SOFTWARE DESIGN

ITACS 5203, Unit 9

LEARNING OBJECTIVES

3.1.1. IS Auditor's Role in SDLC Project Management
3.1.2. Software Development Methods
3.1.2.1. Prototyping
3.1.2.2. Rapid Application Development
3.1.2.3. Agile Development
3.1.2.4. Object Oriented System Development
3.1.2.5. Component Based Development
3.1.2.6. Web Based Application Development
3.1.2.7. Software Reengineering
3.1.2.8. Reverse Engineering
3.1.2.9. DevOps
3.1.2.10. Business Process Reengineering and Process Change
3.1.2.10.1. Benchmarking Process
3.1.2.10.2. IS Auditors Role in Business Process Reengineering
3.1.3. System Development Tools and Productivity Aids
3.1.3.1. Computer Aided Software Engineering
3.1.3.2. Code Generators
3.1.3.3. Fourth Generation Languages
3.1.3.3.1. Query and Report Generators
3.1.3.3.2. Embedded Databases
3.1.3.3.3. Relational Databases
3.1.3.3.4. Application Generators
3.1.3.3.5. Characteristics:
3.1.3.3.5.1. Nonprocedural
3.1.3.3.5.2. Environmental Independence
3.1.3.3.5.3. Software Facilities
3.1.3.3.5.4. Programmer Workbench Concepts
3.1.3.3.5.5. Simple Language Subsets
3.2. Control Identification and Design
3.2.1. Input/Origination Controls
3.2.1.1. Input Authorization
3.2.1.2. Batch Controls and Balancing
3.2.1.3. Error Reporting and Handling
3.2.2. Processing Procedures and Controls
3.2.2.1. Data Validation and Editing Procedures
3.2.2.2. Processing Controls
3.2.2.3. Data File Control Procedures
3.2.3. Output Controls
3.2.4. Application Controls
3.2.4.1. IS Auditor's Role in Reviewing Application Controls
3.2.5. User Procedures3.1. System Development Methodologies
3.1.1. SDLC Phases
3.1.1.1. Implementation
3.1.1.1.1. Configuration
3.1.1.1.2. Development
3.1.1.1.2.1. Programming Methods and Techniques
3.1.1.1.2.2. Integrated Development Environment
3.1.1.1.2.3. Programming Languages
3.1.1.1.2.4. Program Debugging
3.1.1.2. Testing and Implementation
3.1.1.3. Post Implementation Review

1.5. Procedural Security
1.6. Modular Programming
1.7. Sensitive Data Mapping
1.8. Reducing the System Attack Surface

INTRODUCTION

Analysis determines the business needs

Design activities focus on how to build the system

- Major activity is to evolve the models into a design
- Goal is to create a blueprint for the design that makes sense to implement
  - Determine how and where data will be stored
  - Determine how the user will interface with the system (user interface, inputs and outputs)
- Decide on the physical architecture

Analysis and design phases are highly *interrelated* and may require much "going back and forth"

- Example: prototyping may uncover additional information

DESIGN PROCESS

Verify and validate the analysis models

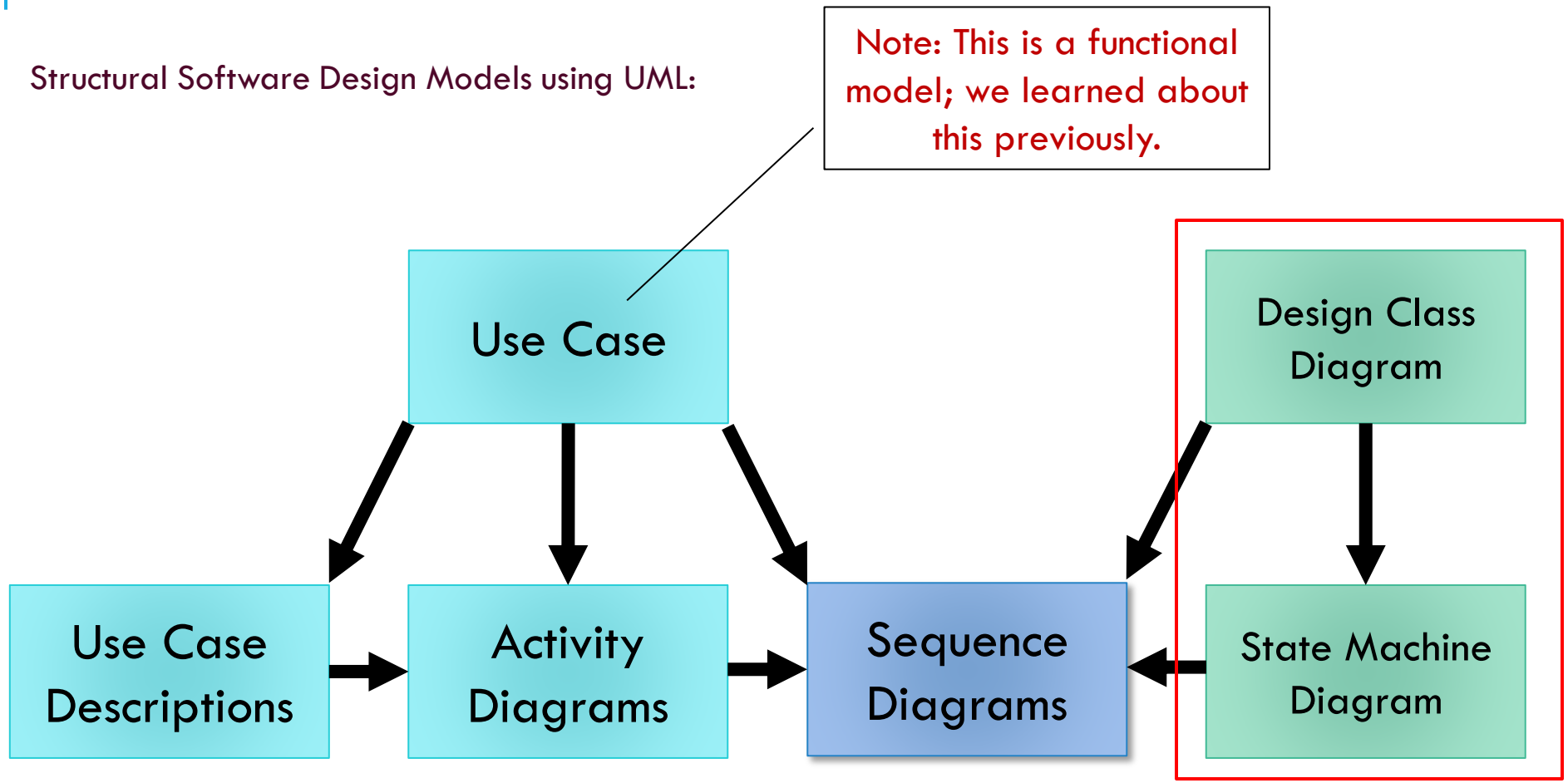
Evolve the analysis models into design models

Create packages and utilize package diagrams

Decide upon a design strategy

STRUCTURAL DESIGN MODELS (UML)

Structural Software Design Models using UML:



DESIGN CLASS DIAGRAM

stereotype a way of categorizing a model element by its characteristics
- indicated by guillemets (<< >>)

persistent class an class whose objects exist after a system is shut down (data remembered)

entity class a design identifier for a problem domain class (usually persistent)

boundary class or view class a class that exists on a system's automation boundary, such as an input window form or Web page

control class a class that mediates between boundary classes and entity classes, acting as a switchboard between the view layer and domain layer

data access class a class that is used to retrieve data from and send data to a database

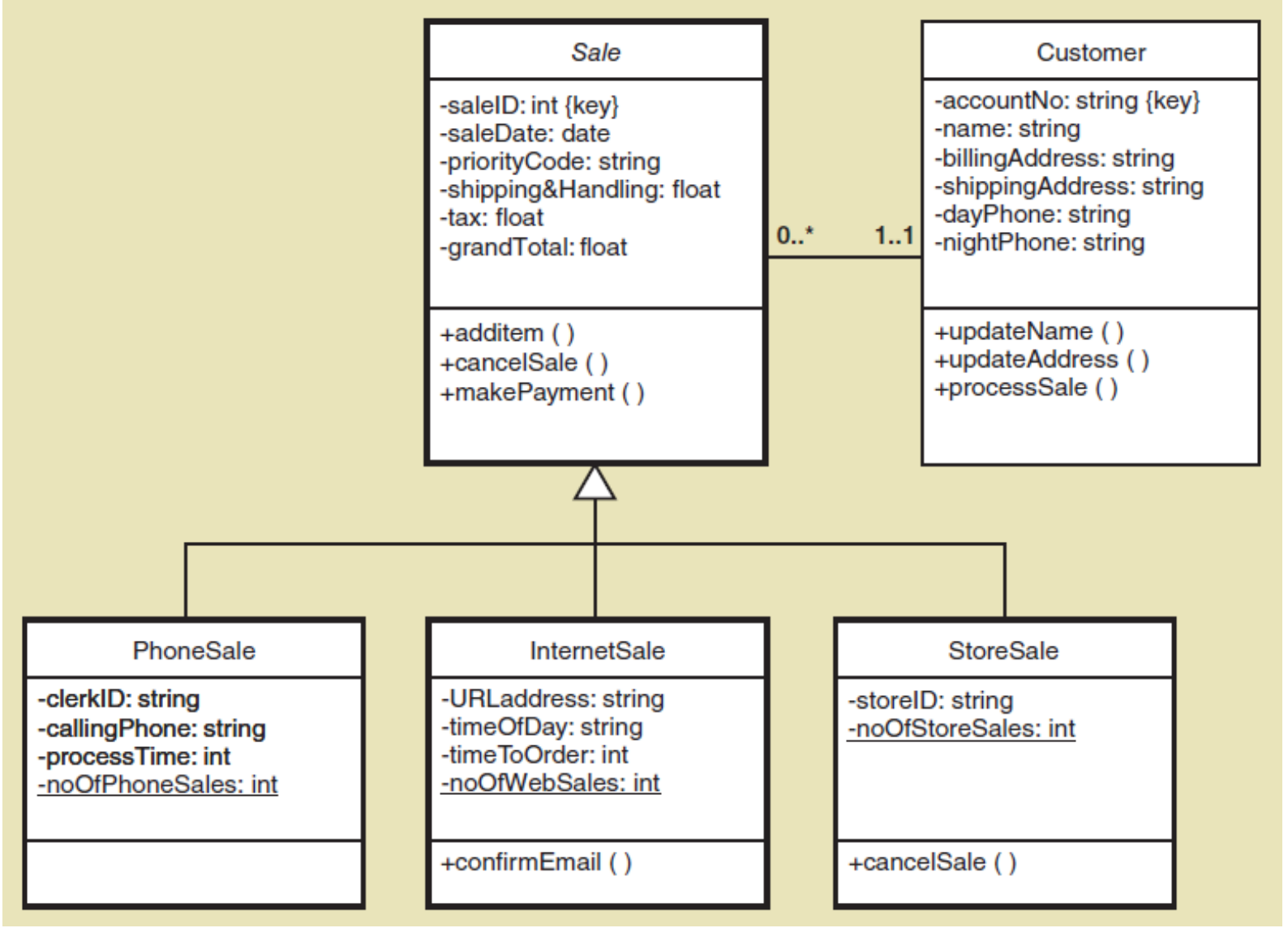
DESIGN CLASS DIAGRAM

An abstract class must be italicized. Concrete classes are not.

<< Stereotype >> Class Name::Parent Class	<< Entity >> AccountHolder::Customer
Attribute List Visibility name:type-expression = initial-value {property}	- SocialSecurityNumber:string=000-00-0000 {key} + Age:Integer
Method List Visibility name (parameter list) : return type-expression	+ ValidateSSN(SocialSecurityNumber) : Boolean

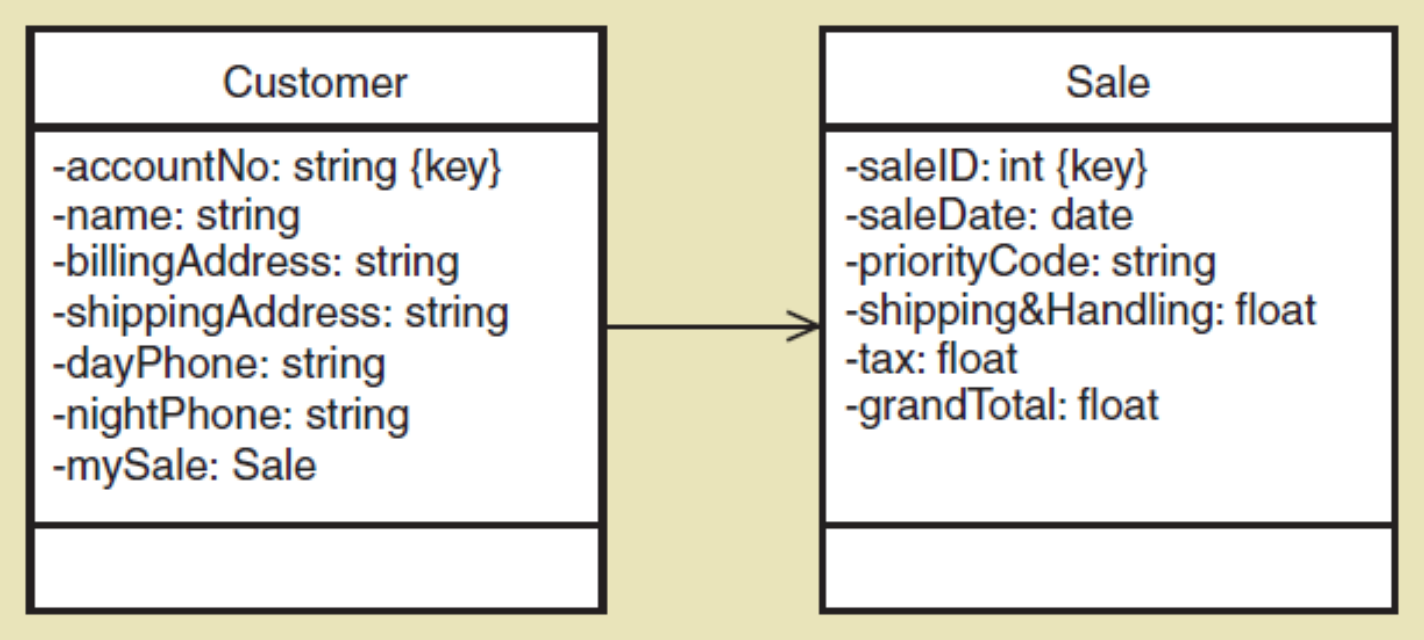
Underline "static" – AKA, applies to the entire class, not a specific object.

DESIGN CLASS DIAGRAM



NAVIGATION VISIBILITY

- The ability of one object to view and interact with another object
- Accomplished by adding an object reference variable to a class.
- Shown as an arrow head on the association line—customer can find and interact with sale because it has mySale reference variable



NAVIGATION VISIBILITY GUIDELINES

One-to-many associations that indicate a superior/subordinate relationship are usually navigated from the superior to the subordinate

Mandatory associations, in which objects in one class can't exist without objects of another class, are usually navigated from the more independent class to the dependent

When an object needs information from another object, a navigation arrow might be required

Navigation arrows may be bidirectional.

START WITH FUNCTIONAL MODELS…

Proceed use case by use case, adding to the diagram

Pick the domain classes that are involved in the use case (see preconditions and post conditions for ideas)

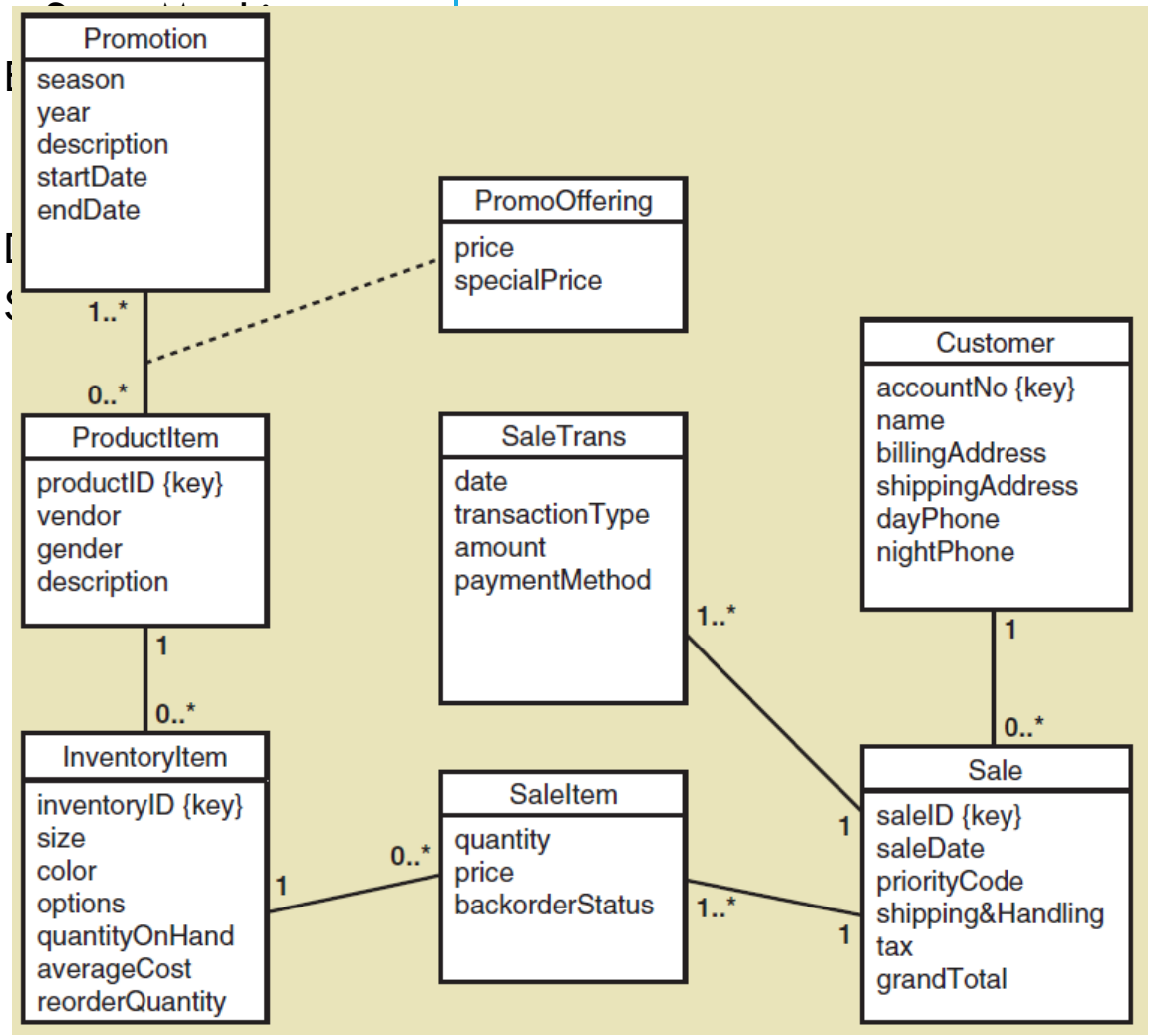
Add a controller class to be in charge of the use case

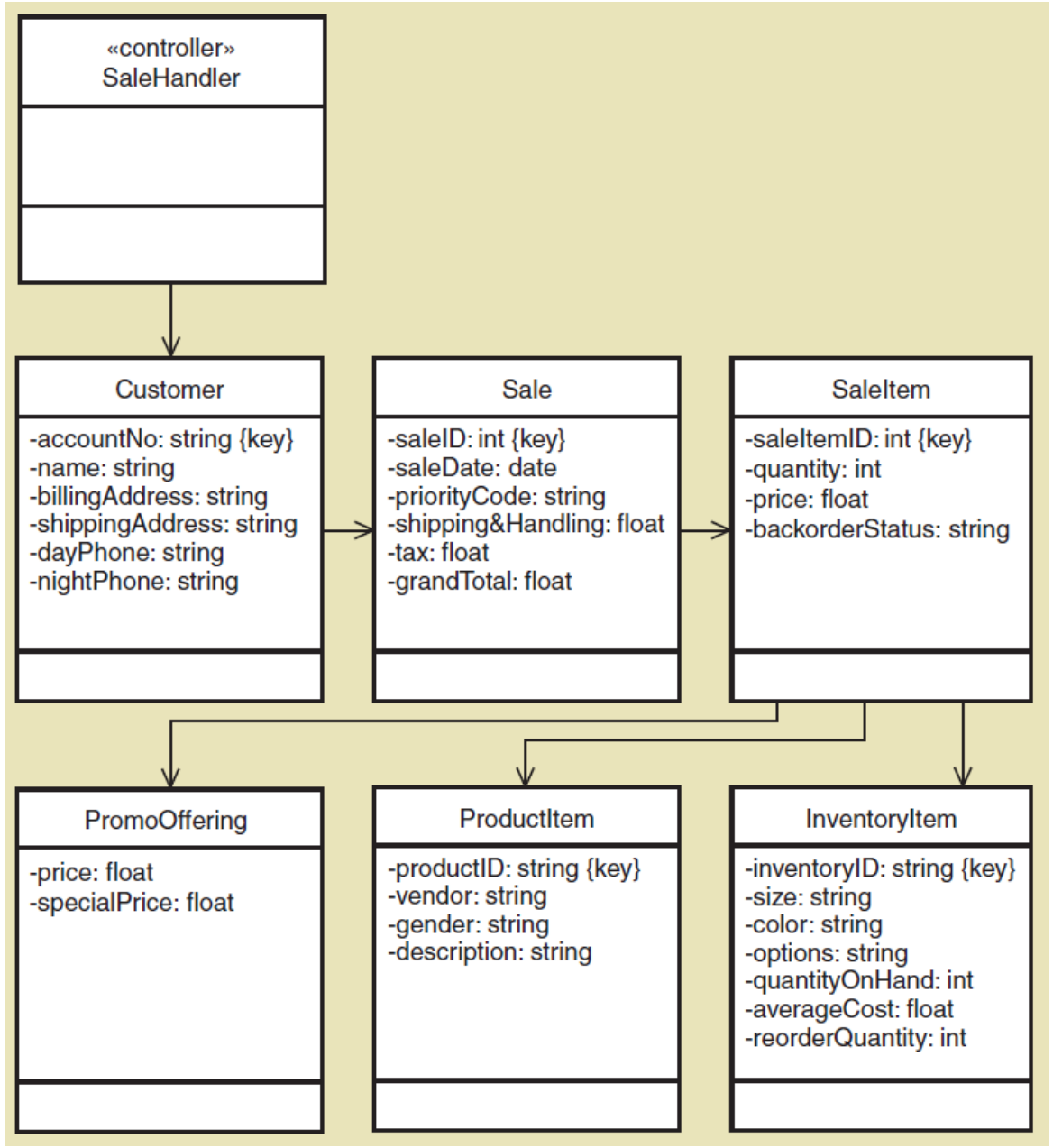
Determine the initial navigation visibility requirements using the guidelines and add to diagram

Elaborate the attributes of each class with visibility and type

Note that often the associations and multiplicity are removed from the design class diagram as in text to emphasize navigation, but they are often left on

EXAMPLE


Promotion
season
year
description
startDate
endDate
PromoOffering
price
specialPrice
1..*
0..*
ProductItem
productID {key}
vendor
gender
description
1
0..*
InventoryItem
inventoryID {key}
size
color
options
quantityOnHand
averageCost
reorderQuantity
SaleTrans
date
transactionType
amount
paymentMethod
1..*
Customer
accountNo {key}
name
billingAddress
shippingAddress
dayPhone
nightPhone
1
0..*
Sale
saleID {key}
saleDate
priorityCode
shipping&Handling
tax
grandTotal
1
SaleItem
quantity
price
backorderStatus
0..*
1
1..*
1


«controller»
SaleHandler
Customer
-accountNo: string {key}
-name: string
-billingAddress: string
-shippingAddress: string
-dayPhone: string
-nightPhone: string
Sale
-saleID: int {key}
-saleDate: date
-priorityCode: string
-shipping&Handling: float
-tax: float
-grandTotal: float
SaleItem
-saleItemID: int {key}
-quantity: int
-price: float
-backorderStatus: string
PromoOffering
-price: float
-specialPrice: float
ProductItem
-productID: string {key}
-vendor: string
-gender: string
-description: string
InventoryItem
-inventoryID: string {key}
-size: string
-color: string
-options: string
-quantityOnHand: int
-averageCost: float
-reorderQuantity: int

USE CRC CARDS TO DEFINE METHODS

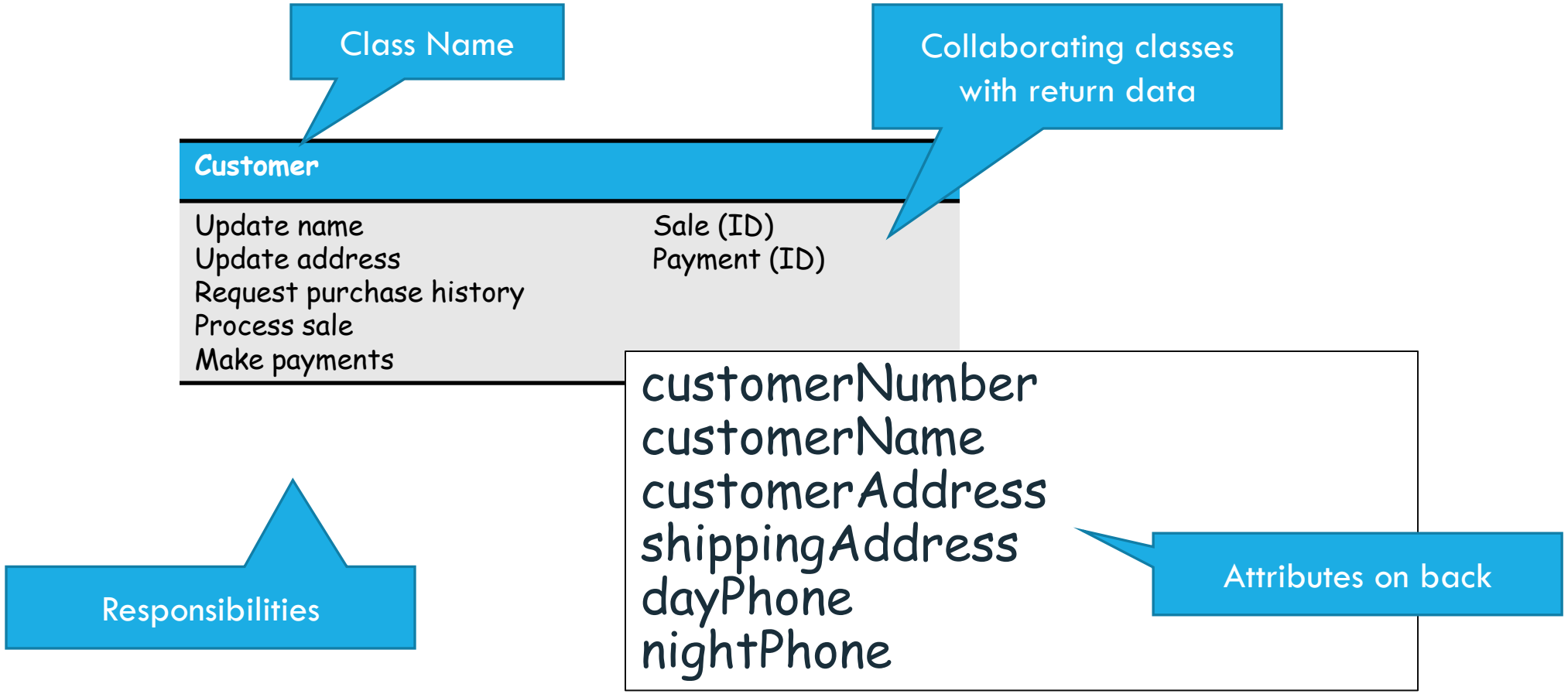
CRC Cards—Classes, Responsibilities, Collaboration Cards

OO design is about assigning Responsibilities to Classes for how they Collaborate to accomplish a use case

Usually a manual process done in a brainstorming session

- 3 X 5 note cards
- One card per class
- Front has responsibilities and collaborations
- Back has attributes needed

CRC CARD EXAMPLE



CRC CARD PROCEDURE

Because the process is to design, or realize, a single use case, start with a set of unused CRC cards. Add a controller class (Controller design pattern).

Identify a problem domain class that has primary responsibility for this use case that will receive the first message from the use case controller. For example, a Customer object for new sale.

Use the first cut design class diagram to identify other classes that must collaborate with the primary object class to complete the use case.

Have use case descriptions and SSDs handy

Start with the class that gets the first message from the controller. Name the responsibility and write it on card.

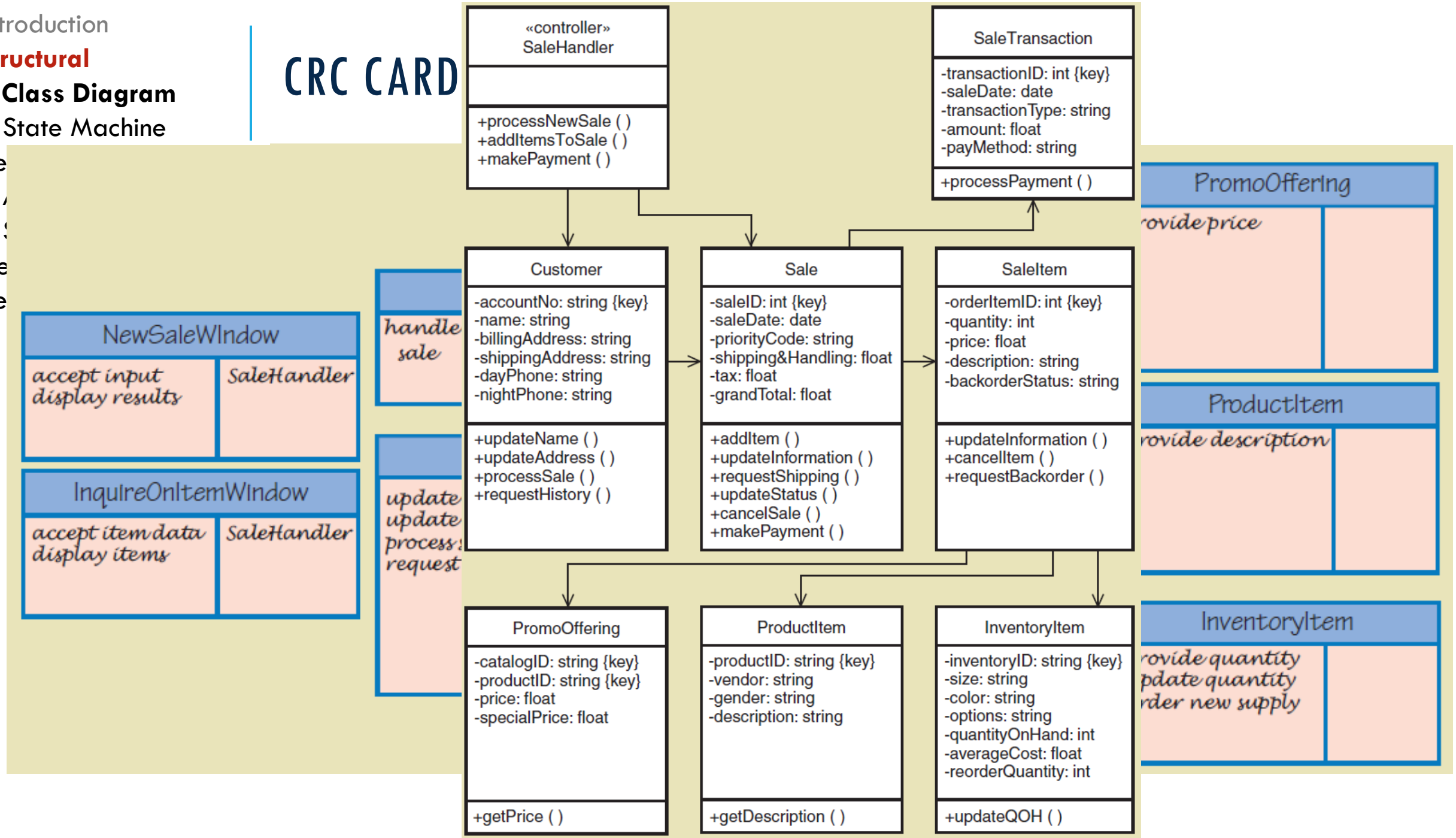
Now ask what this first class needs to carry out the responsibility. Assign other classes responsibilities to satisfy each need. Write responsibilities on those cards.

Sometimes different designers play the role of each class, acting out the use case by verbally sending messages to each other demonstrating responsibilities

Add collaborators to cards showing which collaborate with which. Add attributes to back when data is used

Eventually, user interface classes or even data access classes can be added

CRC CARD


«controller»
SaleHandler
+processNewSale ( )
+addItemsToSale ( )
+makePayment ( )
SaleTransaction
-transactionID: int {key}
-saleDate: date
-transactionType: string
-amount: float
-payMethod: string
+processPayment ( )
Customer
-accountNo: string {key}
-name: string
-billingAddress: string
-shippingAddress: string
-dayPhone: string
-nightPhone: string
+updateName ( )
+updateAddress ( )
+processSale ( )
+requestHistory ( )
Sale
-saleID: int {key}
-saleDate: date
-priorityCode: string
-shipping&Handling: float
-tax: float
-grandTotal: float
+addItem ( )
+updateInformation ( )
+requestShipping ( )
+updateStatus ( )
+cancelSale ( )
+makePayment ( )
SaleItem
-orderItemID: int {key}
-quantity: int
-price: float
-description: string
-backorderStatus: string
+updateInformation ( )
+cancelItem ( )
+requestBackorder ( )
PromoOffering
-catalogID: string {key}
-productID: string {key}
-price: float
-specialPrice: float
+getPrice ( )
ProductItem
-productID: string {key}
-vendor: string
-gender: string
-description: string
+getDescription ( )
InventoryItem
-inventoryID: string {key}
-size: string
-color: string
-options: string
-quantityOnHand: int
-averageCost: float
-reorderQuantity: int
+updateQOH ( )
NewSaleWIndow
accept input
display results
SaleHandler
InquireOnItemWIndow
accept item data
display items
SaleHandler
handle
sale
update
update
process
request
PromoOffering
rovide price
ProductItem
rovide description
InventoryItem
rovide quantity
pdate quantity
rder new supply

STATE MACHINE DIAGRAM

State machine diagram

- A UML diagram showing the life of an object in states and transitions

State

- A condition during an object's life when it satisfies some criterion, performs some action, or waits for an event

Transition

- The movement of an object from one state to another state

Action Expression

- A description of activities performed as part of a transition

Pseudo state

- The starting point of a state machine diagram (black dot)

Origin state

- The original state of an object before transition

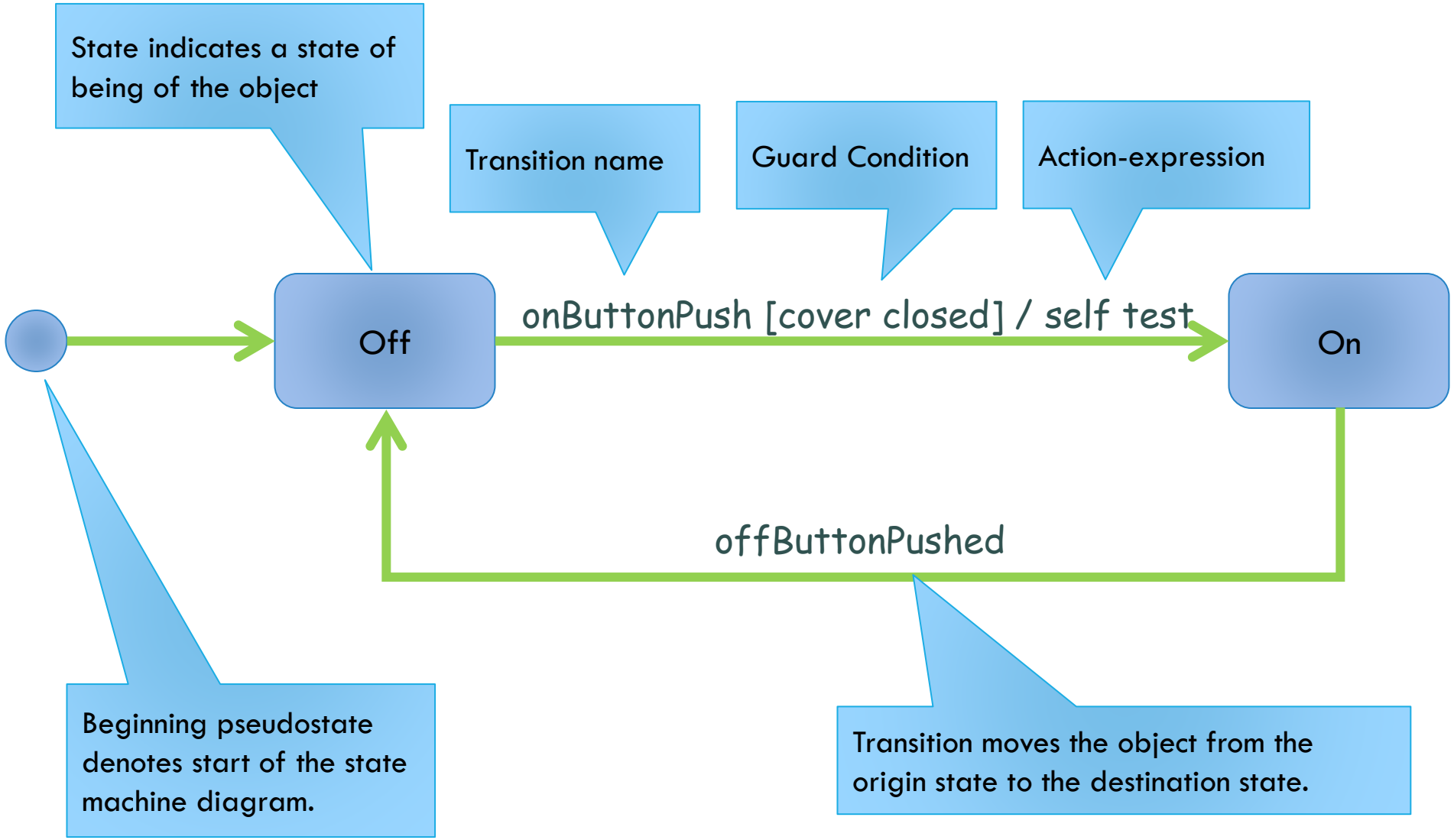
Destination state

- The state to which the object moves after the transition

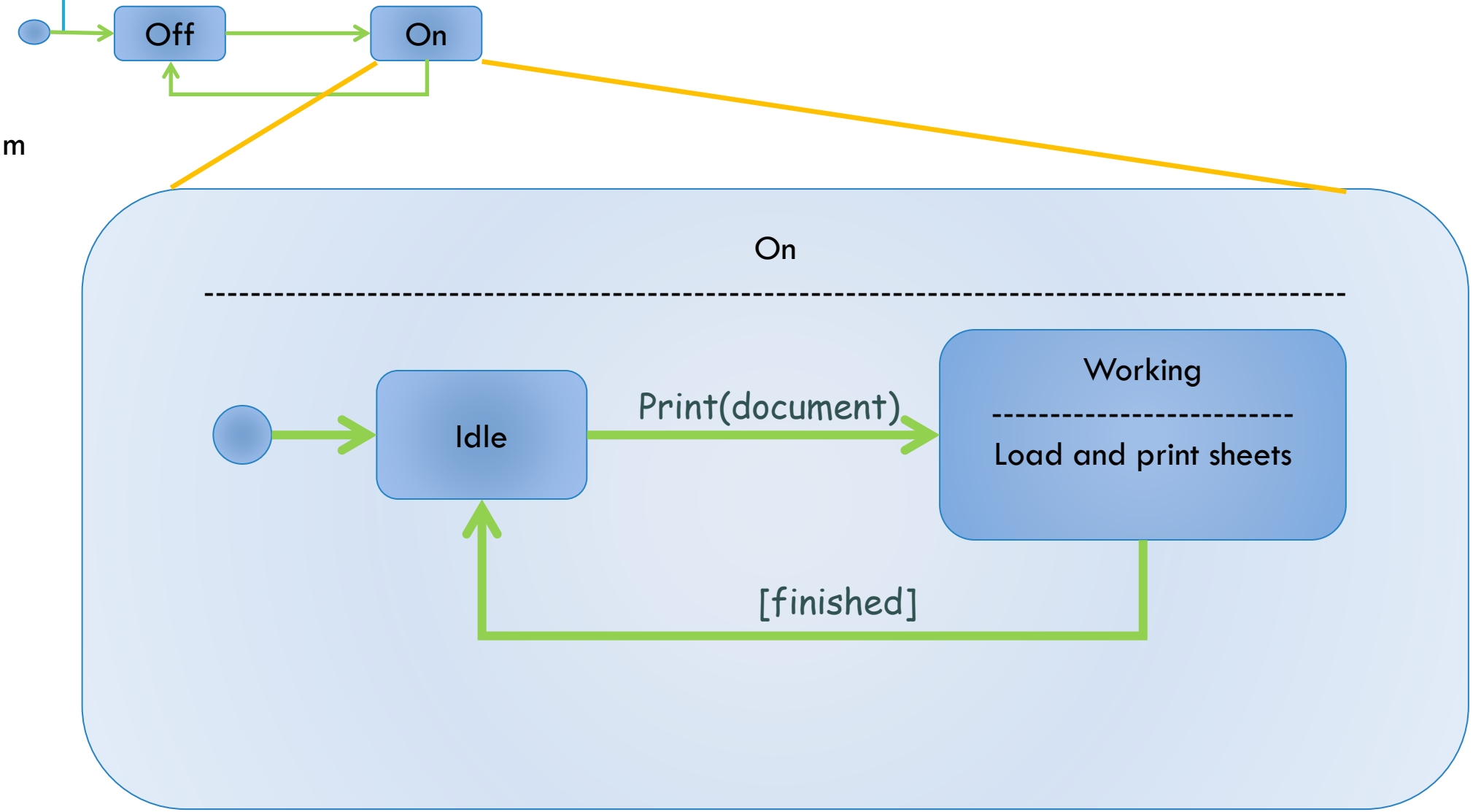
Guard condition

- A true false test to see whether a transition can fire

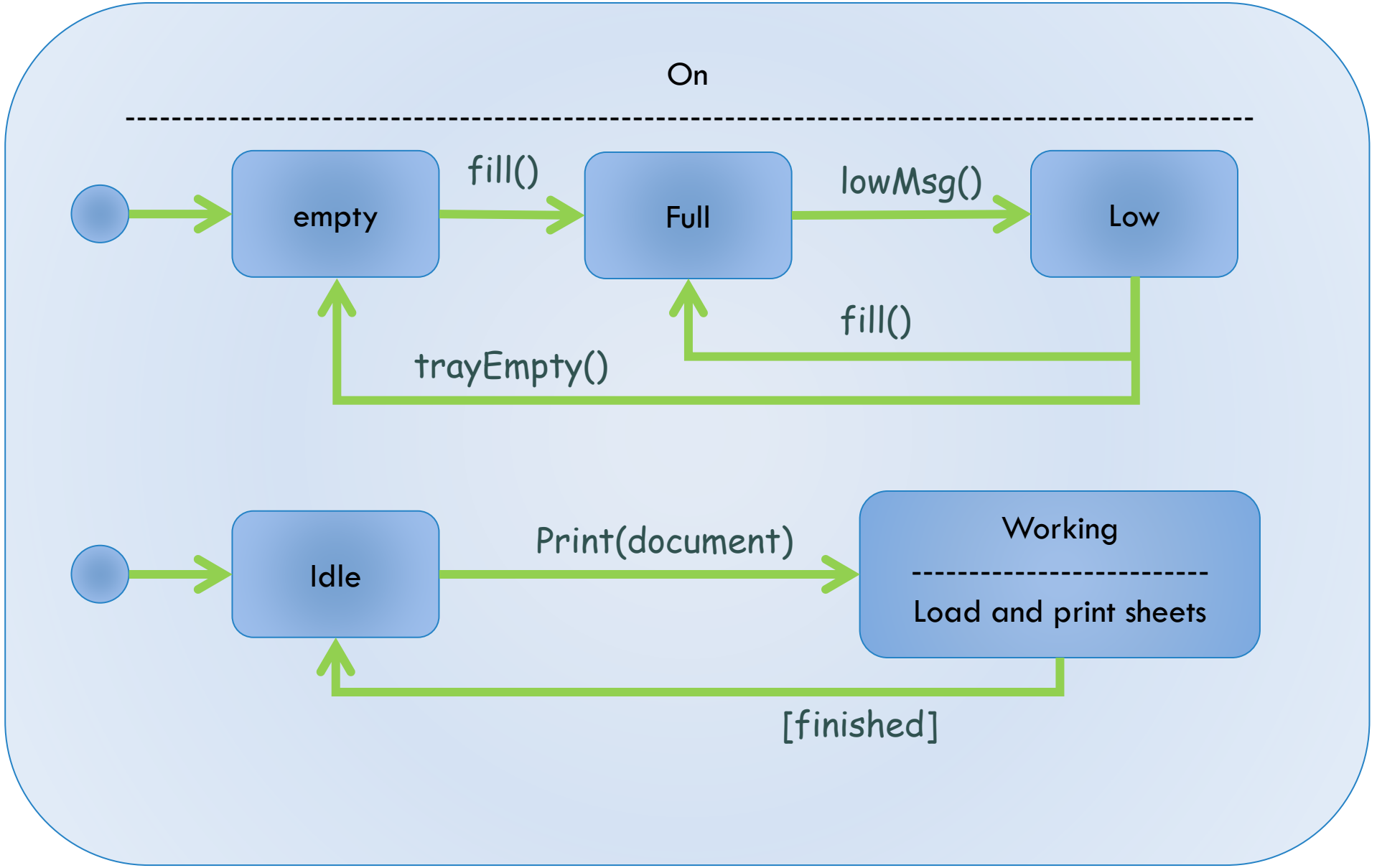
STATE MACHINE EXAMPLE



COMPOSITE STATES



CONCURRENT STATES

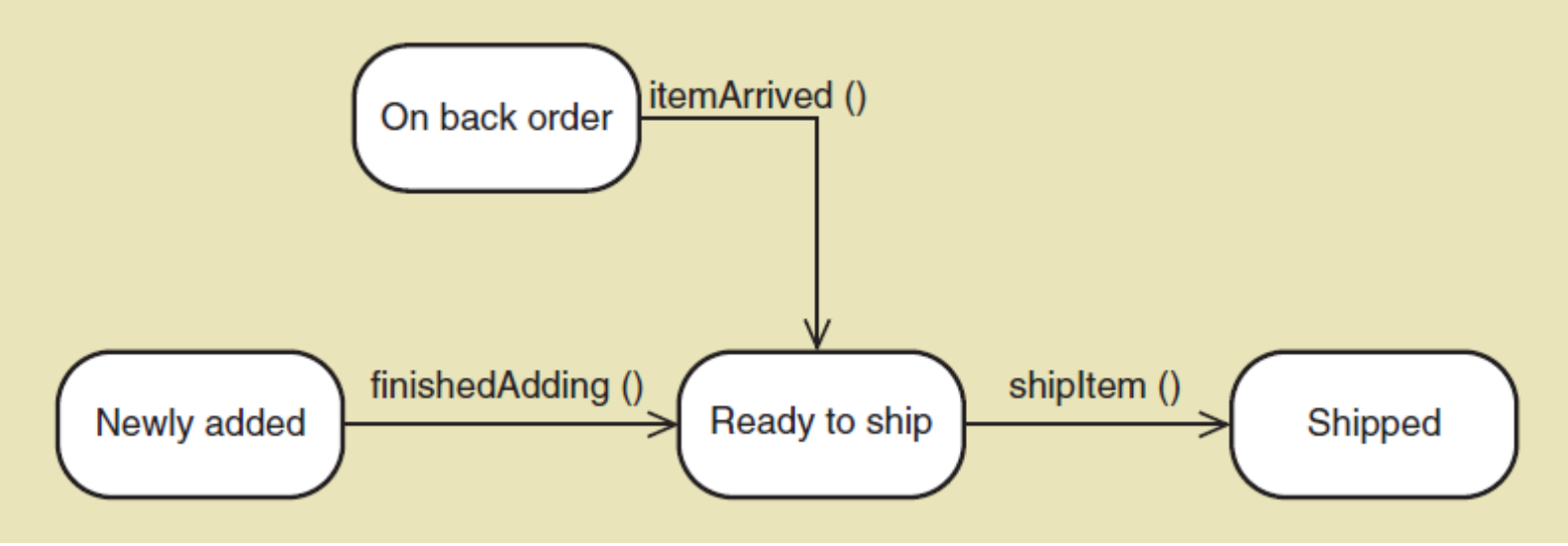


STEPS FOR DEVELOPING A STATE MACHINE

1. Review the class diagram and select classes that might require state machine diagrams
2. For each class, make a list of status conditions (states) you can identify
3. Begin building diagram fragments by identifying transitions that cause an object to leave the identified state
4. Sequence these states in the correct order and aggregate combinations into larger fragments
5. Review paths and look for independent, concurrent paths
6. Look for additional transitions and test both directions
7. Expand each transition with appropriate message event, guard condition, and action expression
8. Review and test the state machine diagram for the class
   - Make sure state are really state for the object in the class
   - Follow the life cycle of an object coming into existence and being deleted
   - Be sure the diagram covers all exception condition
   - Look again for concurrent paths and composite states

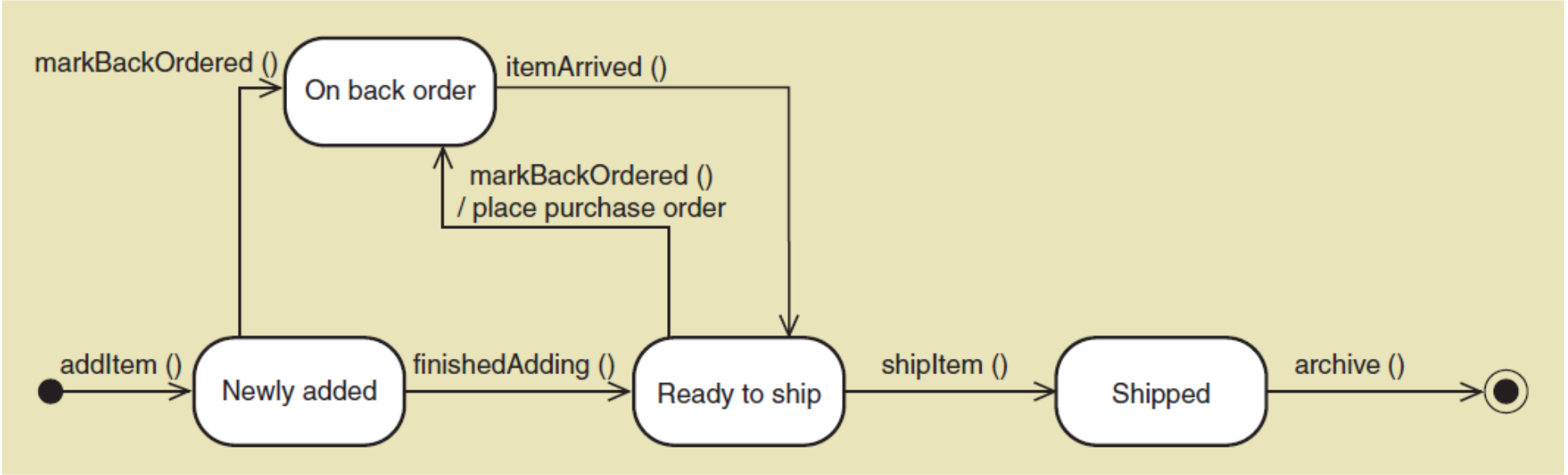
STATE MACHINE EXAMPLE

State	Transition causing exit from state
Newly added	finishedAdding
Ready to ship	shipItem
On back order	itemArrived
Shipped	No exit transition defined

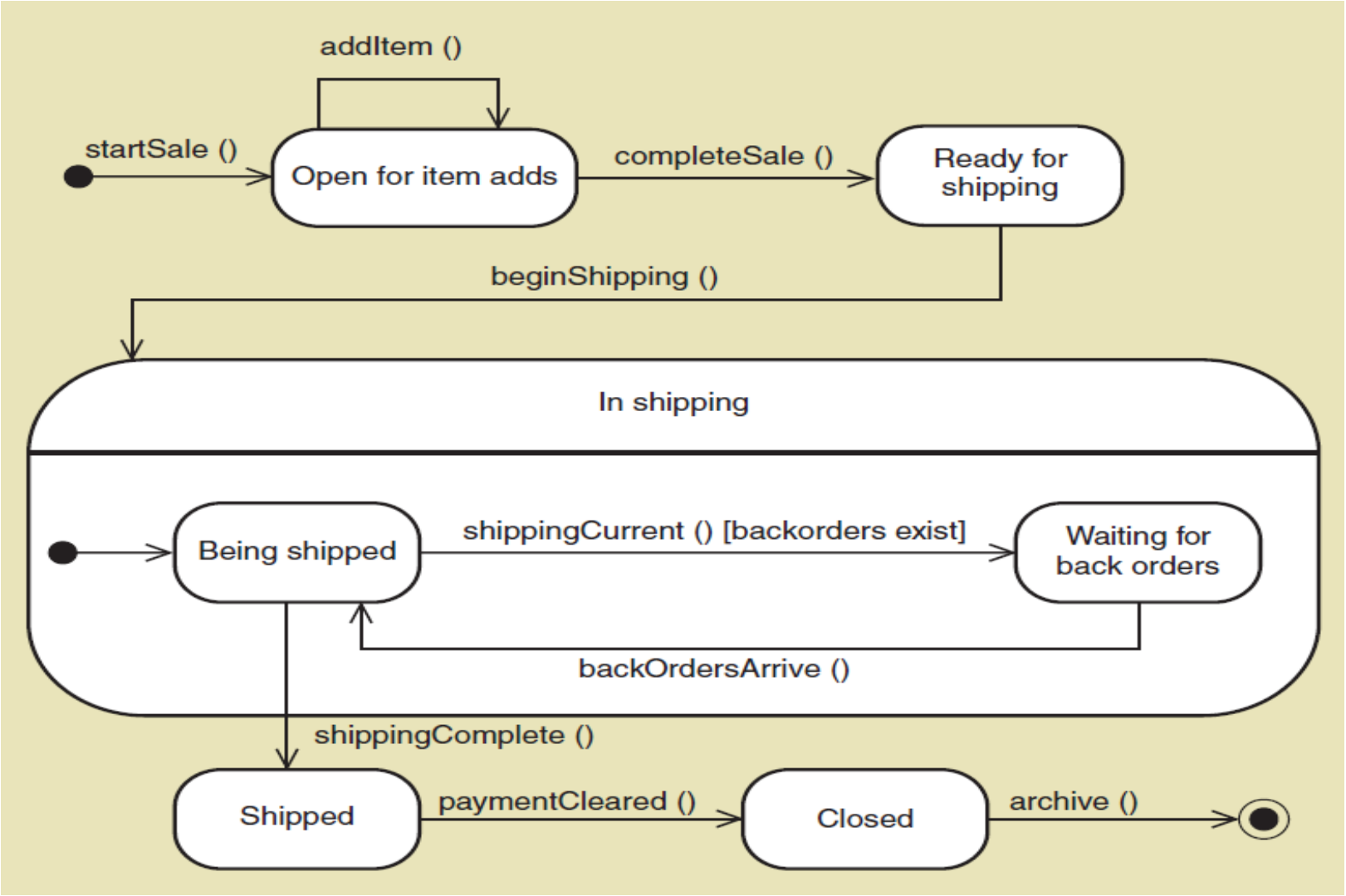


STATE MACHINE EXAMPLE

State	Transition causing exit from state
Newly added	finishedAdding
Ready to ship	shipItem
On back order	itemArrived
Shipped	No exit transition defined

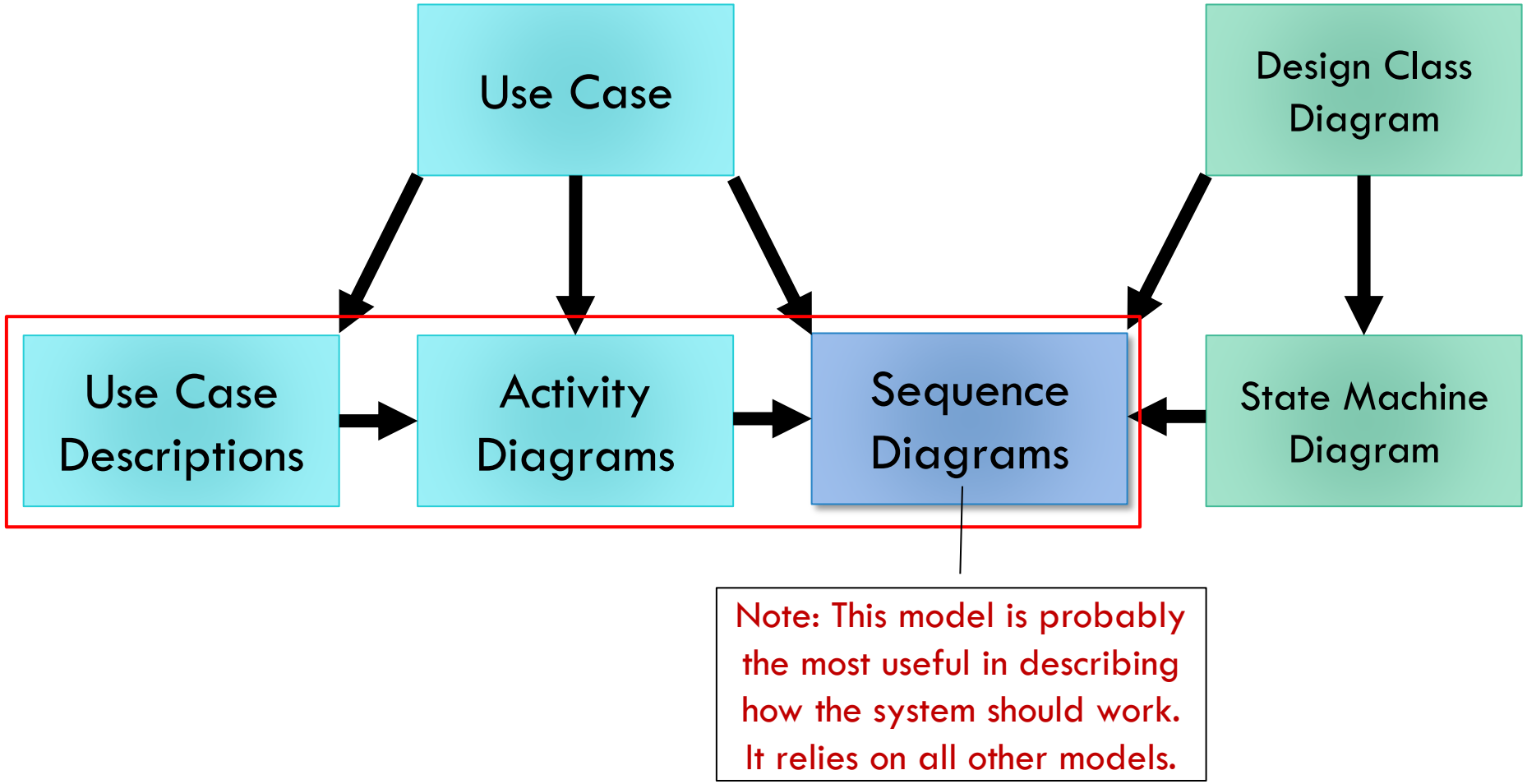


STATE MACHINE EXAMPLE



STRUCTURAL DESIGN MODELS (UML)

Behavioral Software Design Models using UML:



ACTIVITY DIAGRAM

Start with the Use Case; fully develop the use case, then augment with a workflow diagram that described the use case.

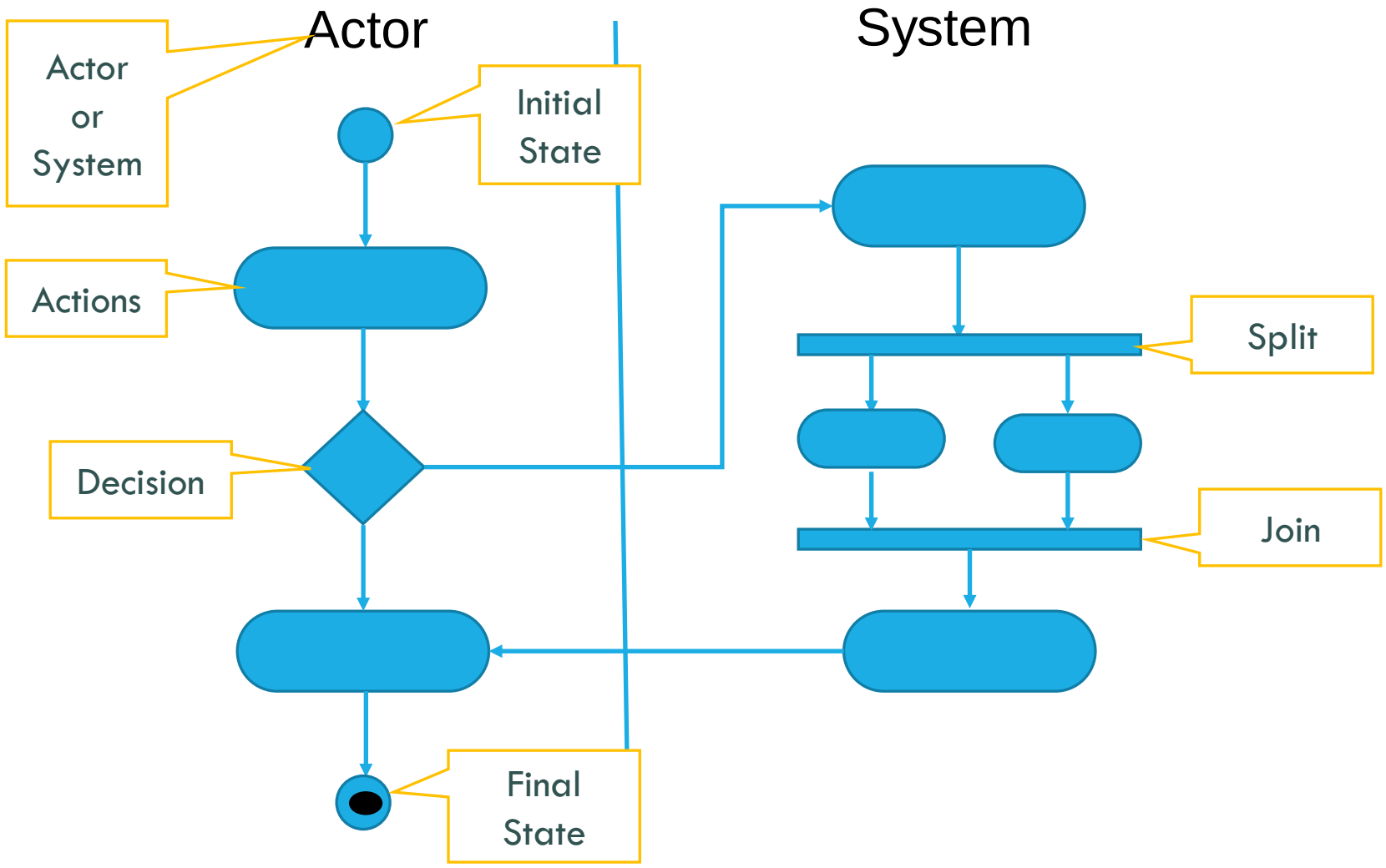
Especially useful for use cases that are complicated, not obvious, or not a generally understood process

- Example: Shipping a package is a generally well understood process.
  - Return Merchandise Authorization is not generally well understood, and should be clarified with an activity diagram in the fully developed use case.

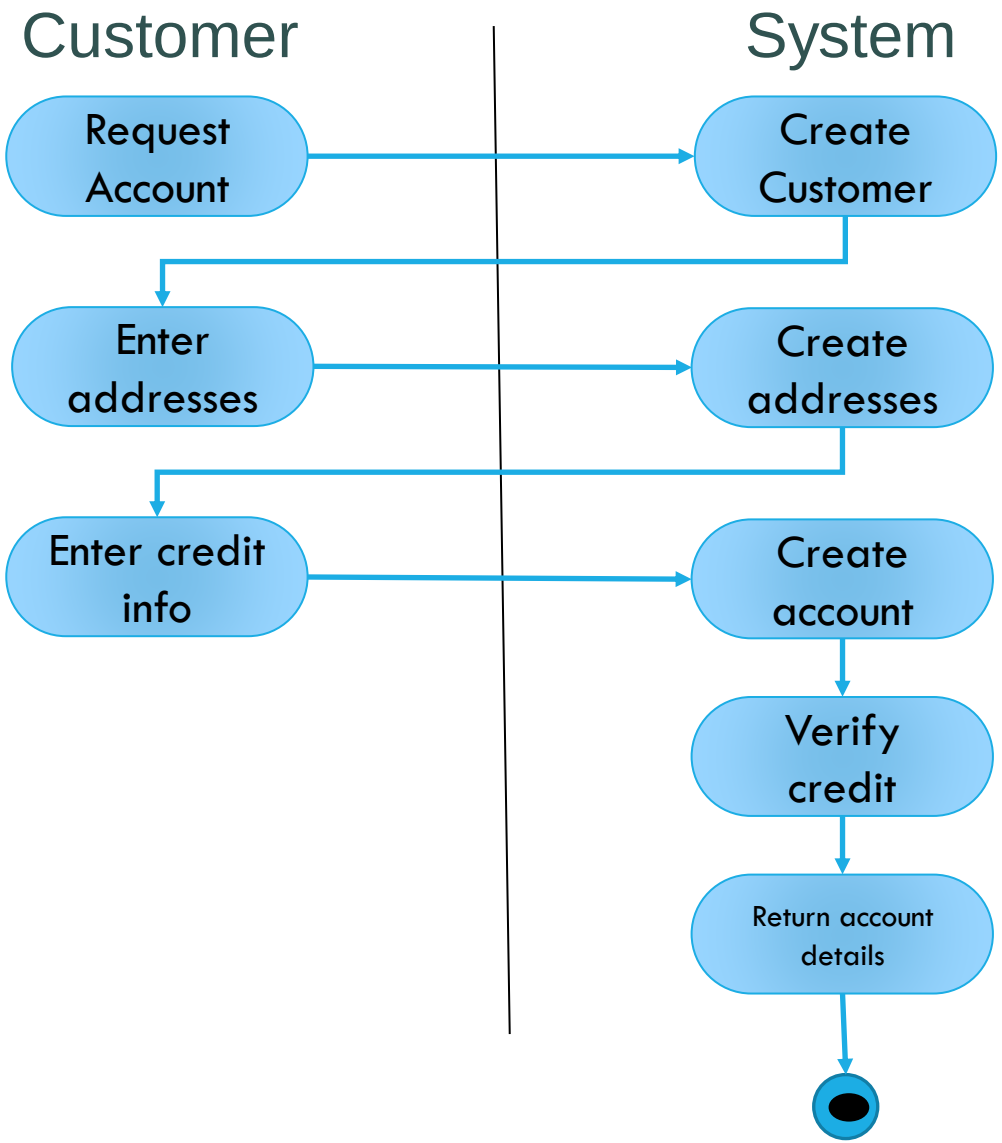
Typical fully developed use case description includes:

- Use case name
- Scenario (if needed)
- Triggering event
- Brief description
- Actors
- Related use cases (<<includes>>)
- Stakeholders
- Preconditions
- Post conditions
- Flow of activities
- Exception conditions

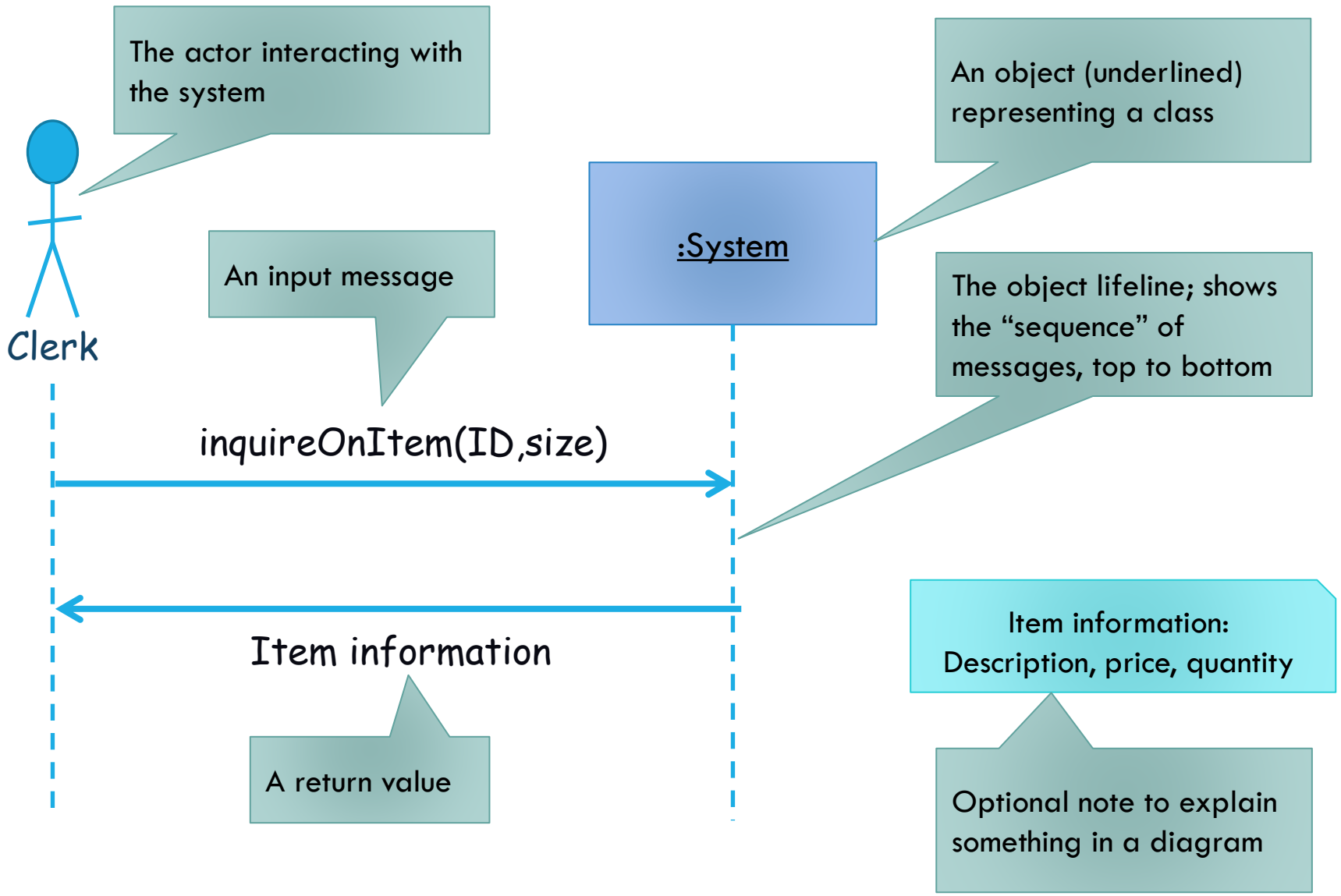
ACTIVITY DIAGRAM SYNTAX



ACTIVITY DIAGRAM EXAMPLE



SEQUENCE DIAGRAM SYNTAX



SEQUENCE DIAGRAM SYNTAX

An asterisk (*) indicates a repeating or looping of the message.

Brackets [ ] indicate a true/false condition. This is a test for that message only. If it evaluates to true, the message is sent. If it evaluates to false, the message isn't sent.

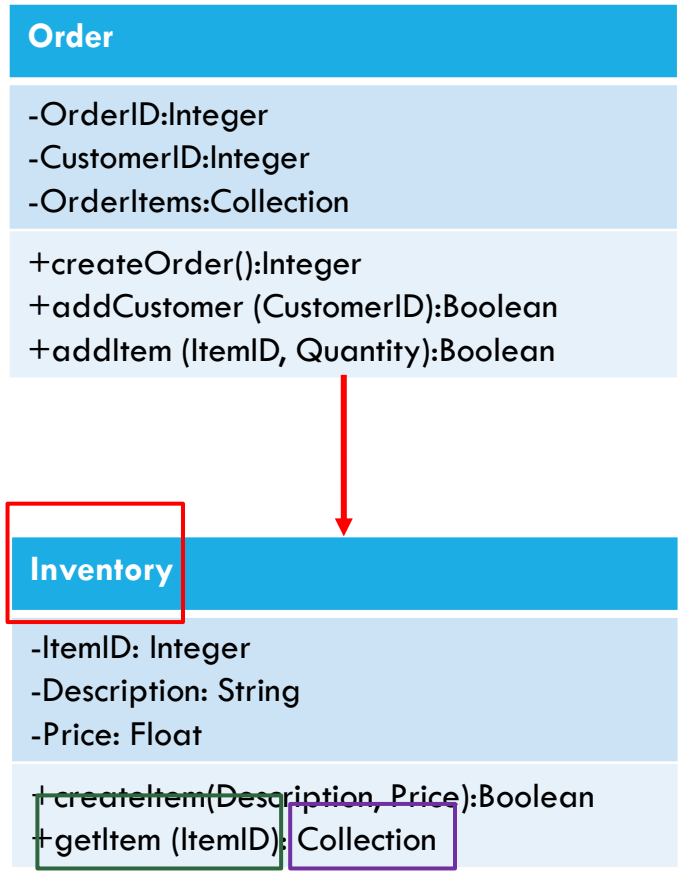
Message-name is the description of the requested service. It is omitted on dashed-line return messages, which only show the return data parameters.

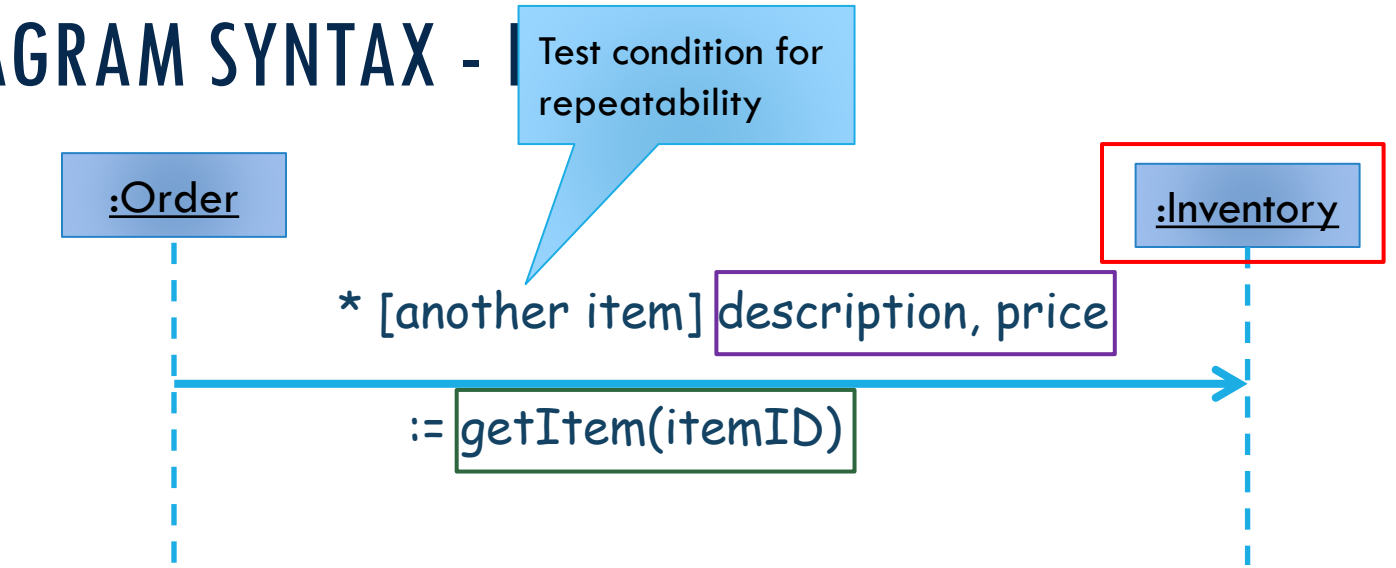
Parameter-list (with parenthesis on initiating messages and without parenthesis on return messages) shows the data that are passed with the message.

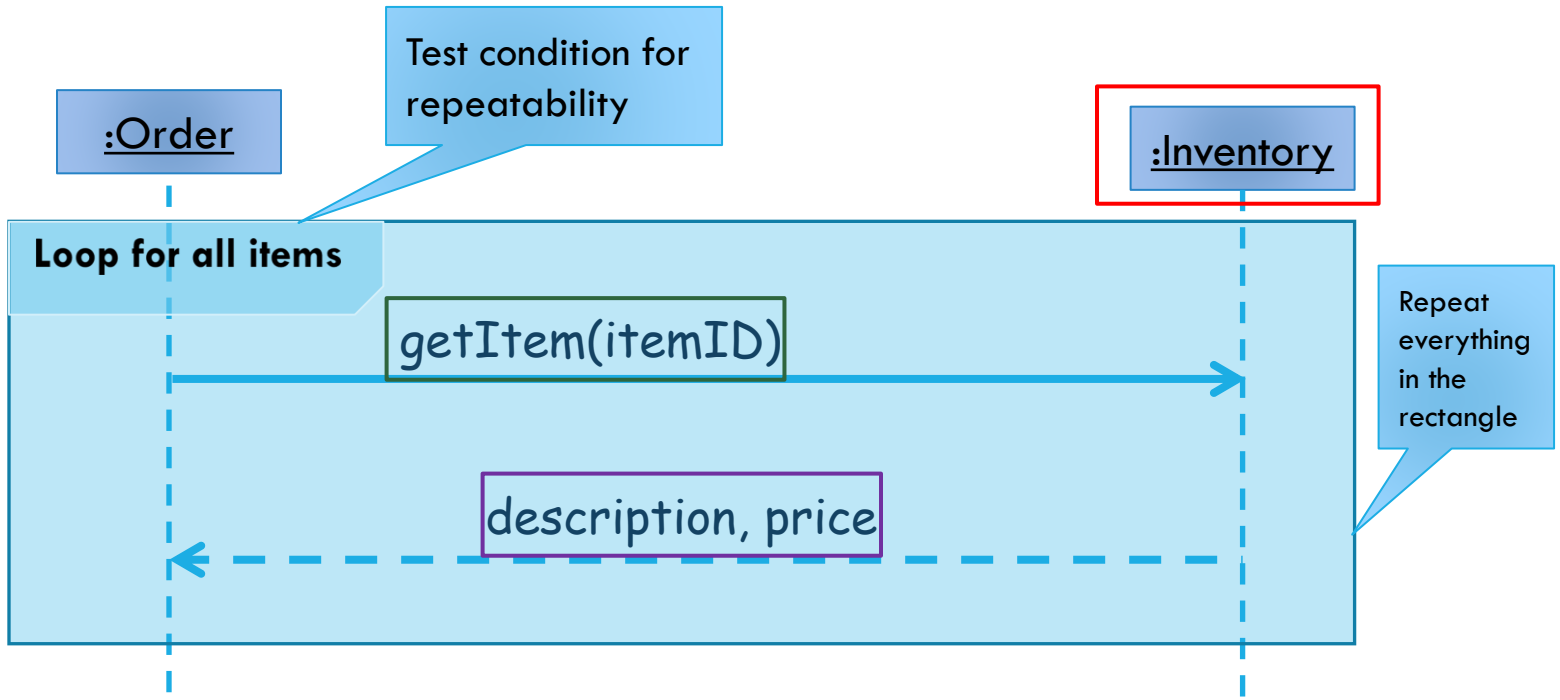
Return-value on the same line as the message (requires :=) is used to describe data being returned from the destination object to the source object in response to the message.

SEQUENCE DIAGRAM SYNTAX - I

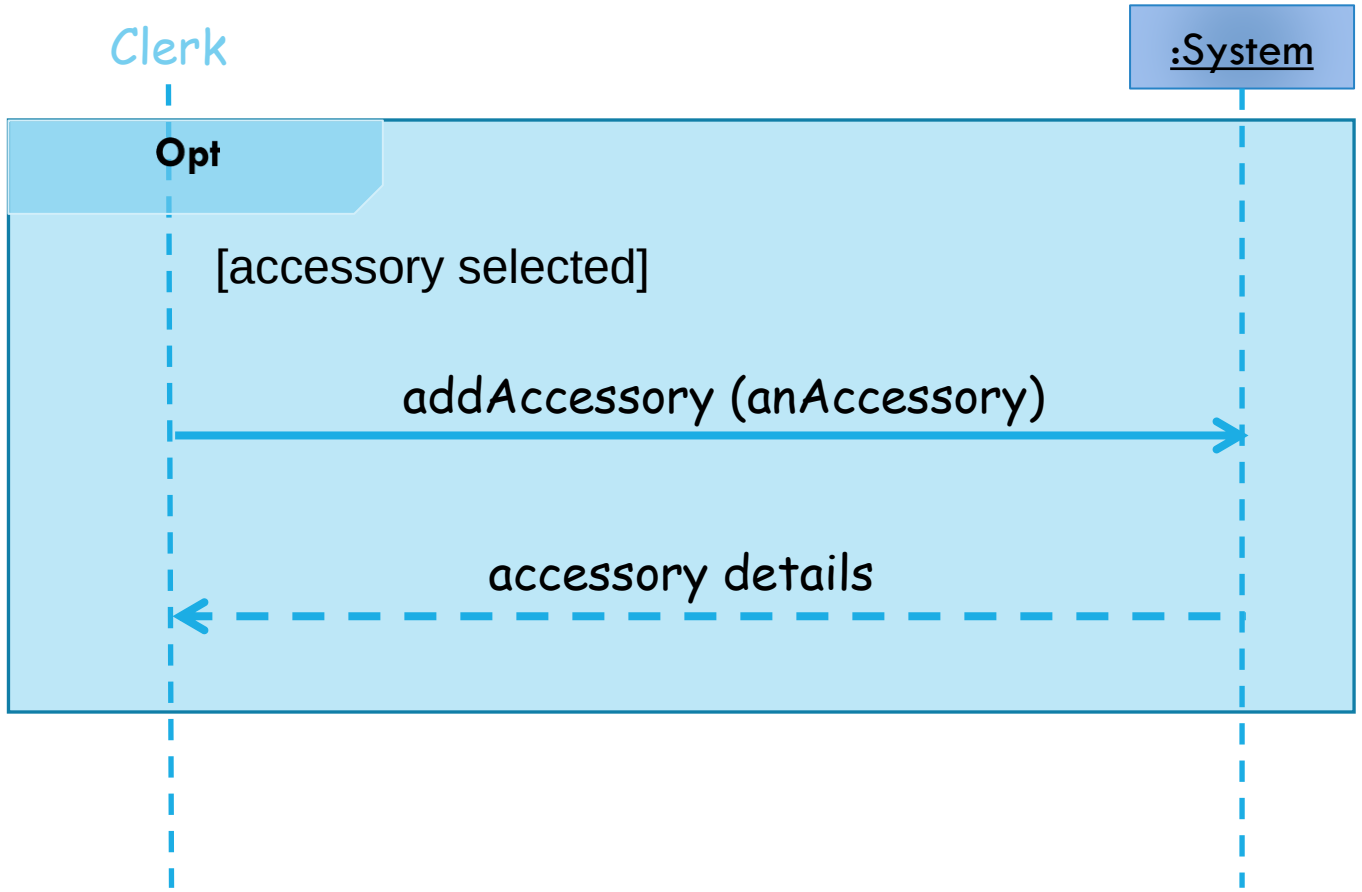
Note: Class and Sequence MUST be consistent with one another!



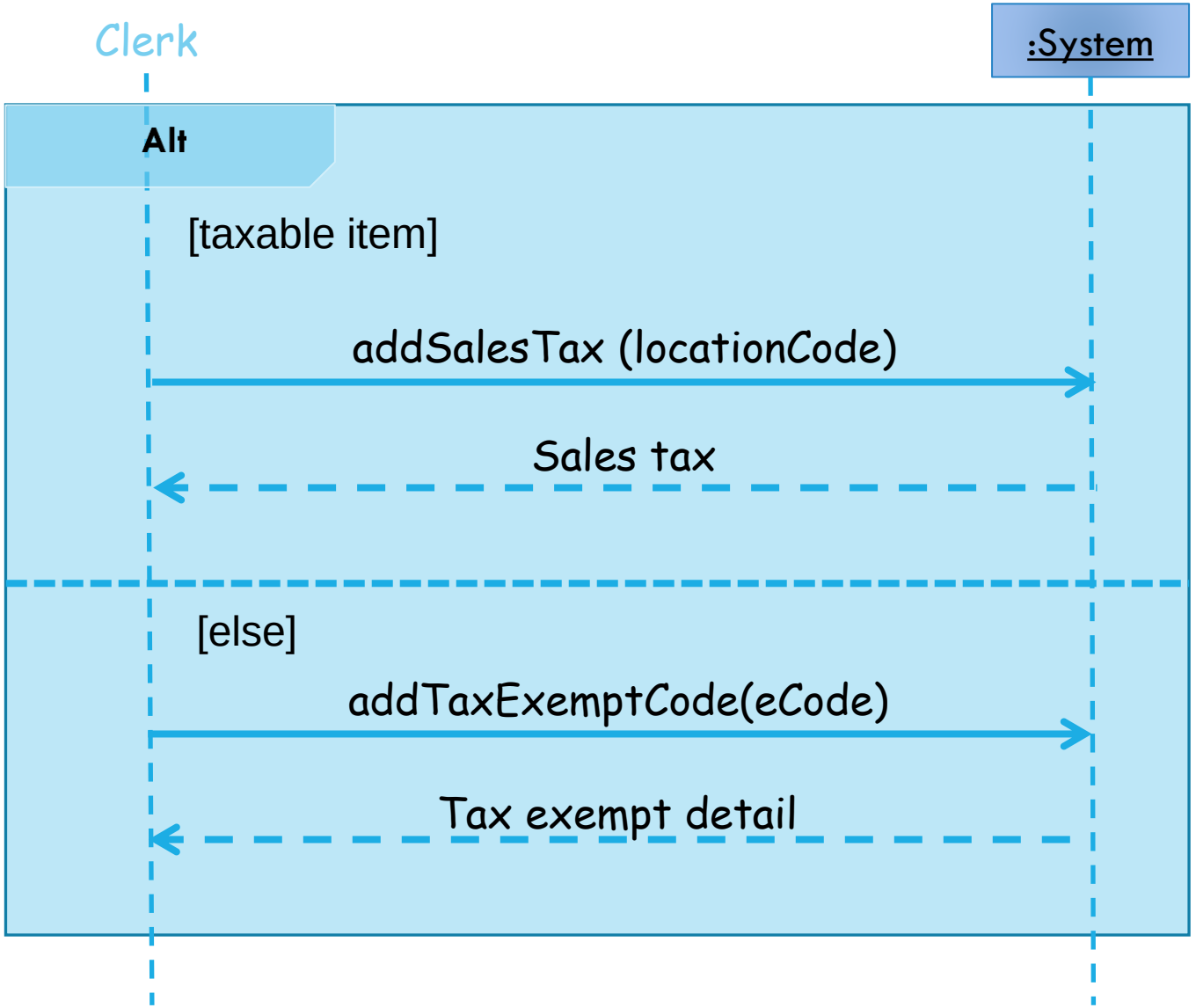




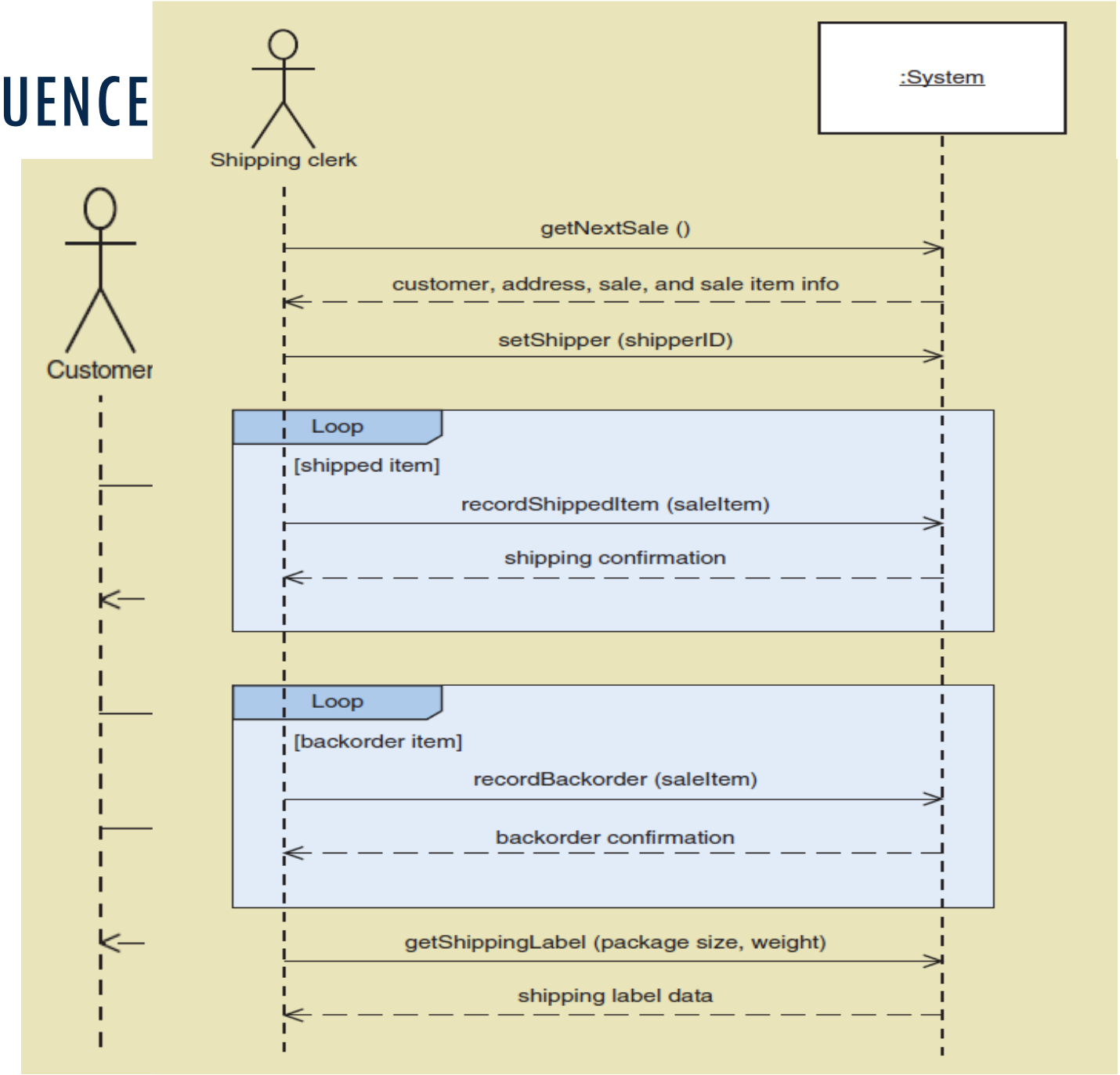
SEQUENCE DIAGRAM SYNTAX - CONDITIONS



SEQUENCE DIAGRAM SYNTAX - CONDITIONS



SEQUENCE



BALANCING FUNCTIONAL, STRUCTURAL AND BEHAVIORAL MODELS

A class on a class diagram must be associated with at least one use-case

An activity in an activity diagram and an event in a use-case description should be related to one or more operations on a class diagram

An object node on an activity diagram must be associated with an instance or an attribute on a class diagram

An attribute or an association/aggregation relationship on a class diagram should be related to the subject or object of a use-case

Sequence & communication diagrams must be associated with a use-case

Actors on sequence & communication diagrams or CRUDE matrices must be associated with actors within a use-case

Messages on sequence & communication diagrams, transitions on behavioral state machines and entries in a CRUD (Create,Read,update,Delete) matrix must relate to activities on an activity diagram and events in a use-case

All complex objects in activity diagrams must be represented in a behavioral state machine

EXAMPLE OF CRUD MATRIX

EBP \ BE	customer	Credit	Account receivable note	Order	Discounts	Invoice	Shipping schedule	Draft	Inventory	Warehouse voucher
Add Customer	C	C								
Add an Account receivable note	R	U	C			R				
Check Credit	R	R		R						
Receive order	R			C						
Calculate discounts				R	R					
Check inventory				R					R	
Calculate price				R	R					
Add discounts					C					
Issue invoice	R	R		R		C				
Schedule shipping						R	C			
Issue draft						R	R	C		
Add an Item									C	
Add a warehouse voucher	R					R			U	C

https://www.researchgate.net/figure/The-CRUD-Matrix-for-Sales-Department-Scenario-16_fig4_221418427

FACTORING

Creating modules that account for similarities and differences between units of interest

New classes formed through a:

- Generalization (a-kind-of) relationship, or a
- Aggregation (has-parts) relationship

Abstraction—create a higher level class (e.g., create an Employee class from a set of job positions)

Refinement—create a detailed class (e.g., create a secretary or bookkeeper from the Employee class)

DESIGN PATTERNS

Use layers to represent and separate elements of the software architecture

- Easier to understand a complex system
- Example:
  - Model-view-controller (MVC) architecture
  - Separates application logic from user interface
- Proposed layers:
  - Foundation (e.g., container classes)
  - Problem domain (e.g., encapsulation, inheritance, polymorphism)
  - Data management (e.g., data storage and retrieval)
  - User interface (e.g., data input forms)
  - Physical architecture (e.g., specific computers and networks)

PACKAGES

Packages group together similar components (e.g., use-cases, class diagrams)

Package diagrams show the packages and their relationships

- Aggregation & association relationships are possible
- Packages may be dependent upon one another
  - If one package is modified, others that depend on it may also require modification

COUPLING

- A quantitative measure of how closely related classes are linked (tightly or loosely coupled)
- Two classes are tightly coupled of there are lots of associations with another class
- Two classes are tightly coupled if there are lots of messages to another class
- It is best to have classes that are **loosely coupled**
- If deciding between two alternative designs, choose the one where overall coupling is less

COHESION

- A quantitative measure of the focus or unity of purpose within a single class (high or low cohesiveness)
- One class has high cohesiveness if all its responsibilities are consistent and make sense for purpose of the class (a customer carries out responsibilities that naturally apply to customers)
- One class has low cohesiveness if its responsibilities are broad or makeshift
- It is best to have classes that are **highly cohesive**
- If deciding between two alternative designs, choose the one where overall cohesiveness is high

PROTECTION FROM VARIATIONS

- A design principle that states parts of a system unlikely to change are separated (protected) from those that will surely change
- Separate user interface forms and pages that are likely to change from application logic
- Put database connection and SQL logic that is likely to change in a separate classes from application logic
- Use adaptor classes that are likely to change when interfacing with other systems
- If deciding between two alternative designs, choose the one where there is protection from variations

INDIRECTION

- A design principle that states an intermediate class is placed between two classes to decouple them but still link them
- A controller class between UI classes and problem domain classes is an example
- Supports low coupling
- Indirection is used to support security by directing messages to an intermediate class as in a firewall
- If deciding between two alternative designs, choose the one where indirection reduces coupling or provides greater security

OBJECT RESPONSIBILITY

- A design principle that states objects are responsible for carrying out system processing
- A fundamental assumption of OO design and programming
- Responsibilities include “knowing” and “doing”
- Objects know about other objects (associations) and they know about their attribute values. Objects know how to carry out methods, do what they are asked to do.
- Note that CRC cards and the design in the next chapter involve assigning responsibilities to classes to carry out a use case.
- If deciding between two alternative designs, choose the one where objects are assigned responsibilities to collaborate to complete tasks (don’t think procedurally).

FUNCTIONAL AND NON-FUNCTIONAL SECURITY

- Why should this be a part of the system?
- What are the constraints on this requirement?
- What are the dependencies on this requirement?
- Who are the stakeholders for this requirement?

ERROR HANDLING

- Fail Case
- Consequence of Failure
- Associated Risks

- What are the exceptions to the normal case for this requirement?
- What sensitive information is included in this requirement?
- What are the consequences if the condition of this requirement are violated
- What happens if this requirement is intentionally violated?