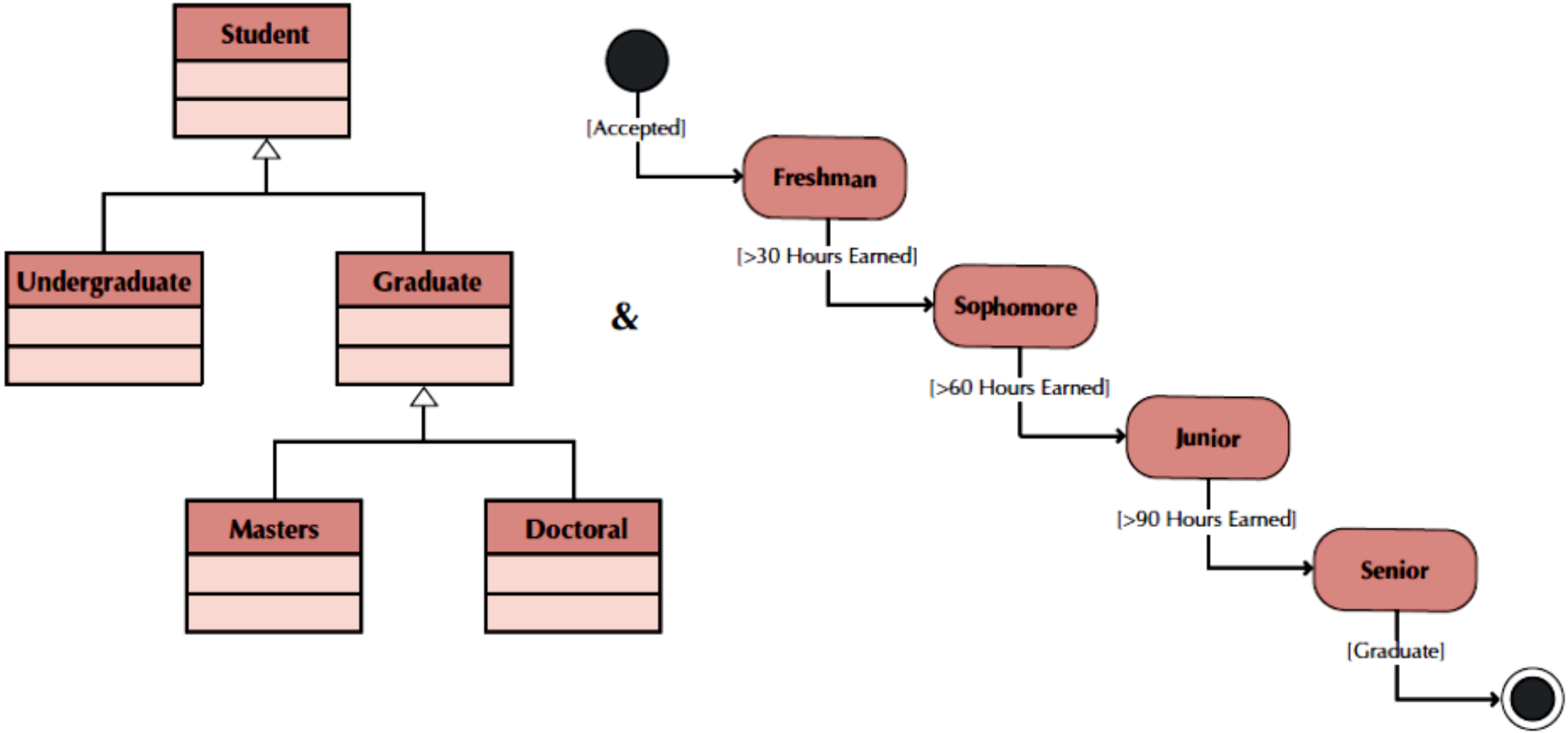


Obviously, because behavioral state machines can uncover additional processing requirements, they can be very useful in filling out the complete description of an evolving system.

Sometimes, states and subclasses can be confused. For example, in Figure, **are the classes Freshman, Sophomore, Junior, and Senior subclasses of the class Undergraduate or are they simply states that an instance of the Undergraduate class goes through during its lifetime?**



VS


Student
Undergraduate
Graduate
Masters
Doctoral
&
[Accepted]
Freshman
[>30 Hours Earned]
Sophomore
[>60 Hours Earned]
Junior
[>90 Hours Earned]
Senior
[Graduate]

Guidelines for Creating Behavioral State Machines

Create a behavioral state machine for objects whose behavior changes based on the state of the object. In other words, do not create a behavioral state machine for an object whose behavior is always the same regardless of its state. These objects are too simple.

Owing to the way western cultures learn to read, from left to right and top to bottom, the initial state should be drawn in the top left corner of the diagram and the final state should be drawn in the bottom right of the diagram.

Make sure that the names of the states are simple, intuitively obvious, and descriptive.

- **Question black hole and miracle states.** These types of states are problematic for the same reason black hole and miracle activities are a problem for activity diagrams. ***Black hole states***, states that an object goes into and never comes out of, most likely are actually final states. ***Miracle states***, states that an object comes out of but never went into, most likely are initial states.

- **Be sure that all guard conditions are mutually exclusive**, (not overlapping). If you created a guard condition of [x >= 0] and a second guard condition [x <= 0], the guard conditions overlap when x = 0, and it is not clear to which state the object would transition. This would obviously cause confusion.

- **All transitions should be associated with a message and operation**. Otherwise, the state of the object could never change. Even though this may be stating the obvious, there have been numerous times that analysts forget to go back andensure that this is indeed true.

Creating a Behavioral State Machine

1. Set Context: Behavioral state machines are drawn to depict an instance of a single class from a class diagram.

2. Identify Object States : This includes establishing the boundaries of the existence of an object by identifying the initial and final states of an object.

3. Lay Out Diagram: Determine the sequence of the states that an object will pass through during its lifetime.

4. Add Transitions: The fourth step is to identify the transitions between the states of the objects and to add the events, actions, and guard conditions associated with the transitions.

5. Validate: The fifth step is to validate the behavioral state machine by making sure that each state is reachable and that it is possible to leave all states except for final states.

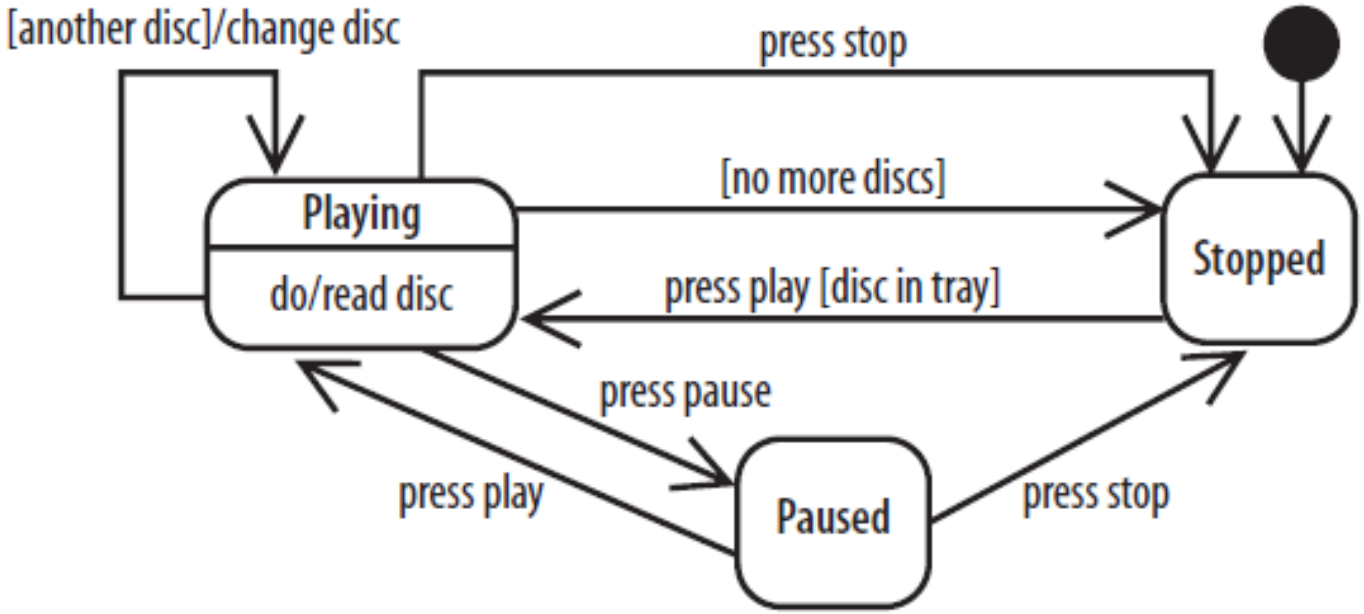
EXAMPLE.

A) Design state diagram for a CD player with 3 possible states:

1) Playing

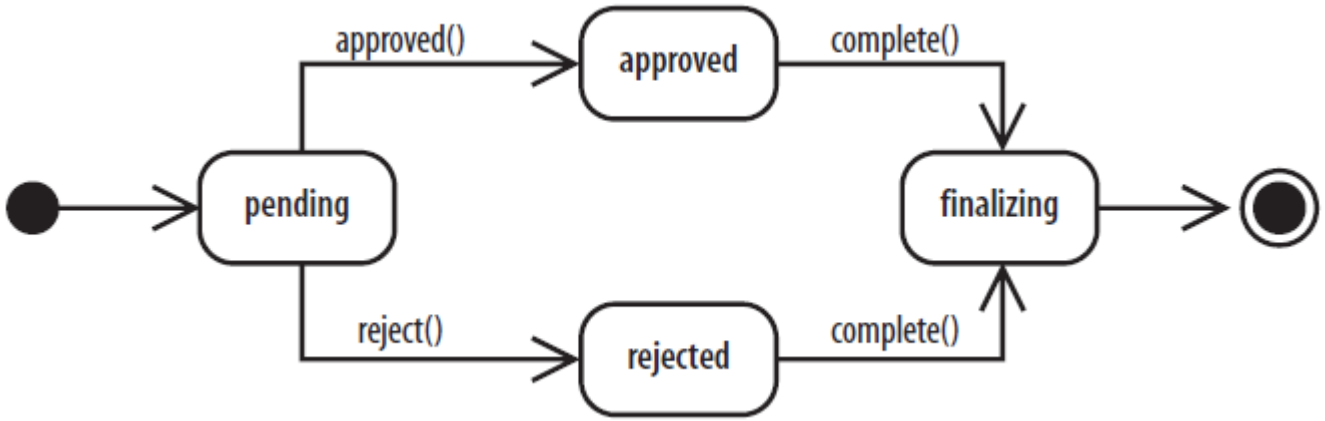
2)Stopped

5) Paused



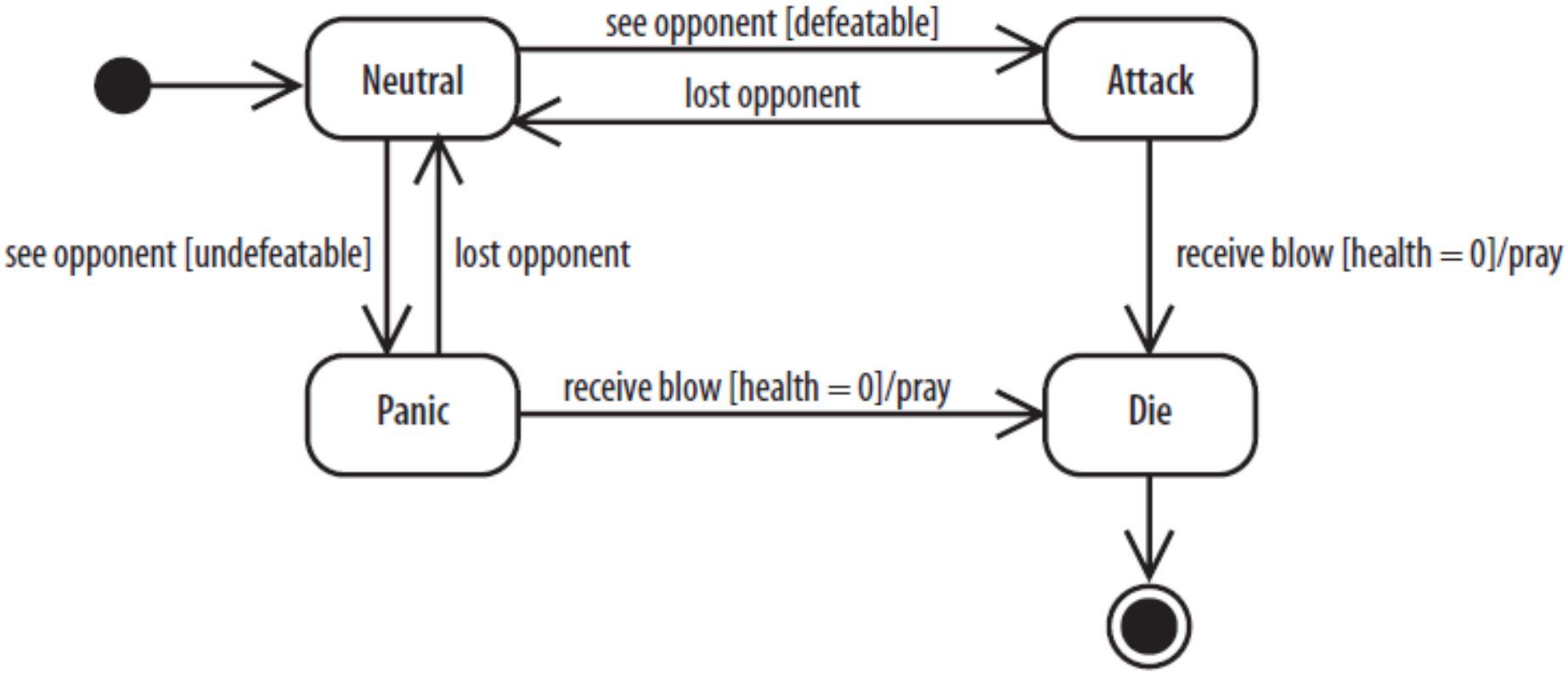
States in Software

If you're a software developer, you're probably wondering when you'll ever need to model the operation of a CD player or coffeemaker. In software, state diagrams model an object's *life cycle*, or the states it goes through during its lifespan.



State diagrams are useful for modeling an object that behaves differently depending on its state.

State diagram are also heavily used in certain software niches, such as first-person shooter (FPS) games.



Example 2

Draw diagram for object of Book class from Library Book Collection Management System.

1. Set Context

Book : Book

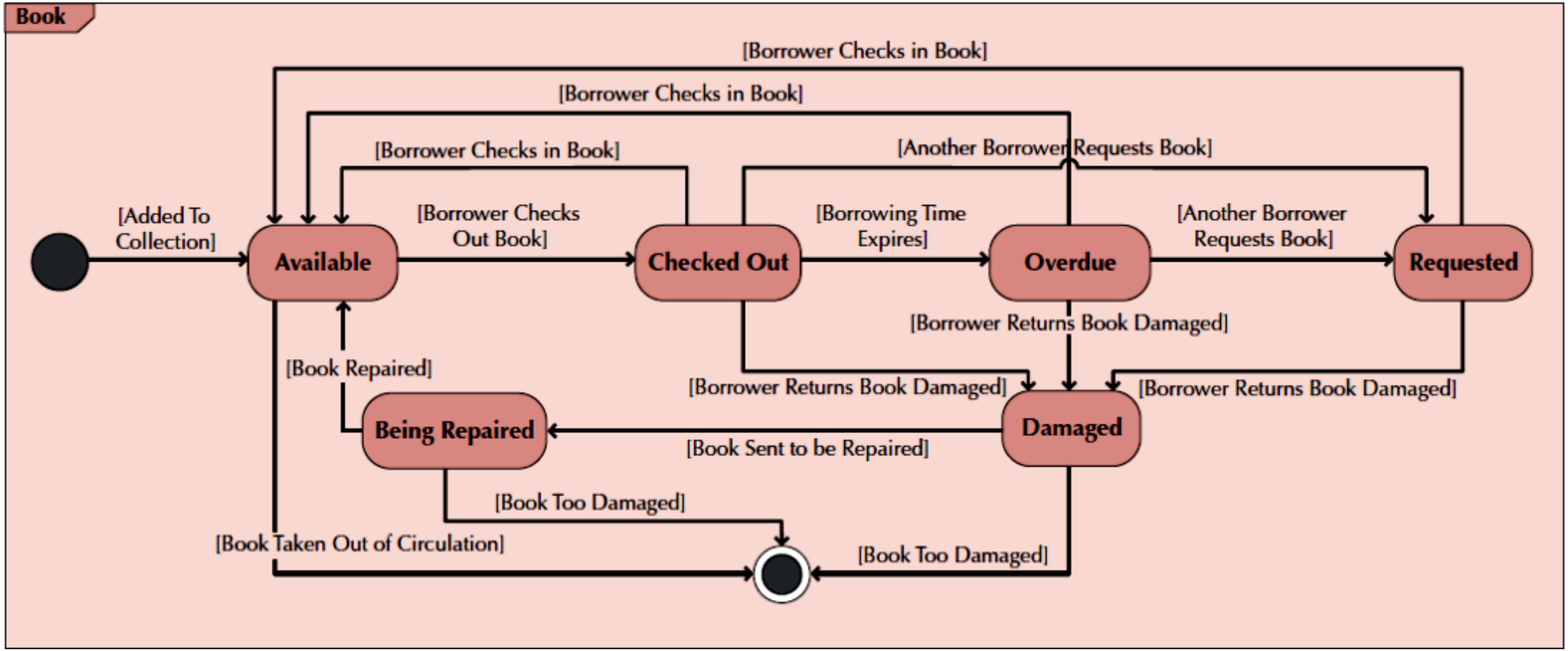
2. Identify Object States

a) Available b)Checked out c)Overdue d)Requested e) Being Repaired f) Damaged

3. Lay Out Diagram

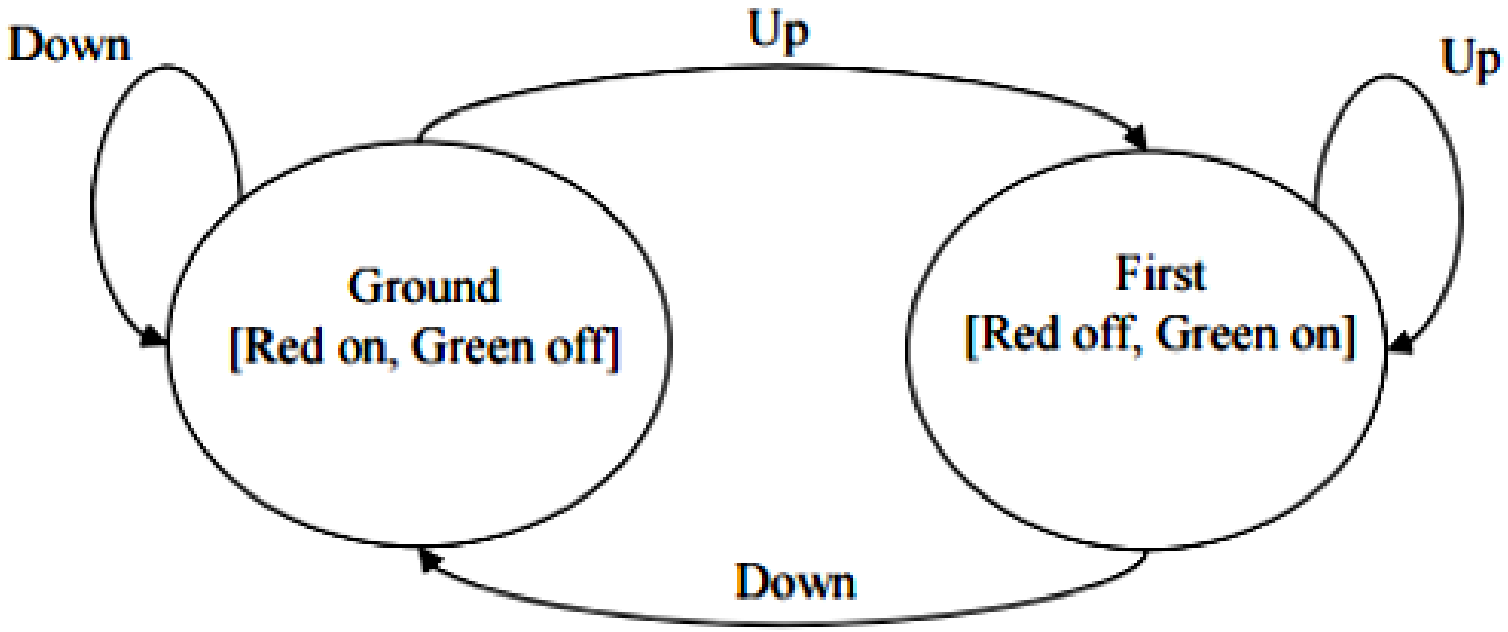
4. Add Transitions

5. Validate


Book
[Borrower Checks in Book]
[Borrower Checks in Book]
[Borrower Checks in Book]
[Another Borrower Requests Book]
[Added To Collection]
Available
[Borrower Checks Out Book]
Checked Out
[Borrowing Time Expires]
Overdue
[Another Borrower Requests Book]
Requested
[Borrower Returns Book Damaged]
[Book Repaired]
[Borrower Returns Book Damaged]
[Borrower Returns Book Damaged]
Being Repaired
Damaged
[Book Sent to be Repaired]
[Book Too Damaged]
[Book Taken Out of Circulation]
[Book Too Damaged]

Example

The elevator can be at one of two floors: Ground or First. There is one button that controls the elevator, and it has two values: Up or Down. Also, there are two lights in the elevator that indicate the current floor: Red for Ground, and Green for First.

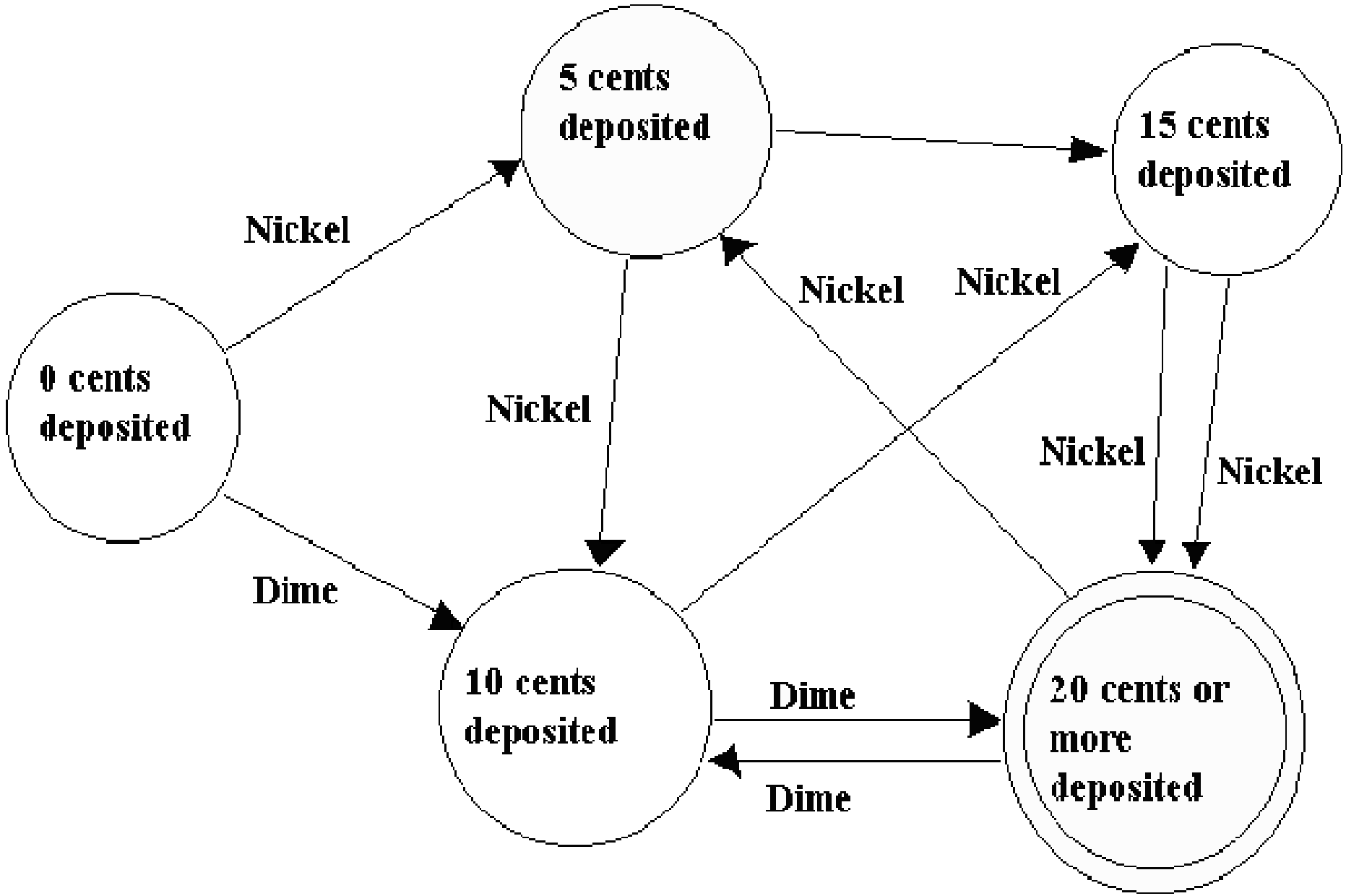


A Simple Vending Machine

A vending r

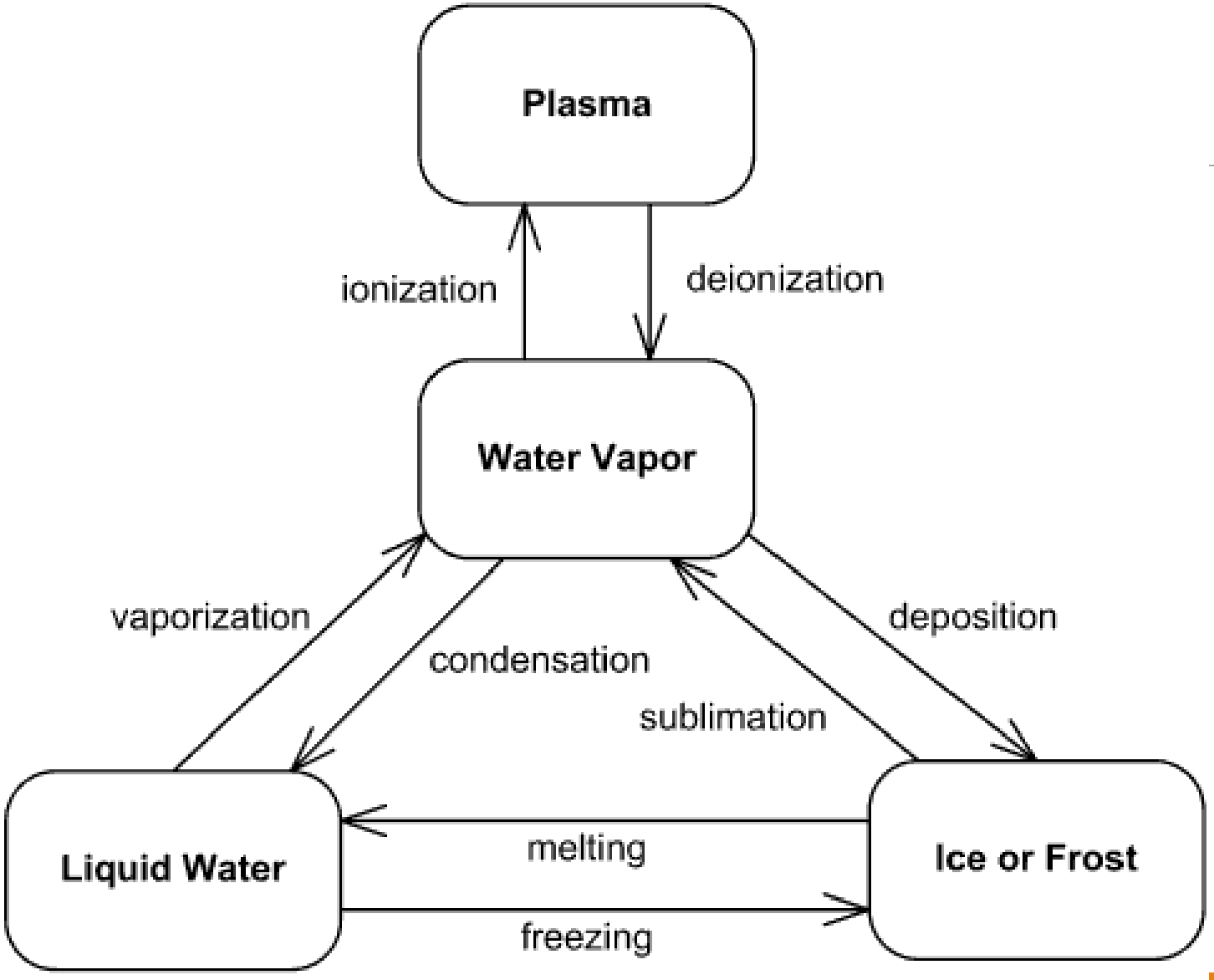
The machir
the amoun
candy.

The next cc



soon as
piece of

Water can exist in several states - liquid, vapor, solid, and plasma. Several transitions are possible from one state to another.



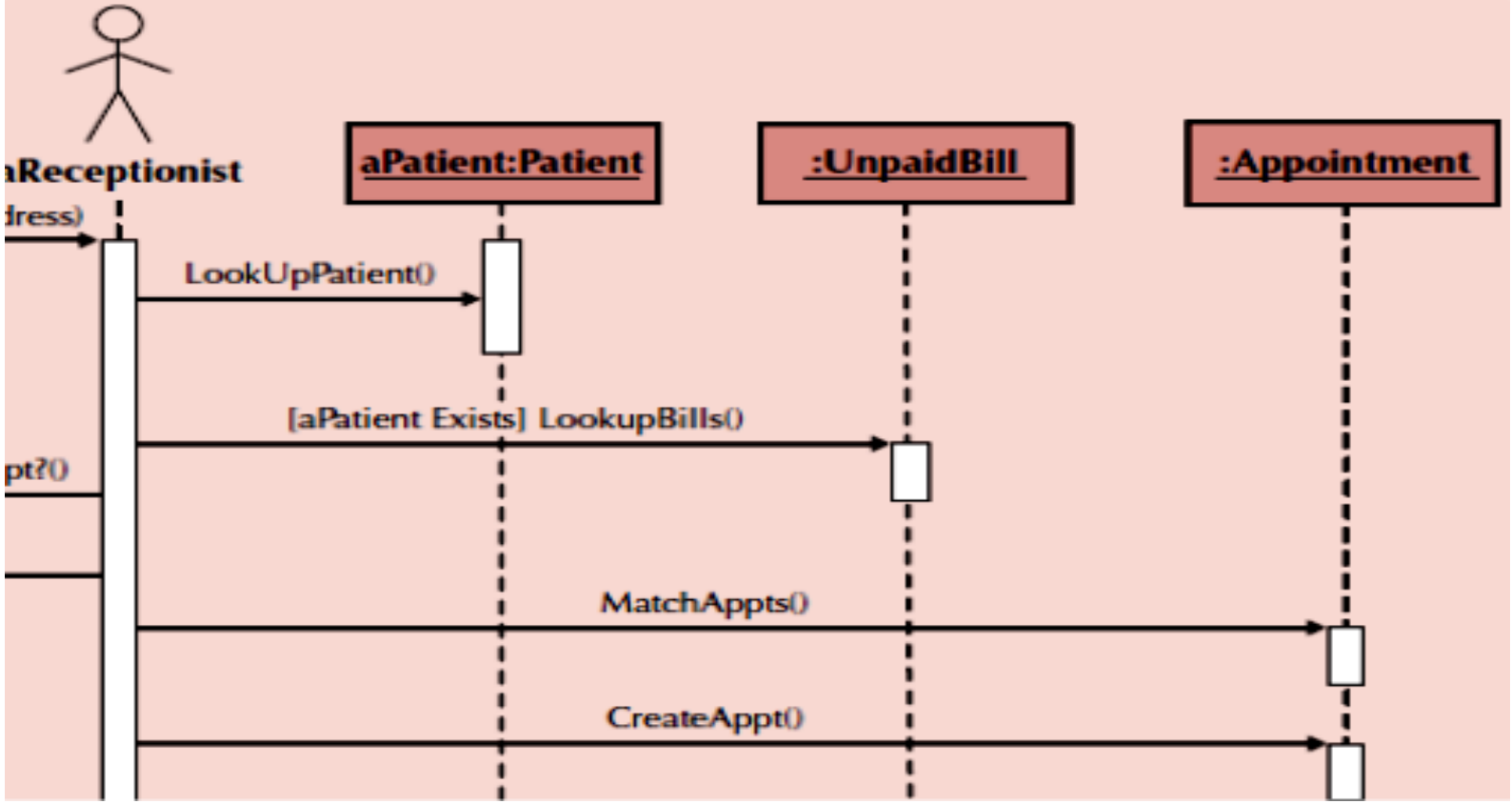
CRUDE ANALYSIS

One useful technique for identifying potential collaborations is ***CRUDE analysis***. CRUDE analysis uses a ***CRUDE matrix*** , in which each interaction among objects is labeled with a letter for the type of interaction:

C for create, R for read or reference, U for update, D for delete, and E for execute.

In an object-oriented approach, a class/actor-by-class/actor matrix is used. Each cell in the matrix represents the interaction between instances of the classes.

For example, in Figure,



an instance of the Receptionist actor creates an instance of the Appointment class. Assuming a Row:Column ordering, a C is placed in the cell Receptionist:Appointment. Also, in Figure, an instance of the Receptionist actor references an instance of the Appointments class. In this case, an R is placed in the Receptionist:Appointments cell.

Figure shows the CRUDE matrix based on the Make Old Patient Appt use case.

	Receptionist	PatientList	Patient	UnpaidBills	Appointments	Appointment
Receptionist		RU	CRUD	R	RU	CRUD
PatientList			R			
Patient						
UnpaidBills						
Appointments						R
Appointment						

Unlike the interaction diagrams and behavioral state machines, a CRUDE matrix is most useful as a system-wide representation. Once a CRUDE matrix is completed for the entire system, the matrix can be scanned quickly to ensure that every class can be instantiated. Each type of interaction can be validated for each class.

VERIFYING AND VALIDATING THE BEHAVIORAL MODEL

In this chapter, we described three different diagrams (sequence diagram, communication diagram, and behavioral state machine) and CRUDE matrices that could be used to represent the behavioral model.

The sequence and communication diagrams modeled the interaction among instances of classes that worked together to support the business processes included in a system, the behavioral state machine described the state changes through which an object traverses during its lifetime, and the CRUDE matrix represented a system level overview of the interactions among the objects in the system.

In this chapter, we combine walkthroughs with CRUDE matrices to more completely verify and validate the behavioral models. Since, in the previous section we covered CRUDE analysis and matrices, we focus only on walkthroughs in this now.

First, every actor and object included on a sequence diagram must be included as an actor and an object on a communication diagram, and vice versa.

Second, if there is a message on the sequence diagram, there must be an association on the communications diagram, and vice versa.

Third, every message that is included on a sequence diagram must appear as a message on an association in the corresponding communication diagram, and vice versa.

Fourth, if a guard condition appears on a message in the sequence diagram, there must be an equivalent guard condition on the corresponding communication diagram, and vice versa.

Fifth, the sequence number included as part of a message label in a communications diagram implies the sequential order in which the message will be sent. Therefore, it must correspond to the top-down ordering of the messages being sent on the sequence diagram.

Sixth, all transitions contained in a behavior state machine must be associated with a message being sent on a sequence and communication diagram, and it must be classified as a (C)reate, (U)pdate, or (D)elete message in a CRUDE matrix.

Seventh, all entries in a CRUDE matrix imply a message being sent from an actor or object to another actor or object. If the entry is a (C)reate, (U)pdate, or (D)elete, then there must be an associated transition in a behavioral state machine that represents the instances of the receiving class.

Finally, many representation-specific rules have been proposed. However, as in the other models, these rules are beyond the scope of this section on verification and validation.

SUMMARY

Behavioral Models

Behavioral models describe the internal logic of the business processes described by the use cases associated with an information system.

Interaction Diagrams

Interaction diagrams are used to describe how objects collaborate to support use cases. There are two different types of interaction diagrams: sequence and communication.

Behavioral State Machine

The behavioral state machine shows the different states through which a single class passes during its life in response to events, along with responses and actions.

CRUDE Analysis

CRUDE analysis is a very useful approach to identifying potential collaborations among classes and to verify and validate a system.

Verifying and Validating Behavioral Models

Verifying and validating the behavioral models are very similar to verifying and validating the functional, and structural models.

System Analysis

DR. DRAGUNOV

Data flow diagrams

INTRODUCTION

A ***process model*** is a graphical way of representing how a business system should operate. It illustrates the processes or activities that are performed and how data move among them.

A process model can be used to document the current system (i.e., as-is system) or the new system being developed (i.e., to-be system), whether computerized or not.

Figure 1

Use Case Name: Request a chemical	**ID:** UC-2	**Priority:** High

Actor: Lawn Chemical Applicator (LCA)

Description: The Lawn Chemical Applicator (LCA) specifies the lawn chemical needed for a job by entering its name or ID number. The system satisfies the request by reserving the quantity requested or the quantity available and notifying the Chemical Supply Warehouse of the pick-up.

Trigger: A Lawn Chemical Applicator (LCA) needs a chemical for a job.

Type: ☑ **External** ☐ **Temporal**

Preconditions:

1. The LCA identity is authenticated.
2. The LCA has necessary training and credentials on file.
3. The Chemical Supply datastore is up-to-date and on-line.

Normal Course:		**Information for Steps:**
1.0 Request a lawn chemical from the chemical supply warehouse.		
1. The LCA specifies the desired lawn chemical	←	Chemical name or ID
2. The system verifies the chemical is approved for usage	←	List of approved chemicals
3. The system displays the quantity of the lawn chemical on hand	←	Quantity on hand
4. The LCA specifies the quantity needed	←	Quantity needed
5. The system asks the LCA to confirm the request for the quantity needed or the quantity available (Alternative Course 1.1)	←	Request confirmation
6. The system gives the LCA a Chemical Pick-up Authorization for the quantity requested	→	Chemical Pick-up Authorization
7. The system notifies the Chemical Supply Warehouse of the chemical pick-up	→	Chemical Pick-up Notice
8. The system stores the Lawn Chemical Request in the Chemical Request datastore	→	Lawn Chemical Request

Alternative Courses:

1.1 Quantity available is less than quantity needed (branch at step 5)

1.	The system asks the LCA if he wants the quantity available or to cancel the request	
2a.	The LCA asks to take the quantity available	← Request quantity available
3a.	The system changes the quantity requested to the quantity available	
4a.	The system gives the LCA a Chemical Pick-up-Authorization for the quantity available	→ Chemical Pick-up Authorization
5a.	The system notifies the Chemical Supply Warehouse of the chemical pick-up	→ Chemical Pick-up Notice
6a.	The system stores the Lawn Chemical Request in the Chemical Management System	→ Lawn Chemical Request
7a.	The system notifies Purchasing of the chemical outage	→ Chemical Outage Notice
2b.	The LCA asks to cancel the request	← Cancellation
3b.	The system terminates the use case	

Postconditions:

1. The Lawn Chemical Request is stored in the Chemical Management System.
2. The Chemical Pick-up Authorization is produced for the LCA.
3. The Chemical Supply Warehouse is notified of the chemical pick-up.
4. Purchasing is notified of chemical outage.

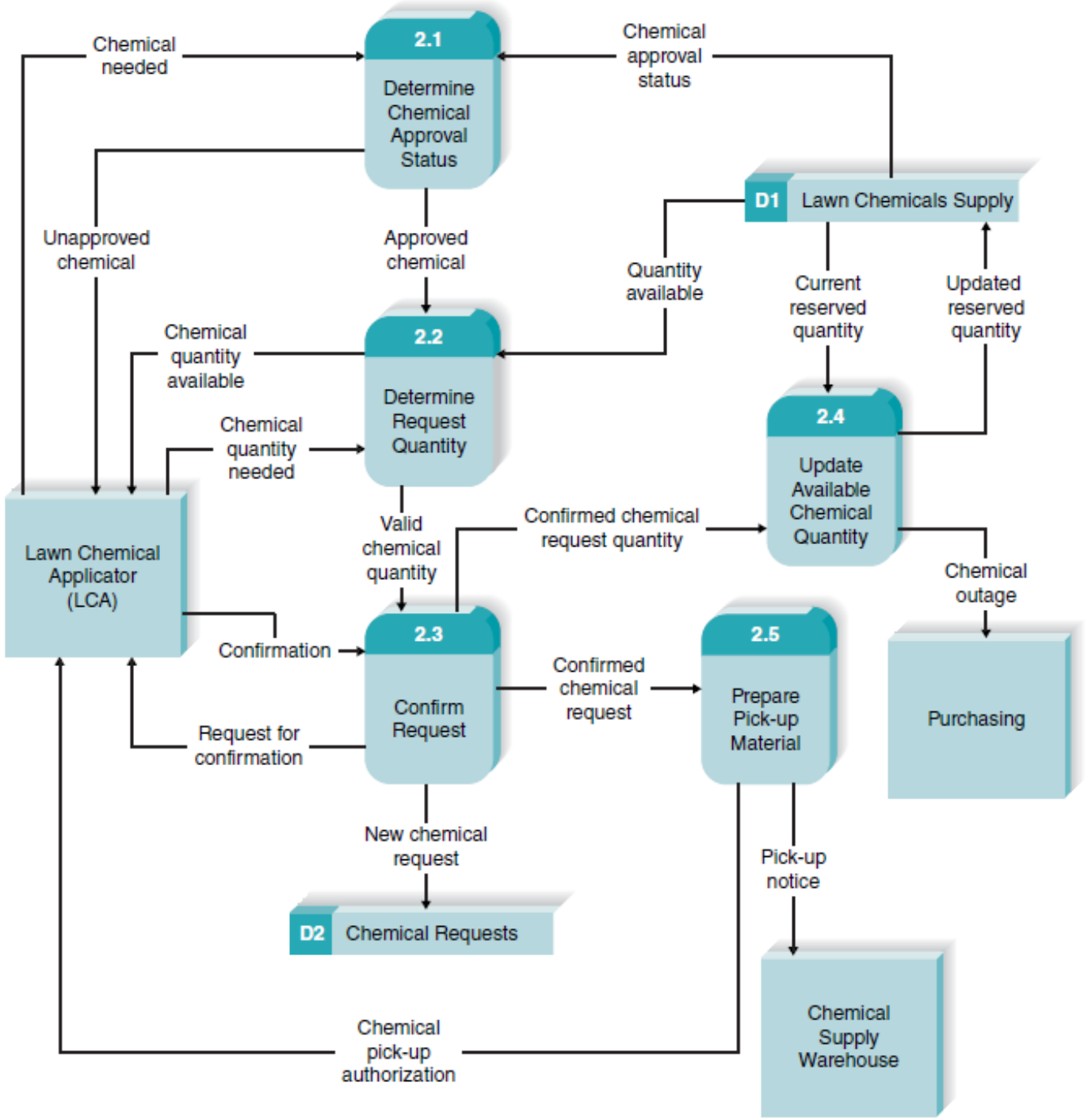
Exceptions:

E1: Chemical is no longer approved for use (occurs at step 2)

1.	The system displays message "That chemical is no longer approved for use"
2.	The system asks the LCA if he wants to request another chemical or to exit
3a.	The LCA asks to request another chemical
4a.	The system starts Normal Course again
3b.	The LCA asks to exit
4b.	The system terminates the use case

Summary Inputs	Source	Outputs	Destination
Chemical name or ID	LCA	Chemical Pick-up Authorization	LCA
List of approved chemicals	Lawn Chemicals Supply datastore		
Chemical quantity on hand	Lawn Chemicals Supply datastore	Chemical Pick-up Notice	Chemical Supply Warehouse
Quantity needed	LCA		
Request confirmation	LCA	Lawn Chemical Request	Chemical Request datastore
Request quantity available or cancellation	LCA	Chemical Outage Notice	Purchasing

Figure 2


2.1
Determine Chemical Approval Status
Chemical needed
Chemical approval status
Unapproved chemical
Approved chemical
D1
Lawn Chemicals Supply
Quantity available
Current reserved quantity
Updated reserved quantity
2.2
Determine Request Quantity
Chemical quantity available
Chemical quantity needed
2.4
Update Available Chemical Quantity
Lawn Chemical Applicator (LCA)
Valid chemical quantity
Confirmed chemical request quantity
Chemical outage
Confirmation
2.3
Confirm Request
2.5
Prepare Pick-up Material
Confirmed chemical request
Purchasing
Request for confirmation
New chemical request
Pick-up notice
D2
Chemical Requests
Chemical pick-up authorization
Chemical Supply Warehouse

Reading Data Flow Diagrams

Most people from Western cultures start reading diagrams from left to right, top to bottom. So, whenever possible, this is where most analysts try to make the DFD begin. The first item on the left side of Figure is the “Lawn Chemical Applicator (LCA)” external entity, which is a rectangle that represents individual employees who must request the chemicals they will use for their lawn care assignments. Th is symbol has three arrows pointing away from it to rounded rectangular symbols. Th ese arrows represent data flows and show that the external entity (LCA) provides three “bundles” of data to processes that use the data.

Now look at the arrow that flows in to the “Determine Chemical Approval Status” process from the right side. In order to determine if the chemical requested is approved for usage, the process has to retrieve some information from storage. Th e open-ended rectangle labeled “Lawn Chemical Supply” is called a data store, and it represents a collection of stored data. Th e “Determine Chemical Approval Status” process uses an identifier for the chemical requested to find the requested chemical and to determine whether it is an approved chemical or a disapproved chemical.

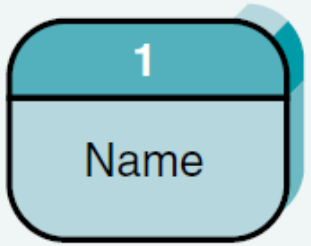
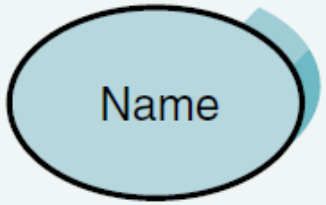
Now look more closely at the Major Steps Performed section of the use case. You can see that a number of steps are listed in the use case. On the DFD, these steps have been organized into five major processes, each performing one main component of the interactions detailed on the use case. On the DFD, as you follow the arrows starting with “Chemical needed” from the LCA to the “Determine Chemical Approval Status” process, imagine the LCA specifying the chemical he needs for a job. Th e system looks up the chemical and responds with a message verifying it as an approved chemical or informing the LCA that the chemical cannot be used. For an approved chemical, the system looks up how much of the chemical is available and informs the LCA. The LCA indicates the quantity he wants.

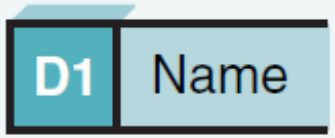
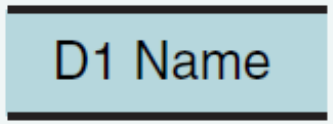
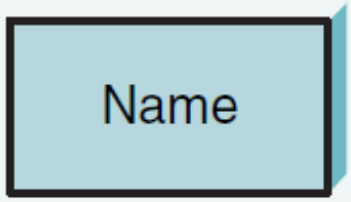
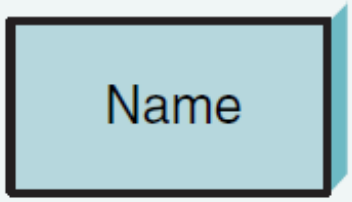
Now look at the description on the use case for steps 1–4 and notice how the use case describes those processes in words. Notice also how the “Information for Steps” section of the use case lists the data elements that are either used or produced by each step, corresponding to the inflows and outflows from the process symbols (2.1, 2.2) on the DFD.

Look at the other three process symbols in the DFD and examine the flows into and out of each process. On the basis of the data fl owing in and fl owing out, try to understand what the process is doing. Check your understanding by looking at the Major Steps Performed and Information for Steps in the use case.

You probably recognized that the “Confirm Request” process (2.3) receives the LCA’s chemical request confirmation and creates and stores a new Chemical Request. You can also see that two additional processes are performed by the system. Th e quantity of the chemical available is modified by marking the quantity requested as “reserved” (and no longer available to another LCA). Finally, the pick-up authorization is provided to the LCA and the Chemical Supply Warehouse is notified of the approved pick-up. You can see by the flows from process 2.3 to processes 2.4 and 2.5 that sometimes a process sends a data fl ow directly to another process.

Elements of Data Flow Diagrams

Data Flow Diagram Element	Typical Computer-Aided Software Engineering Fields	Gane and Sarson Symbol	DeMarco and Yourdon Symbol
Every *process* has a number a name (verb phase) a description at least one output data flow at least one input data flow	Label (name) Type (process) Description (what is it) Process number Process description (structured English) Notes		
Every *data flow* has a name (a noun) a description one or more connections to a process	Label (name) Type (flow) Description Alias (another name) Composition (description of data elements) Notes		

Every *data store* has a number a name (a noun) a description one or more input data flows one or more output data flows	Label (name) Type (store) Description Alias (another name) Composition (description of data elements) Notes		
Every *external entity* has a name (a noun) a description	Label (name) Type (entity) Description Alias (another name) Entity description Notes		

Process

A *process* is an activity or a function that is performed for some specific business reason. Processes can be manual or computerized. Every process should be named starting with a verb and ending with a noun (e.g., “Determine request quantity”). Names should be short, yet contain enough information so that the reader can easily understand exactly what they do. In general, each process performs only one activity, so most system analysts avoid using the word “and” in process names because it suggests that the process performs several activities. In addition, every process must have at least one input data flow and at least one output data flow.

Data Flow

Data flow is a single fact, such as quantity available (sometimes called a data element), or a logical collection of several facts (e.g., new chemical request). Every data flow should be named with a noun. The description of a data flow lists exactly what data elements the flow contains. For example, the pick-up notice data flow would list the LCA name, chemical, and quantity requested as its data elements. Data flows are the glue that holds the processes together. One end of every data flow will always come from or go to a process, with the arrow showing the direction into or out of the process. Data flows show what inputs go into each process and what outputs each process produces. Every process must create at least one output data flow, because if there is no output, the process does not do anything. Likewise, each process has at least one input data flow, because it is difficult, if not impossible, to produce an output with no input.

Data Store

A *data store* is a collection of data that is stored in some way (which is determined later when creating the physical model). Every data store is named with a noun and is assigned an identification number and a description. Data stores form the starting point for the data model (discussed in the next chapter) and form the logical connection between the process model and the data model.

All data stores must have at least one input data flow (or else they never contain any data), unless they are created and maintained by another information system or on another page of the DFD. Likewise, they have at least one output data flow on some page of the DFD. (Why store data if you never use it?) In cases in which the same process both stores data and retrieves data from a data store, there is a temptation to draw one data flow with an arrow on both ends. Th is practice is incorrect, however. Th e data fl ow that stores data and the data flow that retrieves data should always be shown as two separate data flows.

External Entity

An *external entity* is a person, organization, organization unit, or system that is external to the system, but interacts with it (e.g., customer, clearinghouse, government organization, accounting system). The external entity typically corresponds to the primary actor identified in the use case. External entities provide data to the system or receive data from the system, and serve to establish the system boundaries. Every external entity has a name and a description. Th e key point to remember about an external entity is that it is external to the system, but may or may not be part of the organization. People who use the information from the system to perform other processes or who decide what information goes into the system are documented as external entities (e.g., managers, staff).

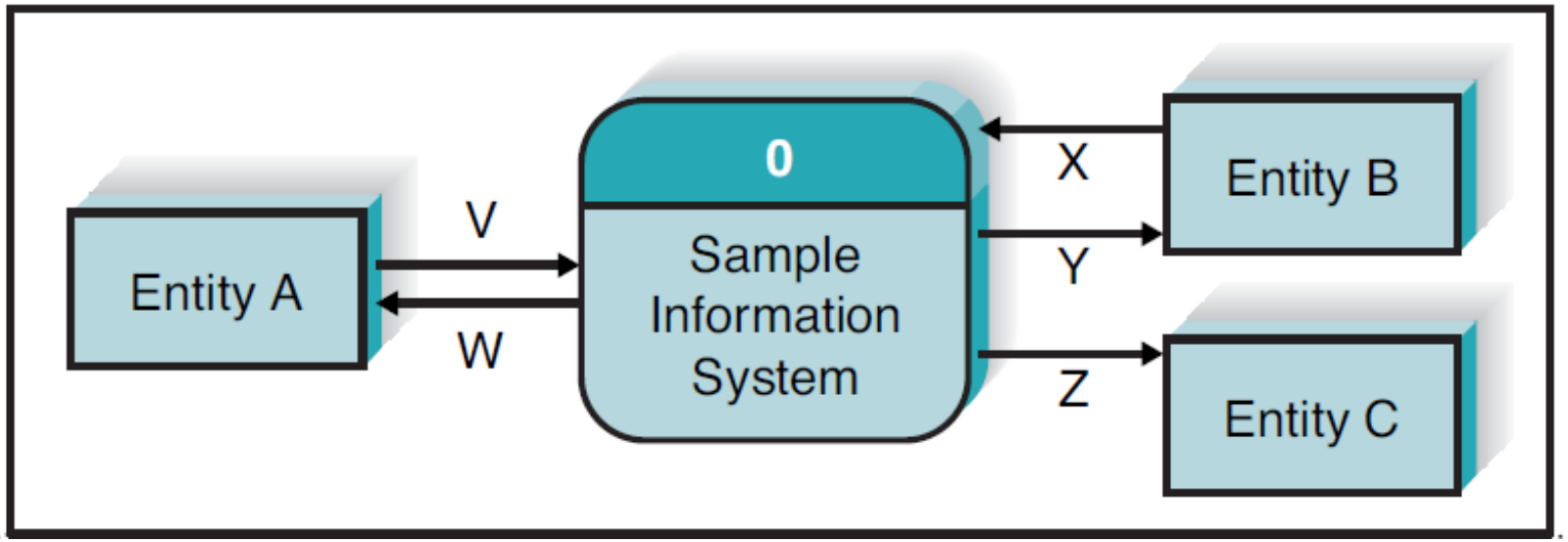
Using Data Flow Diagrams to Define Business Processes

Most business processes are too complex to be explained in one DFD. Most process models are therefore composed of a set of DFDs. Th e first DFD provides a summary of the overall system, with additional DFDs providing more and more detail about each part of the overall business process. Thus, one important principle in process modeling with DFDs is the decomposition of the business process into a hierarchy of DFDs, with each level down the hierarchy consisting of diagrams with less scope but more detail.

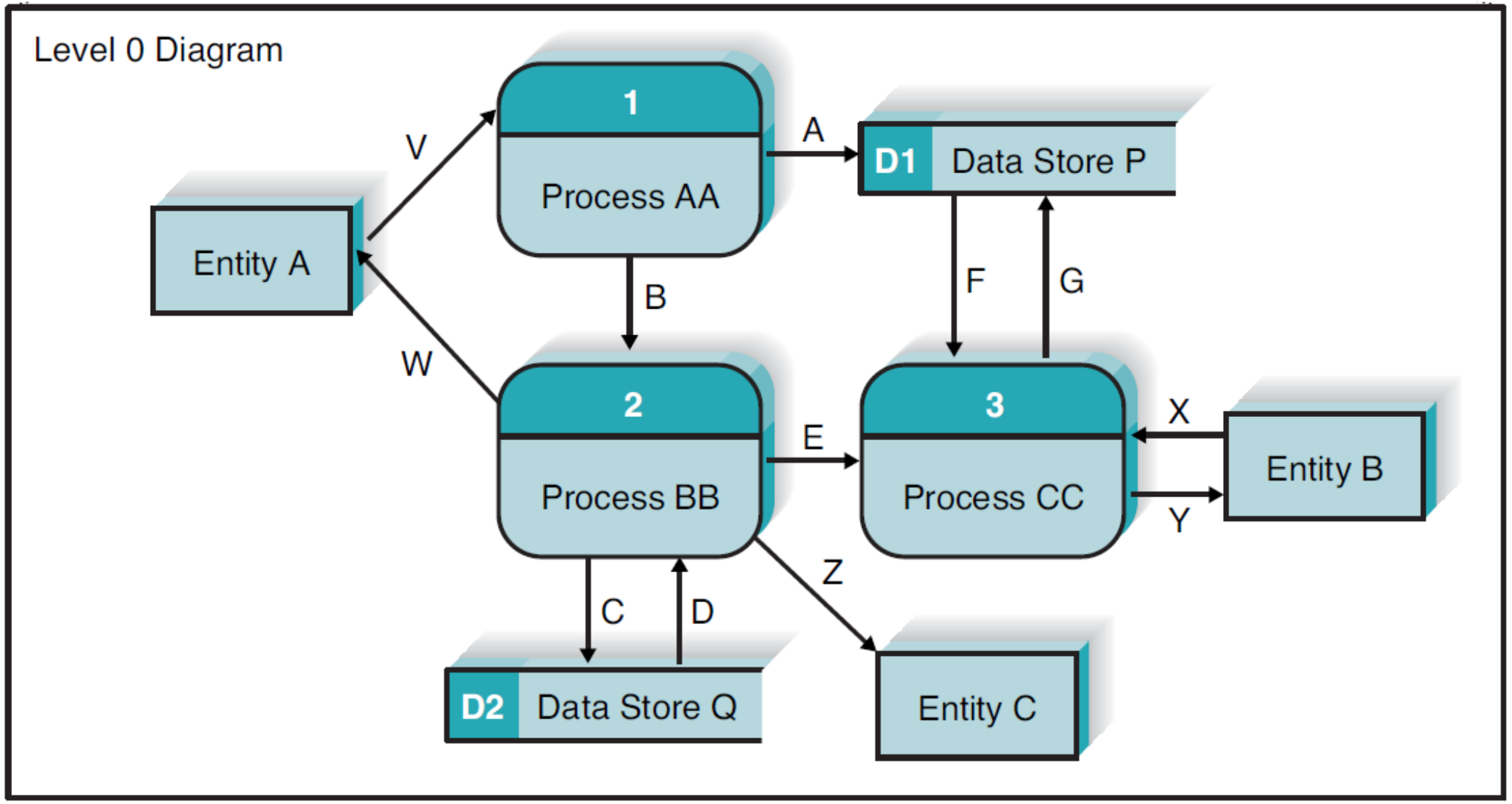
Context Diagram

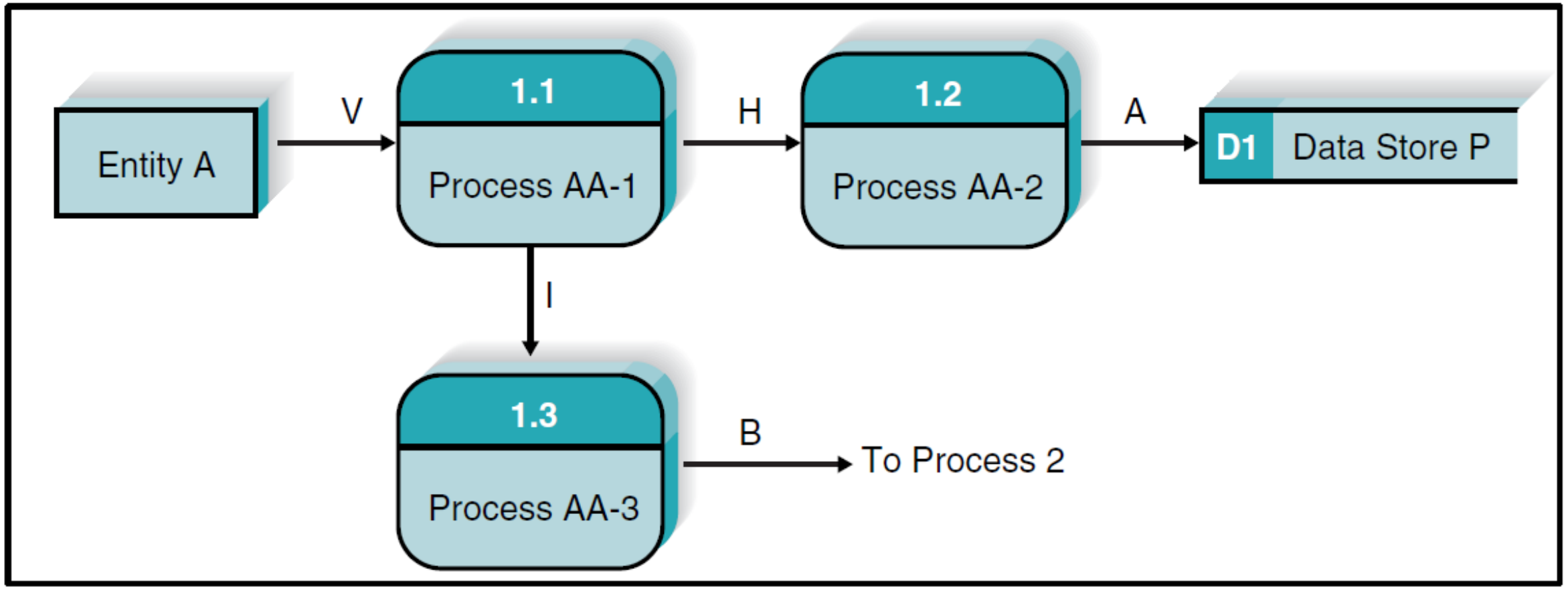
The top-level DFD in every business process model, whether a manual system or a computerized system, is the *context diagram.*

Context Diagram

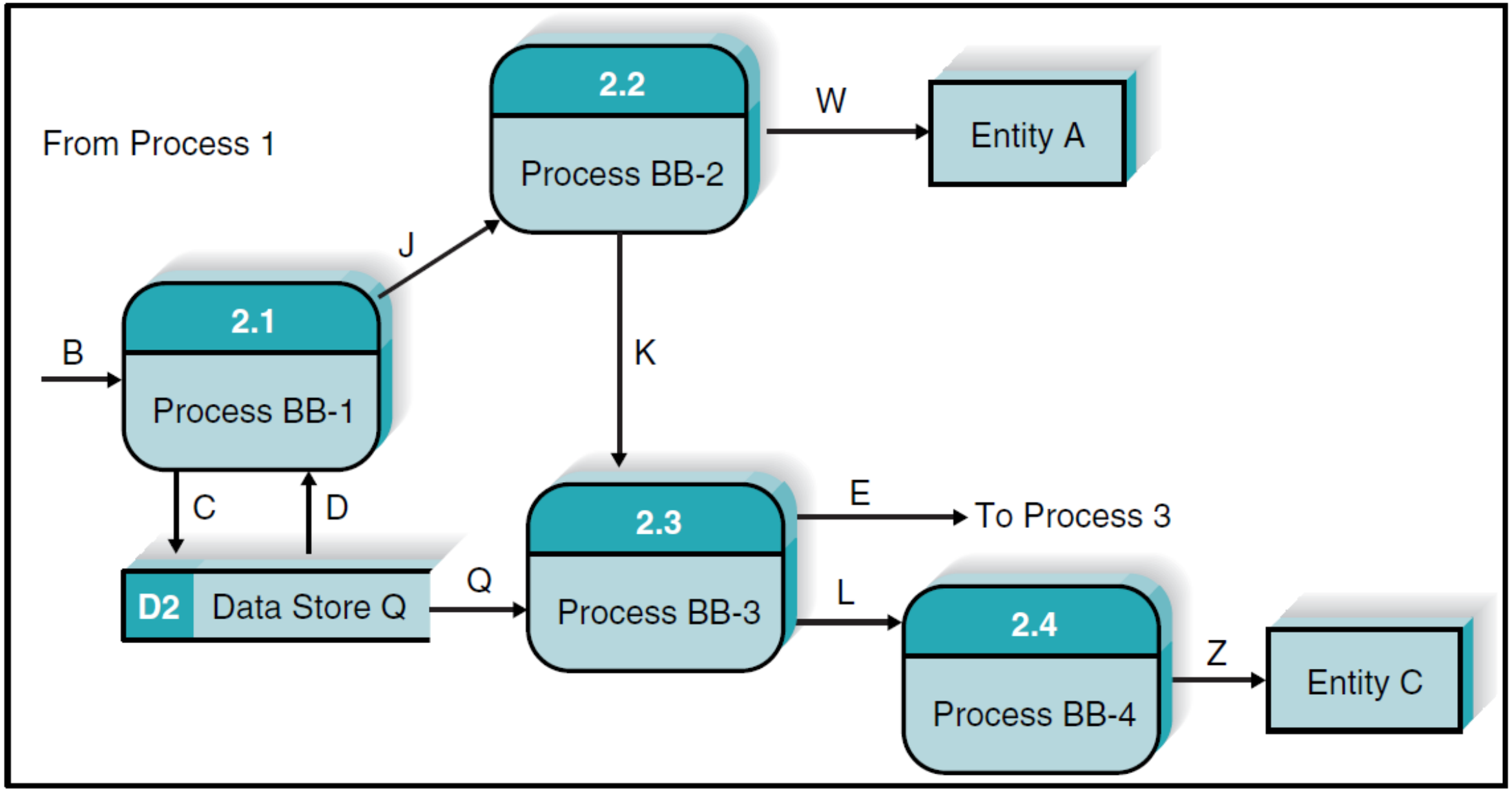


Context Diagram Decomposed into Level 0 Diagram

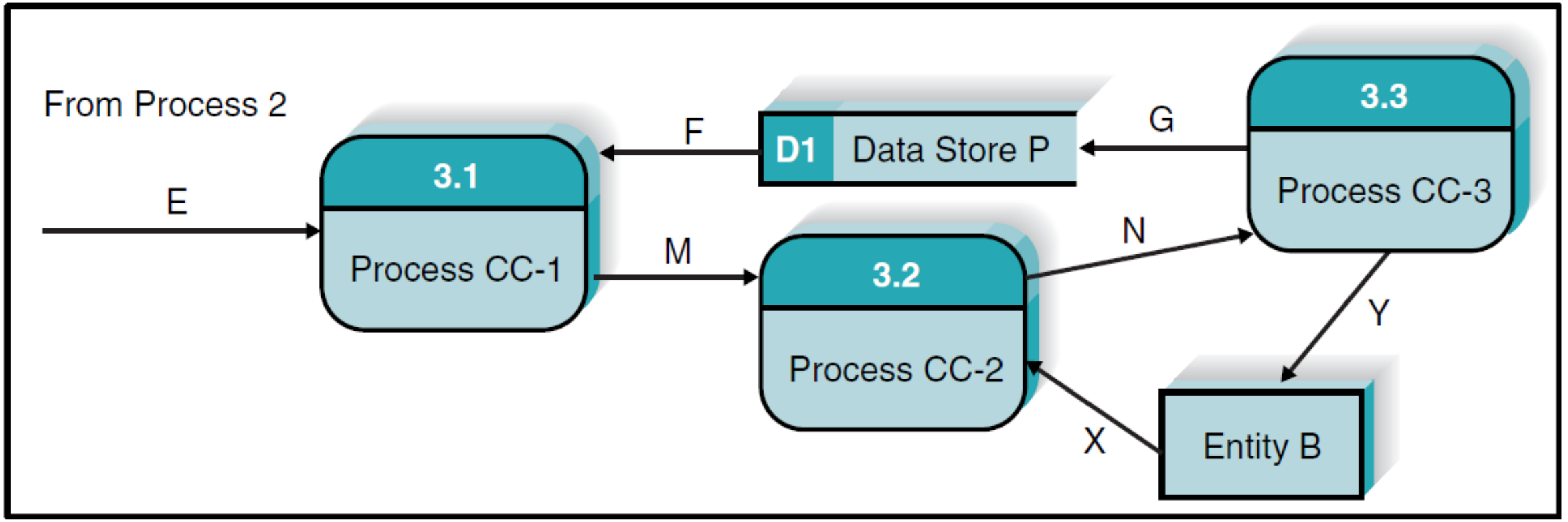




Process 1 Level 1 Diagram



Process 2 Level 1 Diagram



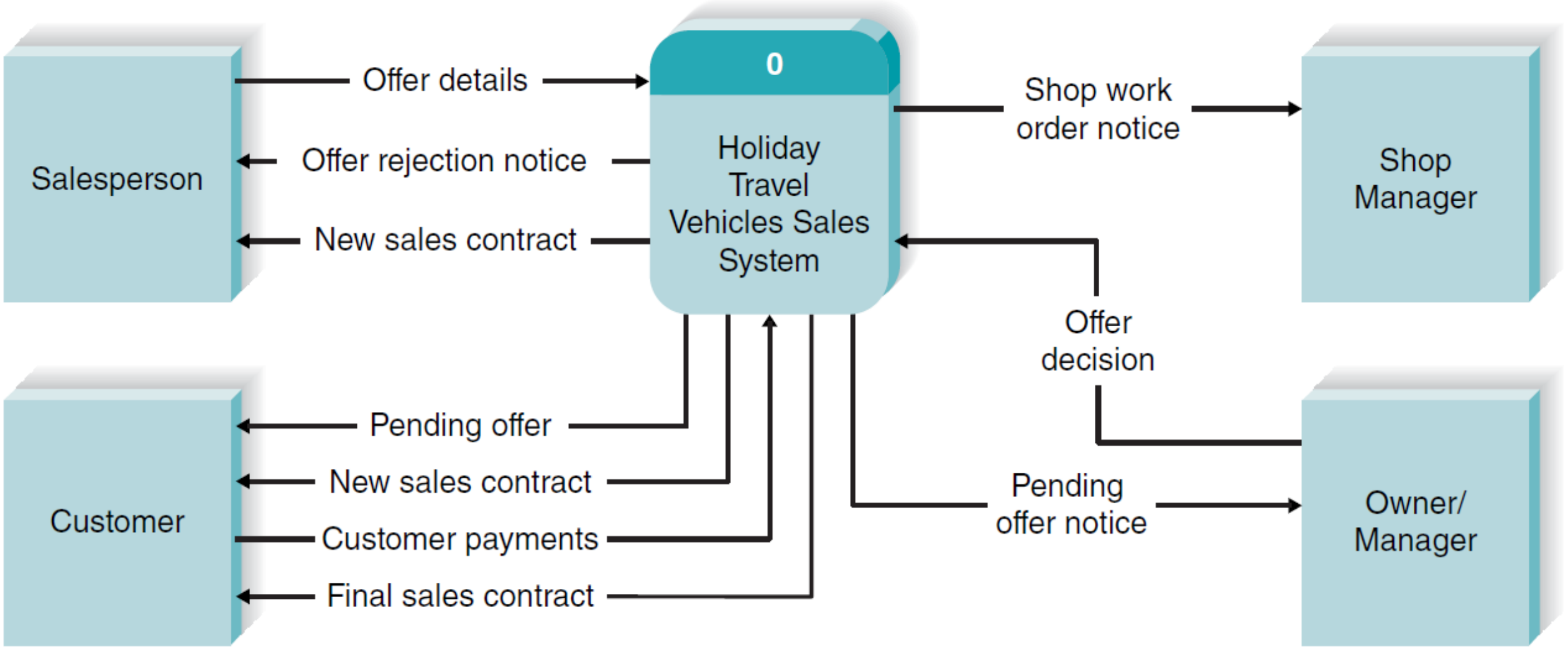
Process 3 Level 1 Diagram

Creating the Context Diagram

The context diagram defines how the business process or computer system interacts with its environment—primarily the external entities.

To create the context diagram, you simply draw one process symbol for the business process or system being modeled (numbered 0 and named for the process or system).

Holiday Travel Vehicles Sales System Context Diagram



Creating Data Flow Diagram Fragments

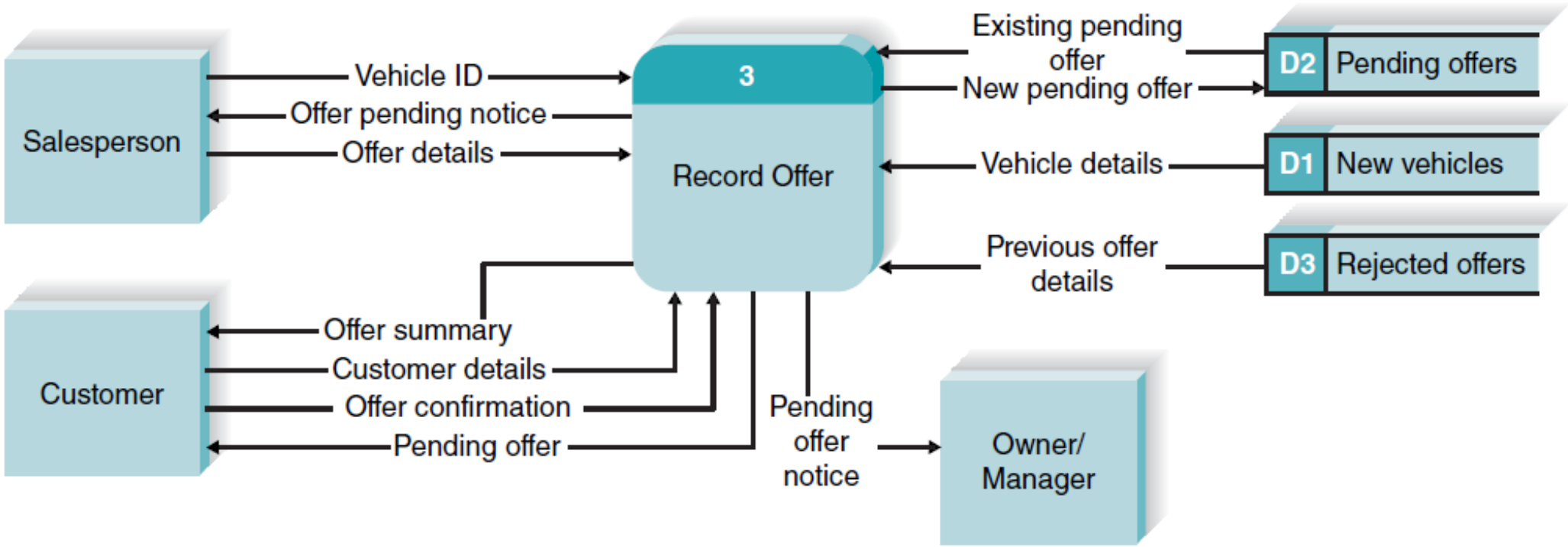
A *DFD fragment* is one part of a DFD that eventually will be combined with other DFD fragments to form a DFD diagram. In this step, each use case is converted into one DFD fragment. You start by taking each use case and drawing a DFD fragment, using the information given on the top of the use case: the name, ID number, and major inputs and outputs.

The information about the major steps that make up each use case is ignored at this point; it will be used in a later step.

Use Case Name: Record an offer	ID: UC-3	Priority: High
Actor: Salesperson		
Description: This use case describes how the salesperson records a customer offer on a vehicle. The offer may be a new offer or a revision of a previously rejected offer.		
Trigger: Customer decides to make an offer on a vehicle. Type: ☑ External ☐ Temporal		

Summary Inputs	Source	Outputs	Destination
Vehicle ID Existing Pending Offer	Salesperson Pending Offers	Offer Pending Notice Offer Summary	Salesperson Customer

Holiday Travel Vehicles Process 3 (Record Offer) DFD Fragment



Creating the Level 0 Data Flow Diagram

Once you have the set of DFD fragments (one for each of the major use cases), you combine them into one DFD drawing that becomes the level 0 DFD. As mentioned earlier, there are no formal layout rules for DFDs. Most systems analysts try to put the process that is first chronologically in the upper left corner of the diagram and work their way from top to bottom, left to right.

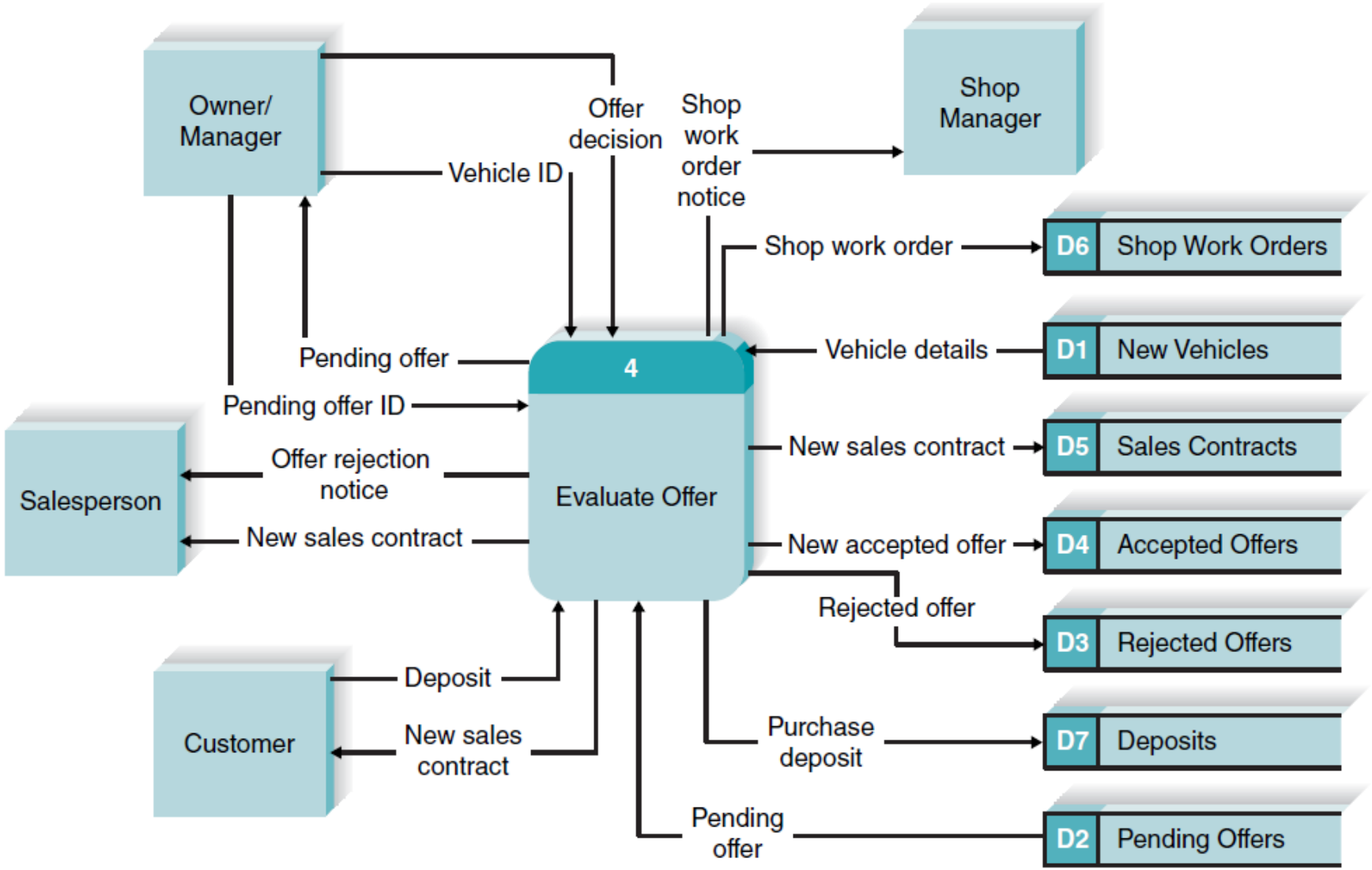
Generally speaking, most analysts try to reduce the number of times that data fl ow lines cross or to ensure that when they do cross, they cross at right angles so that there is less confusion. (Many give one line a little “hump” to imply that one data flow jumps over the other without touching it.) Minimizing the number of data flows that cross is challenging.

Iteration is the cornerstone of good DFD design. Even experienced analysts seldom draw a DFD perfectly the fi rst time. In most cases, they draw it once to understand the pattern of processes, data flows, data stores, and external entities and then draw it a second time on a fresh sheet of paper (or in a fresh fi le) to make it easier to understand and to reduce the number of data flows that cross. Oft en, a DFD is drawn many times before it is fi nished.

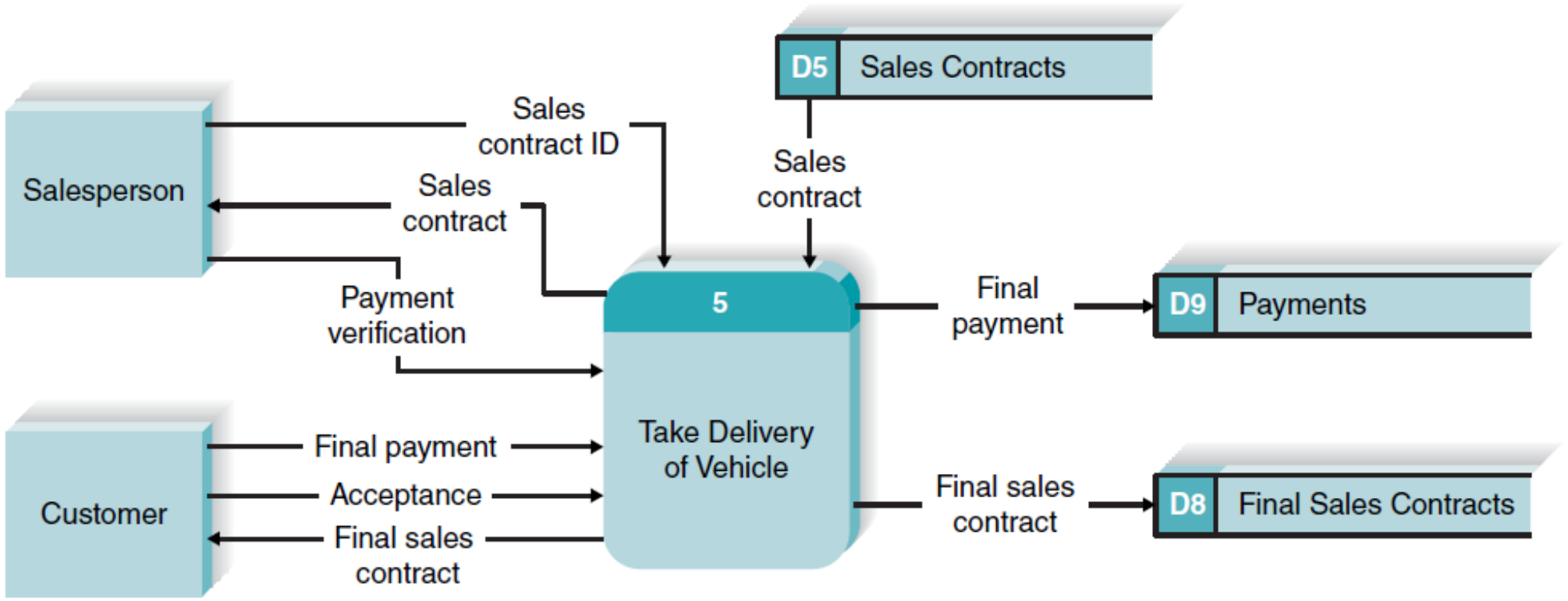
Creating Level 1 Data Flow Diagrams (and Below)

The team now begins to create lower-level DFDs for each process in the level 0 DFD that needs a level 1 DFD. Each one of the use cases is turned into its own DFD. The process for creating the level 1 DFDs is to take the steps as written on the use cases and convert them into a DFD in much the same way as for the level 0 DFD. Usually, each major step in the use case becomes a process on the level 1 DFD, with the inputs and outputs becoming the input and output data flows.

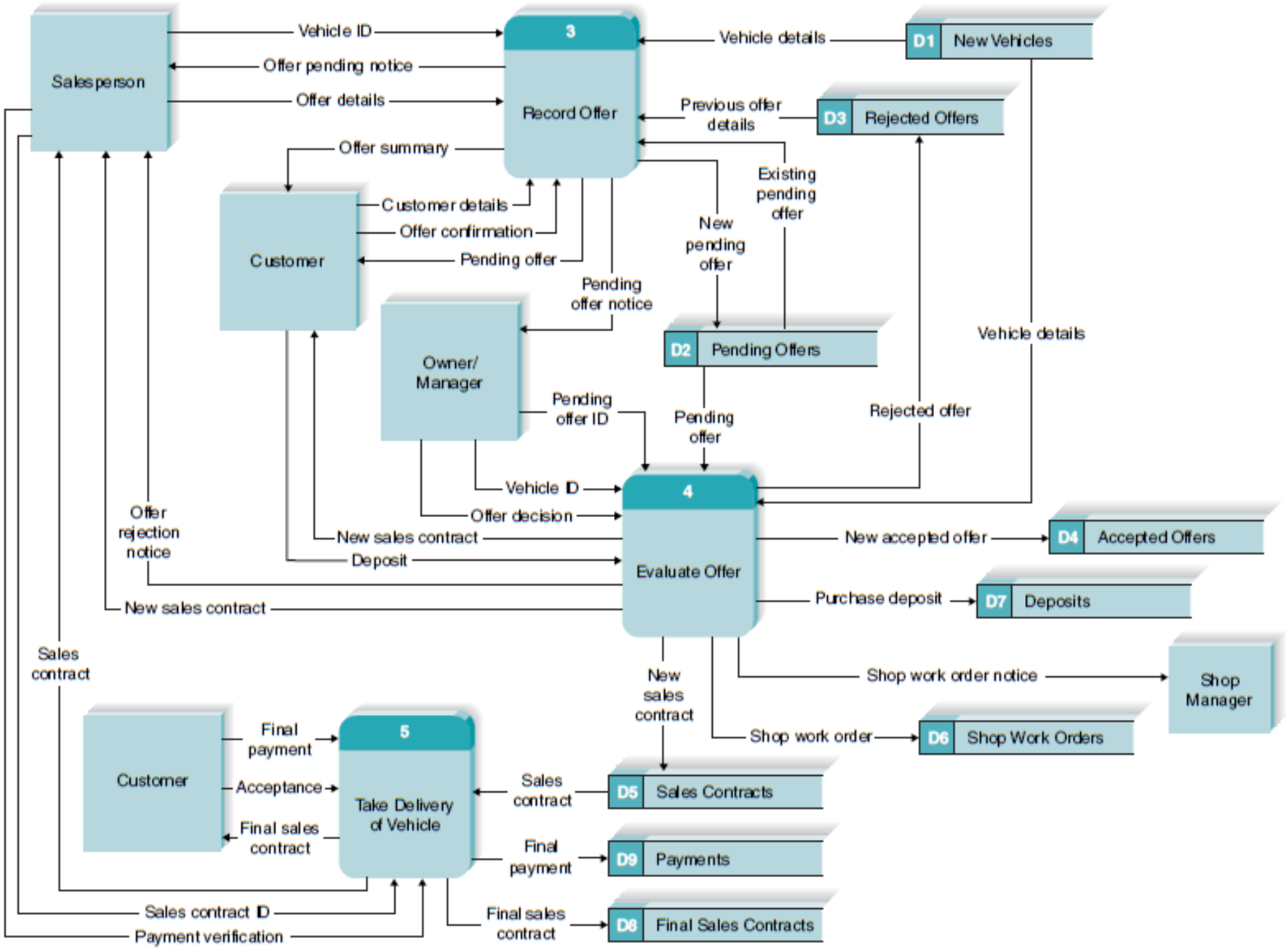
Additional DFD Fragments for Holiday Travel Vehicles



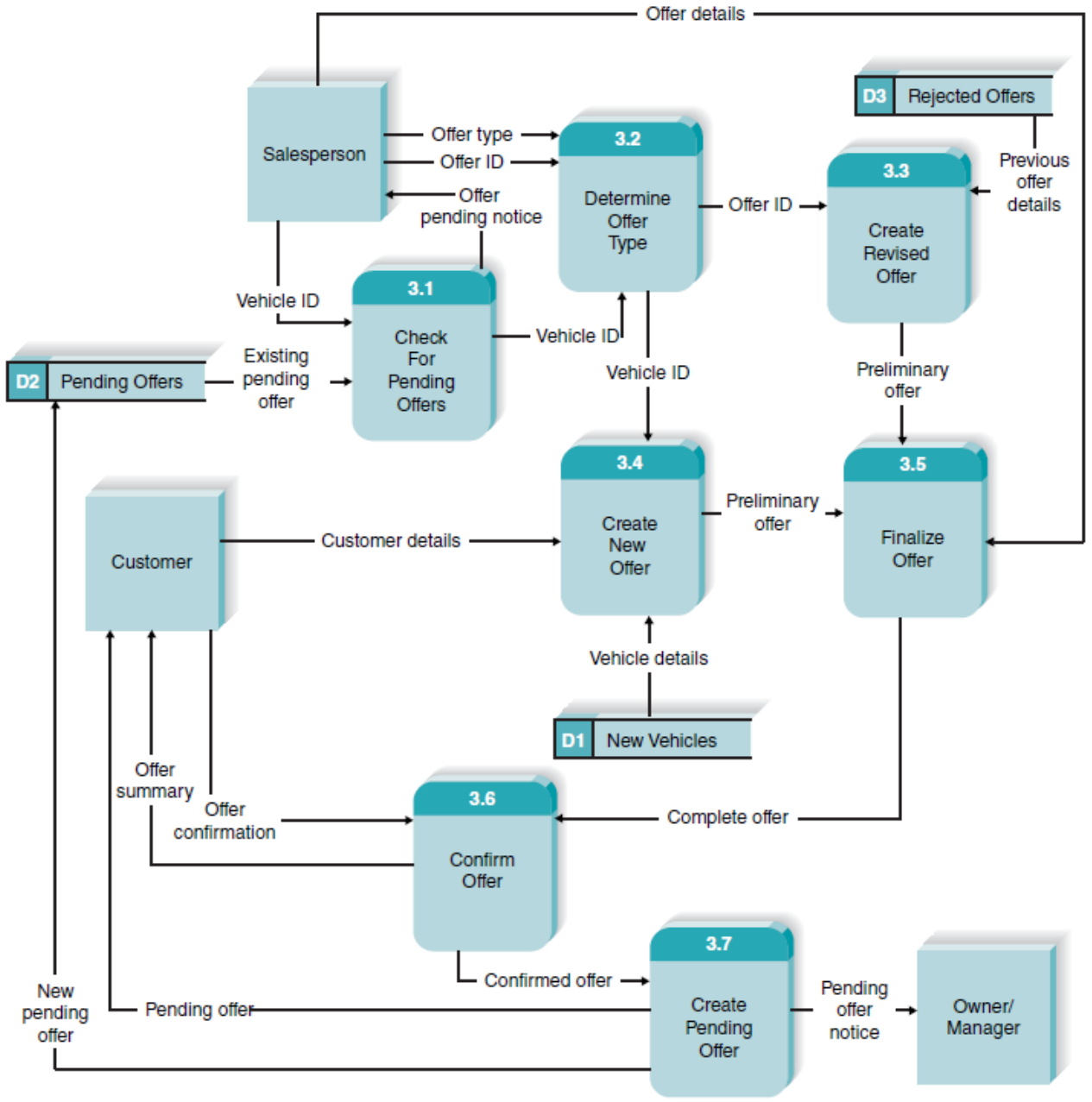
(a) *Process 4 DFD Fragment*



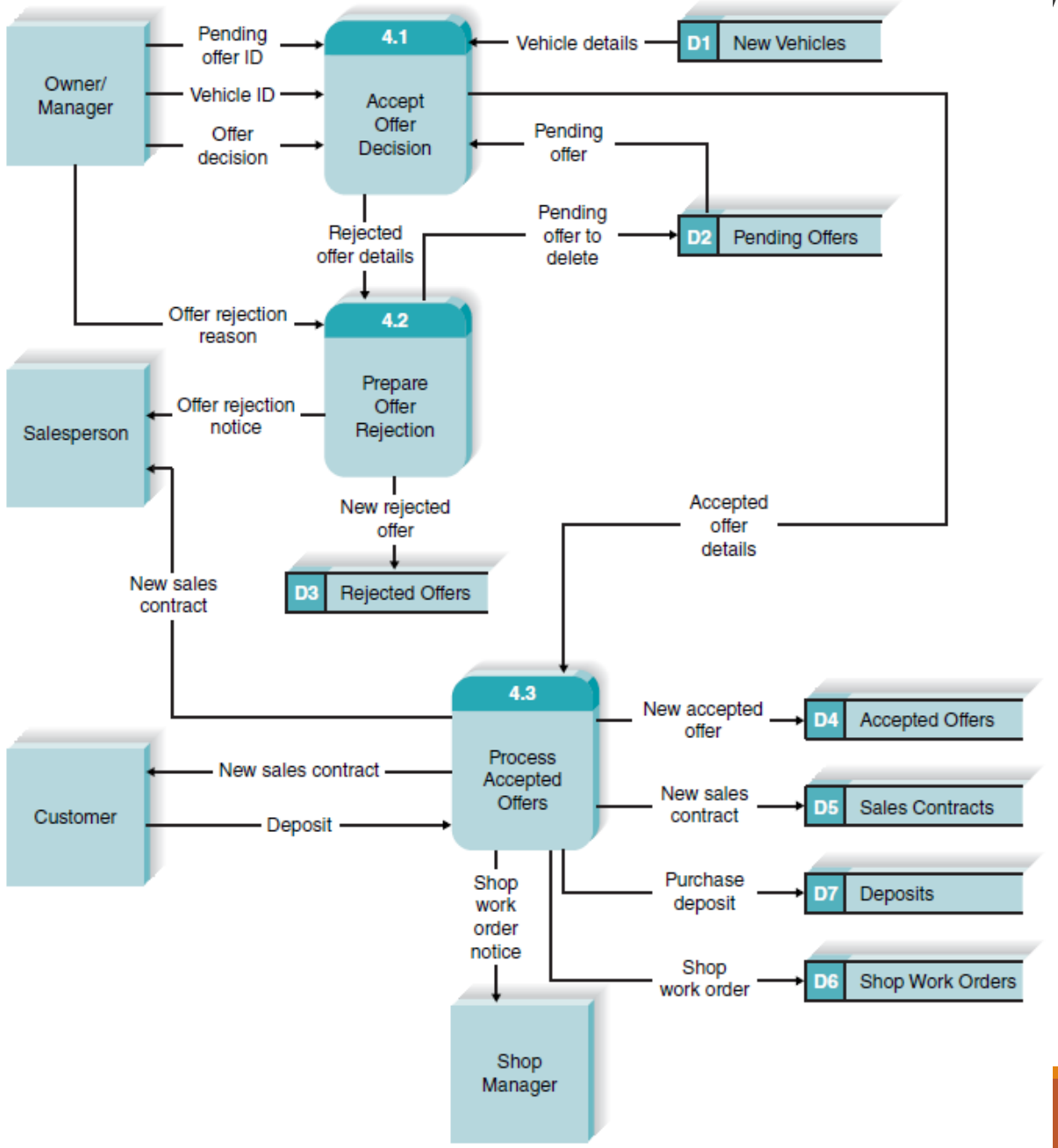
(*b*) *Process 5 DFD Fragment*


Salesperson
Vehicle ID
Offer pending notice
Offer details
3
Record Offer
Vehicle details
D1 New Vehicles
Previous offer details
D3 Rejected Offers
Offer summary
Existing pending offer
Customer
Customer details
Offer confirmation
Pending offer
New pending offer
Pending offer notice
D2 Pending Offers
Vehicle details
Owner/ Manager
Pending offer ID
Pending offer
Rejected offer
Vehicle ID
4
Offer decision
Offer rejection notice
New sales contract
Deposit
Evaluate Offer
New accepted offer
D4 Accepted Offers
Purchase deposit
D7 Deposits
New sales contract
Sales contract
New sales contract
Shop work order notice
Shop Manager
Shop work order
D6 Shop Work Orders
Customer
Final payment
Acceptance
Final sales contract
5
Take Delivery of Vehicle
Sales contract
D5 Sales Contracts
Final payment
D9 Payments
Sales contract ID
Final sales contract
D8 Final Sales Contracts
Payment verification

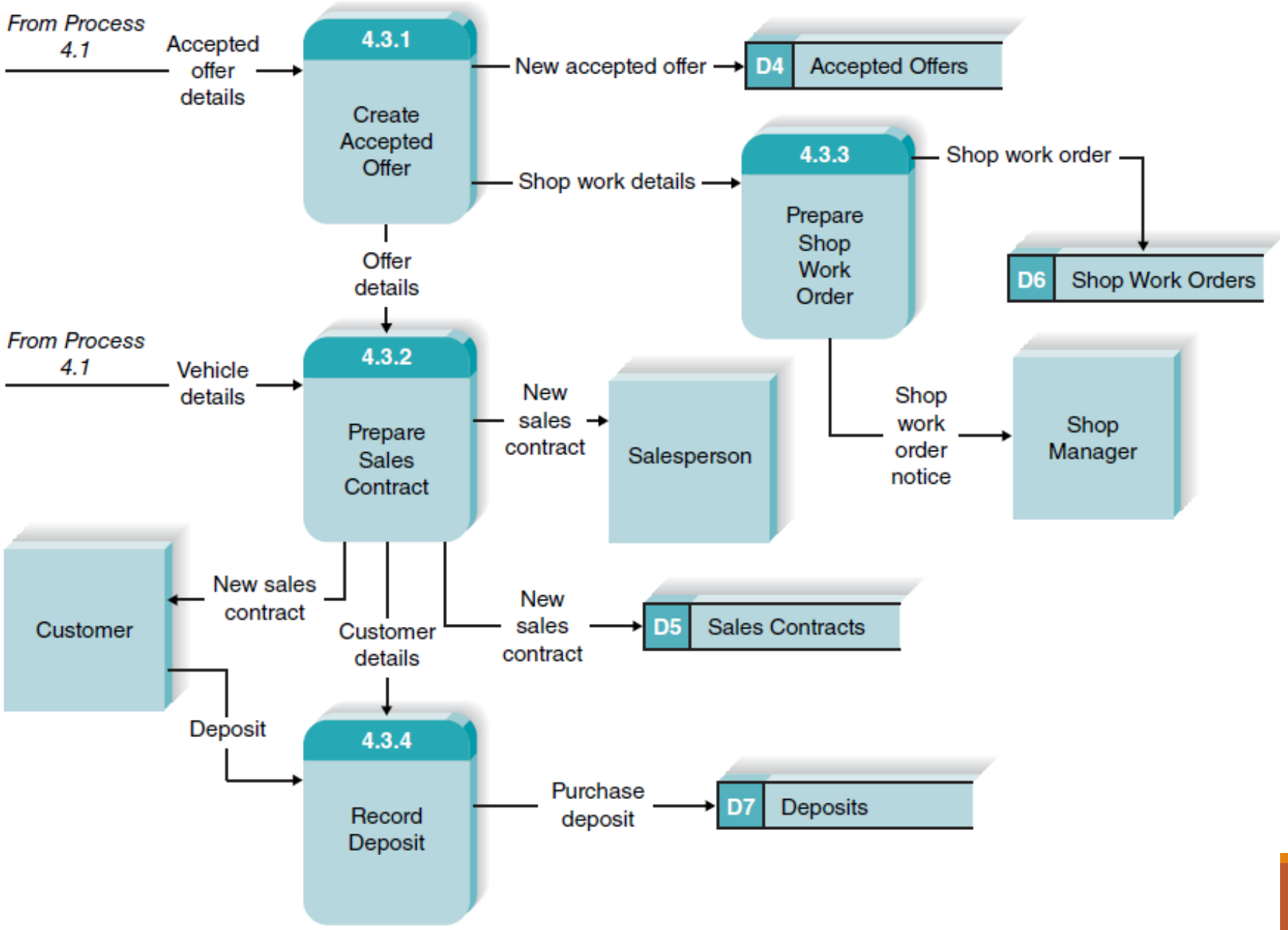
Holiday Travel Vehicles Process 3 (Record Offer) Level 1 DFD



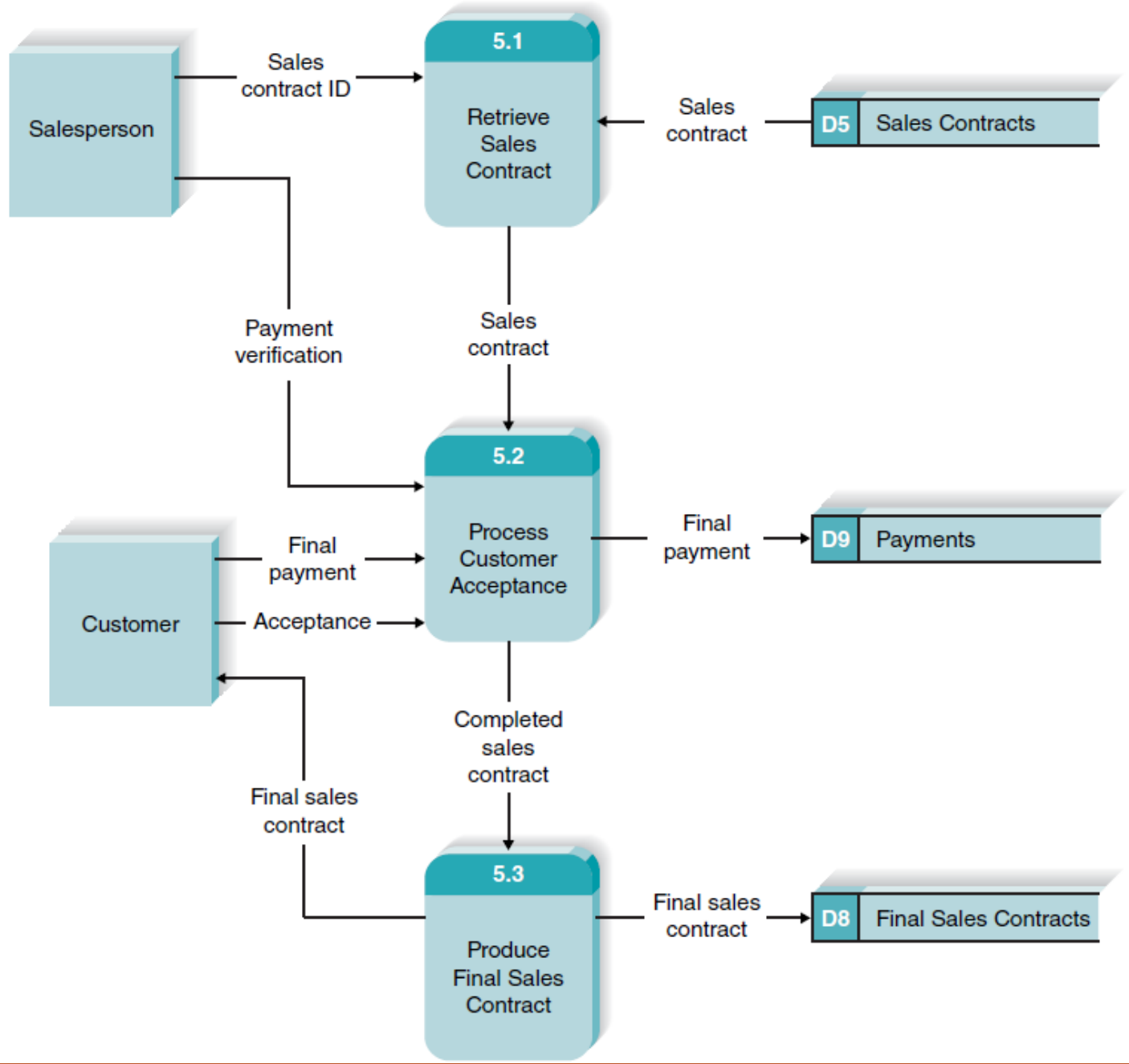
Holiday Travel Vehicles Process 4 (Evaluate Offer) Level 1 DFD



Holiday Travel Vehicles Process 4.3 (Process Accepted Offers) Level 2 DFD



Holiday Travel Vehicles Process 5 (Take Delivery of Vehicle) Level 1 DFD



Validating the Data Flow Diagrams

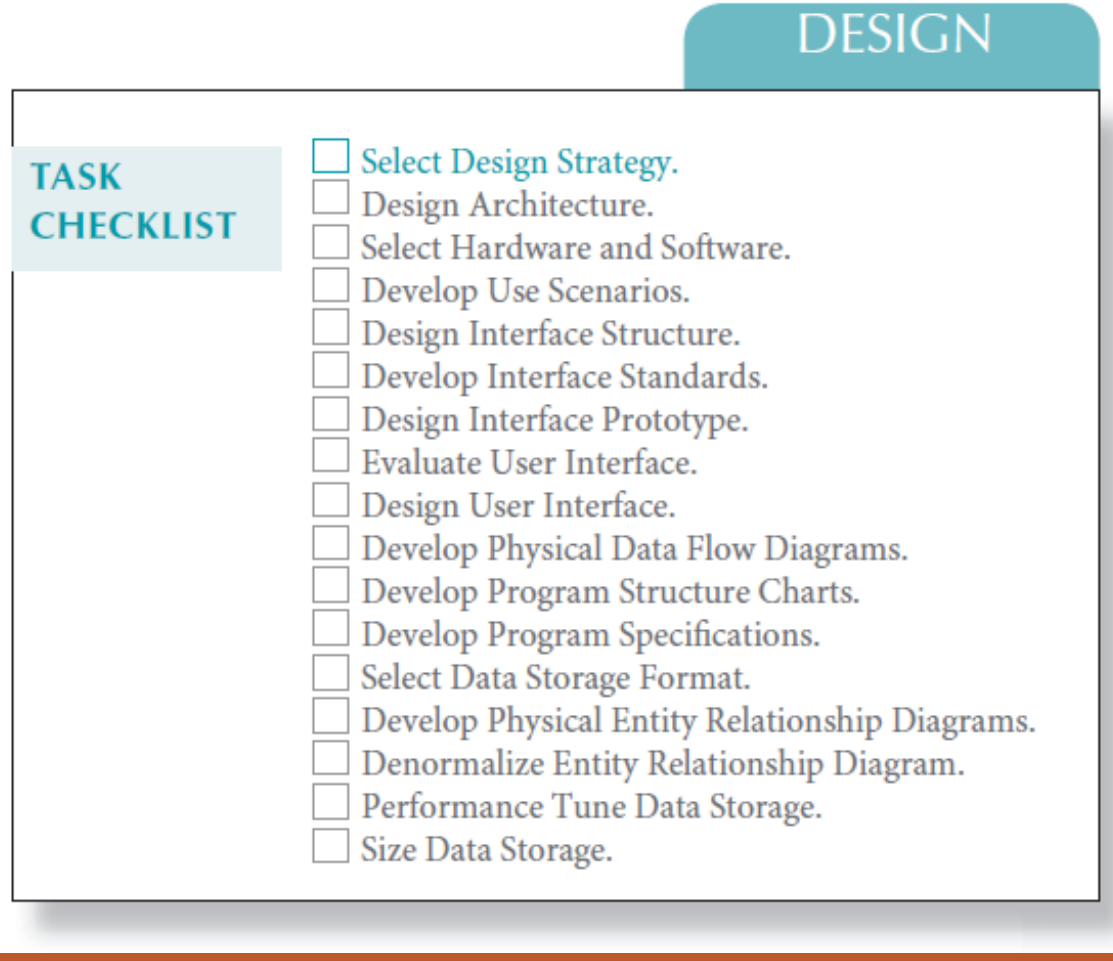
Syntax	
Within DFD	
Process	• Every process has a unique name that is an action-oriented verb phrase, a number, and a description.
	• Every process has at least one input data flow.
	• Every process has at least one output data flow.
	• Output data flows usually have different names than input data flows because the process changes the input into a different output in some way.
	• There are between three and seven processes per DFD.
Data Flow	• Every data flow has a unique name that is a noun, and a description.
	• Every data flow connects to at least one process.
	• Data flows only in one direction (no two-headed arrows).
	• A minimum number of data flow lines cross.
Data Store	• Every data store has a unique name that is a noun, and a description.
	• Every data store has at least one input data flow (which means to add new data or change existing data in the data store) on some page of the process model.
	• Every data store has at least one output data flow (which means to read data from the data store) on some page of the process model.
External Entity	• Every external entity has a unique name that is a noun, and a description.
	• Every external entity has at least one input or output data flow.
Across DFDs	
Context diagram	• Every set of DFDs must have one context diagram.
Viewpoint	• There is a consistent viewpoint for the entire set of DFDs.
Decomposition	• Every process is wholly and completely described by the processes on its children DFDs.
Balance	• Every data flow, data store, and external entity on a higher level DFD is shown on the lower-level DFD that decomposes it.
Semantics	
Appropriate Representation	• User validation
	• Role-play processes
Consistent Decomposition	• Examine lowest-level DFDs
Consistent Terminology	• Examine names carefully

System Analysis

DR. DRAGUNOV

Design

The design phase of the SDLC uses the requirements that were gathered during analysis to create a blueprint for the future system. A successful design builds on what was learned in earlier phases and leads to a smooth implementation by creating a clear, accurate plan of what needs to be done.



INTRODUCTION

The design phase decides *how* the new system will operate. Many activities will be involved as the development team develops the system requirements. This chapter provides an outline of those design phase activities, which culminates in the creation of the system specification.

We also describe three alternative strategies for acquiring the system that are available to the development team.

EVOLVING THE ANALYSIS MODELS INTO DESIGN MODELS

The purpose of the ***analysis phase*** is to figure out what the business needs.

The purpose of the ***design phase*** is to decide how to build it.

System design is the determination of the overall system architecture—consisting of a set of physical processing components, hardware, software, people, and the communication among them—that will satisfy the system's essential requirements.

During the initial part of design, the project team converts the business requirements for the system into *system requirements* that describe the technical details for building the system. Unlike business requirements, which are listed in the requirements definition and communicated through use cases and *logical* process and data models, system requirements are communicated through a collection of design documents and *physical* process and data models. Together, the design documents and physical models make up the blueprint for the new system.

We should note here that our focus is on the design of the technical system blueprint that will satisfy the system's requirements. An important element of the final, complete information system, however, will be redesigned work flows and procedures that users will follow when using the new system. Business analysts oft en turn their attention to the design of these components at this stage of the project, while systems analysts focus on more technical design elements.

Activities of the Design Phase

The design phase has a number of activities that lead to the system blueprint

Activities in the Design Phase	Deliverables	Chapter
✓ Determine preferred system acquisition strategy (make, buy, or outsource).	– Alternative matrix	7
✓ Design the architecture for the system.	– Architecture design	8
✓ Make hardware and software selections.	– Hardware and software specification	
✓ Design system navigation, inputs, and outputs.	– Interface design	9
✓ Convert logical process model to physical process model.	– Physical process model	10
✓ Update CASE repository with additional system details.	– Updated CASE repository	
✓ Design the programs that will perform the system processes.	– Program design specifications	
✓ Convert logical data model to physical data model.	– Physical data model	11
✓ Update CASE repository with additional system details.	– Updated CASE repository	
✓ Revise CRUD matrix.	– CRUD matrix	
✓ Design the way in which data will be stored.	– Data storage design	
✓ Compile final system specification.	– System specification: all of the above deliverables combined and presented to approval committee	7

At the end of the design phase, the project team creates the final deliverable for the phase called the *system specification.* Th is document contains all of the design documents just described: physical process models, physical data model, architecture design, hardware and soft ware specification, interface design, data storage design, and program design. Collectively, the system specification conveys exactly what system the project team will implement during the implementation phase of the SDLC.

- Recommended System Acquisition Strategy
- System Acquisition Weighted Alternative Matrix
- Architecture Design
- Hardware and Software Specification
- Interface Design
- Physical Process Model
- Program Design Specifications
- Physical Data Model
- Data Storage Design
- Updated CRUD Matrix
- Updated CASE Repository Entries

DESIGN STRATEGIES

Until now, we have assumed that the system will be built and implemented by the project team; however, there are actually three ways to approach the creation of a new system:

- developing a custom application in-house
- buying a packaged system and customizing it
- relying on an external vendor, developer, or service provider to build the system

SYSTEM ACQUISITION STRATEGIES

Pros	Cons
Custom Development	
Get **exactly** what we want New system built consistently with existing technology and standards Build and retain technical skills and functional knowledge in-house Allows team flexibility and creativity Unique solutions created for strategic advantage	Requires significant time and effort May add to existing backlogs May require skills we do not have Often costs more Often takes more calendar time Risk of project failure
Packages (purchased or obtained from ASP or SaaS)	
No need to "reinvent the wheel" for common business needs Tested, proven product Cost savings Time savings Utilize vendors' expertise Some customization may be possible	Rarely a perfect fit Organizational processes must adapt to software Reliance on vendor for maintenance and future enhancements Will not develop in-house functional and technical skills Unique needs may go unmet May require system integration

Outsourced Development	
Hire expertise we do not have May save time and money Lower risk	No opportunity to build in-house expertise Reliance on vendor Future options limited Security—potential loss of confidential info Performance based on contract terms

Custom Development

Many project teams assume that *custom development,* or building a new system from scratch, is the best way to create a system. For one thing, teams have complete control over the way the system looks and functions. Custom development also allows developers to be flexible and creative in the way they solve business problems. Additionally, a custom application would be easier to change to include components that take advantage of current technologies that can support such strategic efforts.

Building a system in-house also builds technical skills and functional knowledge within the company.

The risks associated with building a system from the ground up can be quite high, and there is no guarantee that the project will succeed. Developers could be pulled away to work on other projects, technical obstacles could cause unexpected delays, and the business users could become impatient with a growing timeline.

Packaged Software

Many business needs are not unique, and because it makes little sense to reinvent the wheel, many organizations buy *packaged software* that has already been written rather than developing their own custom solution. In fact, there are thousands of commercially available software programs that have already been written to serve a multitude of purposes. Think about your own need for a word processor—did you ever consider writing your own word processing software? That would be very silly considering the number of good software packages available that are relatively inexpensive.

Outsourcing

The design choice that requires the least amount of in-house resources is *outsourcing*— hiring an external vendor, developer, or service provider to create the system. Outsourcing has become quite popular in recent years. Some estimate that as many as 50 percent of companies with IT budgets of more than $5 million are currently outsourcing or evaluating the approach.

Selecting a Design Strategy

Each of the design strategies just discussed has its strengths and weaknesses, and no one strategy is inherently better than the others. Thus, it is important to understand the strengths and weaknesses of each strategy and when to use each. Figure summarizes the characteristics of each strategy.

	Use Custom Development When…	Use a Packaged System When…	Use Outsourcing When…
Business Need	The business need is unique.	The business need is common.	The business need is not core to the business.
In-house Experience	In-house functional and technical experience exists.	In-house functional experience exists.	In-house functional or technical experience does not exist.
Project Skills	There is a desire to build in-house skills.	The skills are not strategic.	The decision to outsource is a strategic decision.
Project Management	The project has a highly skilled project manager and a proven methodology.	The project has a project manager who can coordinate the vendor's efforts.	The project has a highly skilled project manager at the level of the organization that matches the scope of the outsourcing deal.
Time frame	The time frame is flexible.	The time frame is short.	The time frame is short or flexible.

DEVELOPING THE ACTUAL DESIGN

Once the project team has a good understanding of how well each design strategy fits with the project's needs, it must begin to understand exactly *how* to implement these strategies. For example:

What tools and technology would be used if a custom alternative were selected?

What vendors make packaged systems that address the project's needs?

What service providers would be able to build this system if the application were outsourced?

It is likely that the project team will identify several ways that a system could be constructed after weighing the specific design options. For example, the project team might have found three vendors that make packaged systems that potentially could meet the project's needs.

Alternative Matrix

An *alternative matrix* can be used to organize the pros and cons of the design alternatives so that the best solution will be chosen in the end. This matrix is created using the same steps as the **feasibility analysis.**

To create the alternative matrix, draw a grid with the alternatives across the top and different criteria (e.g., feasibilities, pros, cons, and other miscellaneous criteria) along the side. Next, fill in the grid with detailed descriptions about each alternative. This becomes a useful document for discussion because it clearly presents the alternatives being reviewed and comparable characteristics for each one.

Sample Alternative Matrix Using Weights

Evaluation Criteria	Relative Importance (Weight)	Alternative 1: Custom Application Using VB.NET	Score (1–5)*	Weighted Score	Alternative 2: Custom Application Using Java	Score (1–5)*	Weighted Score	Alternative 3: Packaged Software Product ABC	Score (1–5)*	Weighted Score
Technical Issues:										
Criterion 1	20		5	100		3	60		3	60
Criterion 2	10		3	30		3	30		5	50
Criterion 3	10		2	20		1	10		3	30
Economic Issues:										
Criterion 4	25	Supporting	3	75	Supporting	3	75	Supporting	5	125
Criterion 5	10	Information	3	30	Information	1	10	Information	5	50
Organizational Issues										
Criterion 6	10		5	50		5	50		3	30
Criterion 7	10		3	30		3	30		1	10
Criterion 8	5		3	15		1	5		1	5
TOTAL	100			350			270			360

* This denotes how well the alternative meets the criteria. 1 = poor fit; 5 = perfect fit.

An important component of the design of an information system is the design of the physical architecture layer, which describes the system's hardware, software, and network environment.

The physical architecture layer design flows primarily from the nonfunctional requirements, such as operational, performance, security, cultural, and political requirements. The deliverable from the physical architecture layer design includes the architecture and the hardware and software specification.

Architectural Components

The major *architectural components* of any system:

- Software.
- Hardware.

Software basic functions:

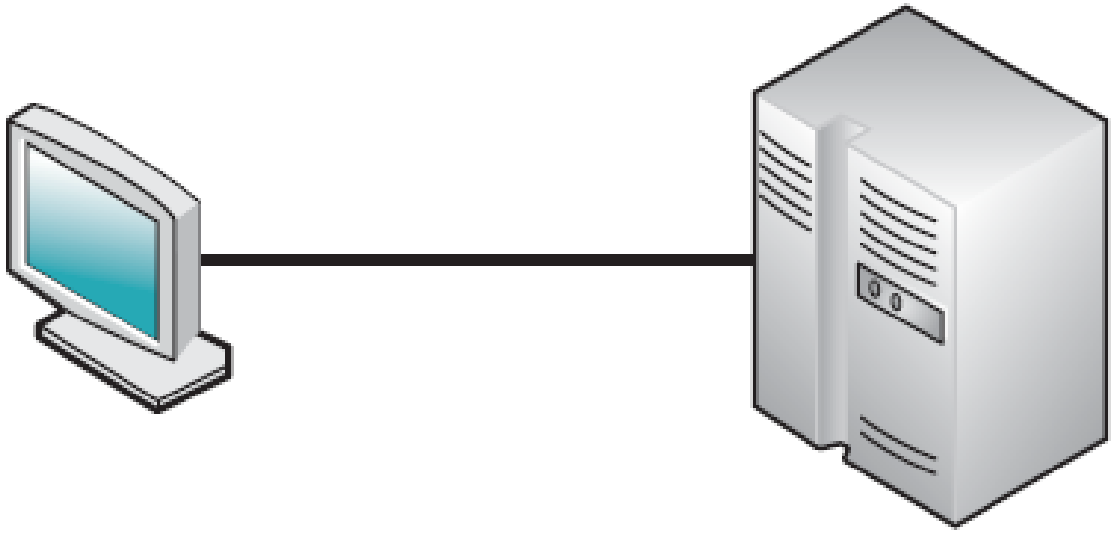
- *data storage*
- *data access logic*
- *application logic*
- *presentation logic*

Primary hardware components:

- *Client computers*
- *Servers*
- *Network*

Server-Based Architectures

The very first computing architectures were server-based architectures

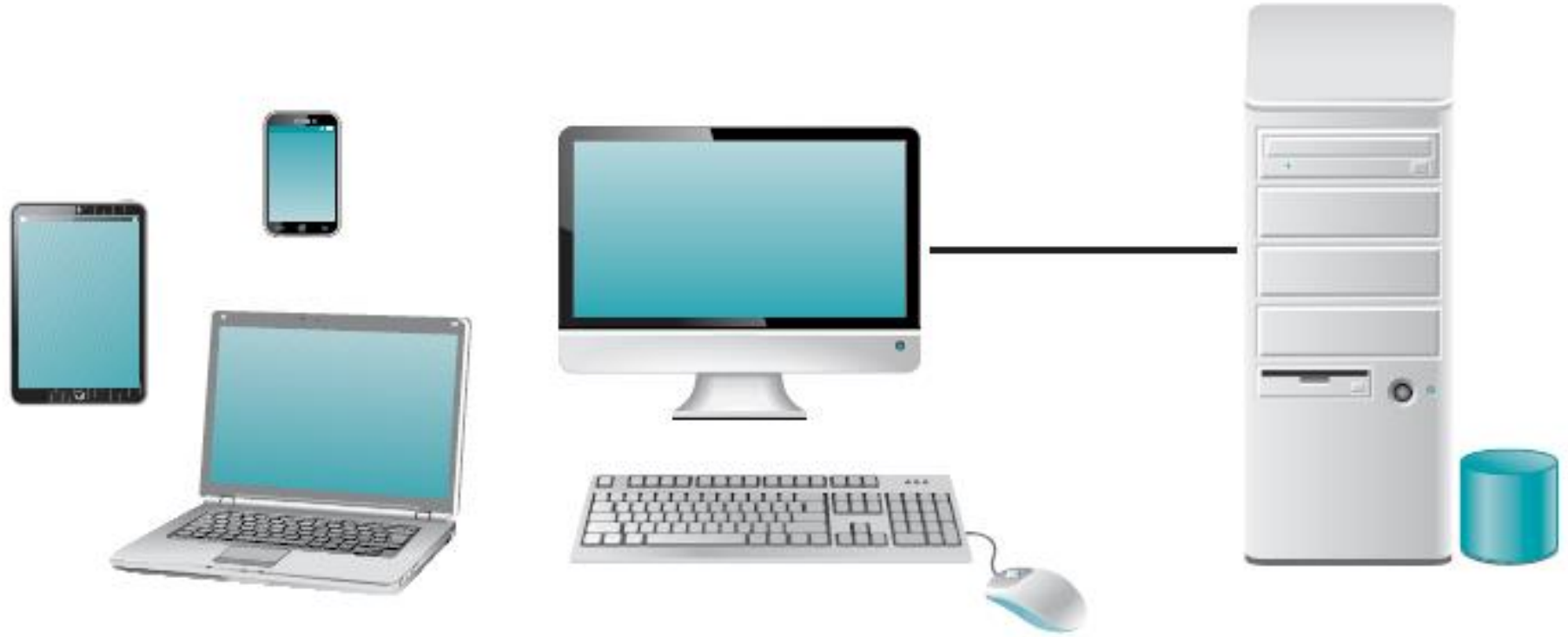


Terminal

Server computer
Handles presentation logic,
application logic,
data access logic, and
data storage

Client-Based Architectures

With client-based architectures, the clients are personal computers on a local area network (LAN), and the server computer is a server on the same network.

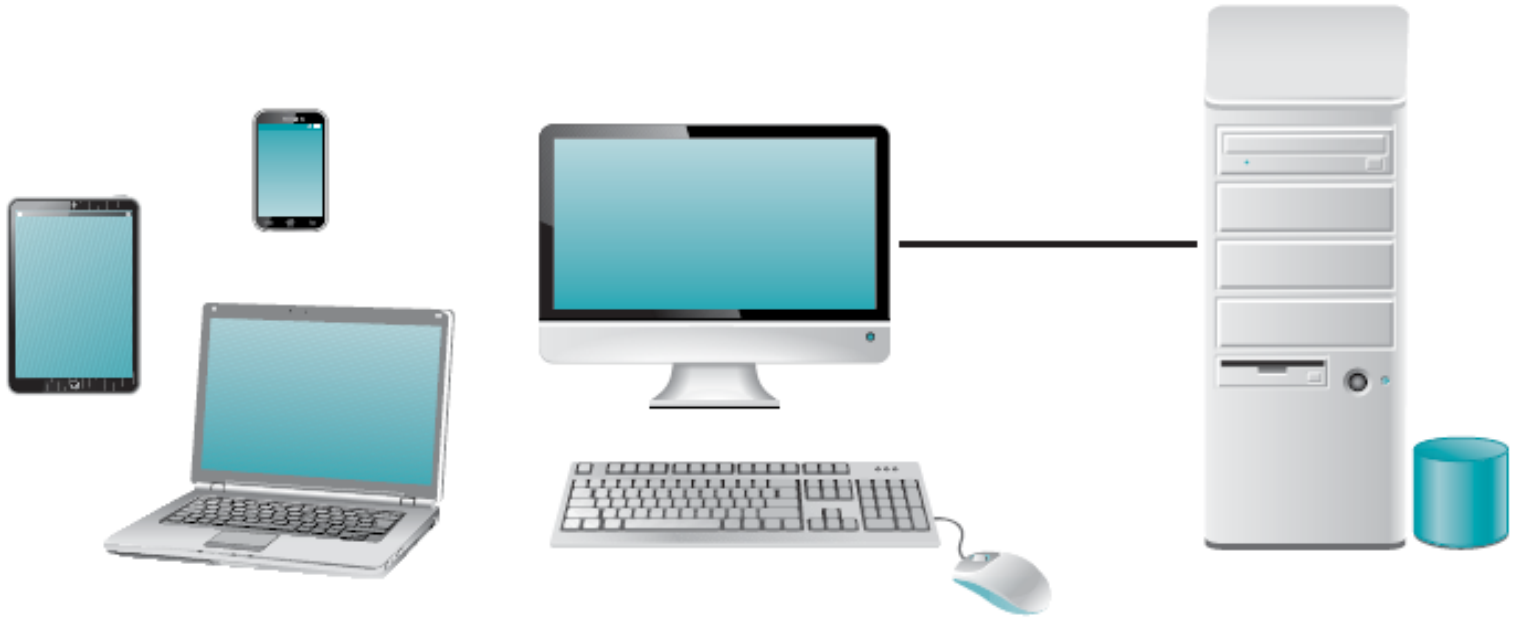


Client devices
Handle presentation logic
Handle application logic
Handles data access logic

Server
Handles data storage

Client–Server Architectures

Most organizations today are moving to client–server architectures, which attempt to balance the processing between the client and the server by having both do some of the application functions. In these architectures, the client is responsible for the presentation logic, whereas the server is responsible for the data access logic and data storage.



Client devices
Handle presentation logic
Handle application logic

Server
Handles data access logic
Handles data storage

Client–server architectures have four important benefits.

First, they are *scalable*. That means it is easy to increase or decrease the storage and processing capabilities of the servers. If one server becomes overloaded, you simply add another server so that many servers are used to perform the application logic, data access logic, or data storage. The cost to upgrade is much more gradual, and you can upgrade in smaller steps rather than spending hundreds of thousands to upgrade a mainframe server.

Second, Client–server architectures can support many different types of clients and servers. It is possible to connect computers that use different operating systems so that users can choose which type of computer they prefer

Third, for thin-client server architectures that use Internet standards, it is simple to clearly separate the presentation logic, the application logic, and the data access logic and design so each is somewhat independent.

Finally, because no single server computer supports all the applications, the network is generally more reliable. There is no central point of failure that will halt the entire network if it fails, as there is in server-based computing.

Limitations

Client–server architectures also have some critical limitations, the most important of which is its complexity. All applications in client–server computing have two parts, the software on the client and the software on the server. Writing this software is more complicated than writing the traditional all-in-one software used in server-based architectures. Updating the network with a new version of the software is more complicated, too. In server based architectures, there is one place where application software is stored; to update the software, we simply replace it there. With client–server architectures, we must update all clients and all servers.

Client–Server Tiers

There are many ways the application logic can be partitioned between the client and the server.

An *n-tiered architecture* uses more than three sets of computers. In this case, the client is responsible for presentation, database servers are responsible for the data access logic and data storage, and the application logic is spread across two or more different sets of servers.

Three-Tiered

Client–Server

Architecture

Selecting a Physical Architecture

Most systems are built to use the existing infrastructure in the organization, so often the current infrastructure restricts the choice of architecture.

Characteristic	Server-based	Client-based	Client–Server
Cost of infrastructure	Very high	Medium	Low
Cost of development	Medium	Low	High
Ease of development	Low	High	Low to medium
Interface capabilities	Low	High	High
Control and security	High	Low	Medium
Scalability	Low	Medium	High

Mobile Application Architecture

In recent years, the variety and capability of mobile devices, such as tablet devices and smartphones, has led to a huge increase in demand for mobile consumer and business applications. Opportunities abound to connect and inform both customers and employees.

Similar to our previous discussion, the system architect must decide how much of the presentation logic, business logic, and data access logic will reside on the mobile device, and how much will reside on server devices. If the application requires local processing using the mobile device's resources, such as the camera or GPS, and is only occasionally connected to the server, it should be created as a ***rich client*****.** With rich clients, the business and data access logic are included on the device along with the presentation logic. When the application can rely on the availability of server processing and will always be connected, a ***thin Web-based client*** can be developed. In this case, the business logic and data access logic will be located on the server side rather than the client side.

If the application requires a rich user interface (UI), only limited access to local device resources, and must be portable to other platforms, design a *rich Internet application* (*RIA*) client. Rich Internet Applications are Web-based applications that function as traditional desktop applications; however, Web browsers are required for access.

While discussion of the details of mobile application development is beyond the scope of this topic, we describe the development technology choices for mobile applications:

- native applications,
- cross-platform frameworks, and
- mobile web apps.

Native applications are written to run on a specific device with a specific operating system. Applications written in a device's native code (e.g., Objective C for Apple devices; Java for Android devices) provide the richest user experience and can take full advantage of the device's resources. Native applications must be re-built for each native operating system, however, and developers with skills in multiple languages are scarce. In addition, the apps must be updated as new device models and operating system versions are released.

Cross-platform frameworks exist that enable developers to create a mobile app in web based technologies like Javascript or HTML and use the framework to adapt the application for multiple devices. Usually the app must be "tweaked" for each device on which it will be deployed. Mobile applications developed with this approach will not provide the same rich user experience as the native apps, so it is best to use this approach with informational applications that do not require heavy use of device functions.

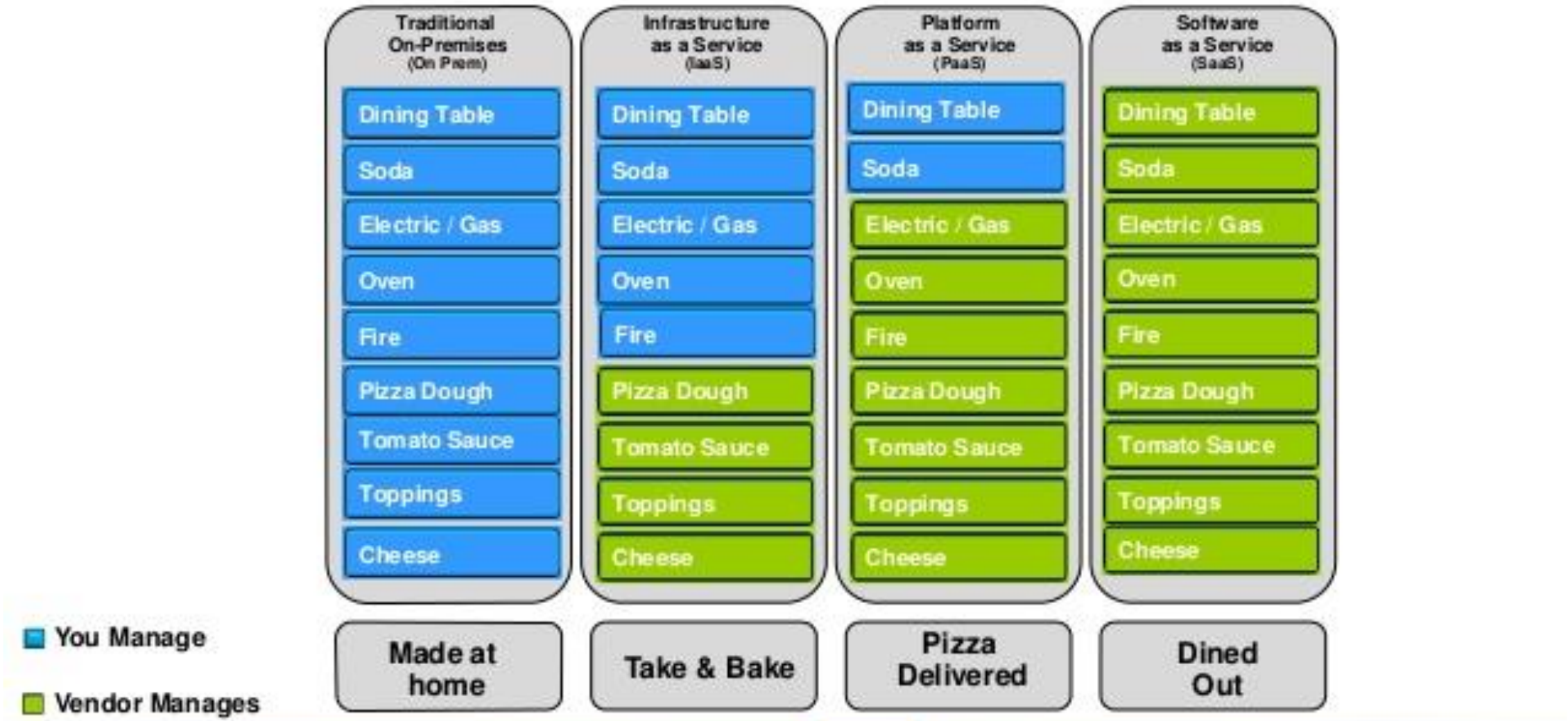
Mobile web apps are platform independent and can be deployed on any device equipped with the Web browser. Mobile web apps are built with web technology such as HTML5. These applications cannot use the hardware and soft ware on the device, require an Internet connection to work, and will provide the most generic user experience.

CLOUD COMPUTING

Cloud computing is the idea of treating IT as a utility or commodity.

Essentially, cloud computing is the latest approach to support distributed computing in a client-server type of architecture where the server is “in the cloud” and the client is on the desktop. The cloud can be the firm’s corporate data center, an external data center, or some combination of the two; however, more and more it generally is seen as an external, rather than an internal, service. Consequently, the idea of *multi tenancy*, where the cloud vendor has multiple customers using the same resource at the same time, becomes a real issue for both the cloud vendor and the cloud customer. Cloud computing may become the greatest enabler for IT *outsourcing*

Pizza as a Service



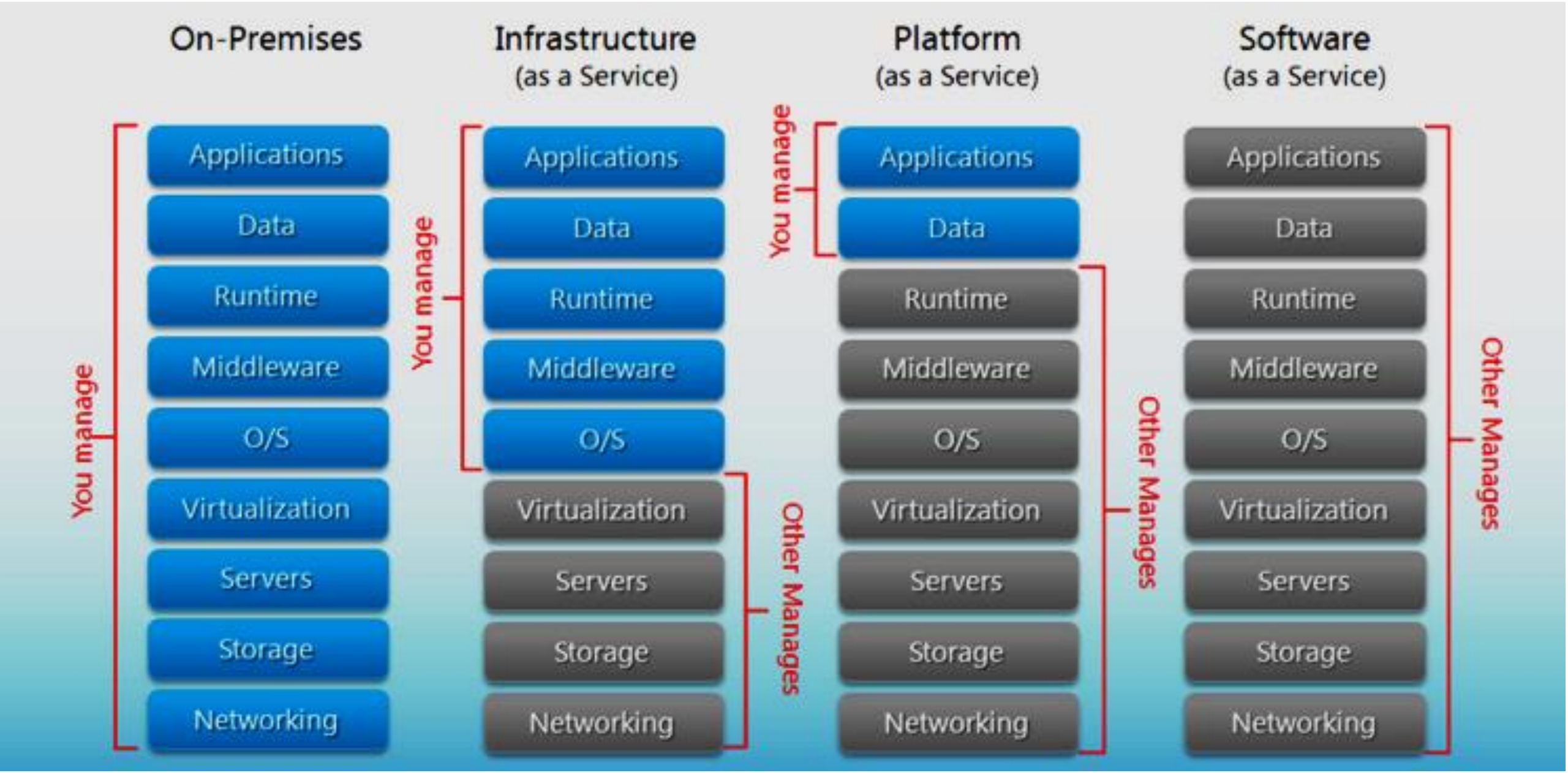
Infrastructure-as-a-service (IaaS)

The most basic category of cloud computing services. With IaaS, you rent IT infrastructure—servers and virtual machines (VMs), storage, networks, operating systems—from a cloud provider on a pay-as-you-go basis. Platform as a service (PaaS)

Platform-as-a-service (PaaS) refers to cloud computing services that supply an on-demand environment for developing, testing, delivering and managing software applications. PaaS is designed to make it easier for developers to quickly create web or mobile apps, without worrying about setting up or managing the underlying infrastructure of servers, storage, network and databases needed for development.

Software as a service (SaaS)

Software-as-a-service (SaaS) is a method for delivering software applications over the Internet, on demand and typically on a subscription basis. With SaaS, cloud providers host and manage the software application and underlying infrastructure and handle any maintenance, like software upgrades and security patching. Users connect to the application over the Internet, usually with a web browser on their phone, tablet or PC.


On-Premises
Infrastructure
(as a Service)
Platform
(as a Service)
Software
(as a Service)
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You manage
Other Manages

There are three different classifications of clouds:

- Private
- Public
- Hybrid

Fundamentally, cloud computing is an umbrella technology that encompasses the ideas of **virtualization**, **service-oriented architectures**, and **grid computing**.

Virtualization

Virtualization is the idea of treating any computing resource, regardless of where it is located, as if it is “in” the client machine. This idea evolved from *virtual memory*. Virtual memory was developed originally in the 1960s. Virtual memory allowed the user/programmer to act as if the amount of main memory in the computer was unlimited.

Service-oriented architectures

A **service-oriented architecture** (**SOA**) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network. **The basic principles of service-oriented architecture are independent of vendors, products and technologies.** A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as retrieving a credit card statement online.

A service has four properties according to one of many definitions of SOA:

It logically represents a business activity with a specified outcome.

It is self-contained.

It is a black box for its consumers.

It may consist of other underlying services.

SOA is a new badge for some very old ideas:

- Divide your code into reusable modules.
- Encapsulate in a module any design decision that is likely to change.
- Design your modules in such a way that they can be combined in different useful ways (sometimes called a "family" or "product line").

What's new in SOA is

- You're doing it on a network.
- Modules are communicating by sending messages to each other over the network, rather than by more traditional programming-language mechanisms like procedure calls. In particular, in a service-oriented architecture the parts generally don't share mutable state (global variables in a traditional program). Or if they do share state, that state is carefully locked up in a database which is itself an agent and which can easily manage multiple concurrent clients.

Sample Hardware and Software Specification

Specification	Standard Client	Standard Web Server	Standard Application Server	Standard Database Server
Operating System	• Windows • Internet Explorer	• Linux	• Linus	• Linux
Special Software	• Acrobat Reader • Adobe Flash • QuickTime	• Apache	• Java	• Oracle
Hardware	• 8 GB Memory • 500 GB disk drive	• 16 GB Memory • 1TB disk drive	• 24 GB Memory • 2–600 GB disk drives	• 32 GB Memory • 4–600 GB Hotplug disk drives
	• Intel Core i5 • 18.5-inch HD monitor	• Intel Xenon X3470 • 18.5-inch HD monitor	• Intel Xenon X5620 • 18.5-inch HD monitor	• Intel Xenon X5650 18.5-inch HD monitor
Network	• 100 Mbps Ethernet High-speed Wireless	• 100 Mbps Ethernet	• 100 Mbps Ethernet	• 100 Mbps Ethernet

NONFUNCTIONAL REQUIREMENTS AND PHYSICAL ARCHITECTURE LAYER DESIGN

The design of the physical architecture layer specifies the overall architecture and the placement of software and hardware that will be used. Each of the architectures discussed before has its strengths and weaknesses. **Most organizations are trying to move to client–server architectures** for cost reasons, so in the event that there is no compelling reason to choose one architecture over another, cost usually suggests client–server.

Creating a physical architecture layer design begins with the nonfunctional requirements. The first step is to refine the nonfunctional requirements into more-detailed requirements that are then used to help select the architecture to be used (sever-based, client-based, or client–server) and what software components will be placed on each device.

Operational Requirements

Operational requirements specify the operating environment(s) in which the system must perform and how those might change over time. This usually refers to operating systems, system software, and information systems with which the system must interact, but on occasion it also includes the physical environment if the environment is important to the application

Type of Requirement	Definition	Examples
Technical Environment Requirements	Special hardware, software, and network requirements imposed by business requirements	• The system will work over the Web environment with Internet Explorer. • All office locations will have an always-on network connection to enable real-time database updates. • A version of the system will be provided for customers connecting over the Internet via a tablet or smartphone.
System Integration Requirements	The extent to which the system will operate with other systems	• The system must be able to import and export Excel spreadsheets. • The system will read and write to the main inventory database in the inventory system.
Portability Requirements	The extent to which the system will need to operate in other environments	• The system must be able to work with different operating systems (e.g., Linux, Mac OS, and Windows). • The system might need to operate with handheld devices such as a Android and Apple iOS devices.
Maintainability Requirements	Expected business changes to which the system should be able to adapt	• The system will be able to support more than one manufacturing plant with six months' advance notice. • New versions of the system will be released every six months.

Performance Requirements

Performance requirements focus on performance issues, such as response time, capacity, and reliability.

Type of Requirement	Definition	Examples
Speed Requirements	The time within which the system must perform its functions	• Response time must be less than 7 seconds for any transaction over the network. • The inventory database must be updated in real time. • Orders will be transmitted to the factory floor every 30 minutes.
Capacity Requirements	The total and peak number of users and the volume of data expected	• There will be a maximum of 100–200 simultaneous users at peak use times. • A typical transaction will require the transmission of 10K of data.
Availability and Reliability Requirements	The extent to which the system will be available to the users and the permissible failure rate due to errors	• The system will store data on approximately 5,000 customers for a total of about 2 MB of data. • Scheduled maintenance shall not exceed one 6-hour period each month. • The system shall have 99% uptime performance.

Security Requirements

Security is the ability to protect the information system from disruption and data loss, whether caused by an intentional act (e.g., a hacker, a terrorist attack) or a random event (e.g., disk failure, tornado).

Type of Requirement	Definition	Examples
System Value Estimates	Estimated business value of the system and its data	• The system is not mission critical but a system outage is estimated to cost $50,000 per hour in lost revenue. • A complete loss of all system data is estimated to cost $20 million.
Access Control Requirements	Limitations on who can access what data	• Only department managers will be able to change inventory items within their own department. • Telephone operators will be able to read and create items in the customer file but cannot change or delete items.
Encryption and Authentication Requirements	Defines what data will be encrypted Where and whether authentication will be needed for user access	• Data will be encrypted from the user's computer to the website to provide secure ordering. • Users logging in from outside the office will be required to authenticate.
Virus Control Requirements	Requirements to control the spread of viruses	• All uploaded files will be checked for viruses before being saved in the system.

Cultural and political requirements are those specific to the countries in which the system will be used.

Type of Requirement	Definition	Examples
Customization Requirements	Specification of what aspects of the system can be changed by local users	• Country managers will be able to define new fields in the product database to capture country-specific information. • Country managers will be able to change the format of the telephone number field in the customer database.
Legal Requirements	The laws and regulations that impose requirements on the system	• Personal information about customers cannot be transferred out of European Union countries into the United States. • It is against U.S. federal law to divulge information on who rented what videotape, so access to a customer's rental history is permitted only to regional managers.

System Analysis

DR. DRAGUNOV

Construction / Development

DESIGNING TESTS

Testing is more critical to object-oriented systems than systems developed in the past. Based on encapsulation (and information hiding), polymorphism (and dynamic binding), inheritance, reuse, and the actual object-oriented products, thorough testing is much more difficult and critical.

The purpose of testing is not to demonstrate that the system is free of errors. It is not possible to prove that a system is error free, which is especially true with object-oriented systems.

This is similar to theory testing. You cannot prove a theory. If a test fails to find problems with a theory, your confidence in the theory is increased.

There are four general stages of tests:

- unit tests,
- integration tests,
- system tests, and
- acceptance tests.

Test Planning

Testing starts with the development of a *test plan,* which defines a series of tests that will be conducted. Because testing takes place throughout the development of an object-oriented system, a test plan should be developed at the very beginning of system development and continuously updated as the system evolves.

Unit Tests

Unit tests focus on a single unit—the class. There are two approaches to unit testing: black-box testing and white-box testing.

Black-box testing is the most commonly used because each class represents an encapsulated object. Black-box testing is driven by the CRC cards, behavior state machines, and contracts associated with a class, not by the programmers' interpretation. In this case, the test plan is developed directly from the specification of the class: each item in the specification becomes a test, and several test cases are developed for it.

White-box testing is based on the method specifications associated with each class.

Integration Tests

Integration tests assess whether a set of classes that must work together do so without error.

System Tests

System tests are usually conducted by the systems analysts to ensure that all classes work together without error. System testing is similar to integration testing but is much broader in scope.

Acceptance Tests

Acceptance testing is done primarily by the users with support from the project team. The goal is to confirm that the system is complete, meets the business needs that prompted the system to be developed, and is acceptable to the users. Acceptance testing is done in two stages: *alpha testing*, in which users test the system using made-up data, and *beta testing*, in which users begin to use the system with real data but are carefully monitored for errors

DEVELOPING DOCUMENTATION

Like testing, developing documentation of the system must be done throughout system development. There are two fundamentally different types of documentation:

- system documentation
- user documentation.

System documentation is intended to help programmers and systems analysts understand the application software and enable them to build it or maintain it after the system is installed.

User documentation (such as user's manuals, training manuals, and online help systems) is designed to help the user operate the system.