



Object Oriented Analysis & Design

Module-1 (RL 1.2.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Overview of Unified Process (UP)

The Unified Process (UP)

- For simple systems, it might be feasible to sequentially define the whole problem, design the entire solution, build the software, and then test the product.
- For complex and sophisticated systems, this linear approach is not realistic.
- The **Unified Process (UP)** is a process for building object-oriented systems.
- The goal of the UP is to enable the production of high quality software that meets users needs within predictable schedules and budgets.

The Unified Process (UP)

- Development is organized into a series of short fixed-length mini-projects called **iterations**.
- The outcome of each iteration is a tested, integrated and executable system.
- An iteration represents a complete development cycle: it includes its own treatment of requirements, analysis, design, implementation and testing activities.
- UML is compulsory to use for Modeling

Iteration Length and Timeboxing

- The UP recommends short iteration lengths to allow for rapid feedback and adaptation.
- Long iterations increase project risk.
- Iterations are fixed in length (**timeboxed**). If meeting deadline seems to be difficult, then remove tasks or requirements from the iteration and include them in a future iteration.
- The UP recommends that an iteration should be between two and six weeks in duration.



Object Oriented Analysis & Design

Module-1 (RL 1.2.2)

Sanjay Joshi

BITS Pilani

Pilani|Duba|Goa|Hyderabad





UP : An Iterative & Evolutionary Development

UP: An Iterative & Evolutionary Development



- The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations with feedback and adaptation.
- The system grows incrementally over time, iteration by iteration.
- The system may not be eligible for production deployment until after many iterations.

UP: An Iterative & Evolutionary Development



- The output of an iteration is not an experimental prototype but a production subset of the final system.
- Each iteration tackles new requirements and incrementally extends the system.
- An iteration may occasionally revisit existing software and improve it.

UP: An Iterative & Evolutionary Development



[Iteration N]

Requirements – Analysis - Design- Implementation - Testing



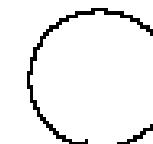
[Iteration N+1]

Requirements – Analysis - Design- Implementation - Testing



Feedback from iteration N leads to refinement and adaptation of the requirements and design in iteration N+1.

The system grows incrementally.



Embracing Change

- Stakeholders usually have changing requirements.
- Each iteration involves choosing a small subset of the requirements and quickly design, implement and testing them.
- This leads to rapid feedback, and an opportunity to modify or adapt understanding of the requirements or design.



Object Oriented Analysis & Design

Module-1 (RL 1.2.3)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



UP : Phases & Disciplines

Phases of the Unified Process

- A UP project organizes the work and iterations across four major phases:
 - Inception - Define the scope of project.
 - Elaboration - Plan project, specify features, baseline architecture.
 - Construction - Build the product
 - Transition - Transition the product into end user community

The UP Disciplines

Process Disciplines

Business Modeling

Requirements

Analysis & Design

Implementation

Test

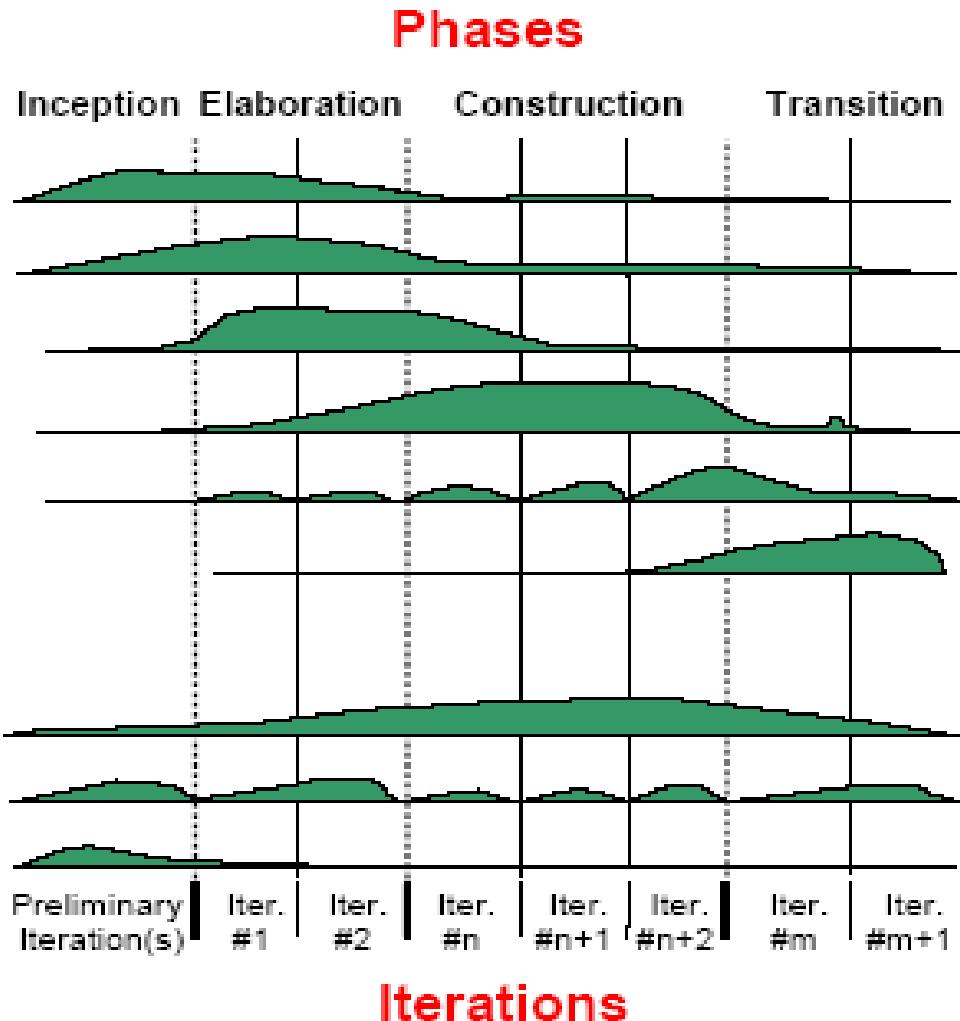
Deployment

Supporting Disciplines

Configuration Mgmt

Management

Environment





Object Oriented Analysis & Design

Module-1 (RL 1.3.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Agile Principles & Manifesto

Common Fears for Developers



- The project will produce the wrong product.
- The project will produce a product of inferior quality.
- The project will be late.
- We'll have to work 80 hour weeks.
- We'll have to break commitments.
- We won't be having fun.

What is “Agility”?

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed

Yielding ...

- Rapid, incremental delivery of software

An Agile Process

- Is driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple ‘software increments’
- Adapts as changes occur

Principles of Agility

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development.
- Business people and developers must work together daily throughout the project.

Principles of Agility

- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace continuously.



The Manifesto for Agile Software Development

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value”

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan



Object Oriented Analysis & Design

Module-1 (RL 1.3.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



eXtreme Programming (XP)



eXtreme Programming (XP)



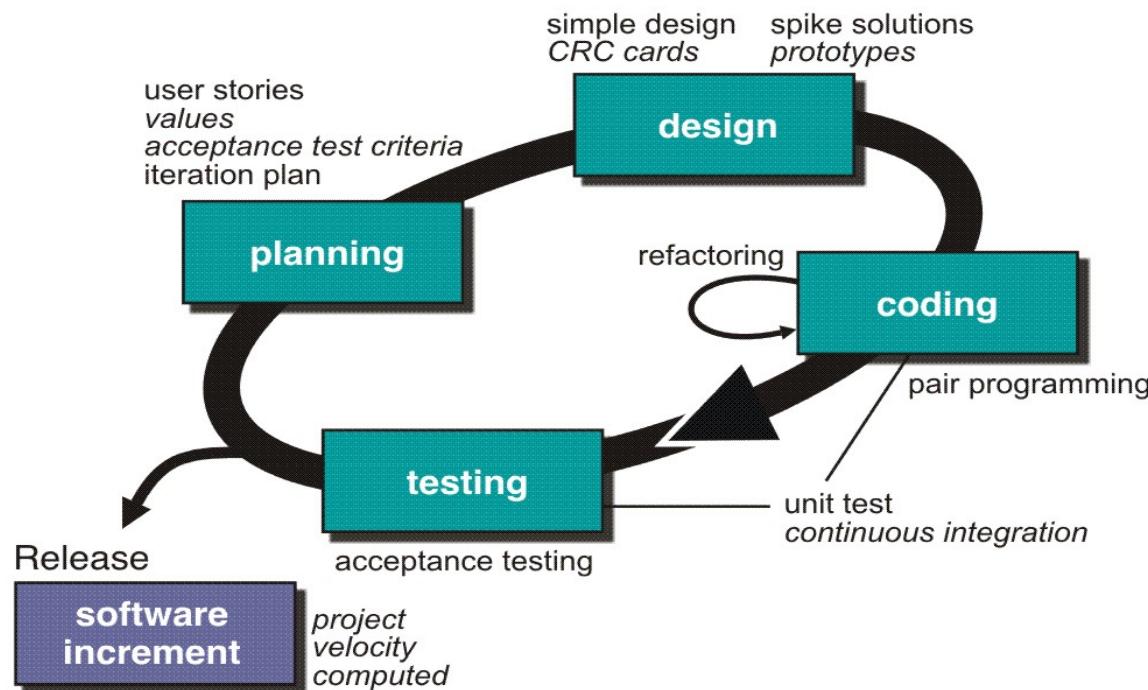
- The most widely used agile process, originally proposed by Kent Beck
- XP Planning
 - Begins with the creation of user stories
 - Agile team assesses each story and assigns a cost
 - Stories are grouped to form a deliverable increment
 - A commitment is made on delivery date
 - After the first increment project velocity (measure of how much work is getting done on your project) is used to help define subsequent delivery dates for other increments



eXtreme Programming (XP)

- XP Design
 - Follows the KIS principle
 - Encourage the use of CRC cards
 - For difficult design problems, suggests the creation of spike solutions — a design prototype
 - Encourages refactoring — an iterative refinement of the internal program design
- XP Coding
 - Recommends the construction of a unit test for a story *before* coding commences
 - Encourages pair programming
- XP Testing
 - All unit tests are executed daily
 - Acceptance tests are defined by the customer and executed to assess customer visible functionality

eXtreme Programming (XP)





Object Oriented Analysis & Design

Module-1 (RL 1.3.3)

Sanjay Joshi

BITS Pilani

Pilani|Duba|Goa|Hyderabad



BITS Pilani

Pilani | Dubai | Goa | Hyderabad



SCRUM

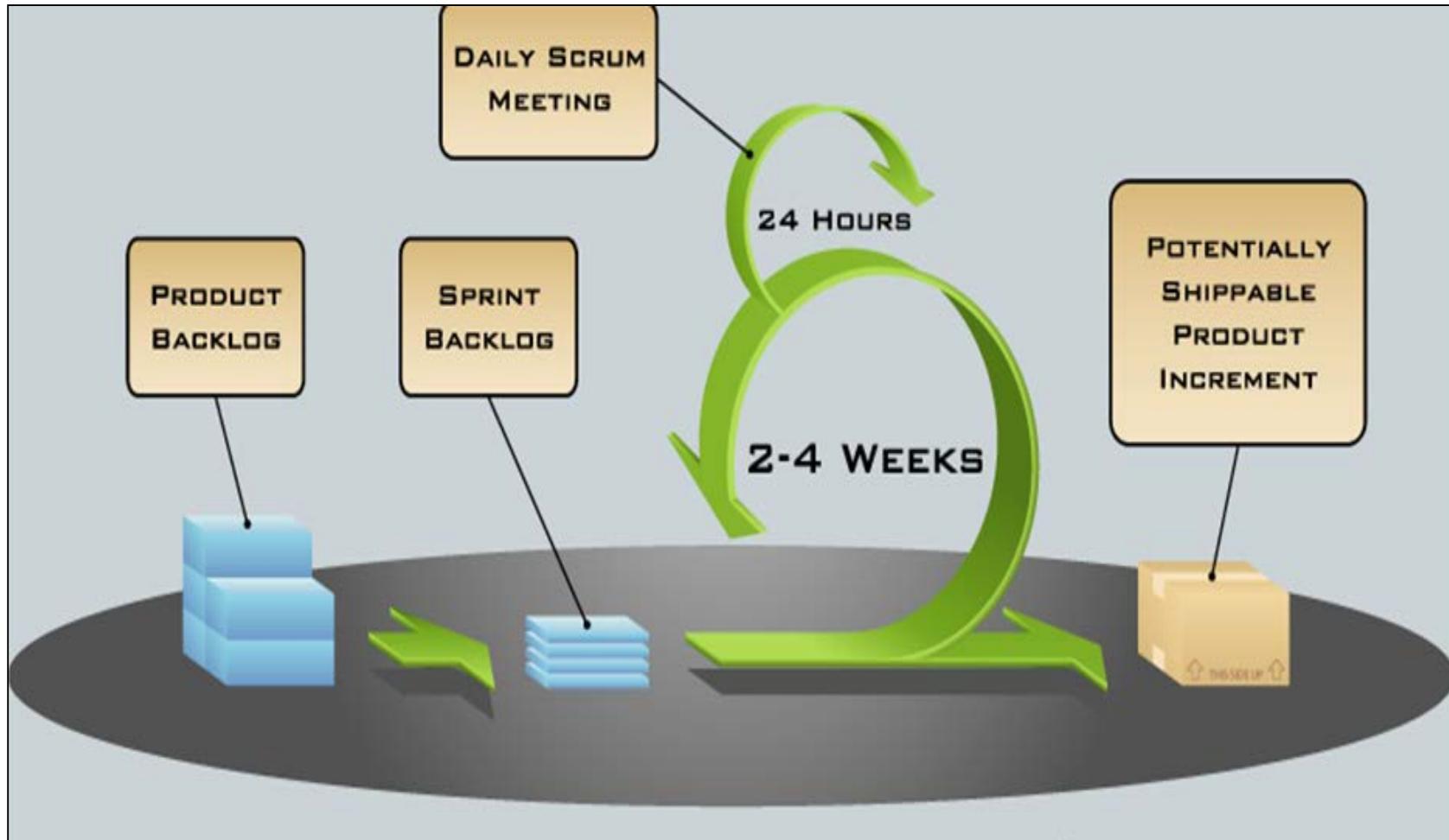
SCRUM

- Project Management Methodology
- Wrapper for existing engineering practices
- Advocates small team (7-9)
- Consists of three roles
 1. Product Owner
 2. Scrum Master
 3. Team
- Tracks progress regularly

Scrum Practices

- Sprint Planning Meeting
- Sprint Daily Scrum
- Sprint Review Meeting
- Sprint Retrospective

SCRUM Process Flow





Object Oriented Analysis & Design

Module-1 (RL 1.3.4)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Agile Modelling

Agile Modeling (AM)

- Agile Modeling (AM) is a practice based methodology for effective modeling and documentation of s/w based systems
- Following are modeling principles
 - Model with a purpose
 - Use multiple models
 - Travel light
 - Content is more important than representation
 - Know the models & tools you use
 - Adapt locally



Object Oriented Analysis & Design

Module-1 (RL 1.3.5)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi

BITS Pilani

Pilani|Dubai|Goa|Hyderabad



Test Driven Development

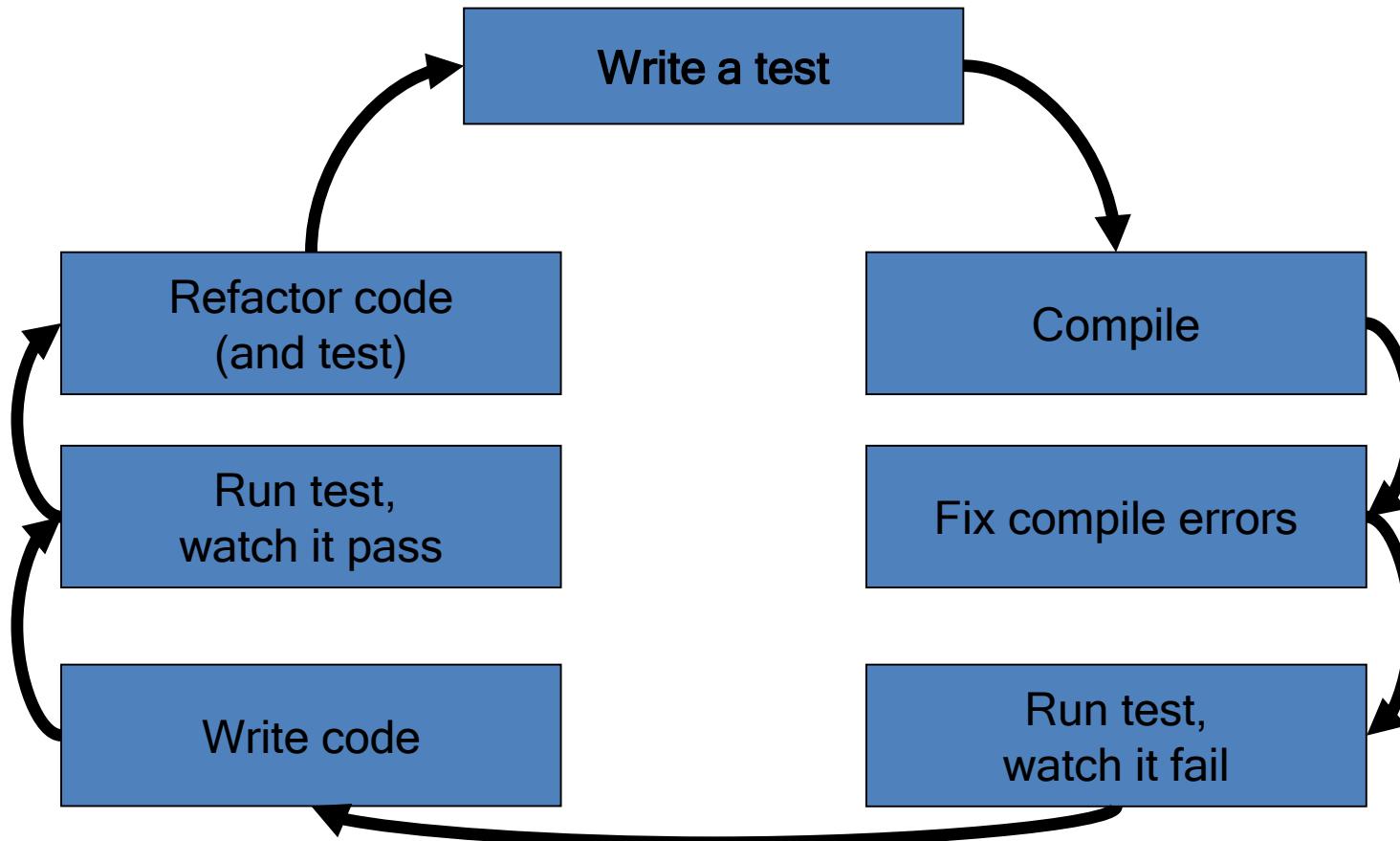
Test Driven Development

- TDD is a technique whereby you write your test cases **before** you write any implementation code
- Tests drive or dictate the code that is developed
- An indication of “intent”
 - Tests provide a specification of “what” a piece of code actually does
 - Some might argue that “tests are part of the documentation”

Why is TDD

- TDD can lead to more modularized, flexible, and extensible code
- Clean code
- Leads to better design
- Better code documentation
- More productive
- Good design

Introduction to TDD



Introduction to TDD

- In Extreme Programming Explored (The Green Book), Bill Wake describes the test / code cycle:
 1. Write a single test
 2. Compile it. It shouldn't compile because you've not written the implementation code
 3. Implement **just enough** code to get the test to compile
 4. Run the test and see it **fail**
 5. Implement **just enough** code to get the test to pass
 6. Run the test and see it **pass**
 7. **Refactor** for clarity
 8. Repeat

Introduction to TDD

- Example
- Given a string swap the last two characters
- Sample input & output set is

Blank String => Blank String

A => A

AB => BA

ABCD => ABDC



Object Oriented Analysis & Design

Module-1 (RL 1.3.6)

Sanjay Joshi

BITS Pilani

Pilani|Duba|Goa|Hyderabad





Refactoring & Continuous Integration

Refactoring

- What is Refactoring?
- Revisit, Optimize
- Refactoring in
 - Programming
 - Design

Continuous Integration (CI)

- 2 Ways of testing
 - Big-bang fashion
 - Continuous Integration (CI)
- You decide which one is better
- Need to take a call with many parameters



Object Oriented Analysis & Design

Module-1 (RL 1.4.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Introduction to OOA & OOD

Object-Oriented Analysis

- An investigation of the problem (rather than how a solution is defined)
- During OO analysis, there is an emphasis on finding and describing the objects (or concepts) in the problem domain.
 - For example, concepts in a Library Information System include *Book* and *Library*.

Object-Oriented Design

- Emphasizes a conceptual solution that fulfills the requirements.
- Need to define software objects and how they collaborate to fulfill the requirements.
 - For example, in the Library Information System, a *Book* software object may have a *title* attribute and a *getChapter* method.
- Designs are implemented in a programming language.
 - In the example, we will have a *Book* class in Java.



Object Oriented Analysis & Design

Module-1 (RL 1.4.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Overview of UML

What is UML?

- Unified Modeling Language
 - modeling language to draw diagrams in Uniform way Object Oriented Analysis & Design
- Need of UML
- Who uses UML?



Object Oriented Analysis & Design

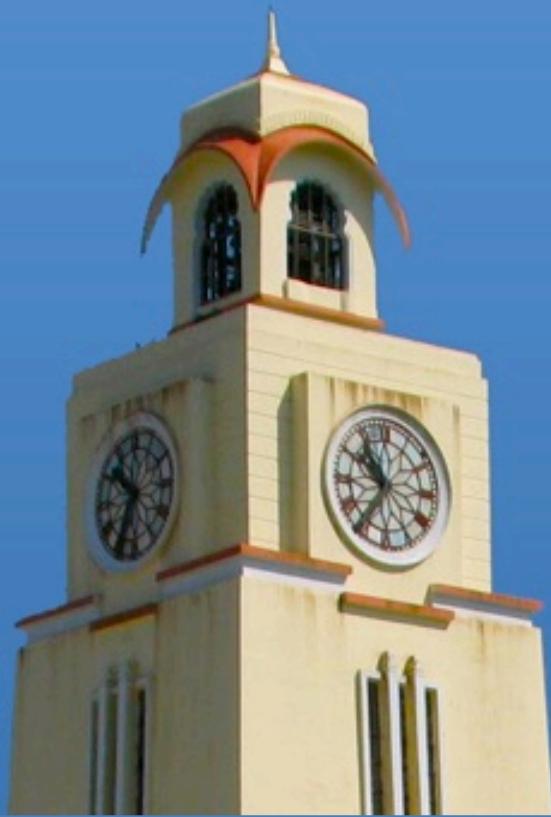
Module-1 (RL 1.5.1)



BITS Pilani

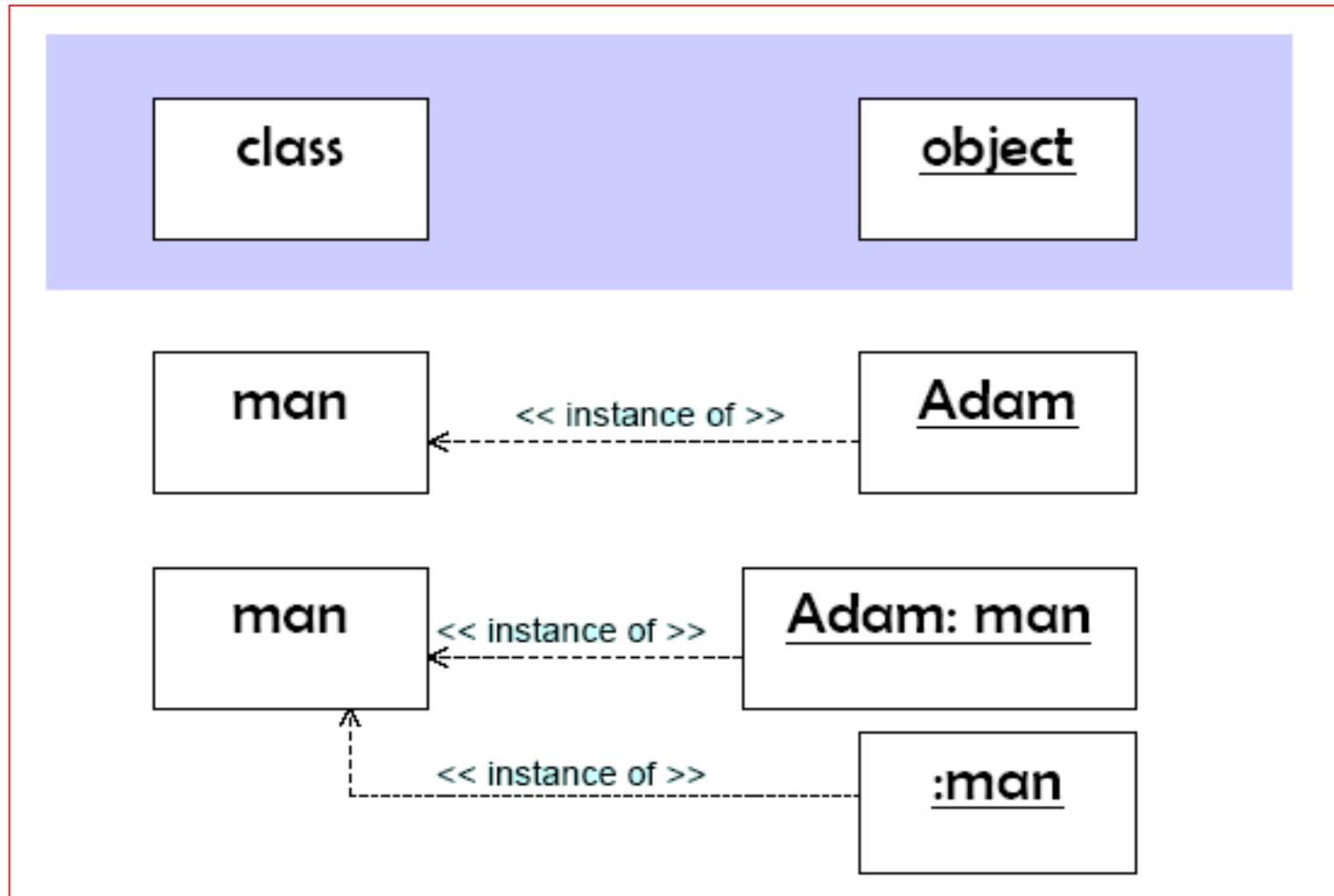
Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Concept of Class & Object

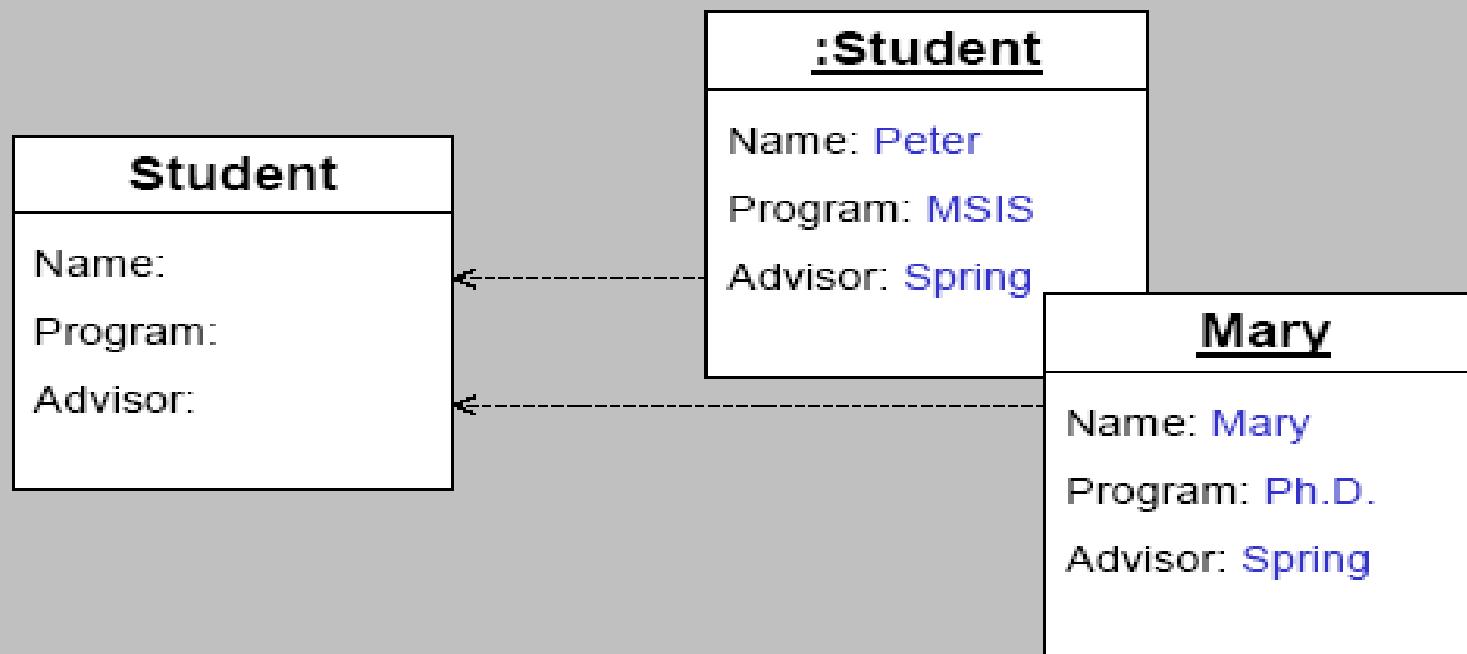
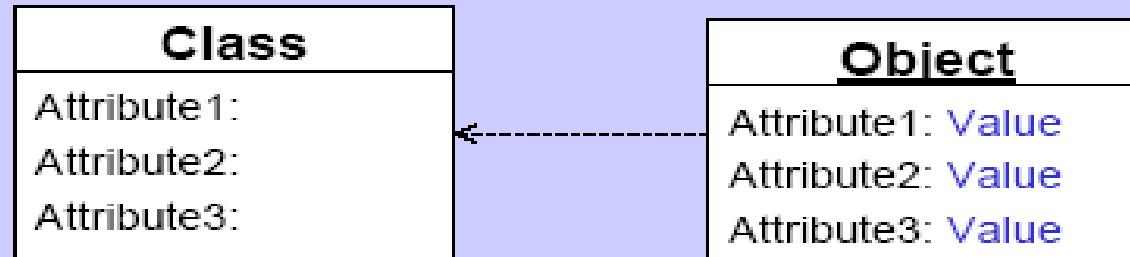
Class and Object: in UML



Attributes

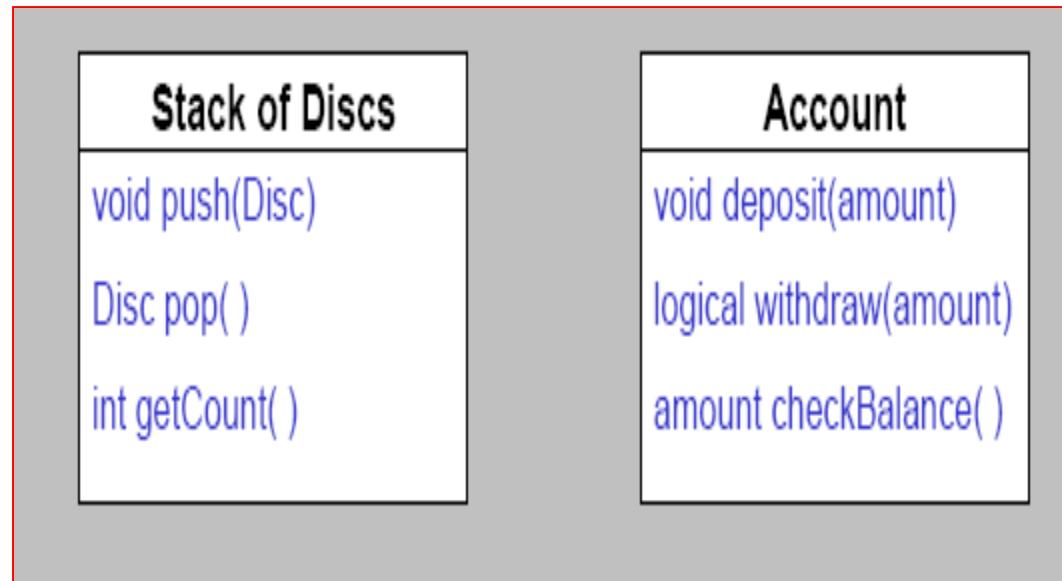
- The class captures the abstraction of properties in the set of objects.
- **An attribute** of a class is identified by **name**, and it identifies a property of the objects of the class, for which each object takes a **value**.

Attributes: in UML



Operations

- The operations are the “responsibilities” – the things we can ask an object to do.
- Note that these are classes
- Should we have open() and close() with Account?



Operations: in UML

Class

```
return-type name(...parameters list...)  
· · ·  
create(...) // Constructor - CTOR  
destroy( ) // Destructor - DTOR
```

:Account

deposit(\$1000)



```
void deposit(amount)  
logical withdraw(amount)  
amount checkBalance( )
```



Object Oriented Analysis & Design

Module-1 (RL 1.5.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi

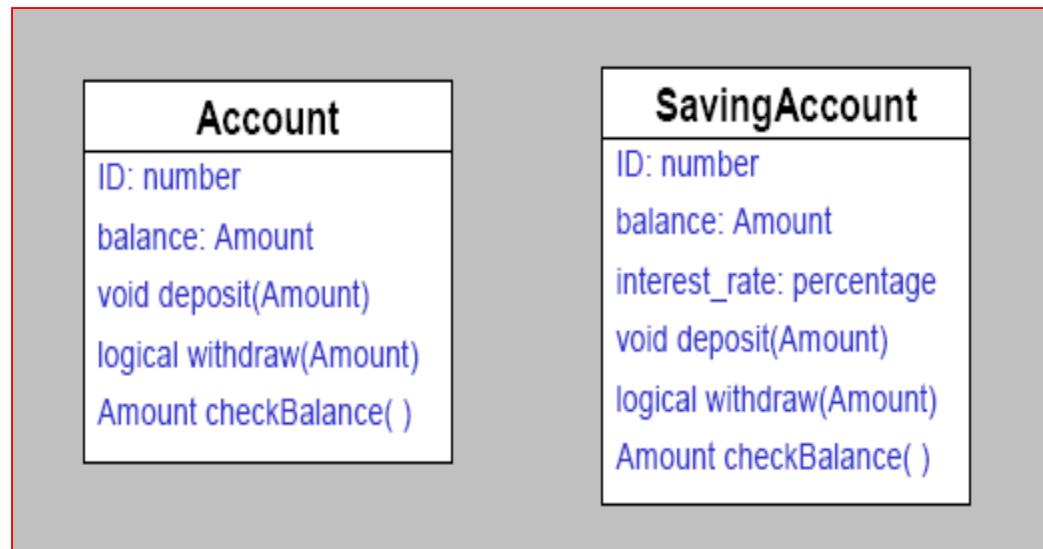


Class Relationships in UML

Inheritance: superclass & subclass

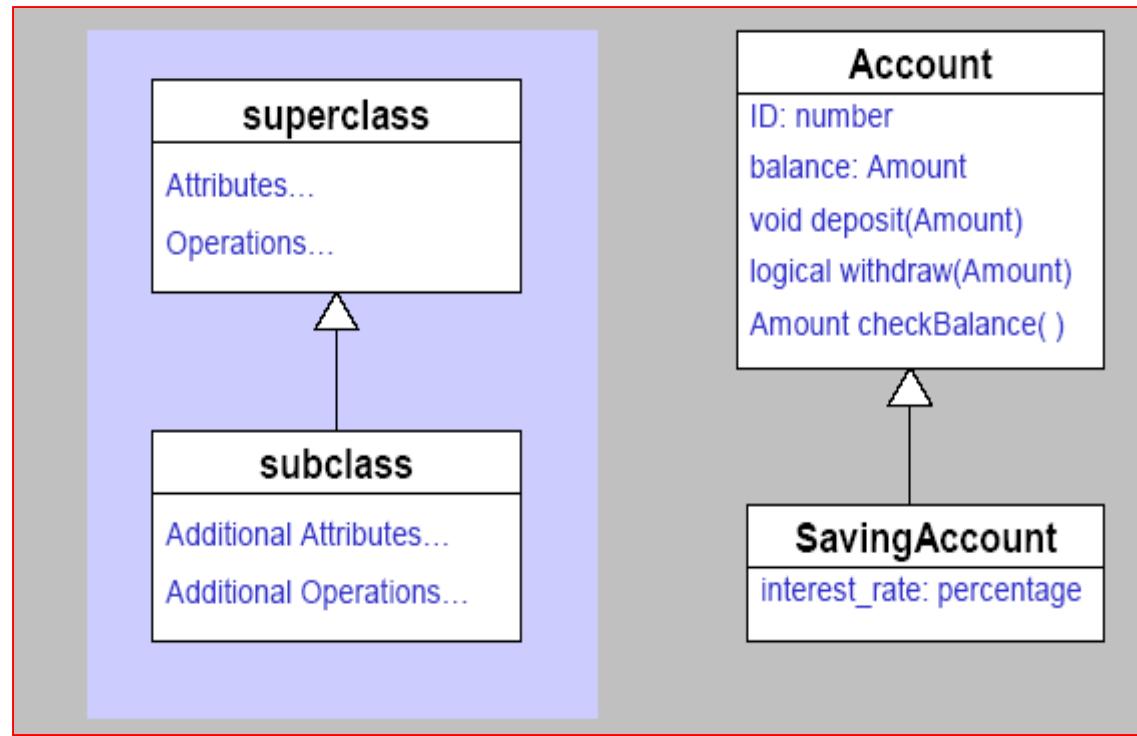


- A class may be the subset of another class...
- A saving account is also an account.
- Not all accounts are saving accounts.



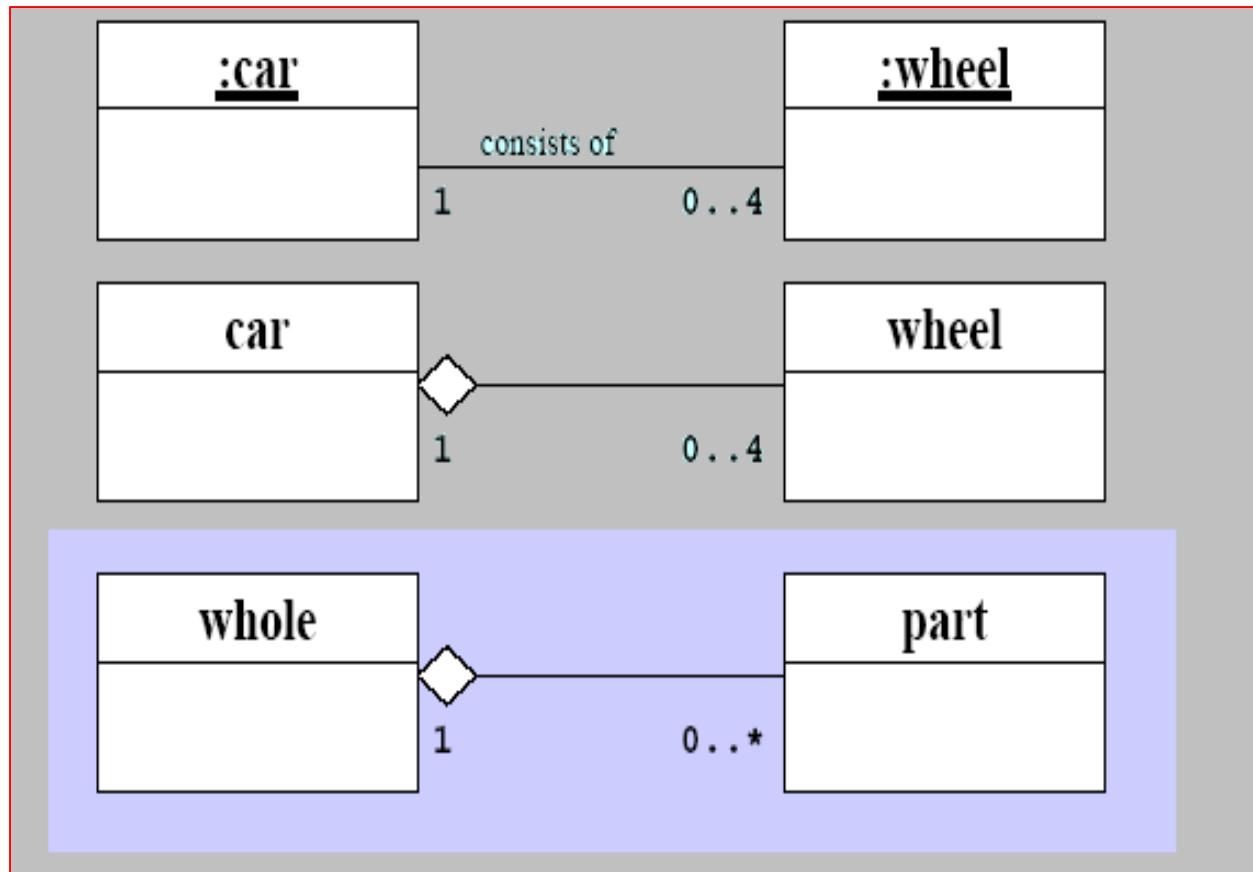
Inheritance in UML

- The subclass inherits from the superclass the attributes as well as the operations.



Aggregation: in UML

- An object may be part of another object...





Object Oriented Analysis & Design

Module-2 (RL 2.1.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Point of Sale (PoS) Case Study



PoS (Point Of Sale) System

- Mall System
- Customer purchasing goods
- Cashier making entry in the system
- Salesperson is having commission
- Customer can return goods as well.



Object Oriented Analysis & Design

Module-2 (RL 2.1.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Requirement Categories – Functional & Non Functional Requirements

Requirement Categorization

- Functional requirements
 - Features and capabilities.
 - Recorded in the Use Case model (see next), and in the systems features list of the Vision artifact.
- Non-functional (or quality requirements)
 - Usability (Help, documentation, ...), Reliability (Frequency of failure, recoverability, ...), Performance (Response times, availability, ...), Supportability (Adaptability, maintainability, ...)
 - Recorded in the Use Case model or in the Supplementary Specifications artifact.



Object Oriented Analysis & Design

Module-2 (RL 2.1.3)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



What is Use Case Diagram & Use Cases?

What is Use Case Diagram (UCD)



- It shows interaction among actors (external parties) and Use Cases (Modules) of the System
- It is not Data Flow Diagram !!
- Significance of Use Case Diagram

What is Use Case?

- Use Cases is detailing of the Use Case Diagram (UCD)
 - Use Cases are textual artifacts
 - Use Case depicts functional requirements primarily.
 - Significance of Use Case
-



Object Oriented Analysis & Design

Module-2 (RL 2.2.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

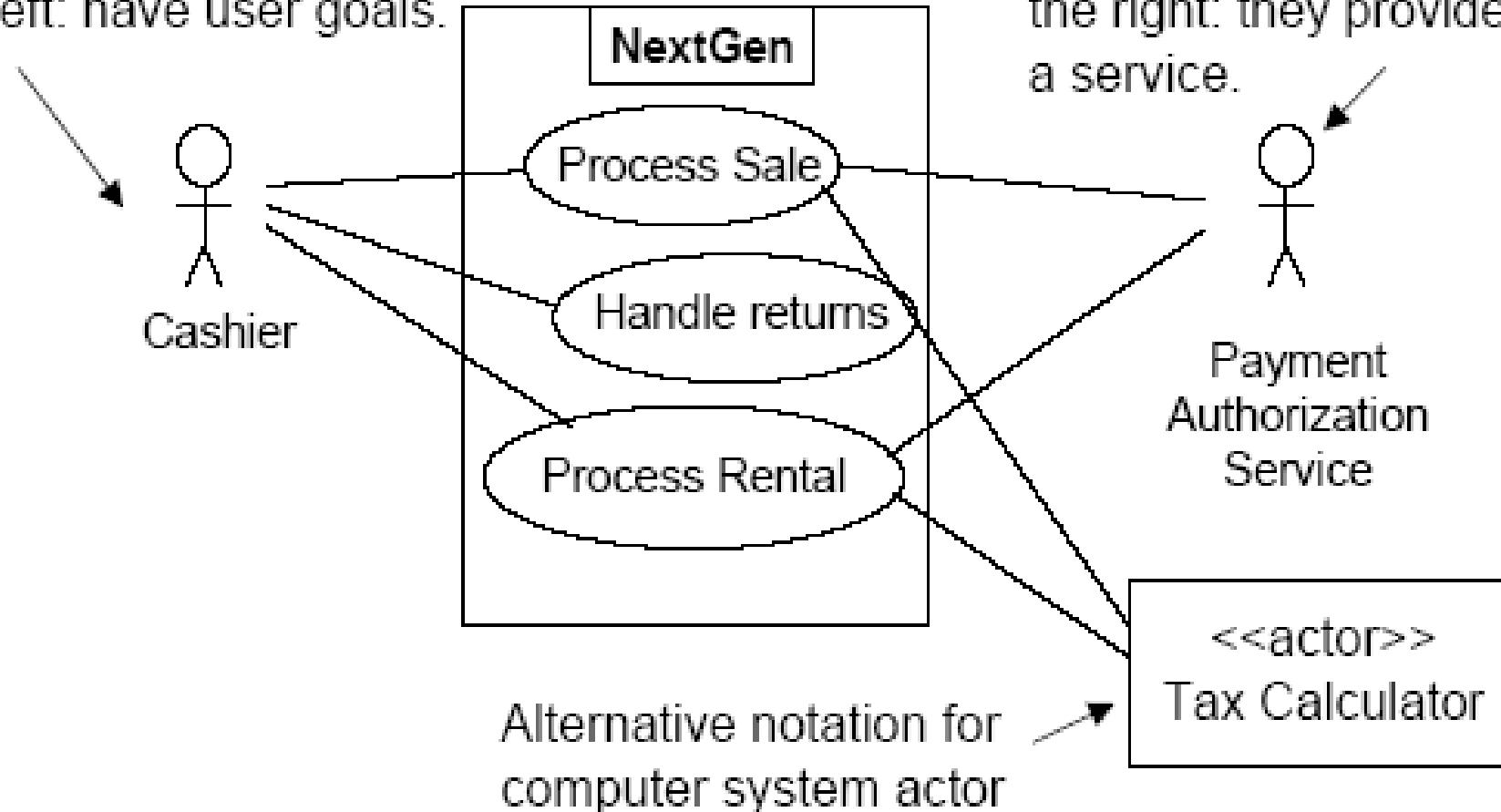
Sanjay Joshi



Drawing Use Case Diagram for PoS

Use Case Diagram for PoS

Primary actors to the left: have user goals.



Supporting actors to the right: they provide a service.



Object Oriented Analysis & Design

Module-2 (RL 2.3.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Use Cases Types

Use case types and formats

- Black-box use cases describe system responsibilities, i.e. define what the system must do.
- Use cases may be written in three formality types
 - Brief: one-paragraph summary, usually of the main success scenario.
 - Casual: Informal paragraph format (e.g. Handle returns)
 - Fully dressed: elaborate. All steps and variations are written in detail.

Use case types and formats

- **Handle returns**

Main success scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item...

Alternate scenarios:

If the credit authorization is reject, inform customer and ask for an alternative payment method.

If item identifier not found in the system, notify the Cashier and suggest manual entry of the identifier code.



Object Oriented Analysis & Design

Module-2 (RL 2.3.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Fully Dressed Use Case for PoS

Fully-dressed example:

Process Sale

Use case UC1: Process Sale

Primary Actor: Cashier

Stakeholders and Interests:

- Cashier: Wants accurate and fast entry, no payment errors, ...
- Salesperson: Wants sales commissions updated.

...

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions):

- Sale is saved. Tax correctly calculated.

...

Main success scenario (or basic flow): [see next slide]

Extensions (or alternative flows): [see next slide]

Special requirements: Touch screen UI, ...

Open issues: What are the tax law variations? ...

Fully dressed example:

Process Sale (cont.)

Main success scenario (or basic flow):

1. The Customer arrives at a POS checkout with items to purchase.
2. The cashier records the identifier for each item. If there is more than one of the same item, the Cashier can enter the quantity as well.
3. The system determines the item price and adds the item information to the running sales transaction. The description and the price of the current item are presented.
4. On completion of item entry, the Cashier indicates to the POS system that item entry is complete.
5. The System calculates and presents the sale total.
6. The Cashier tells the customer the total.
7. The Customer gives a cash payment (“cash tendered”) possibly greater than the sale total.

Extensions (or alternative flows):

- 2a. If sticker is tampered. Enter item id manually
If invalid identifier entered. Indicate error.
If customer didn't have enough cash, cancel sales transaction.
*If Power failure. Restart the transaction.



Object Oriented Analysis & Design

Module-2 (RL 2.3.3)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Styles of Use Cases

Essential vs. Concrete style

- Essential: Focus is on intend.
 - Avoid making UI decisions
- Concrete: UI decisions are embedded in the use case text.
 - e.g. “Admin enters ID and password in the dialog box (see picture X)”
 - Concrete style not suitable during early requirements analysis work.



Object Oriented Analysis & Design

Module-3 (RL 3.1.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



What is Domain Model?

Domain Model

- A Domain Model illustrates meaningful concepts in a problem domain.
- It is a representation of real-world things, not software components.
- It is a set of static structure diagrams; no operations are defined.
- It may show:
 - concepts
 - associations between concepts
 - attributes of concepts



Object Oriented Analysis & Design

Module-3 (RL 3.1.2)



BITS Pilani

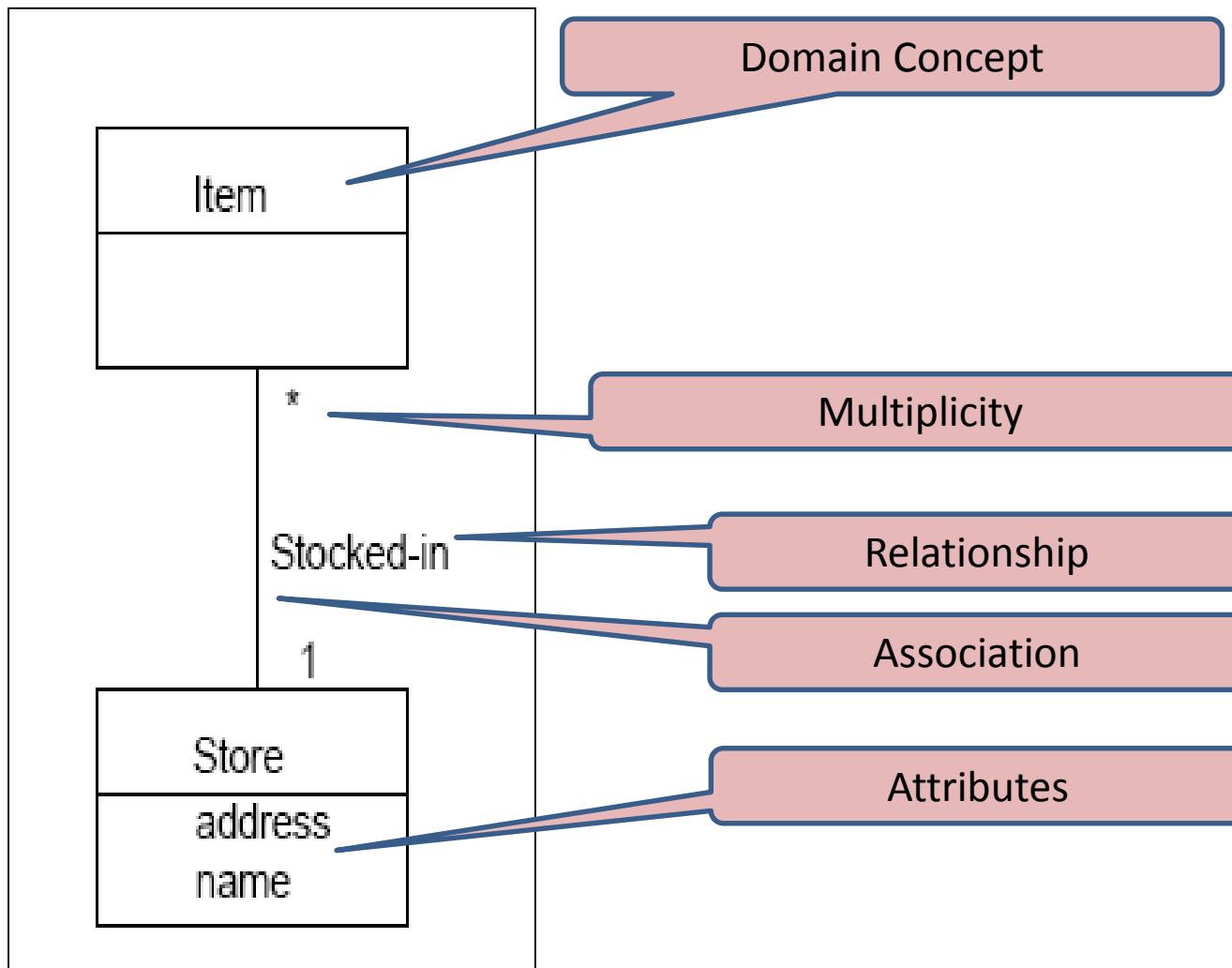
Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



How Domain Model is represented in UML?

Domain Model in UML





Object Oriented Analysis & Design

Module-3 (RL 3.1.3)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

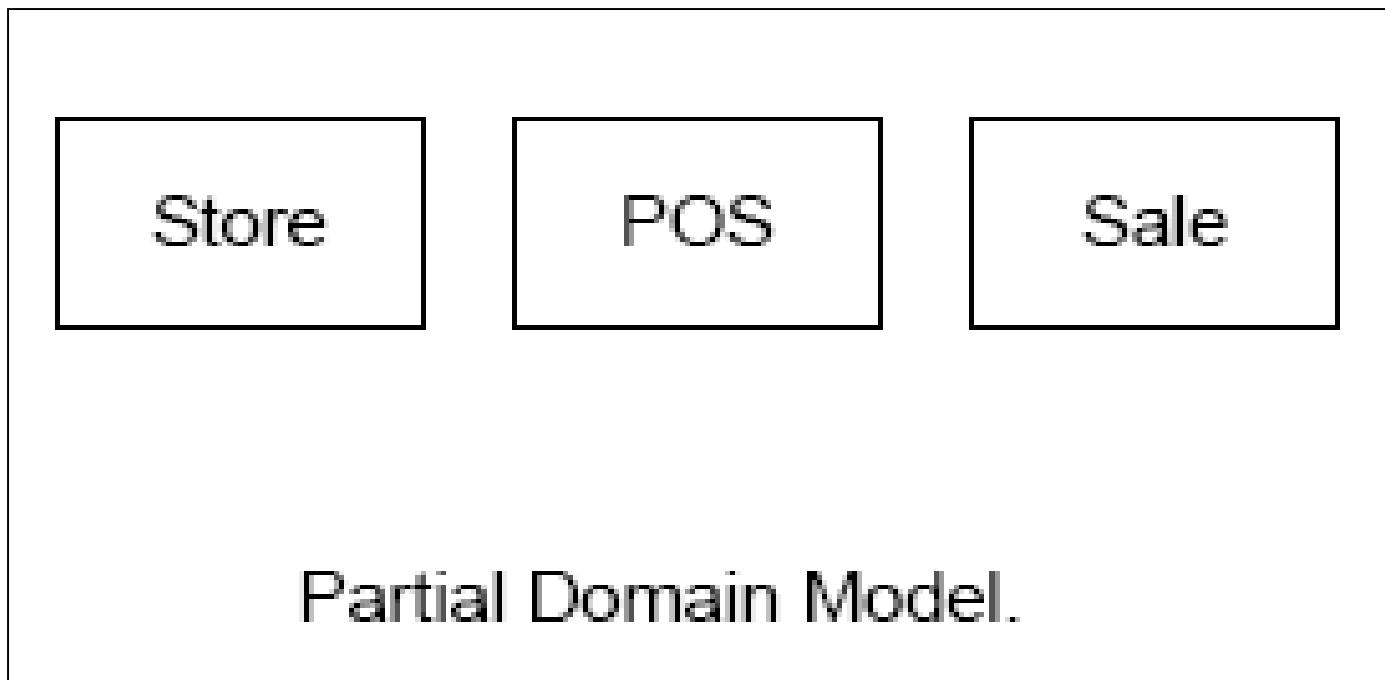
Sanjay Joshi



Identification of Domain Concepts from Use Case

Domains Concepts in PoS

- A central distinction between Object oriented and Procedure Oriented: division by concepts (objects) rather than division by functions.



Strategy to Identify Conceptual Classes



- Use noun phrase identification.
 - Identify noun (and noun phrases) in textual descriptions of the problem domain, and consider them as concepts or attributes.
 - Use Cases are excellent description to draw for this analysis.

Finding Conceptual Classes with Noun Phrase Identification

- 1. This use case begins when a **Customer** arrives at a **POS checkout** with items to purchase.
- 2. The **Cashier** starts a new sale.
- 3. **Cashier** enters item identifier.
...
- The fully addressed Use Cases are an excellent description to draw for this analysis.
- Some of these noun phrases are candidate concepts; some may be attributes of concepts.
- A mechanical noun-to-concept mapping is not possible, as words in a natural language are (sometimes) ambiguous.

Fully-dressed Use Case: Process Sale

Use case UC1: Process Sale

Primary Actor: Cashier

Stakeholders and Interests:

- Cashier: Wants accurate and fast entry, no payment errors, ...
- Salesperson: Wants sales commissions updated.

...

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions):

- Sale is saved. Tax correctly calculated.

...

Main success scenario (or basic flow): [see next slide]

Extensions (or alternative flows): [see next slide]

Special requirements: Touch screen UI, ...

Open issues: What are the tax law variations? ...

Fully dressed example:

Process Sale

Main success scenario (or basic flow):

1. The Customer arrives at a POS checkout with items to purchase.
2. The cashier records the identifier for each item. If there is more than one of the same item, the Cashier can enter the quantity as well.
3. The system determines the item price and adds the item information to the running sales transaction. The description and the price of the current item are presented.
4. On completion of item entry, the Cashier indicates to the POS system that item entry is complete.
5. The System calculates and presents the sale total.
6. The Cashier tells the customer the total.
7. The Customer gives a cash payment ("cash tendered") possibly greater than the sale total.

Extensions (or alternative flows):

- 2a. If sticker is tampered. Enter item id manually
If invalid identifier entered. Indicate error.
If customer didn't have enough cash, cancel sales transaction.
*If Power failure. Restart the transaction.

The NextGen POS (partial) Domain Model





Object Oriented Analysis & Design

Module-3 (RL 3.1.4)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi

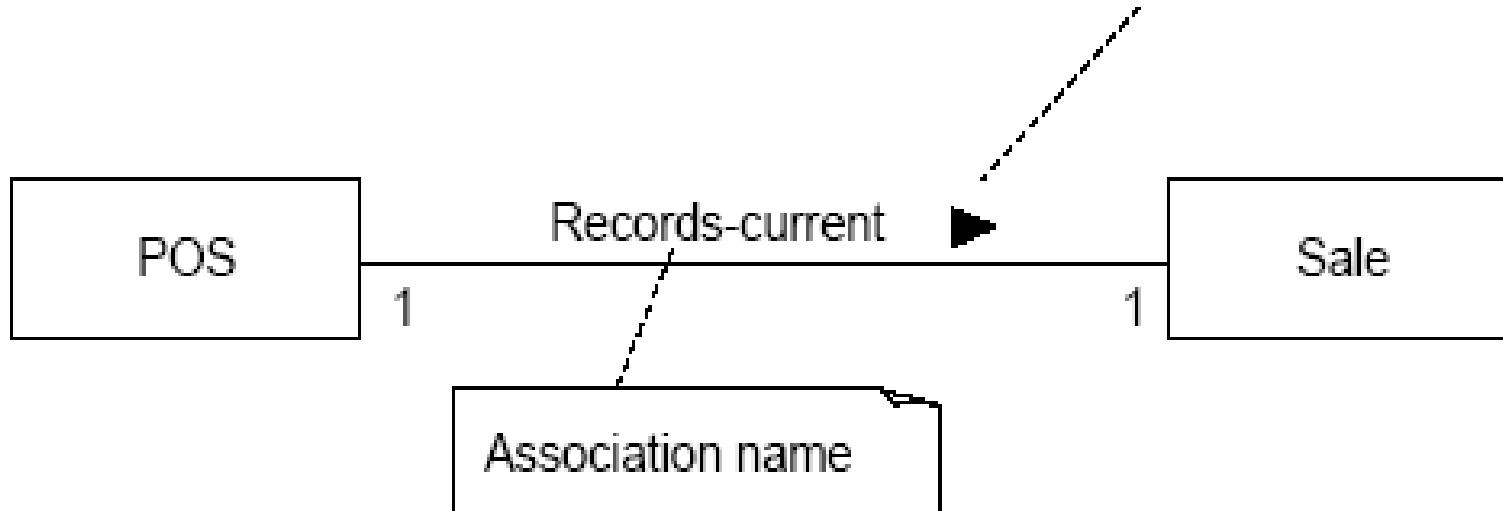


Identification of relationship among domain concepts

Adding Associations

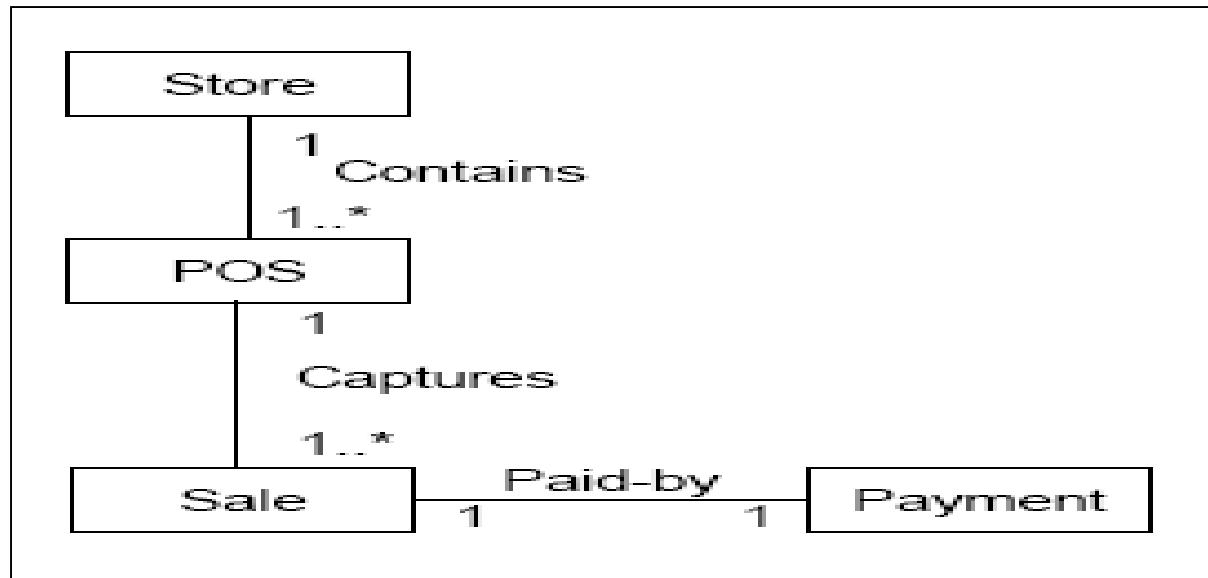
An association is a relationship between concepts that indicates some meaningful and interesting connection.

“Direction reading arrow” has no meaning other than to indicate direction of reading the association label.
Optional (often excluded)



Naming Associations

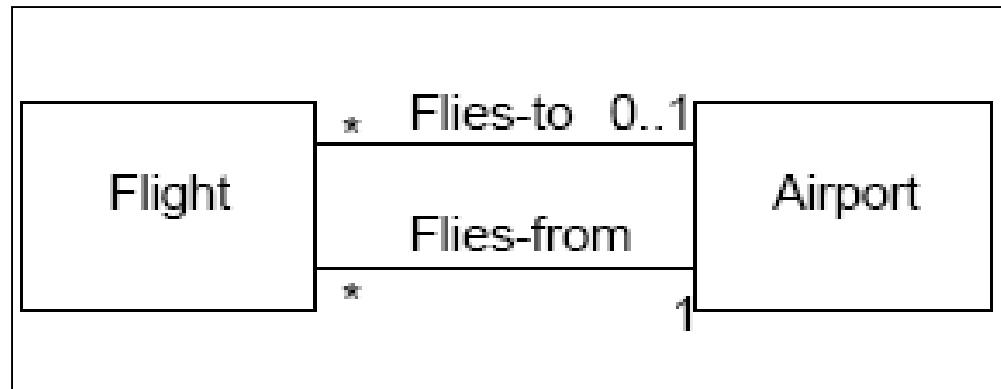
- Name an association based on a TypeName-VerbPhrase-TypeName format.
- Association names should start with a capital letter.
- A verb phrase should be constructed with hyphens.
- The default direction to read an association name is left to right, or top to bottom.



Multiple Associations Between Two Types



- It is not uncommon to have multiple associations between two types.
- In the example, not every flight is guaranteed to land at an airport.





Object Oriented Analysis & Design

Module-3 (RL 3.1.5)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

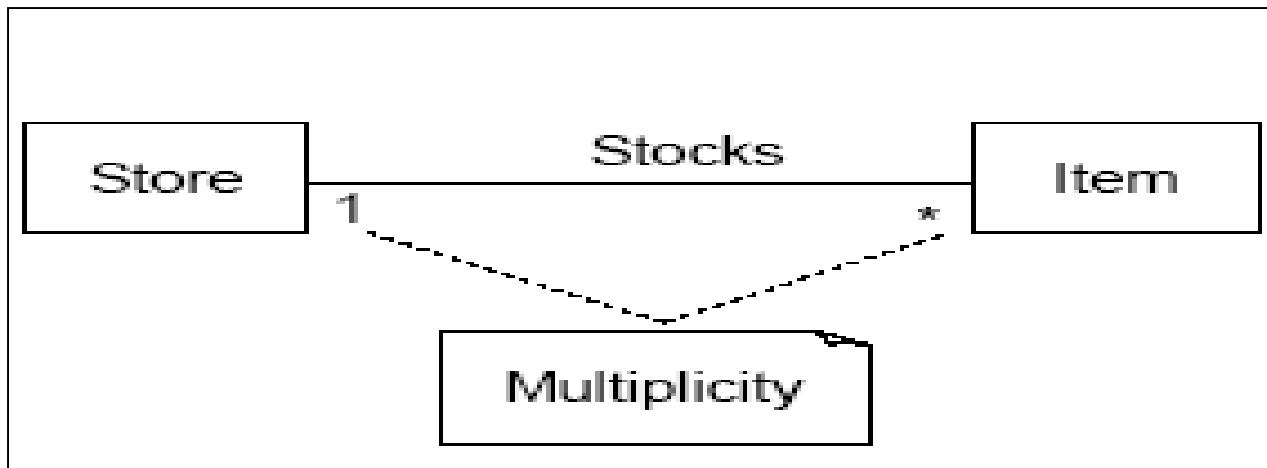
Sanjay Joshi



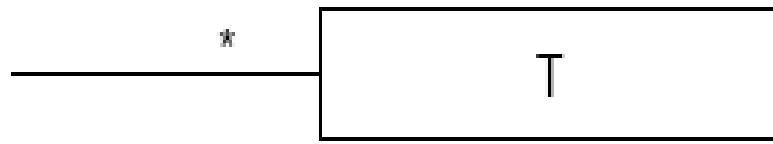
Finding multiplicity among Domain Concepts

Multiplicity

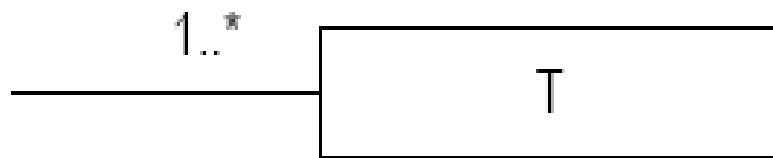
- Multiplicity defines how many instances of a type A can be associated with one instance of a type B, at a particular moment in time.
- For example, a single instance of a Store can be associated with “many” (zero or more) Item instances.



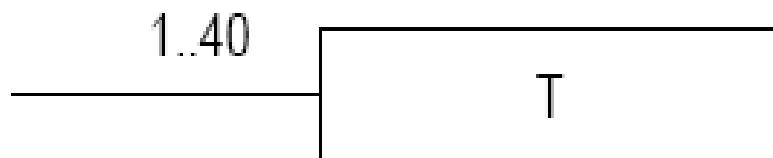
Multiplicity



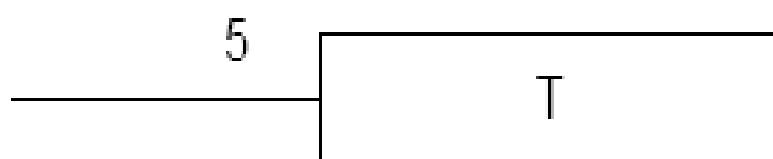
Zero or more;
“many”



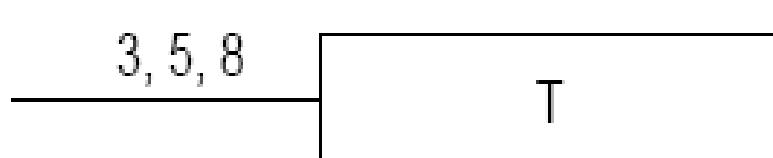
One or more



One to forty



Exactly five



Exactly three, five
or eight.



Object Oriented Analysis & Design

Module-3 (RL 3.1.6)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

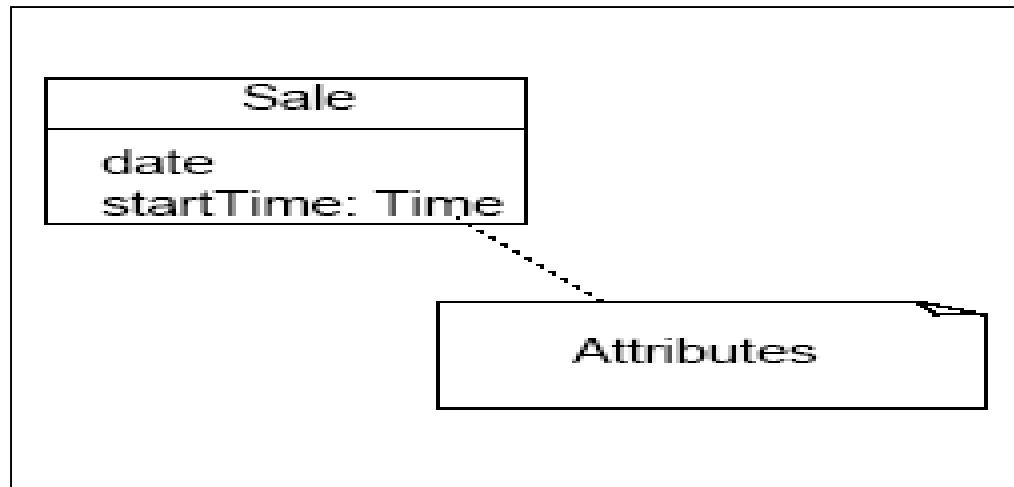
Sanjay Joshi



Adding attributes to Domain Model

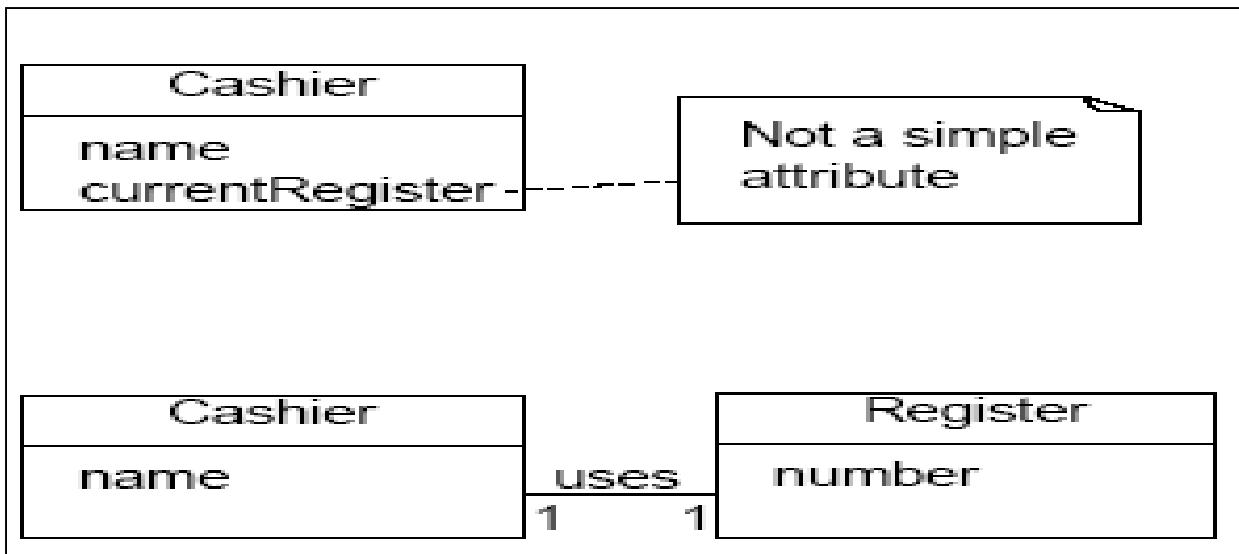
Adding Attributes

- An attribute is a logical data value of an object.
- Include the following attributes: those for which the requirements suggest or imply a need to remember information.
- For example, a Sales receipt normally includes a date and time.
- The Sale concept would need a date and time attribute.



Valid Attribute Types

- Keep attributes simple.
- The type of an attribute should not normally be a complex domain concept, such as Sale or Airport.
- Attributes in a Domain Model should preferably be
 - Pure data values: Boolean, Date, Number, String, ...
 - Simple attributes: color, phone number, zip code, universal product code (UPC), ...





Object Oriented Analysis & Design

Module-3 (RL 3.1.7)



BITS Pilani

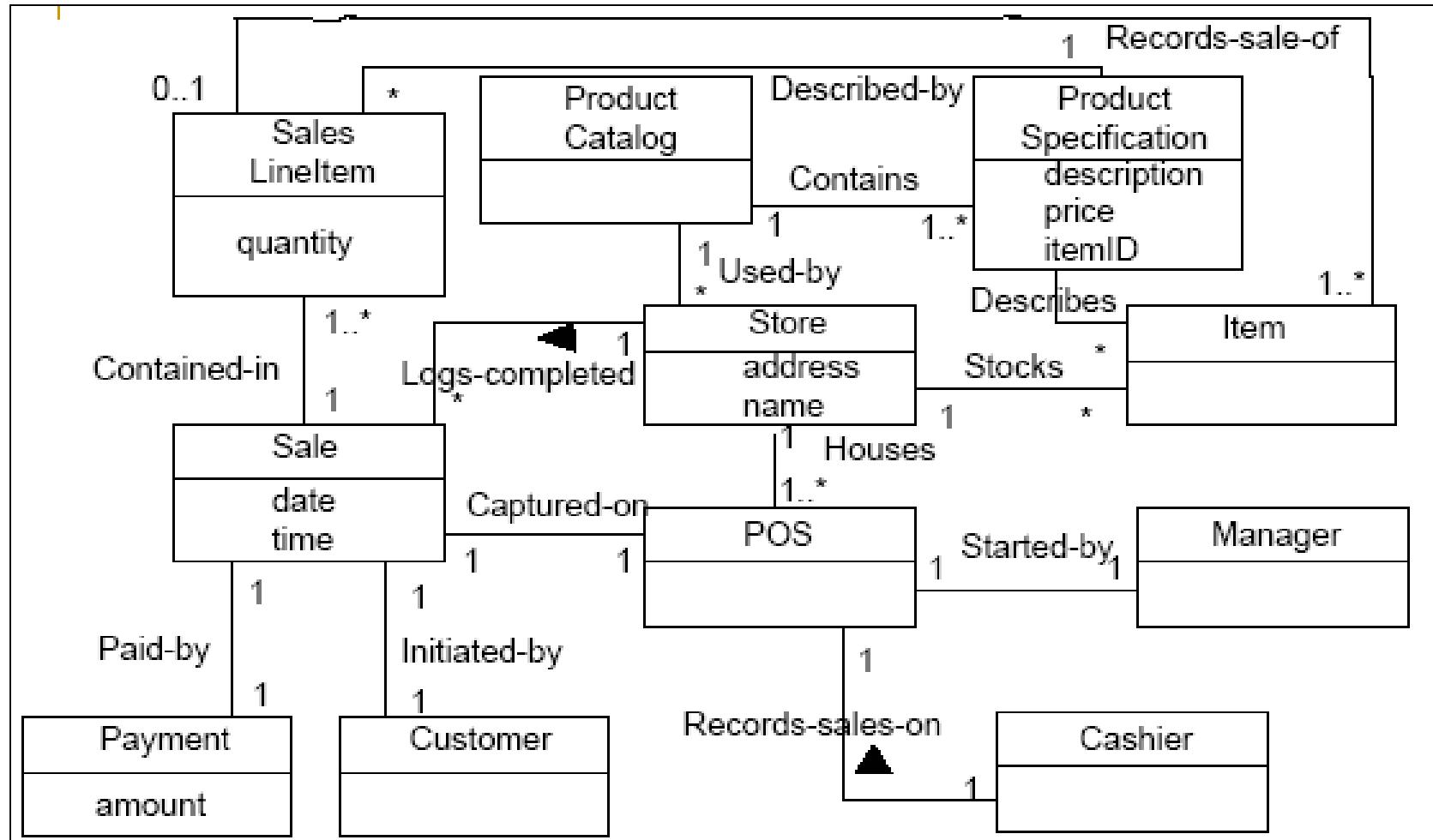
Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Significance of Domain Model

Domain Model : Partial?



Significance of Domain Model

- Domain Model is base for Designer to draw Class Diagram
- Not necessary that all Domain Concepts will be carried forward.
- More implementation classes can be added by Designed in Class Diagram



Object Oriented Analysis & Design

Module-3 (RL 3.2.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



What is SSD (System Sequence Diagram)

What is SSD (System Sequence Diagram) ?

- It is useful to investigate and define the behavior of the software as a “black box”.
- System behavior is a description of *what the system does* (without an explanation of how it does it).
- Use cases describe how external actors interact with the software system. During this interaction, an actor generates events.
- A request event initiates an operation upon the system.



Object Oriented Analysis & Design

Module-3 (RL 3.2.2)



BITS Pilani

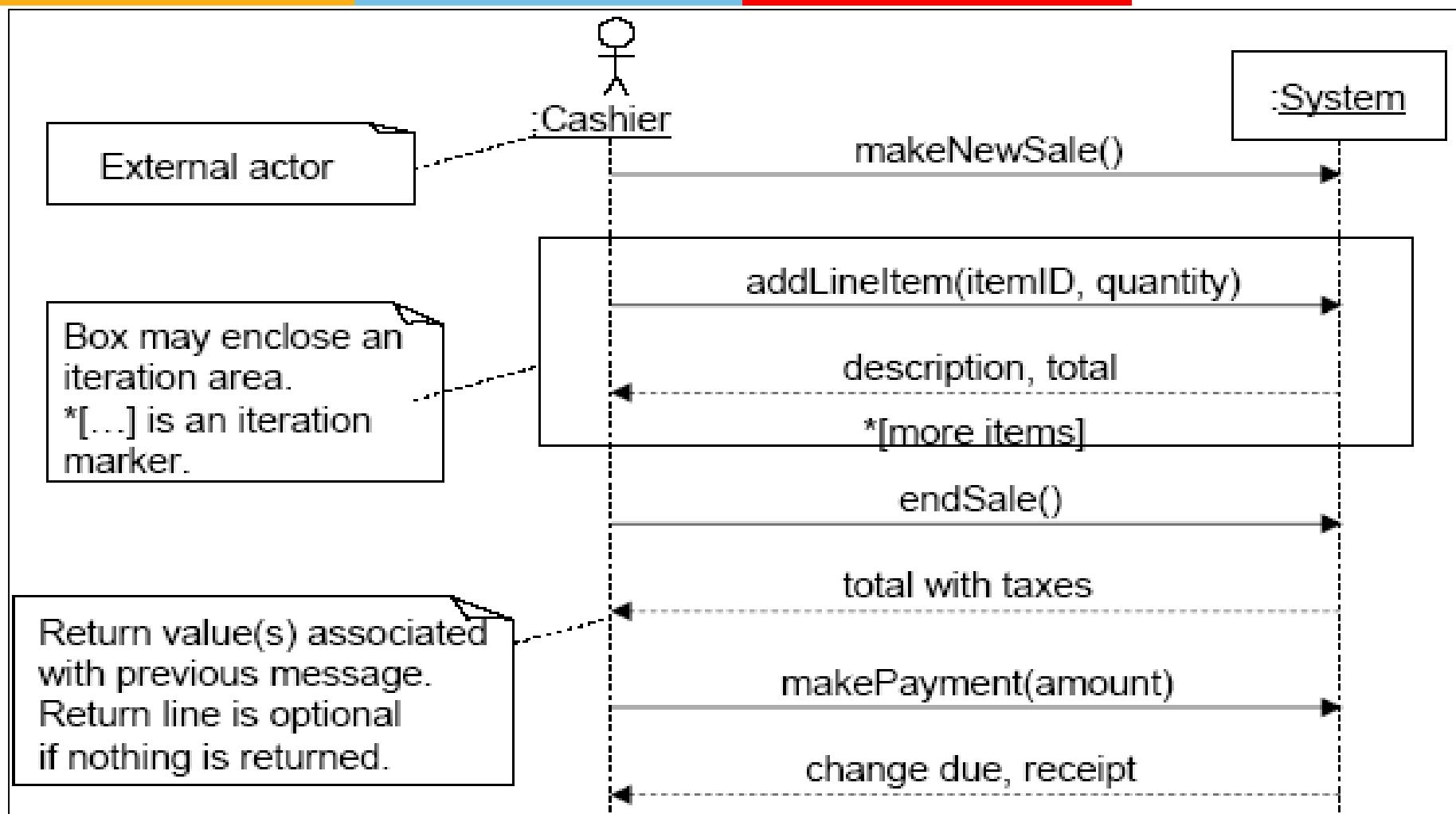
Pilani|Duba|Goa|Hyderabad

Sanjay Joshi

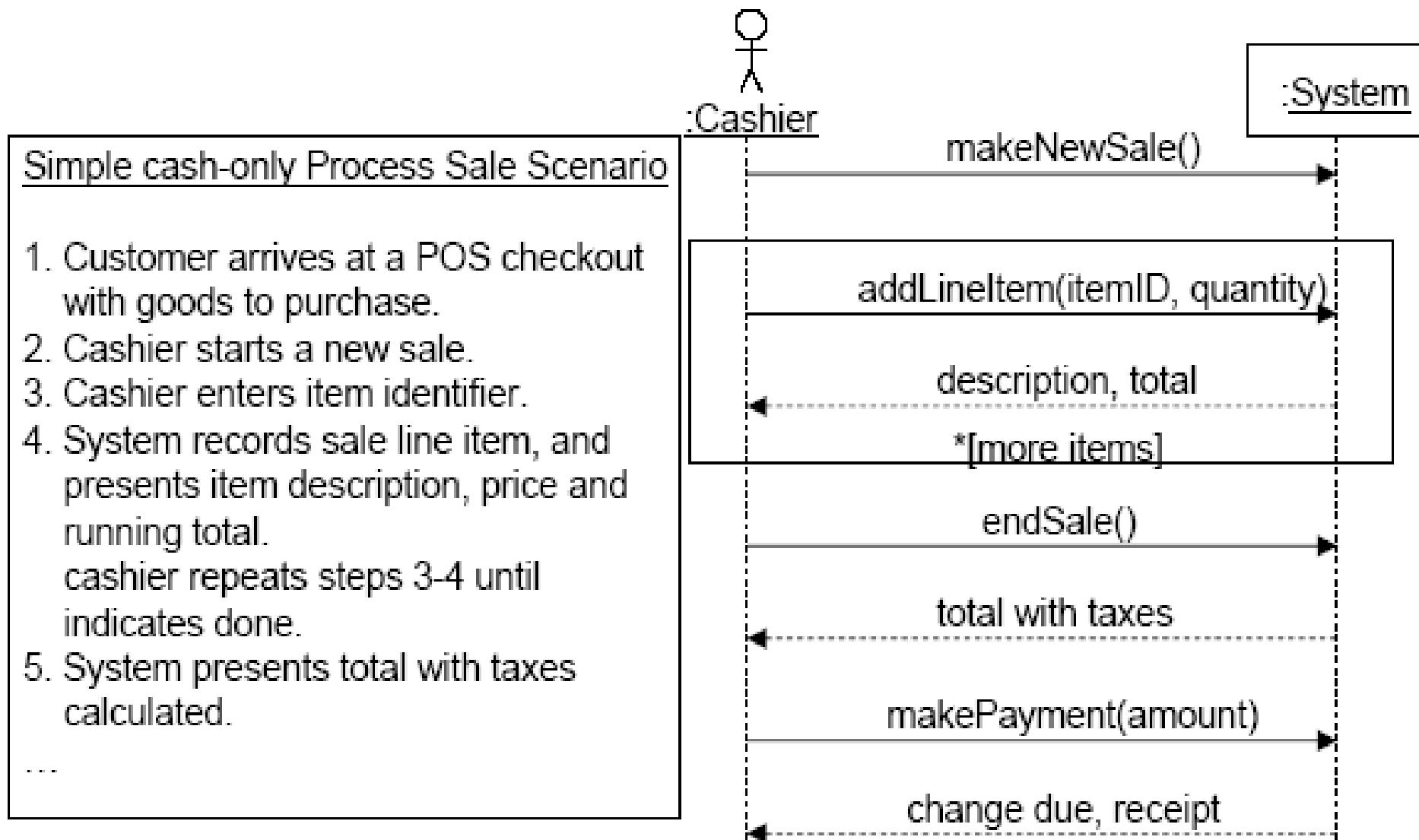


Drawing SSD for PoS

SSDs for Process Sale Scenario



SSD and Use Cases



Naming System Events and Operations

- The set of all required system operations is determined by identifying the system events.
 - makeNewSale()
 - addLineItem(itemID, quantity)
 - endSale()
 - makePayment(amount)



Object Oriented Analysis & Design

Module-3 (RL 3.2.3)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Significance of SSD

Significance of SSD

- Interaction of the system with the outside world
- It is used to depict how System responses to external events
- It plays key role in GUI design of the system



Object Oriented Analysis & Design

Module-3 (RL 3.3.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



What is Operation Contracts

Why Operation Contracts?

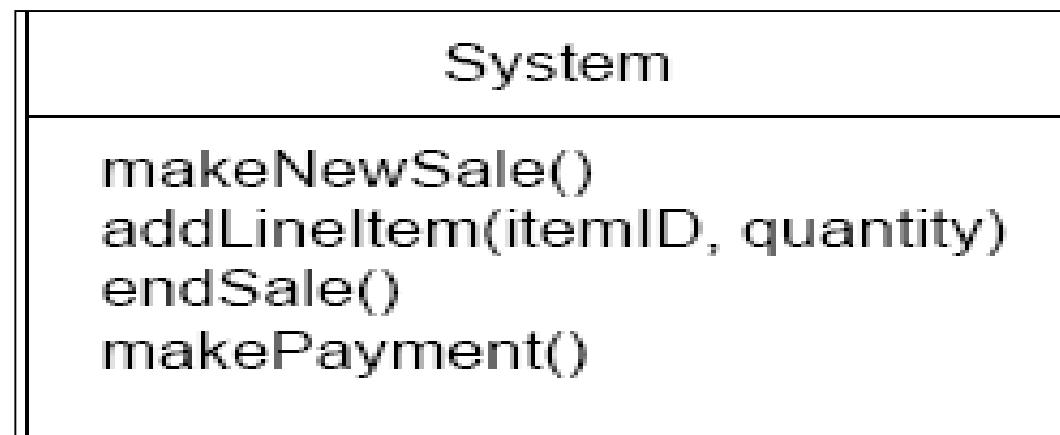
- Details missing from System Sequence Diagram
- What an Analyst Reflect in Operation Contract?

What is Operation Contract?

- Operation Contracts are documents that describe system behavior.
- Operation Contracts may be defined for **system operations**.
 - Operations that the system (as a black box) offers in its public interface to handle incoming system events.
- The entire set of system operations across all use cases, defines the public system interface.

System Operations and the System Interface

- In the UML the system as a whole can be represented as a class.
- Contracts are written for each system operation to describe its behavior.





Object Oriented Analysis & Design

Module-3 (RL 3.3.2)



BITS Pilani

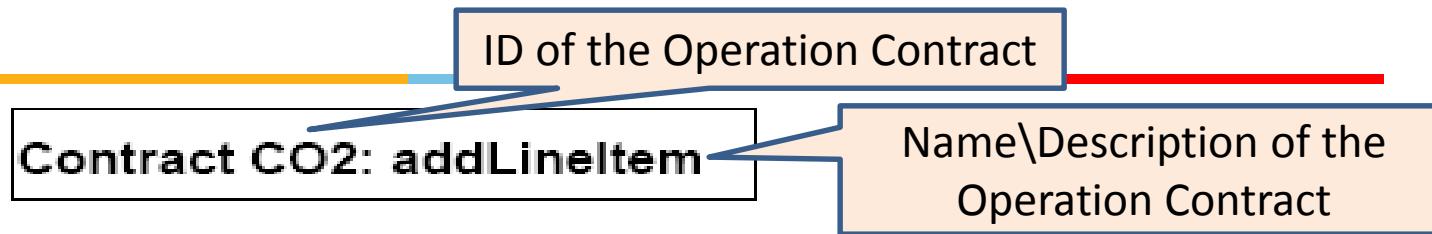
Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Represent Operation Contract in UML

Operation Contract in UML



Operation: addLineItem (itemID: integer, quantity: integer)

Cross References: Use Cases: Process Sale.

Operation Signature

Pre-conditions: There is a sale underway.

Use Case Reference

Post-conditions:

Prerequisite

- A SalesLineItem instance *sli* was created. (instance creation)
- *sli* was associated with the Sale. (association formed)
- *sli.quantity* was set to quantity. (attribute modification)
- *sli* was associated with a ProductSpecification, based on itemID match (association formed)

Post Condition in Past Tense



Object Oriented Analysis & Design

Module-3 (RL 3.3.3)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Writing Operation Contract for PoS

Operation Contract: addLineItem

Contract CO2: addLineItem

Operation: addLineItem (itemID: integer, quantity: integer)

Cross References: Use Cases: Process Sale.

Pre-conditions: There is a sale underway.

Post-conditions:

- A SalesLineItem instance *sli* was created. (instance creation)
- *sli* was associated with the Sale. (association formed)
- *sli.quantity* was set to quantity. (attribute modification)
- *sli* was associated with a ProductSpecification, based on itemID match (association formed)

Pre- and Postconditions

- Preconditions are assumptions about the state of the system before execution of the operation.
- A postcondition is an assumption that refers to the state of the system after completion of the operation.
 - The postconditions are not actions to be performed during the operation.
 - Describe changes in the state of the objects in the Domain Model (instances created, associations are being formed or broken, and attributes are changed)

addLineItem postconditions

- Instance Creation and Deletion

After the itemID and quantity of an item have been entered by the cashier, what new objects should have been created?

- A SalesLineItem instance *sli* was created.

addLineItem postconditions

- Attribute Modification

After the itemID and quantity of an item have been entered by the cashier, what attributes of new or existing objects should have been modified?

- sli.quantity was set to quantity (attribute modification).

addLineItem postconditions

- Associations Formed and Broken

After the itemID and quantity of an item have been entered by the cashier, what associations between new or existing objects should have been formed or broken?

- sli was associated with the current Sale (association formed).
- sli was associated with a ProductSpecification, based on itemID match (association formed).



Object Oriented Analysis & Design

Module-4 (RL 4.1.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi

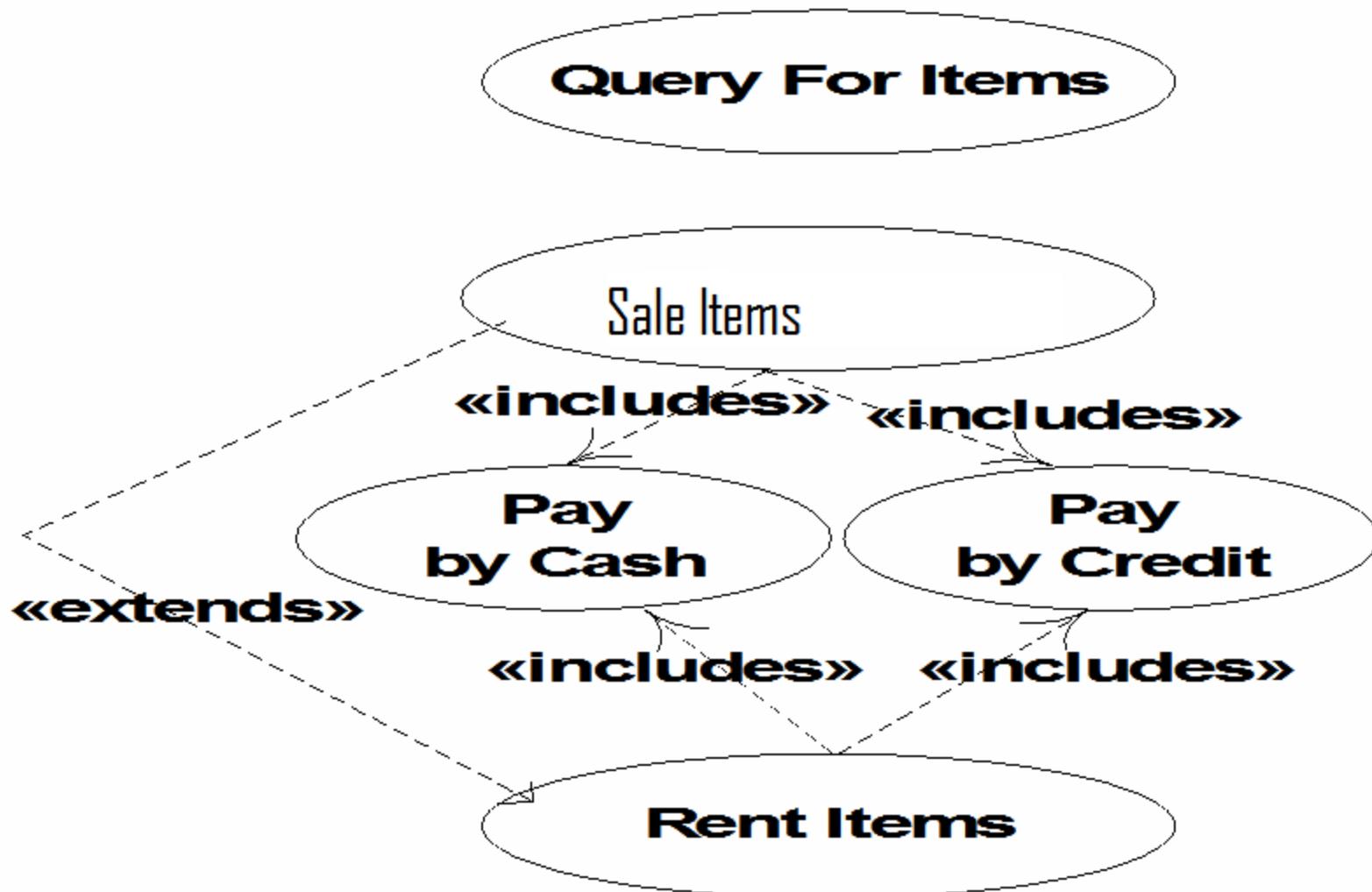


Relating Use Cases : includes, extends relationships

Relating Use Cases

- When creating the use case diagram, it can be useful (in terms of comprehension and simplification) to:
 - factor out shared sub-processes
 - use the <<includes>> relationship
 - show extensions
 - use the <<extends>> relationship

Video Store Information System





Object Oriented Analysis & Design

Module-4 (RL 4.1.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

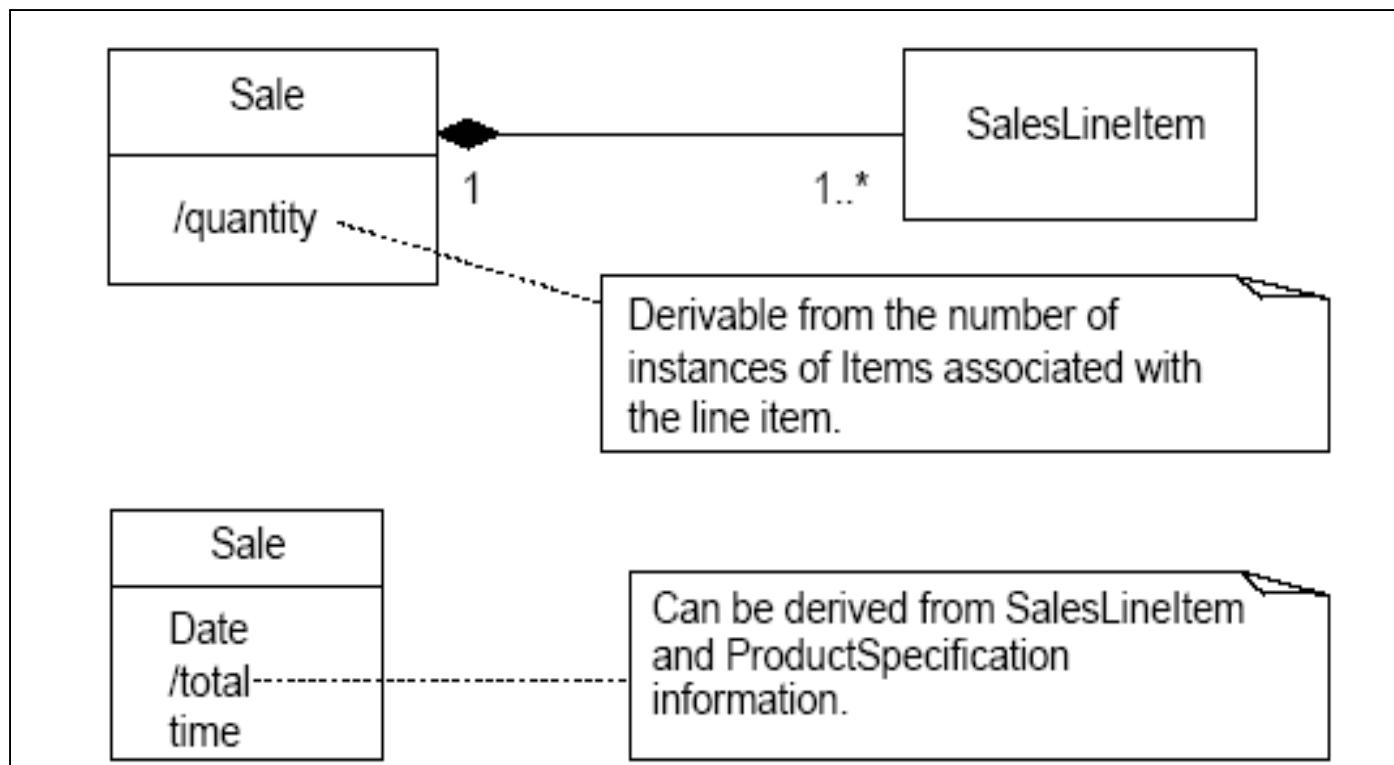
Sanjay Joshi



Refining Domain Model : Derived Attributes

Derived Elements

- A derived element can be determined from others.





Object Oriented Analysis & Design

Module-4 (RL 4.1.3)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



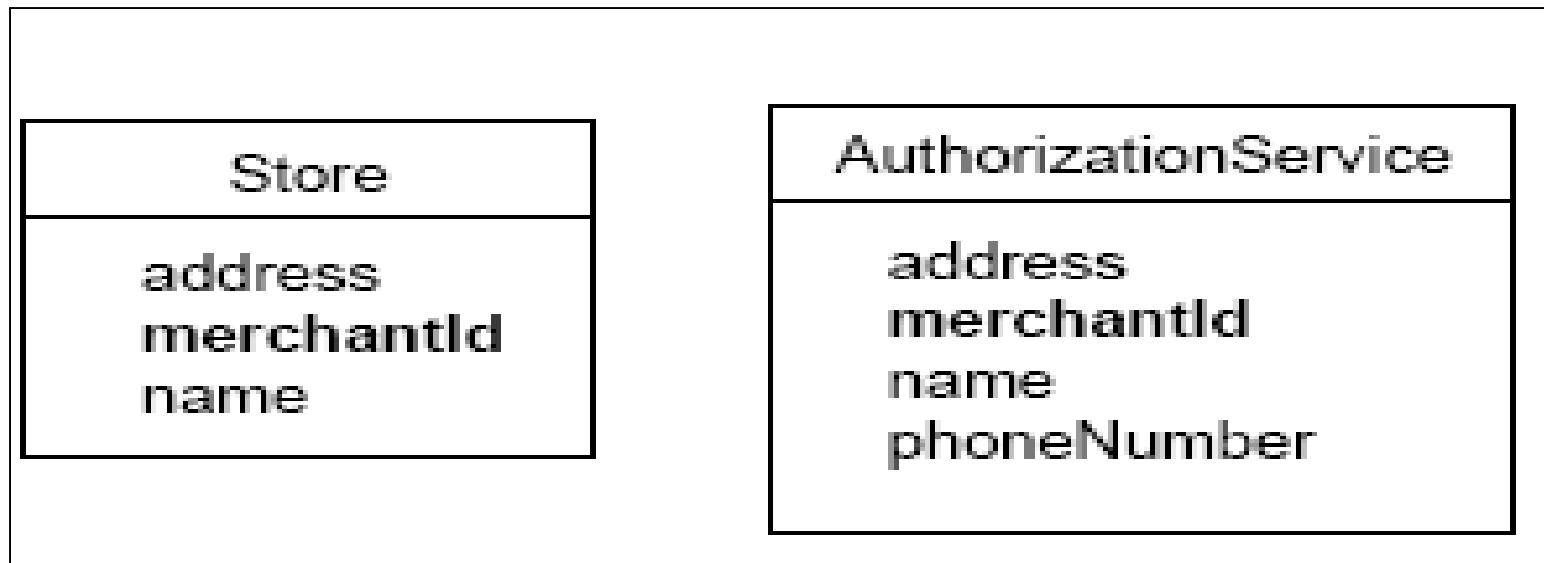
Refining Domain Model : Association Classes

Association Classes

- Authorization services assign a merchant ID to each store for identification during communications.
- A payment authorization request from the store to an authorization service requires the inclusion of the merchant ID that identifies the store to the service.
- Consider a store that has a different merchant ID for each service. (e.g. Id for Visa is XXX, Id for MC is YYY, etc.)

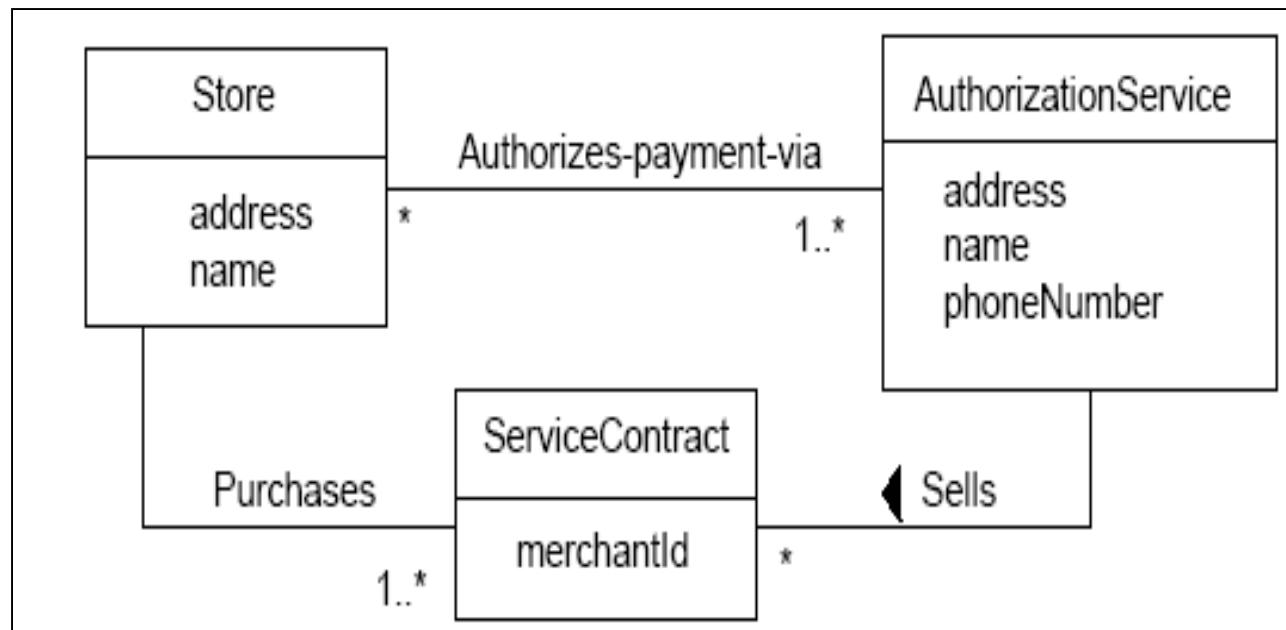
Association Classes

- Where in the conceptual model should the merchant ID attribute reside?
- Placing the merchantId in the Store is incorrect, because a Store may have more than one value for merchantId.
- The same is true with placing it in the AuthorizationService



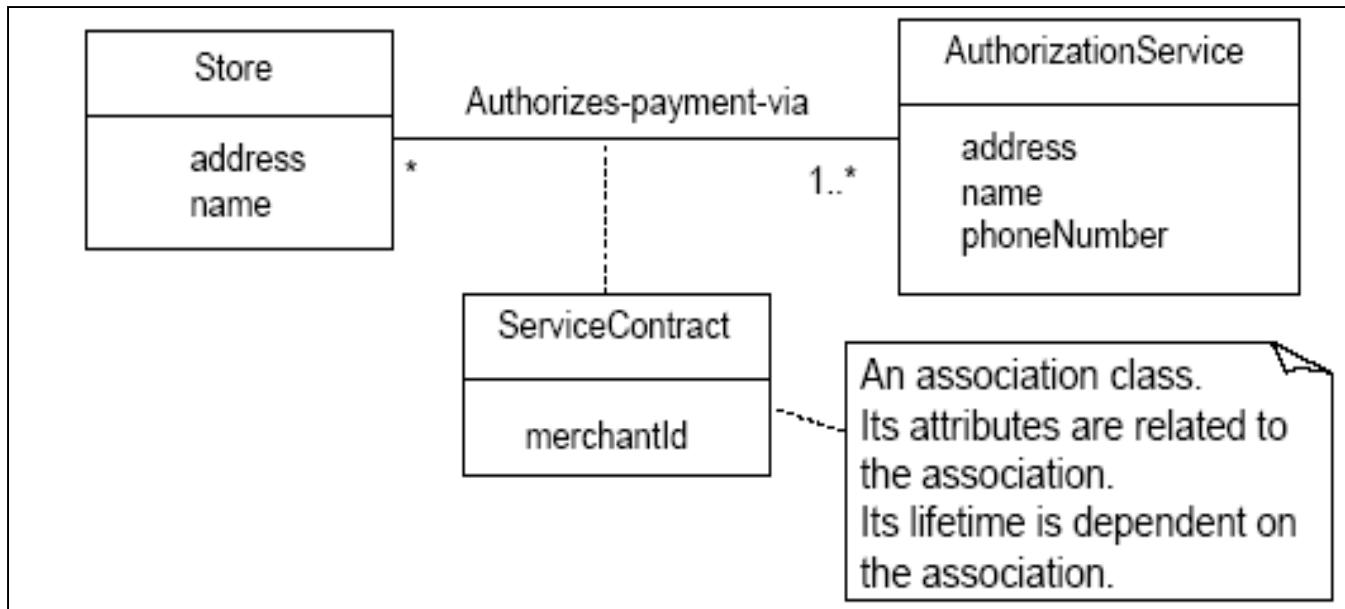
Association Classes

- In a conceptual model, if a class C can simultaneously have many values for the same kind of attribute A, do not place attribute A in C. Place attribute A in another type that is associated with C.



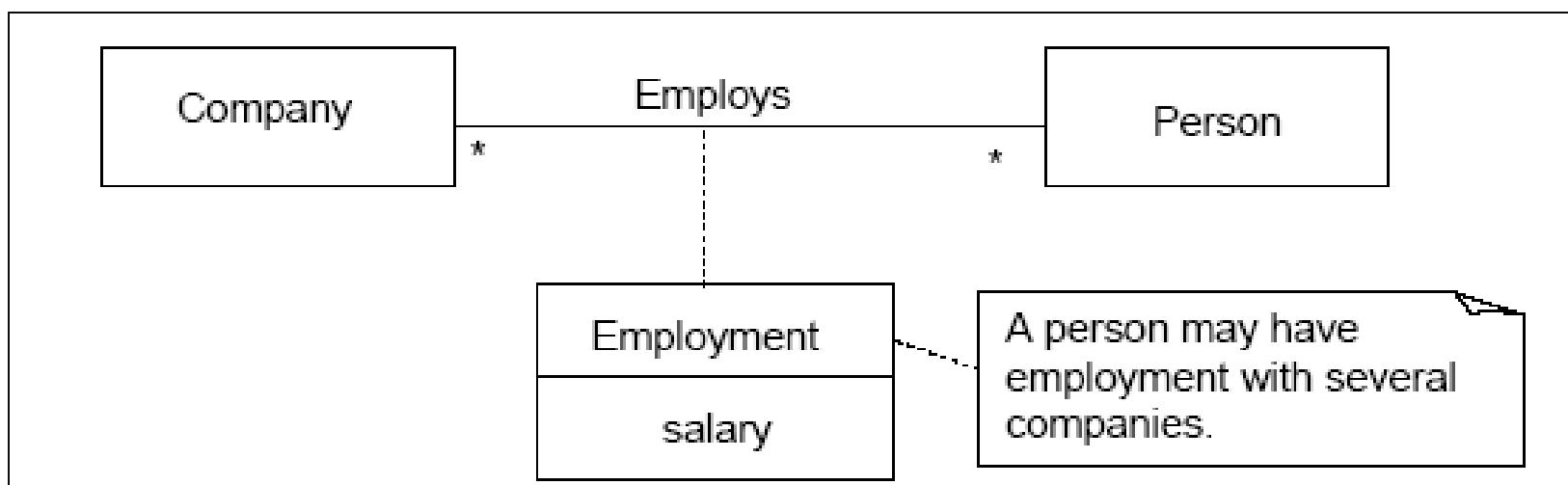
Association Classes

- The merchantId is an attribute related to the association between the Store and AuthorizationService; it depends on their relationship.
- ServiceContract may then be modeled as an association class.



Guidelines for Association Classes

- An attribute is related to an association.
- Instances of the association class have a life-time dependency on the association.
- There is a many-to-many association between two concepts.
- The presence of a many-to-many association between two concepts is a clue that a useful associative type should exist in the background somewhere.





Object Oriented Analysis & Design

Module-4 (RL 4.2.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



What is Interaction Diagram?

What is Interaction Diagram?

- *Shows interaction among Objects within the system*
- *It is scenario specific diagram*
- *Interaction by means of exchange of messages*
- *It is part of Lower Level Design*
- *Is it same as SSD (System Sequence Diagram)?*



Object Oriented Analysis & Design

Module-4 (RL 4.2.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



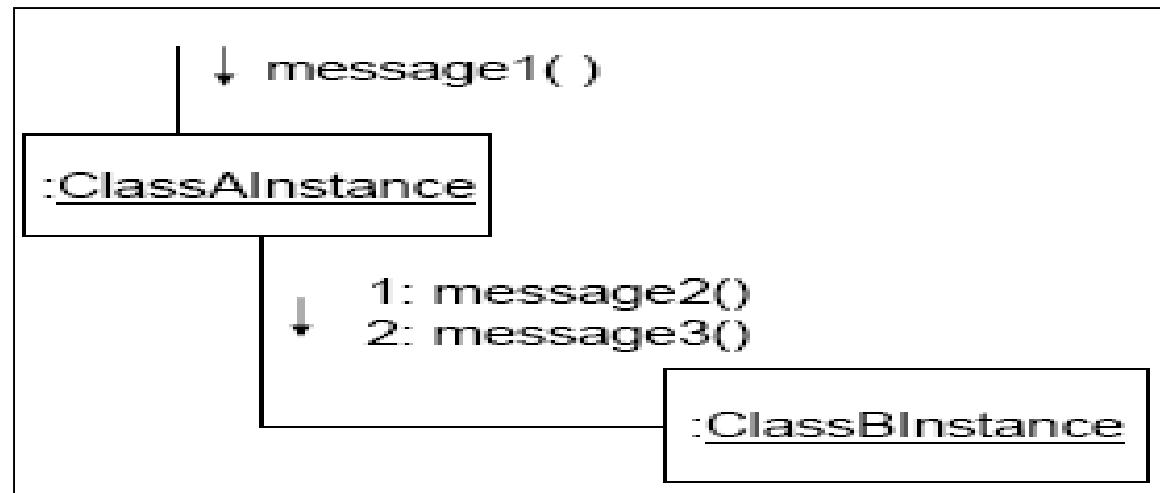
Types of Interactions Diagram

Types of Interaction Diagrams



- *Two Types : Collaboration Diagram & Sequence Diagram*

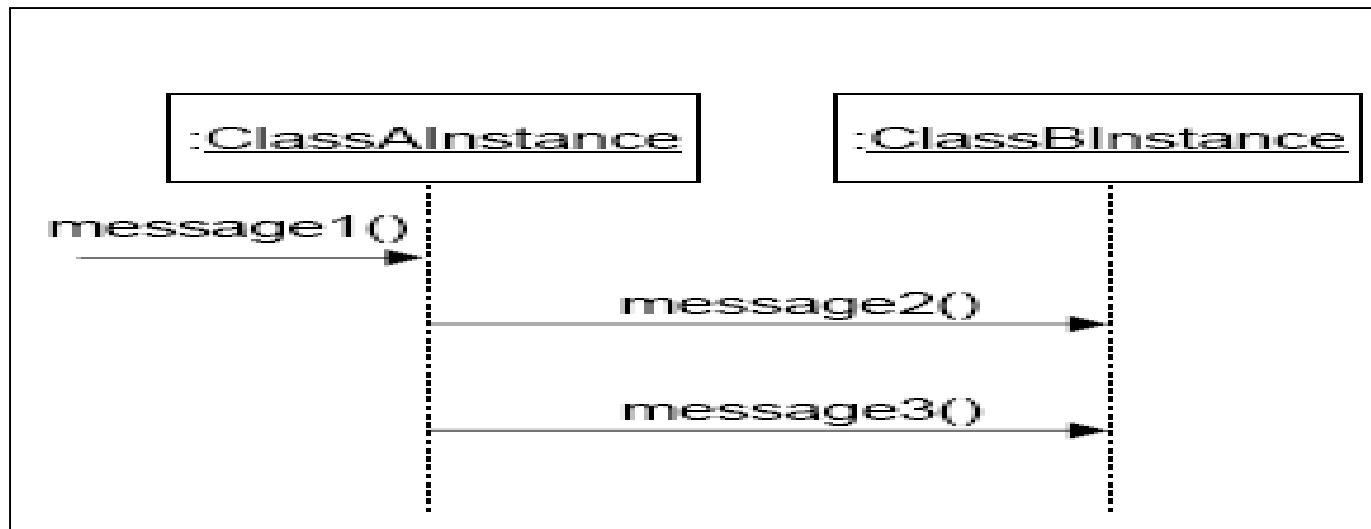
Collaboration diagrams illustrate object interactions in a graph or network format.



Types of Interaction Diagrams



- *Sequence diagrams* illustrate interactions in a kind of fence format.





Object Oriented Analysis & Design

Module-4 (RL 4.2.3)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

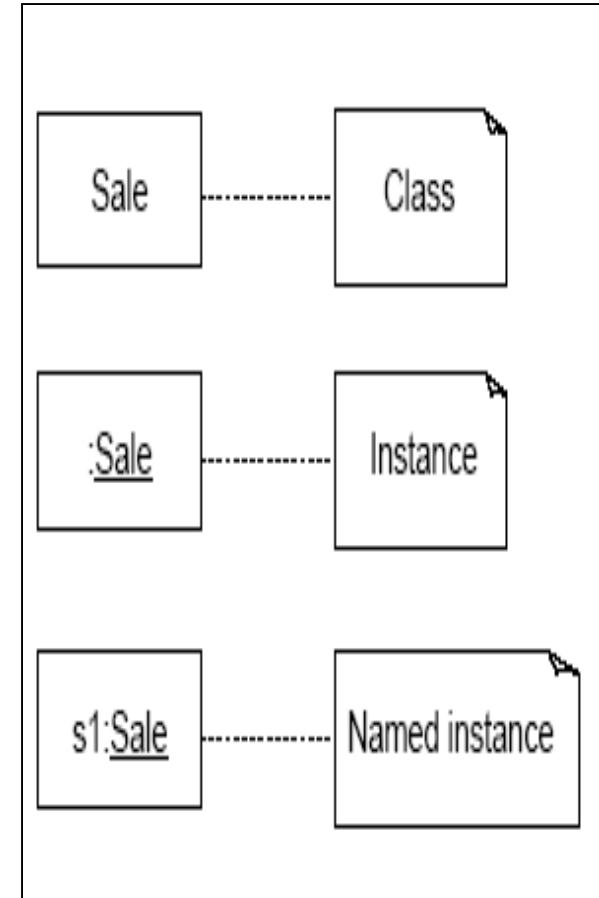
Sanjay Joshi



Representation of Interaction Diagrams in UML

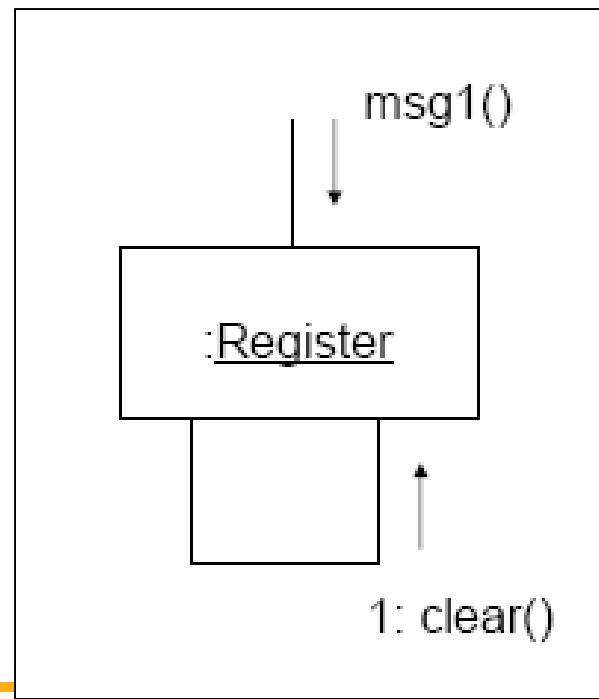
Illustrating Classes and Instances

- To show an instance of a class, the regular class box graphic symbol is used, but the name is underlined.
Additionally a class name should be preceded by a colon.
- An instance name can be used to uniquely identify the instance.



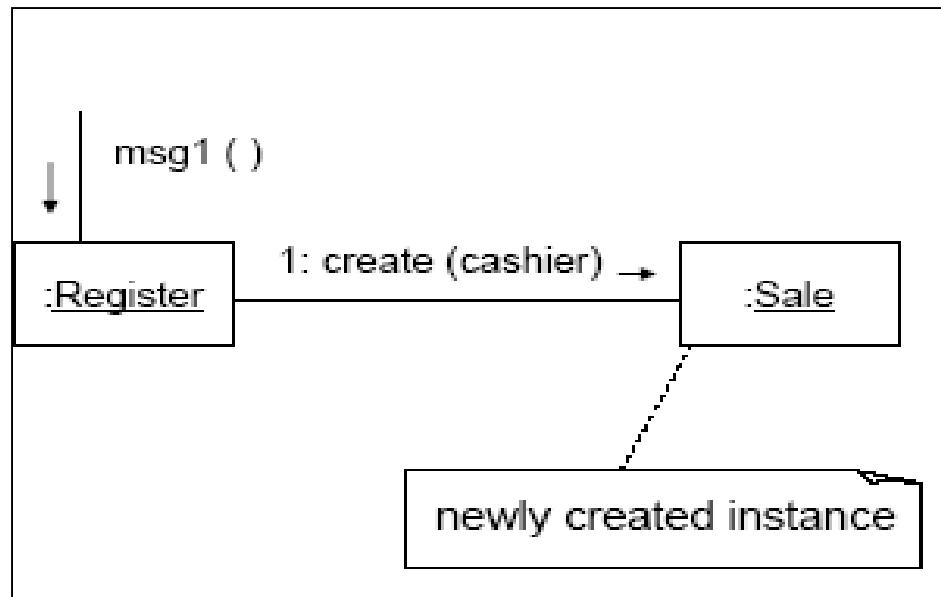
Messages to “self” or “this”

- A message can be sent from an object to itself.
- This is illustrated by a link to itself, with messages flowing along the link.



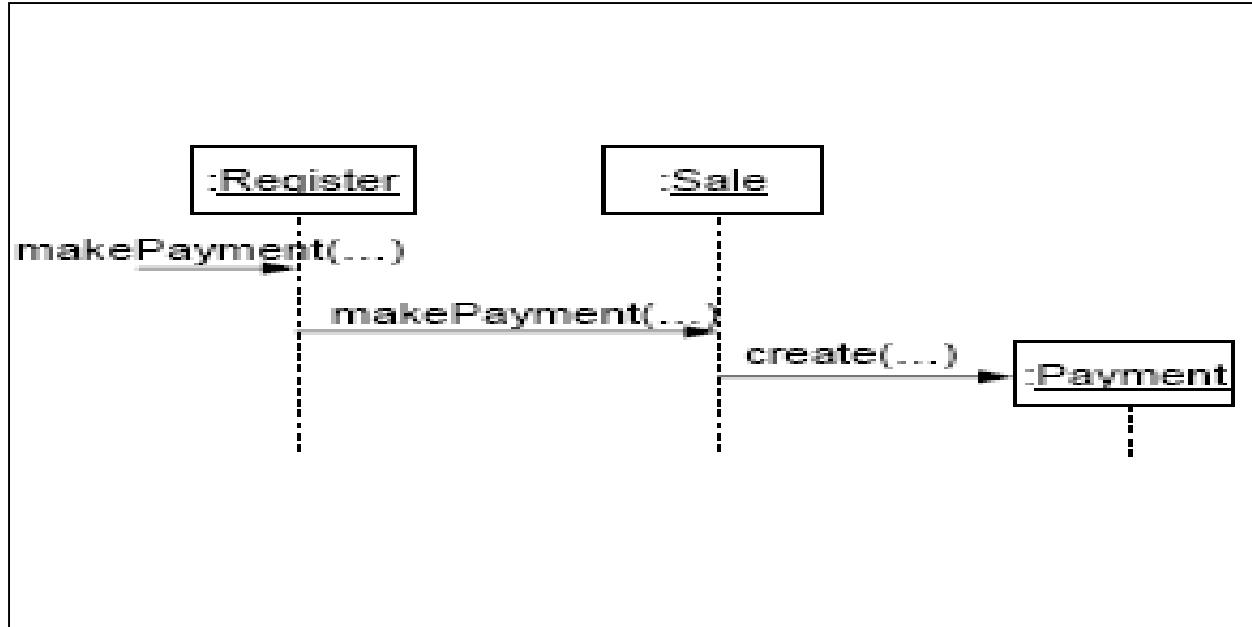
Creation of Instances

- The language independent creation message is create, being sent to the instance being created.
- The create message may include parameters, indicating passing of initial values.



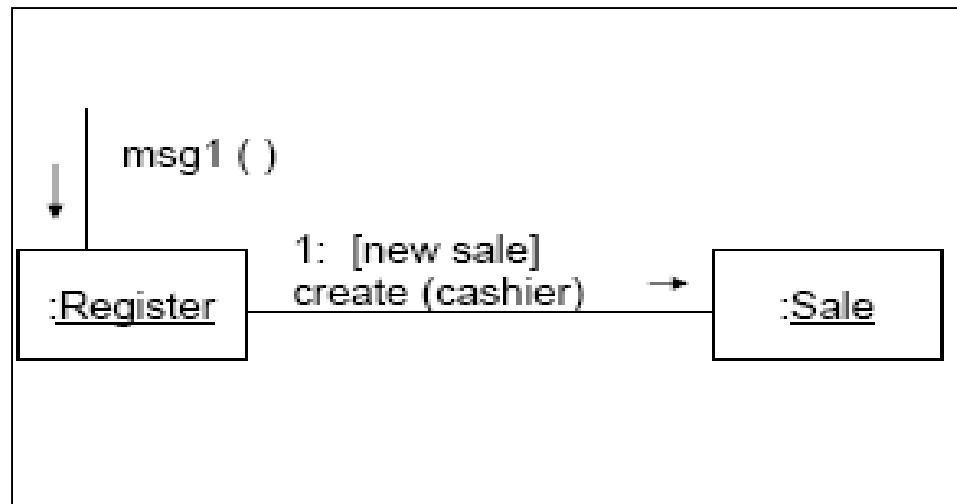
Creation of Instances

- An object lifeline shows the extend of the life of an object in the diagram.
- Note that newly created objects are placed at their creation height.

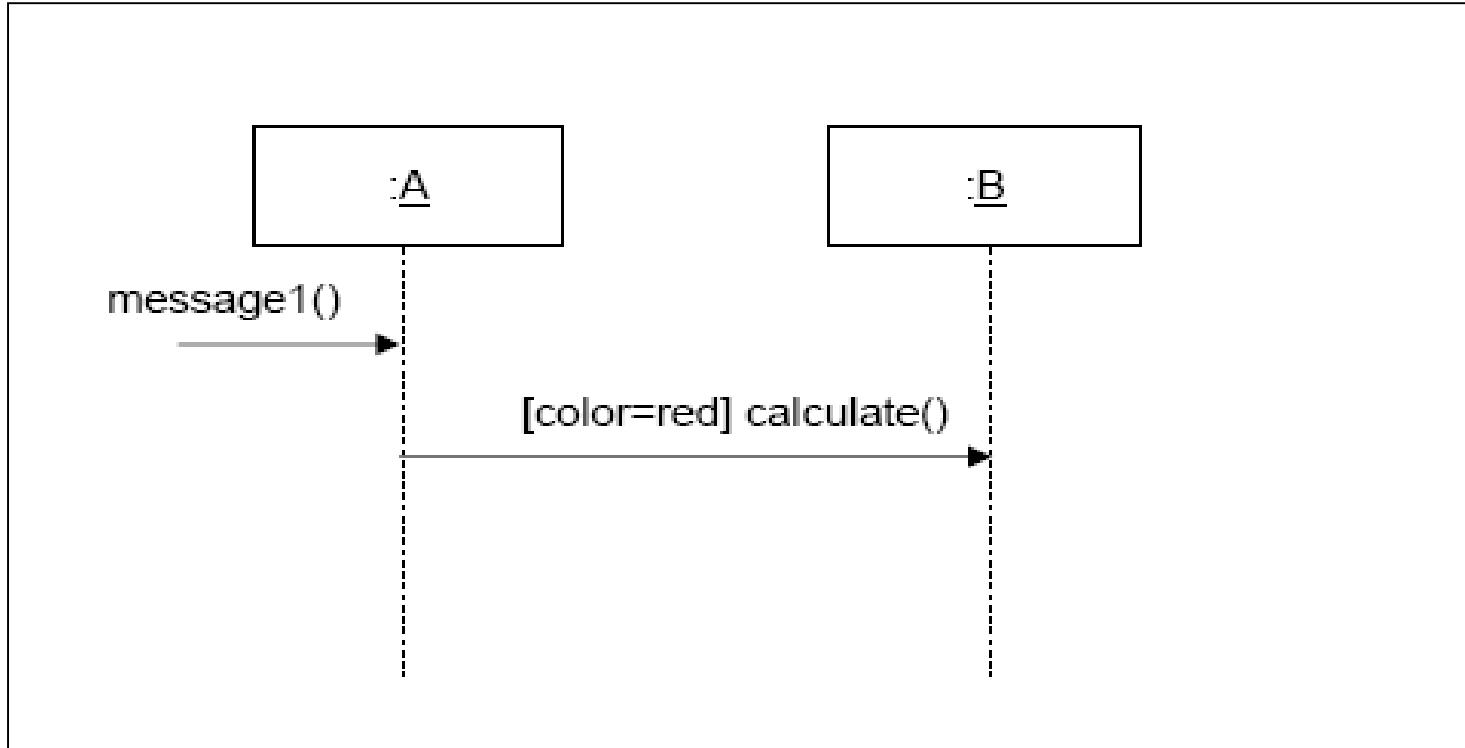


Conditional Messages

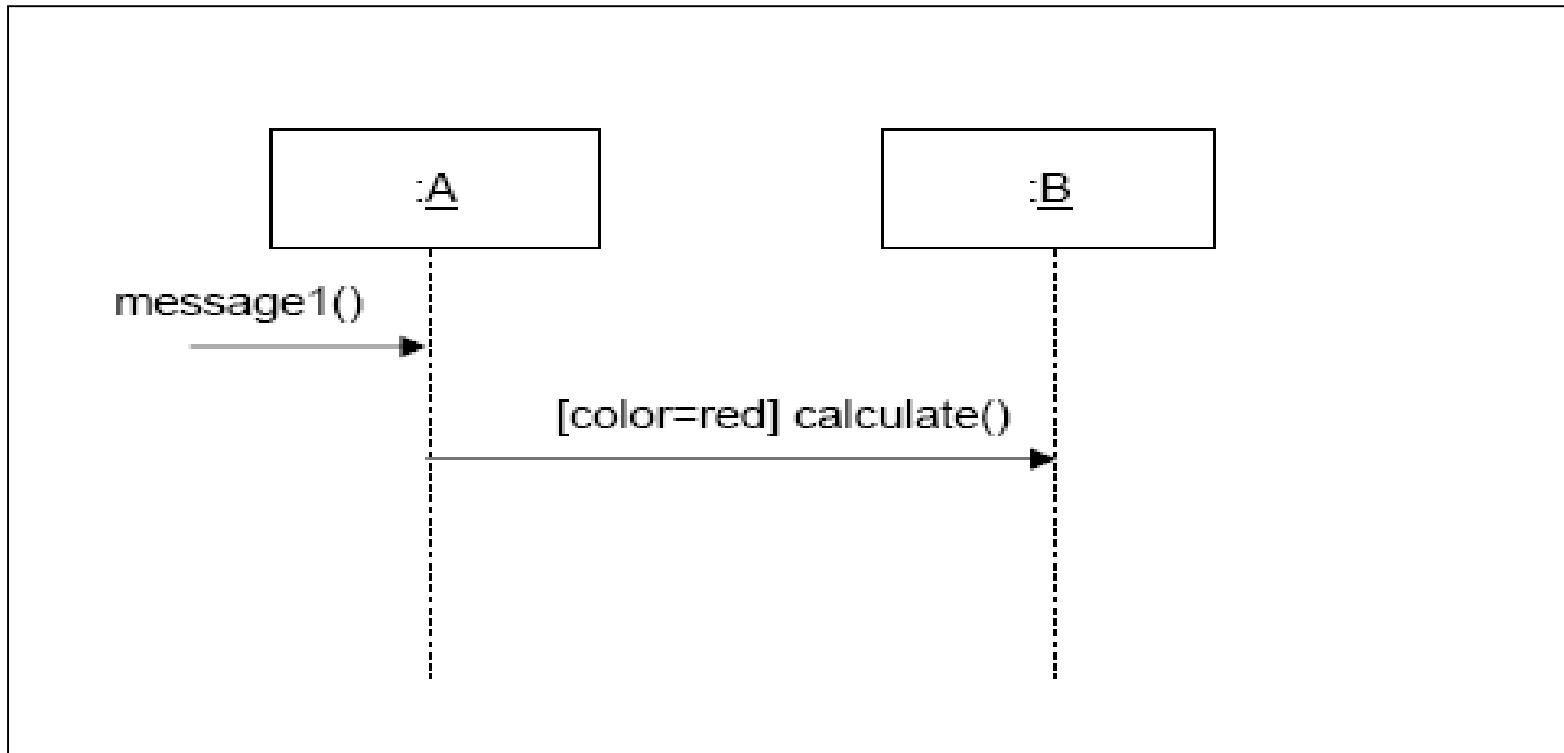
- A conditional message is shown by following a sequence number with a conditional clause in square brackets, similar to the iteration clause.
- The message is sent only if the clause evaluates to true.



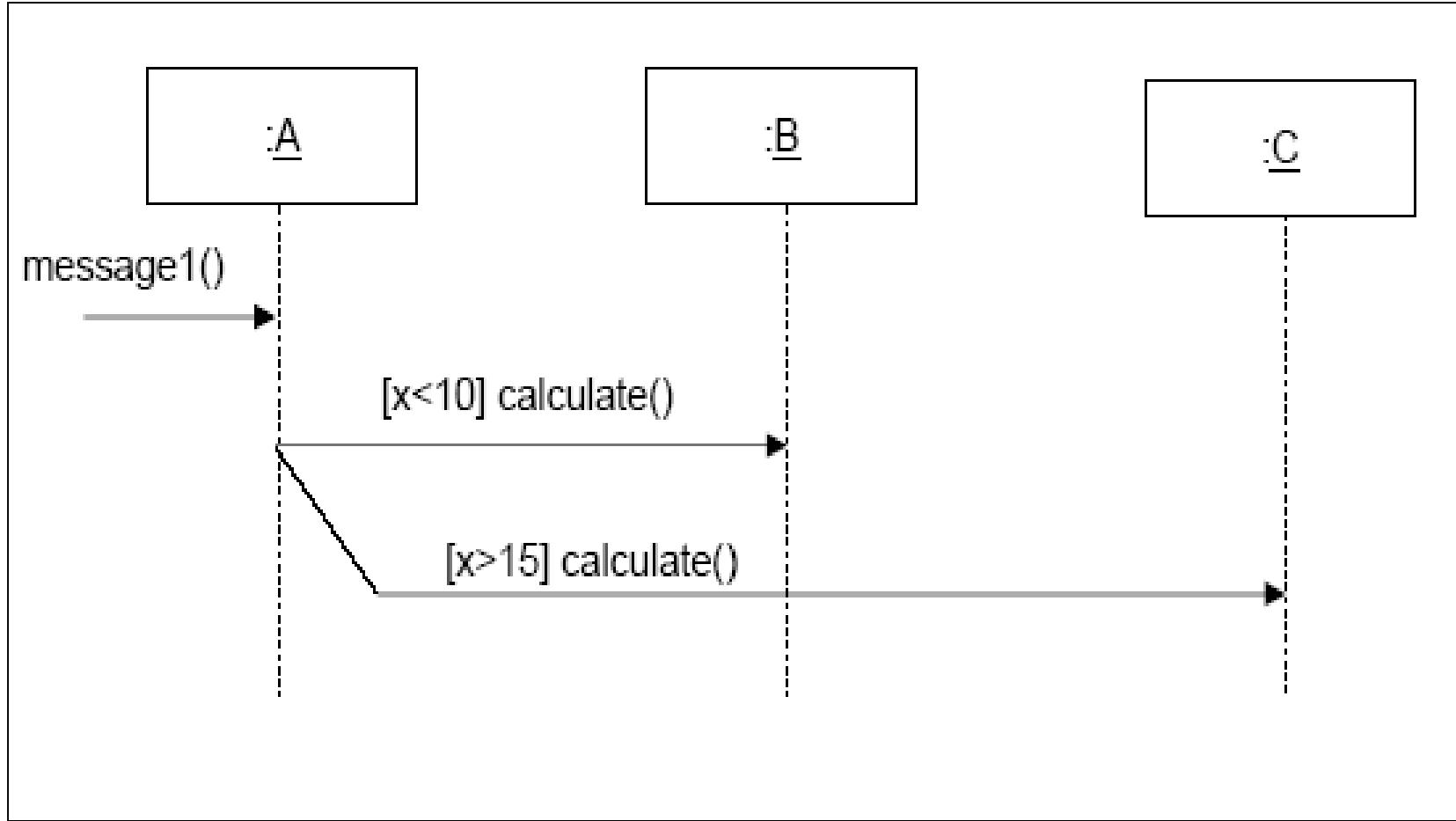
Conditional Messages



Mutually Exclusive Conditional Paths

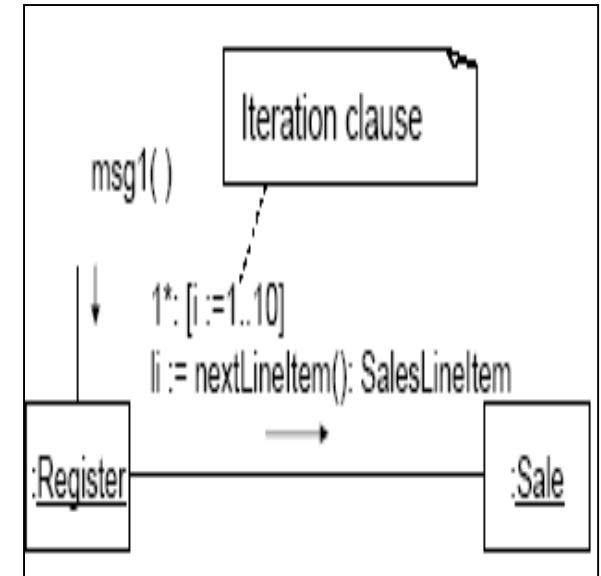
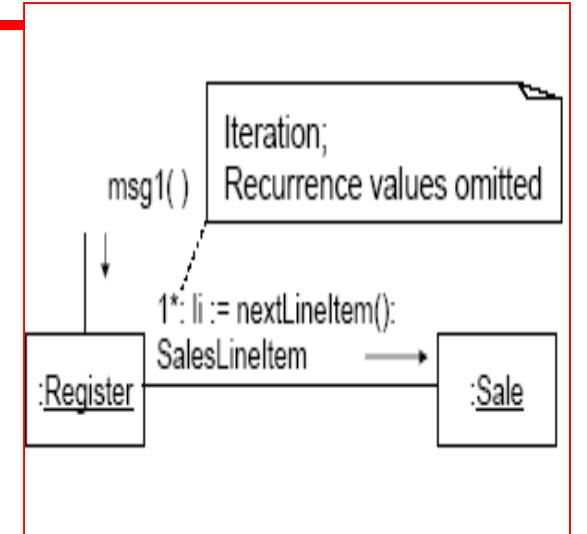


Mutually Exclusive Conditional Messages



Iteration or Looping

- Iteration is indicated by following the sequence number with a star *
- This expresses that the message is being sent repeatedly, in a loop, to the receiver.
- It is also possible to include an iteration clause indicating the recurrence values.





Object Oriented Analysis & Design

Module-4 (RL 4.2.4)



BITS Pilani

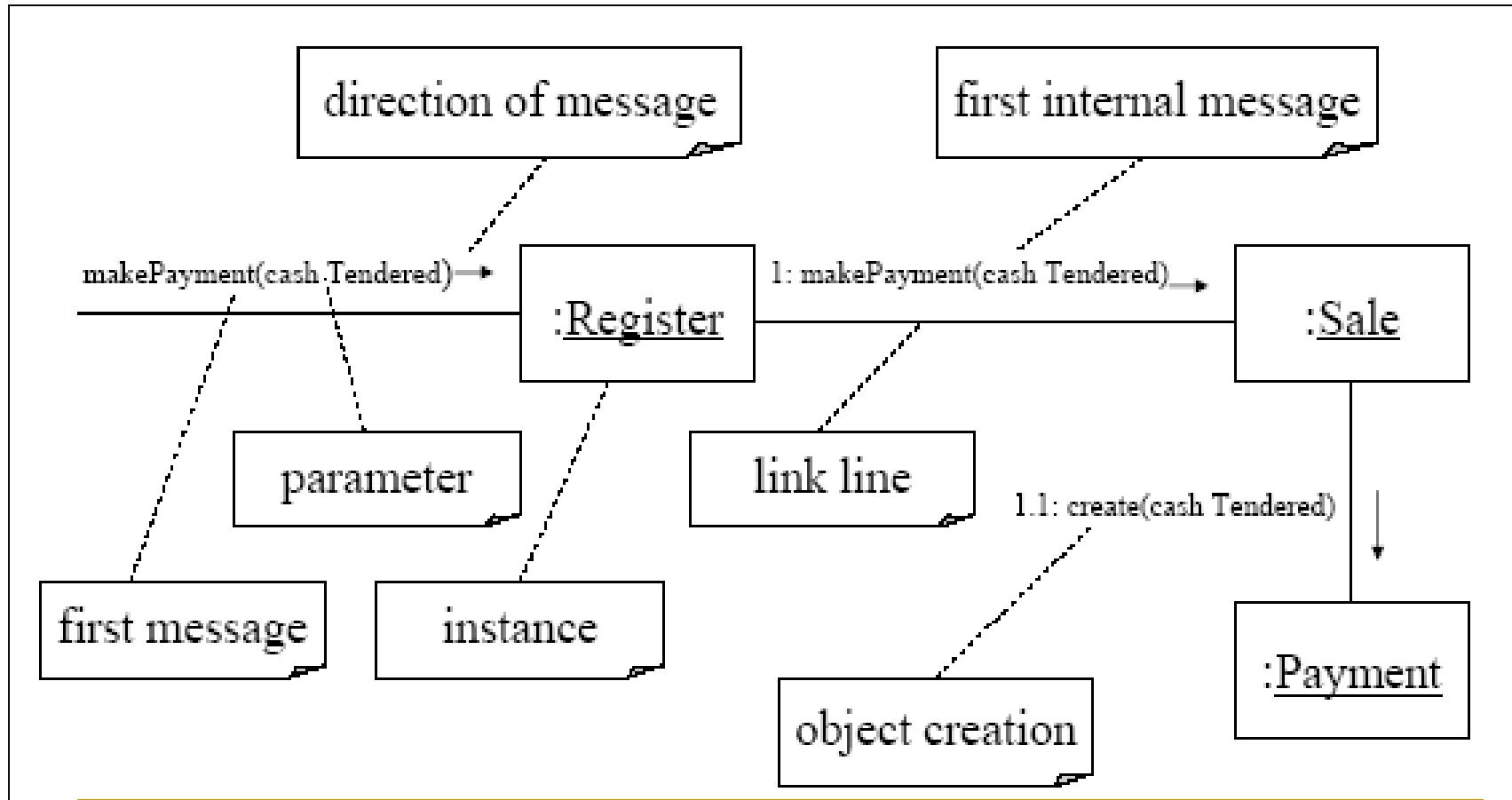
Pilani|Duba|Goa|Hyderabad

Sanjay Joshi

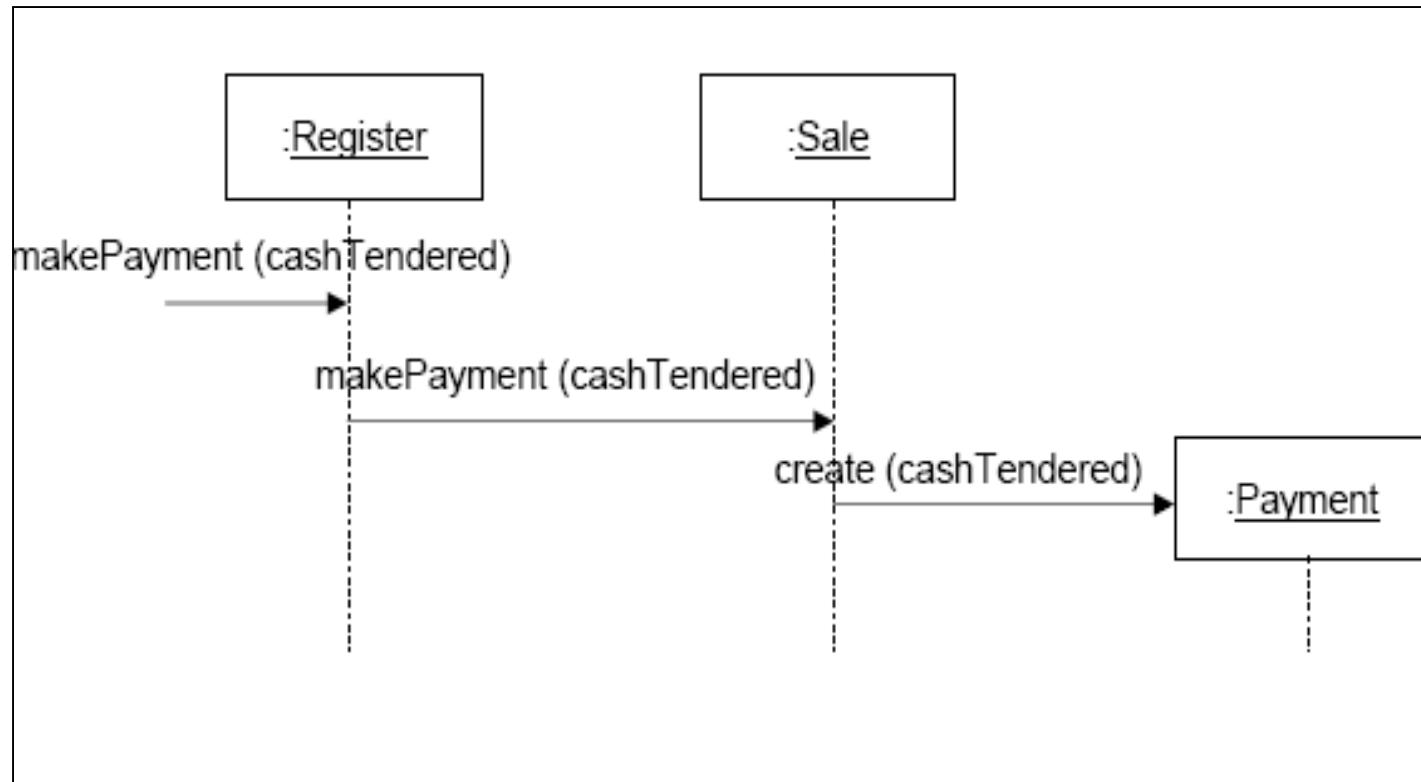


Drawing Interaction Diagrams for PoS

Collaboration Diagram: makePayment



Sequence Diagram : makePayment





Object Oriented Analysis & Design

Module-4 (RL 4.3.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Introduction to State Transition Diagram

Introduction to State Transition Diagram



- A state diagram (also state transition diagram) illustrates the events and the states of things.
- It can be drawn for System, Subsystem or an Object in the System.

Events, States and Transitions



- An event is a trigger, or occurrence.
 - e.g. a telephone receiver is taken off the hook.
- A state is the condition of an entity (object) at a moment in time - the time between events.
 - e.g. a telephone is in the state of being idle after the receiver is placed on the hook and until it is taken off the hook.

Events, States and Transitions

- A transition is a relationship between two states; It indicates that when an event occurs, the object moves from the prior state to the subsequent state.
 - e.g. when an event off the hook occurs, transition the telephone from the idle state to active state.

State Transition Diagrams

- A statechart diagram shows the life-cycle of an object; what events it experiences, its transitions and the states it is in between events.
- A state diagram need not illustrate every possible event; if an event arises that is not represented in the diagram, the event is ignored as far as the state diagram is concerned.
- Thus, we can create a state diagram which describes the life-cycle of an object at any simple or complex level of detail, depending on our needs.



Object Oriented Analysis & Design

Module-4 (RL 4.3.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi

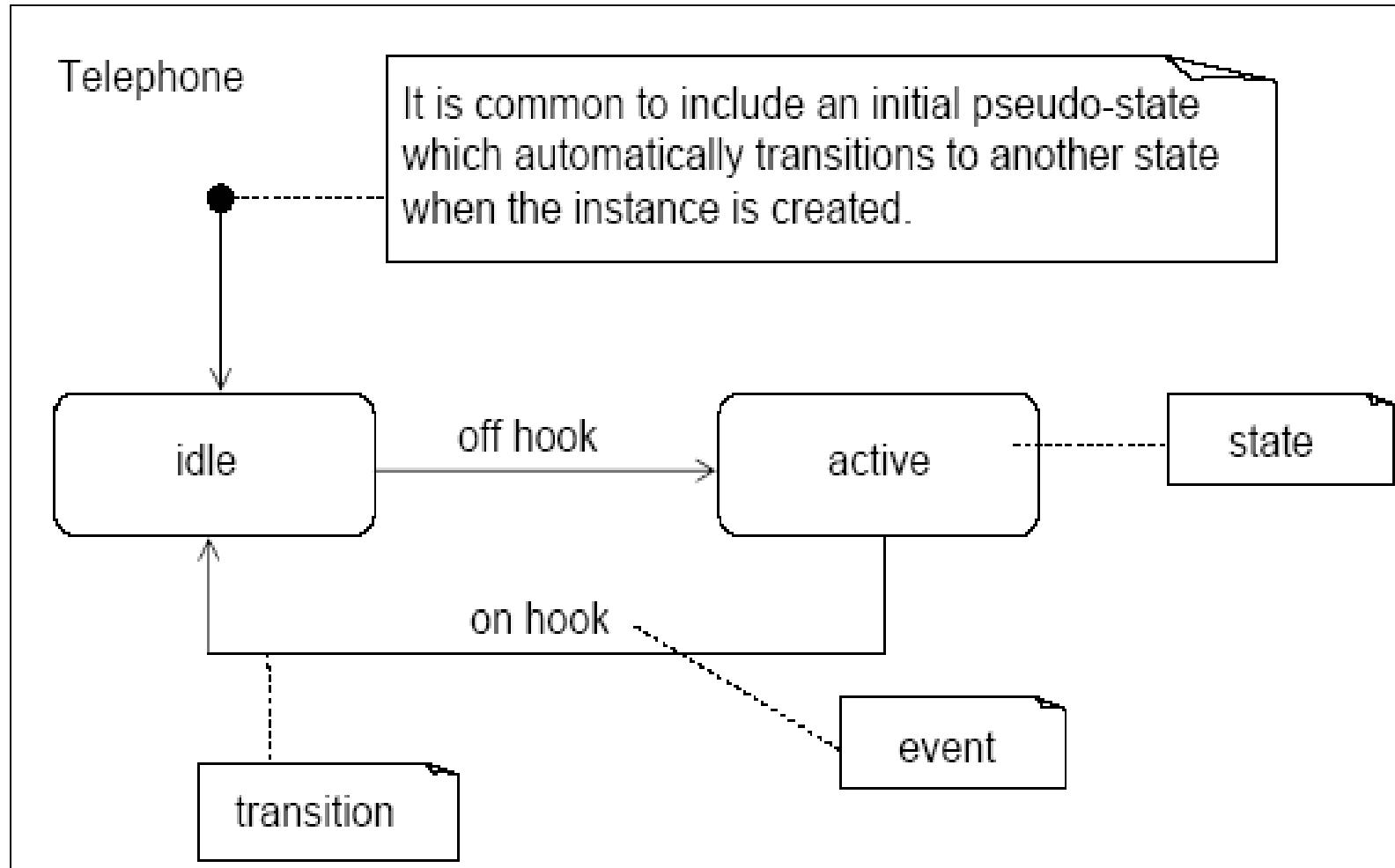


Representing State Transition Diagram in UML

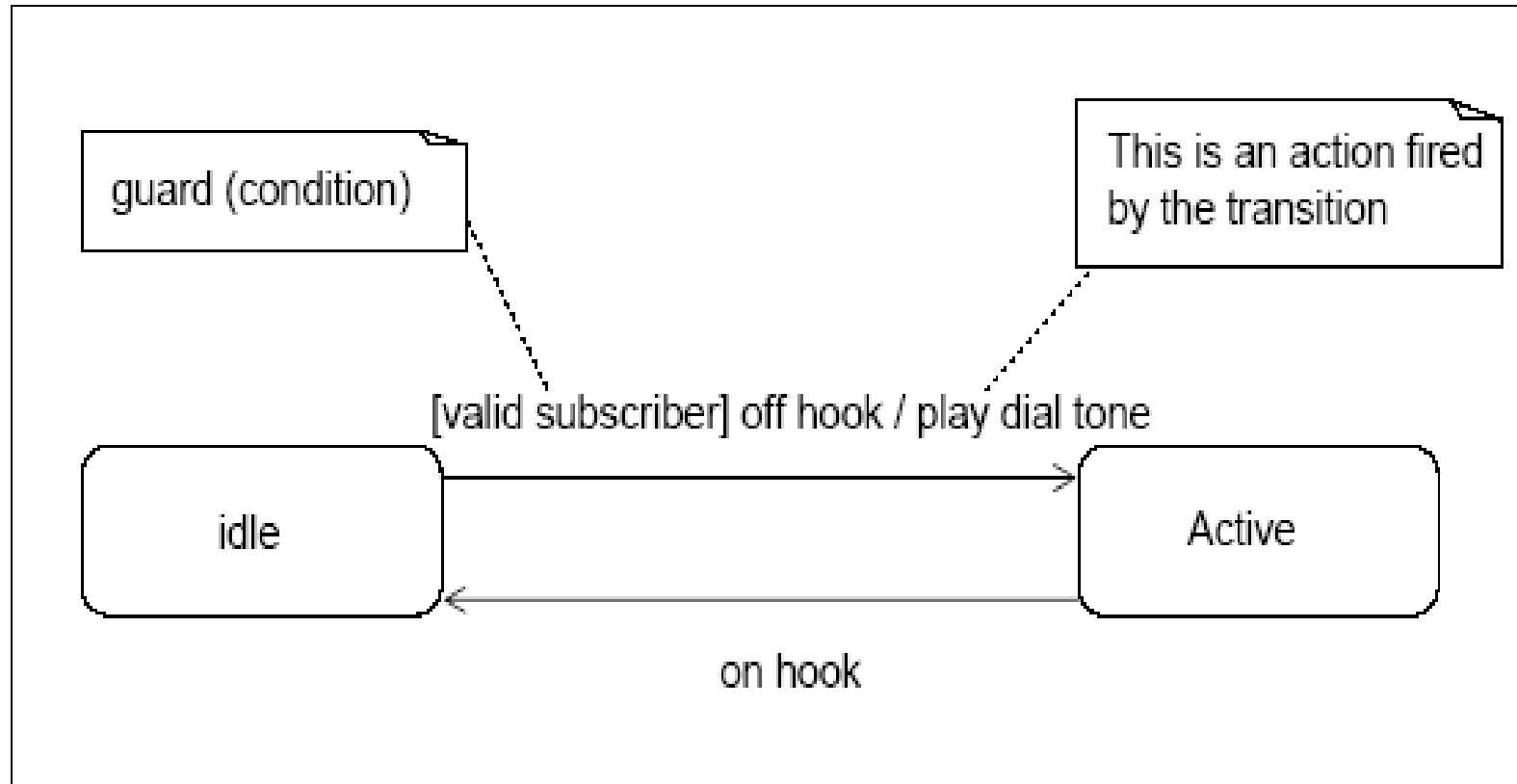
State Transition Diagram



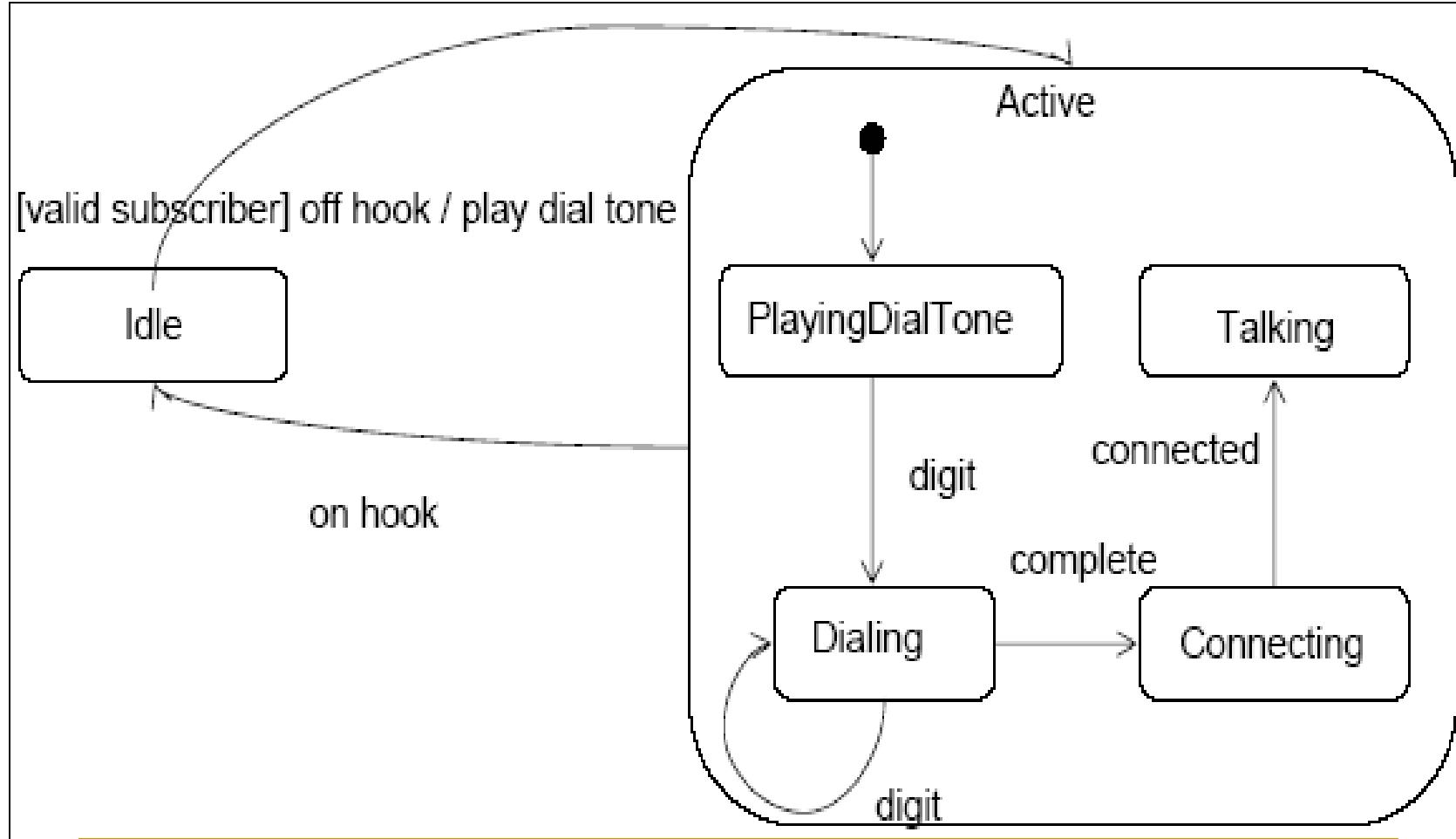
in UML



Additional State Transition Diagram Notation



Additional State Transition Diagram Notation



Additional State Transition Diagram Notation

- A state allows nesting to contain substates. A substate inherits the transitions of its superstate (the enclosing state).
- Within the Active state, and no matter what substate the object is in, if the on hook event occurs, a transition to the idle state occurs.



Object Oriented Analysis & Design

Module-4 (RL 4.3.3)



BITS Pilani

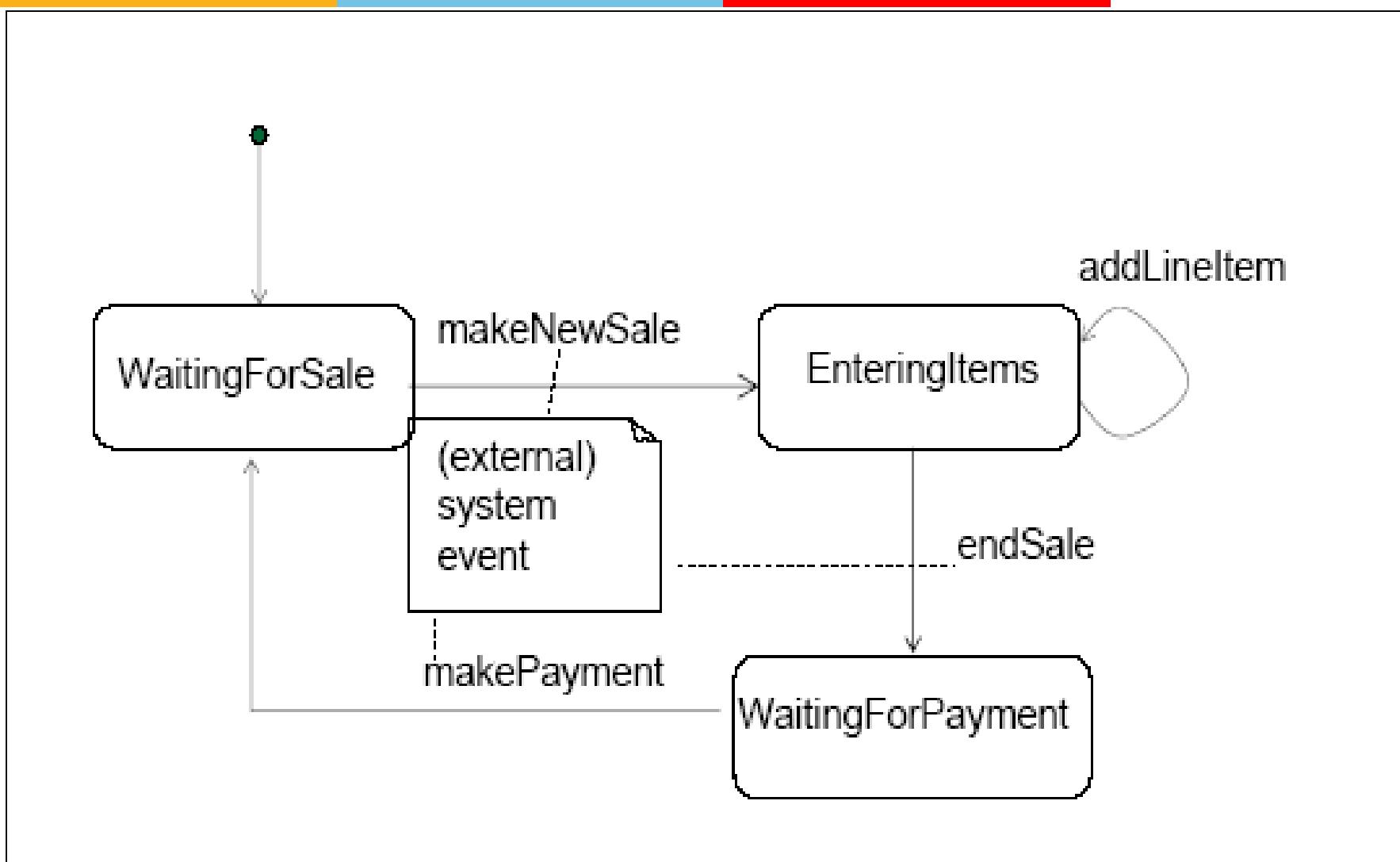
Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



Drawing State Transition diagram for PoS

State Transition Diagram for PoS





Object Oriented Analysis & Design

Module-4 (RL 4.4.1)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



What is Activity Diagram?

What is Activity Diagram

- They show the sequence of flow activities involved in a process.
- Used to model dynamic aspects of a system
- Like Flowchart showing flow of control from activity to activity
- Are used when you have multiple activities going on at the same time.



Object Oriented Analysis & Design

Module-4 (RL 4.4.2)



BITS Pilani

Pilani|Duba|Goa|Hyderabad

Sanjay Joshi



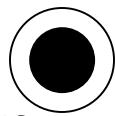
Representing Activity Diagram in UML

Activity Diagram in UML

- Initial state



- Final state



- Fork and join

– model concurrent flows



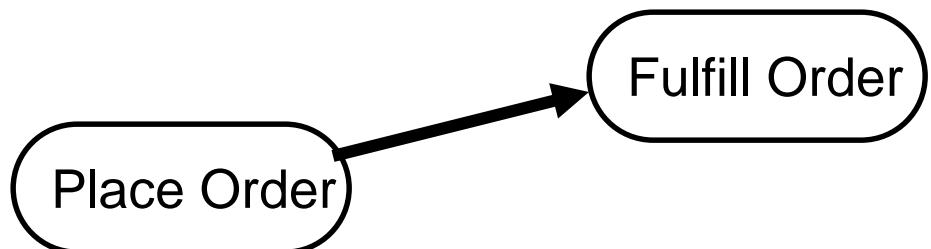
- Activities

– Step in overall Process



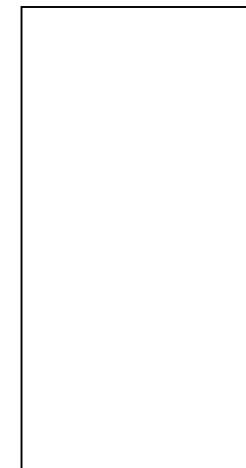
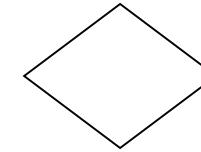
- Transitions

– Triggered by end of previous activity and initiates next activity



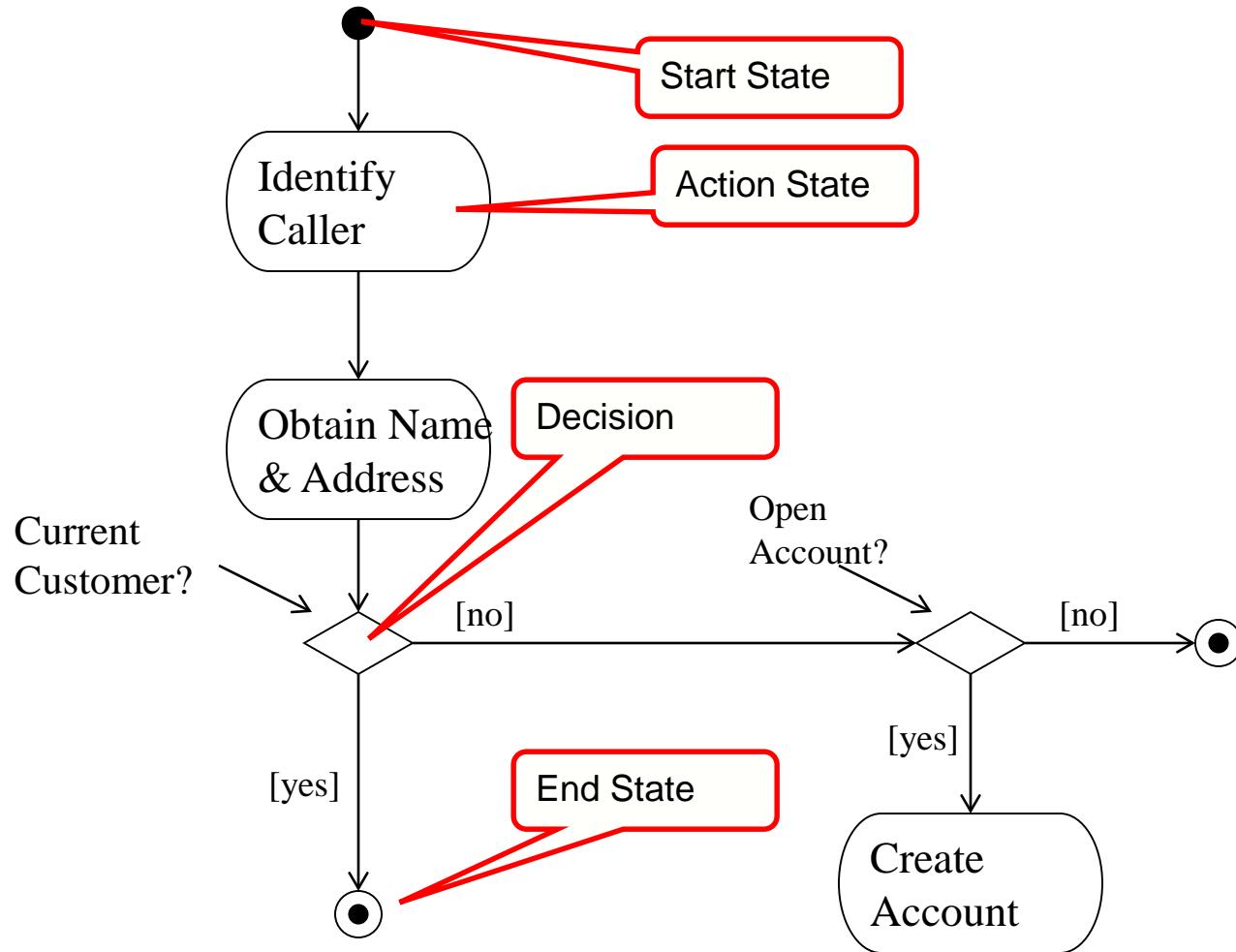
Activity Diagram in UML

- Branching: specifies alternate paths taken based Boolean expression
- swimlanes: Allow to partition the activity states into groups, each group representing the business organization responsible for those activities





Activity Diagram in UML





Object Oriented Analysis & Design

Module-5 (RL 5.1.1)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



What is Visibility among Objects

Visibility in UML Class Diagrams



3 common compartments
1. classifier name
2. attributes
3. operations

an interface shown with a keyword

«interface»
Runnable

run()

interface implementation and subclassing

SuperclassFoo
or
SuperClassFoo { abstract }

```

-classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [ * ]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute

```

```

+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
# protectedMethod()
~ packageVisibleMethod()
<constructor> SuperclassFoo( Long )
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }

```

SubclassFoo

```

...
run()
...

```

- ellipsis "..." means there may be elements, but not shown
- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common

java.awt.Font
or
java.awt.Font

```

plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...

```

getFont(name : String) : Font
getName() : String
...

dependency

Fruit

```

...

```

PurchaseOrder

```

...

```

order

association with multiplicities

Elements in a UML Class diagram are:

Visibility
Parameters
Compartments

These elements are optional

Visibility Between Objects

For Systems Operations to take place messages need to pass between objects.

- Sender Object Sends Message
- To a Received Object.

The UML provides four abbreviations for visibility:

- + (public),
- - (private),
- ~ (package), and
- # (protected)

Sender must be visible to receiver.

Sender must have some sort of pointer or reference to Receiver.

Visibility

The full format of the attribute text notation is:

Visibility name:type multiplicity = default [property-string]

- Visibility is a subject that is simple in principle but has complex subtleties.

The Simple idea is that any class has

- public and
- private elements

Public and Private Elements

- Public elements can be used by any other class;
 - Private elements can be used only by the owning class.
-
- However, each language makes its own rules.
 - Many languages use such terms as Public, private, and protected, they mean different things in different languages.
 - These differences are Small, but they lead to confusion, especially for those of us who use more than one language.

UML and Visibility

- The UML tries to address this without getting into a horrible tangle .
- Essentially, within the UML, you can tag any attribute or operation with a visibility indicator.
- You can use any marker you like, but its meaning is language dependent.

UML Syntax for Visibility

- These four levels are used within the UML meta-model and are defined within it, but their definitions vary subtly from those in other languages.

The UML provides four abbreviations for visibility:

- + (public),
- - (private),
- ~ (package), and
- # (protected)

Visibility and programming

When you are using visibility, use the rules of the language in which you are working.

- When you are looking at a UML model from elsewhere, be wary of the meanings of the visibility markers.
- Be aware of how those meanings can change from language to language.

Tips on use of Visibility

- don't draw visibility markers in diagrams;
- use them only if you need to highlight the differences in visibility of certain features
- even then, try to get away with + and -, which at least are easy to remember

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.



Object Oriented Analysis & Design

Module-5 (RL 5.1.2)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



Significance of finding Visibility

Visibility Between Objects

For Systems Operations to take place messages need to pass between objects.

- Sender Object Sends Message
- To a Received Object.

The UML provides four abbreviations for visibility:

- + (public),
- - (private),
- ~ (package), and
- # (protected)

Sender must be visible to receiver.

Sender must have some sort of pointer or reference to Receiver.

Significance

Sender:

Register

Receiver:

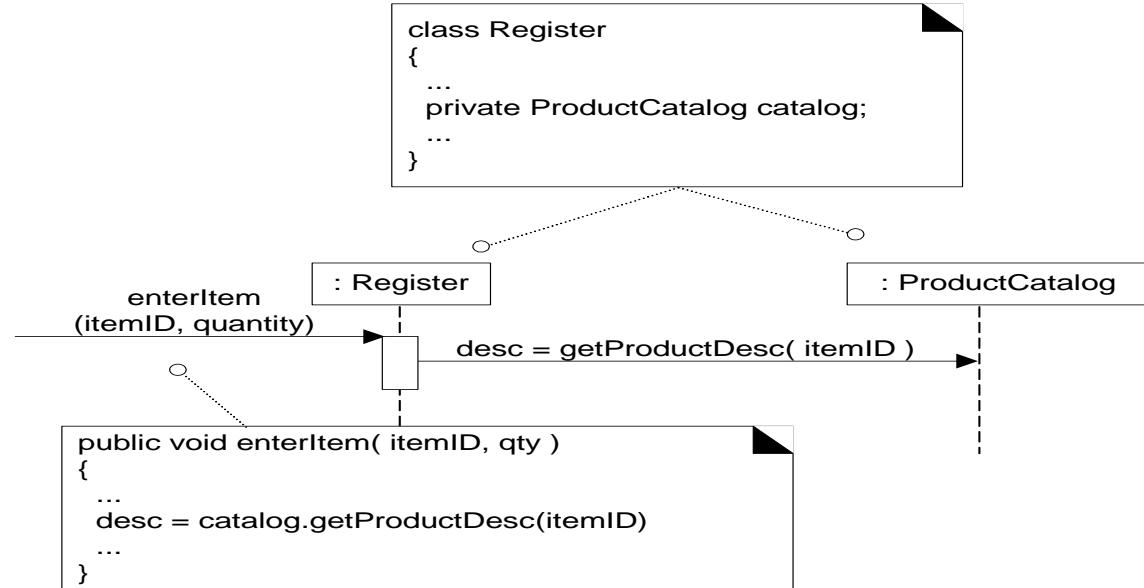
ProductCatalogue

Message:

getProductDesc

Visibility from the Register to ProductCatalogue is required

ProductCatalogue must be visible to Register for interaction to take place.



Significance of finding Visibility

- Visibility is the ability of an object to “see” or have a reference to another object.
- This is an issue with relation to scope.
- It settles the issue as to whether one object is within the scope of another object/ resource/ instance.

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.



Object Oriented Analysis & Design

Module-5 (RL 5.1.3)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



Types of Visibility – Attribute, Parameter, Local & Global Visibility

Ways to achieve visibility

- There are 4 common ways of achieving visibility
 - Attribute Visibility
 - Parameter Visibility
 - Local Visibility
 - Global Visibility

Attribute Visibility

Object B will be visible to Object A

If

B is an attribute of A.

In the declaration of Class A all Objects that exist as attributes are visible to Objects instantiated from Class A.

Parameter Visibility

Object B will be visible to Object A

If

Object B is a parameter of a method in A

All Objects that are passed as parameter to various methods in Objects instantiated from Class A will be visible to these Objects of Class A.

Local Visibility

Object B may be said to be visible to Object A

If

Object B is a local object (non-parameter) in a method of Object A.

All Objects that are instantiated in a method are visible in that method and are said to be visible to the Object to which the method belongs.

Global Visibility

Object B will be visible to Object A

If

Object B has been instantiated as a globally visible object.

All globally visible Objects
are visible to each Class in
the environment to which
they apply.

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.



Object Oriented Analysis & Design

Module-5 (RL 5.1.4)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



Attribute Visibility

Attribute Visibility

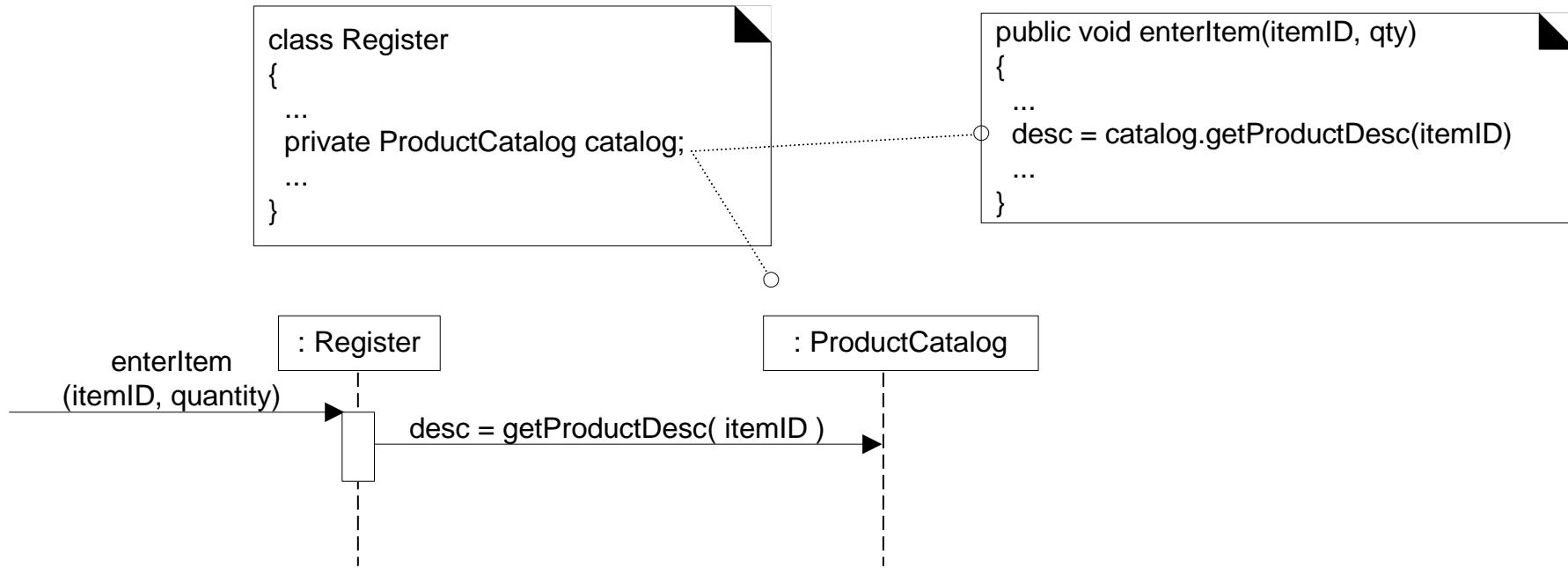
Object B will be visible to Object A

If

B is an attribute of A.

In the declaration of Class A all Objects that exist as attributes are visible to Objects instantiated from Class A.

Permanent Visibility Persists as long as A and B exist



A `ProductCatalog` object (`catalog`)

Is visible to a `Register` Object

Because `catalog` is an attribute of `Register`.

This is necessary as the message
`getProdDesc` is to be sent

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.



Object Oriented Analysis & Design

Module-5 (RL 5.1.5)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



BITS Pilani

Pilani|Dubai|Goa|Hyderabad



Parameter Visibility

Parameter Visibility

Object B will be visible to Object A

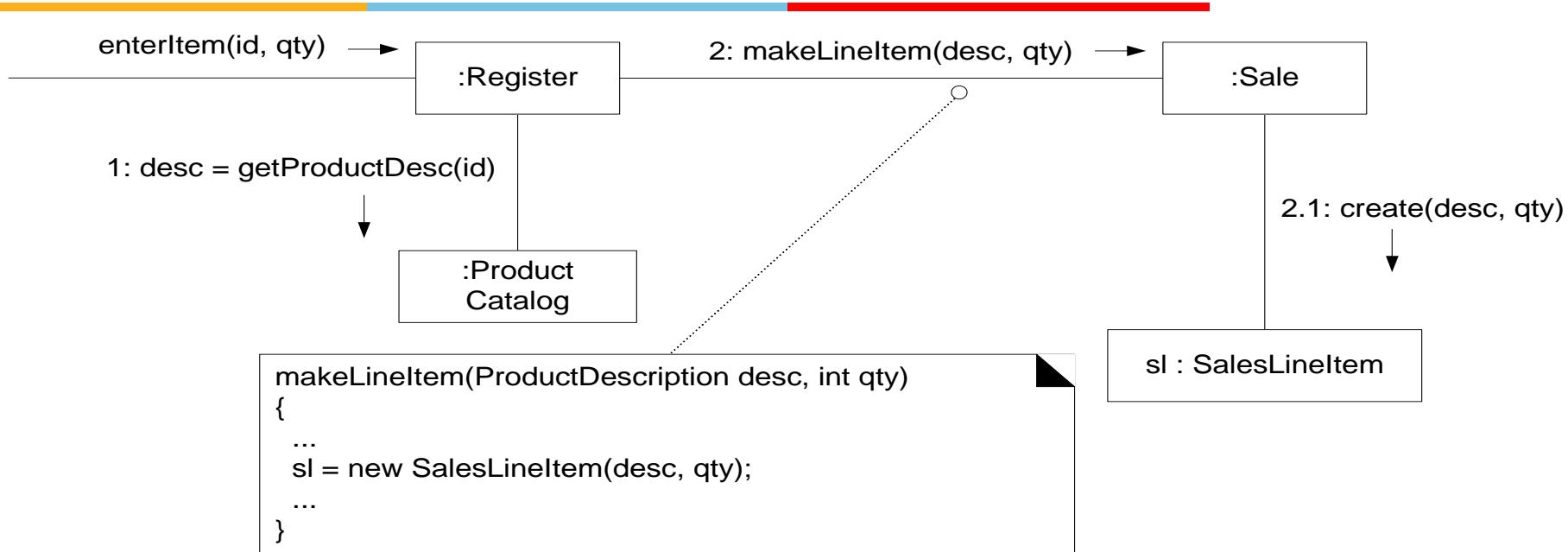
If

Object B is a parameter of a method in A

All Objects that are passed as parameter to various methods in Objects instantiated from Class A will be visible to these Objects of Class A.

Temporary Visibility

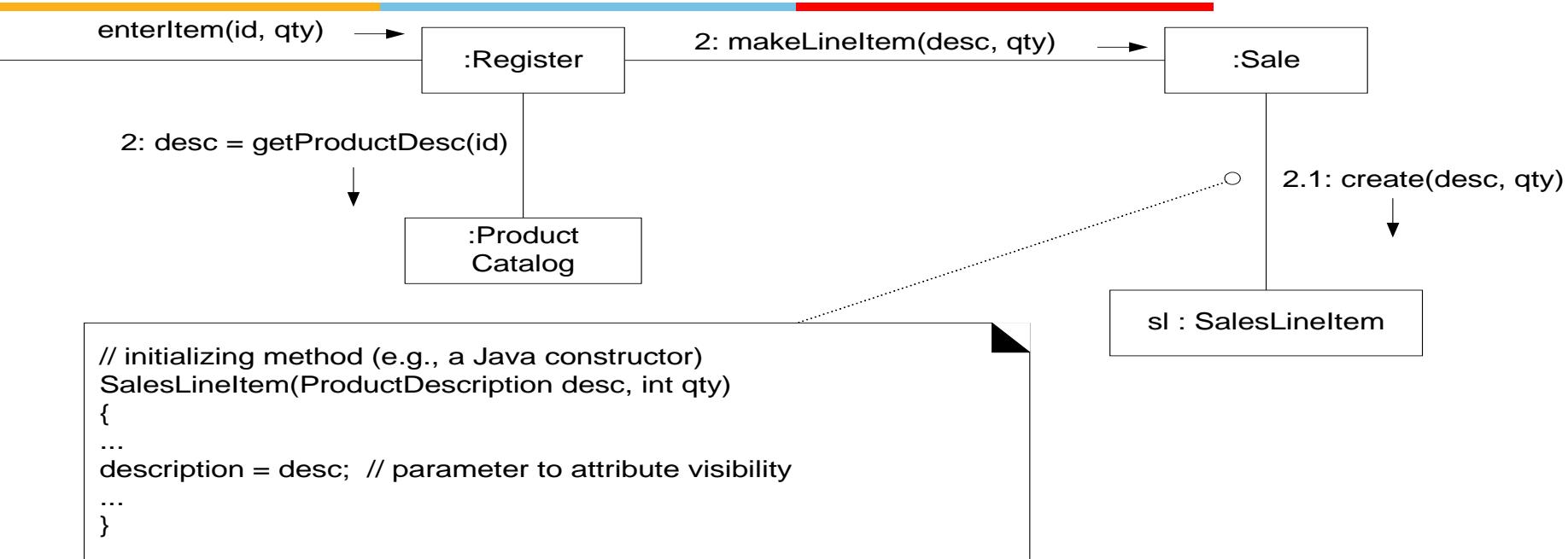
Persists only within the scope of the method



desc is an Object of Type ProductDescription
and is passed as a parameter by a Register
Object to method in an Object of Sale.

desc is visible to the Sale Object while the
method is being executed.

Parameter to Attribute Visibility



The Object received as a parameter may be used in a Constructor to assign the Object to an Attribute of the Class.

This establishes Attribute Visibility.

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.



Object Oriented Analysis & Design

Module-5 (RL 5.1.6)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



BITS Pilani

Pilani|Dubai|Goa|Hyderabad



Local Visibility

Local Visibility

Object B may be said to be visible to Object A

If

Object B is a local object (non-parameter) in a method of Object A.

All Objects that are instantiated in a method are visible in that method and are said to be visible to the Object to which the method belongs.

Temporary Visibility

Persists only within the scope of the method

```
enterItem(id, qty)
{
...
// local visibility via assignment of returning object
ProductDescription desc = catalog.getProductDes(id);
...
}
```



An interesting case is when an Object is Created as a result of a returning object from a method invocation:

anObject.getFoo().doBar();



Achieving Local Visibility

- Create a **new local instance** and assign it to a **local variable**.
 - Description desc = new Description();
- Assign the **returning object from a method invocation** to a **local variable**.
 - Description desc = catalog.getDescription(ItemID);

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.



Object Oriented Analysis & Design

Module-5 (RL 5.1.7)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



BITS Pilani

Pilani|Dubai|Goa|Hyderabad



Global Visibility

Global Visibility

Object B will be visible to Object A

If

Object B has been instantiated as a globally visible object.

All globally visible Objects are visible to each Class in the environment to which they apply.

Singleton Pattern

- Not a common form of Visibility in Object Oriented Systems.
- Relatively permanent as it exists as long as Objects of both type exist.
- Can be achieved by assigning an instance of a global variable.
- Best achieved through a Singleton Pattern. A single instance of a Class is generated and is used with a global presence where-ever and when-ever required.

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.



Object Oriented Analysis & Design

Module-5 (RL 5.2.1)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



Use Domain Model to draw Class Diagram

Domain Models

- “A domain model captures the most important types of objects in the context of the business. The domain model represents the ‘things’ that exist or events that transpire in the business environment.” – I. Jacobsen

Why Draw a Domain Model

- Gives a conceptual framework of the things in the problem space
- Helps you think – focus on semantics
- Provides a glossary of terms – noun based
- It is a static view - meaning it allows us convey time invariant business rules
- Foundation for use case/workflow modelling
- Based on the defined structure, we can describe the state of the problem domain at any time.

Features

- The following features enable us to express time invariant static business rules for a domain:-
 - **Domain classes** – each domain class denotes a type of object.
 - **Attributes** – an attribute is the description of a named slot of a specified type in a domain class; each instance of the class separately holds a value.
 - **Associations** – an association is a relationship between two (or more) domain classes that describes links between their object instances. Associations can have roles, describing the multiplicity and participation of a class in the relationship.
 - **Additional rules** – complex rules that cannot be shown with symbology can be shown with attached notes.

Domain Classes

- Each domain class denotes a type of object. It is a descriptor for a set of things that share common features. Classes can be:-
 - *Business objects* - represent things that are manipulated in the business e.g. *Order*.
 - *Real world objects* – things that the business keeps track of e.g. *Contact, Site*.
 - *Events that transpire* - e.g. *sale and payment*.
- A domain class has attributes and associations with other classes (discussed below). It is important that a domain class is given a good description

Domain Modelling

Perform the following in very short iterations:

- o Make a list of candidate domain classes.
 - o Draw these classes in a UML class diagram.
 - o If possible, add brief descriptions for the classes.
 - o Identify any associations that are necessary.
 - o Decide if some domain classes are really just attributes.
 - o Where helpful, identify role names and multiplicity for associations.
 - o Add any additional static rules as UML notes that cannot be conveyed with UML symbols.
 - o Group diagrams/domain classes by category into packages.
- Concentrate more on just identifying domain classes in early iterations !

Identifying Domain Classes

- An obvious way to identify domain classes is to identify nouns and phrases in textual descriptions of a domain.

Consider a use case description as follows:-

1. Customer arrives at a checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records the sale line item and presents the item **description, price** and running **total**.

Identifying Attributes

- A domain class sounds like an attribute if: -
 - It relies on an associated class for its identity
 - e.g. ‘order number’ class associated to an ‘order’ class. The ‘order number’ sounds suspiciously like an attribute of ‘order’.
 - It is a simple data type – e.g. ‘order number’ is a simple integer. Now it really sounds like an attribute!

Class Diagram from Domain Model

- UML design class diagrams (DCD) show software class definitions. They are based on the collaboration diagram. Attribute visibility is shown for permanent connections. Classes are shown with their simple attributes and methods listed.

Design Class Diagrams and UP

- Typical information in a DCD includes:
 - Classes, associations and attributes
 - Interfaces (with operations and constants)
 - Methods
 - Attribute type information
 - Navigability
 - Dependencies
- The DCD depends upon the Domain Model and interaction diagrams.
- The UP defines a Design Model which includes interaction and class diagrams.

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
 - <http://creately.com/diagram-type/article/simple-guidelines-drawing-uml-class-diagrams>



Object Oriented Analysis & Design

Module-5 (RL 5.2.2)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



Representing Class in UML

Design Class Diagrams and UP

- Typical information in a DCD includes:
 - Classes, associations and attributes
 - Interfaces (with operations and constants)
 - Methods
 - Attribute type information
 - Navigability
 - Dependencies
- The DCD depends upon the Domain Model and interaction diagrams.
- The UP defines a Design Model which includes interaction and class diagrams.

Guideline for Drawing



UML Class Diagrams

- When it comes to system construction, a **class diagram** is the most widely used diagram. This diagram generally consists of **interfaces, classes, associations and collaborations**. Such a diagram would illustrate the object-oriented view of a system, which is static in nature. The object orientation of a system is indicated by a class diagram.
- Since class diagrams are used for many different purposes, such as making stakeholders aware of requirements to highlighting your detailed design, you need to apply a different style in each circumstance.
- The points that are going to be covered are indicated as follows:
- General issues
- Classes
- Interfaces
- Relationships
- Inheritance
- Aggregation and Composition

General Issues

A - Analysis and design versions of a class

Analysis	Design
Order	Order
Placement Date	- deliveryDate: Date
Delivery Date	- orderNumber: int
Order Number	- placementDate: Date
Calculate Total	- taxes: Currency
Calculate Taxes	- total: Currency
	# calculate Taxes (Country, State): Currency
	# calculate Total (): Currency
	getTaxEngine () {visibility=implementation}

Show visibility only on design models

Assess responsibilities on domain class diagrams

Highlight language-dependent visibility with property strings

Highlight types only on design models

Highlight types on analysis models only when the type is an actual requirement

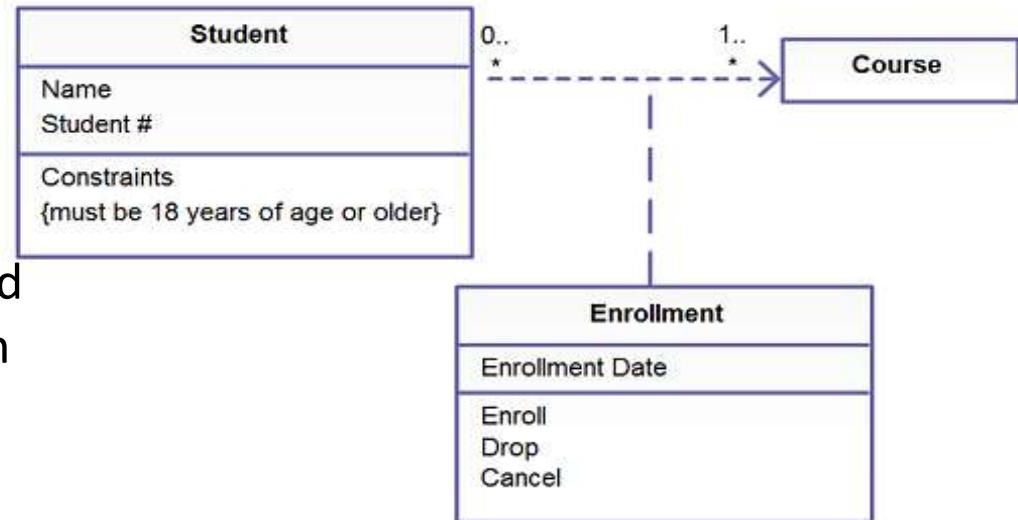
Association Classes

Model association classes on analysis diagrams.

The image that shows the “Modelling association classes” indicates the association classes that are depicted as class attached via a dashed line to an association – the association line, the class, and the dashed line are considered one symbol in the UML.

- Do not name associations that have association classes.
- Center the dashed line of an association class.

B - Modeling association classes



Class Style

A class is basically a template from which objects are created.

Classes define attributes, information that are relevant to their instances, operations, and functionality that the objects support.

Some of the more important guidelines pertinent to classes are listed in the next slide.

Scaffolding code refers to the attributes and operations required to use basic functionality within your classes, such as the code required to implement relationships with other classes.

Without scaffolding

```
OrderItem
#numberOrdered:int
+findForItem(Item): Vector
+findForOrder(Order): Vector
#calculateTaxes(): Currency
#calculateTotal(): Currency
-getTaxEngine()
```

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
 - <http://creately.com/diagram-type/article/simple-guidelines-drawing-uml-class-diagrams>



Object Oriented Analysis & Design

Module-5 (RL 5.2.3)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

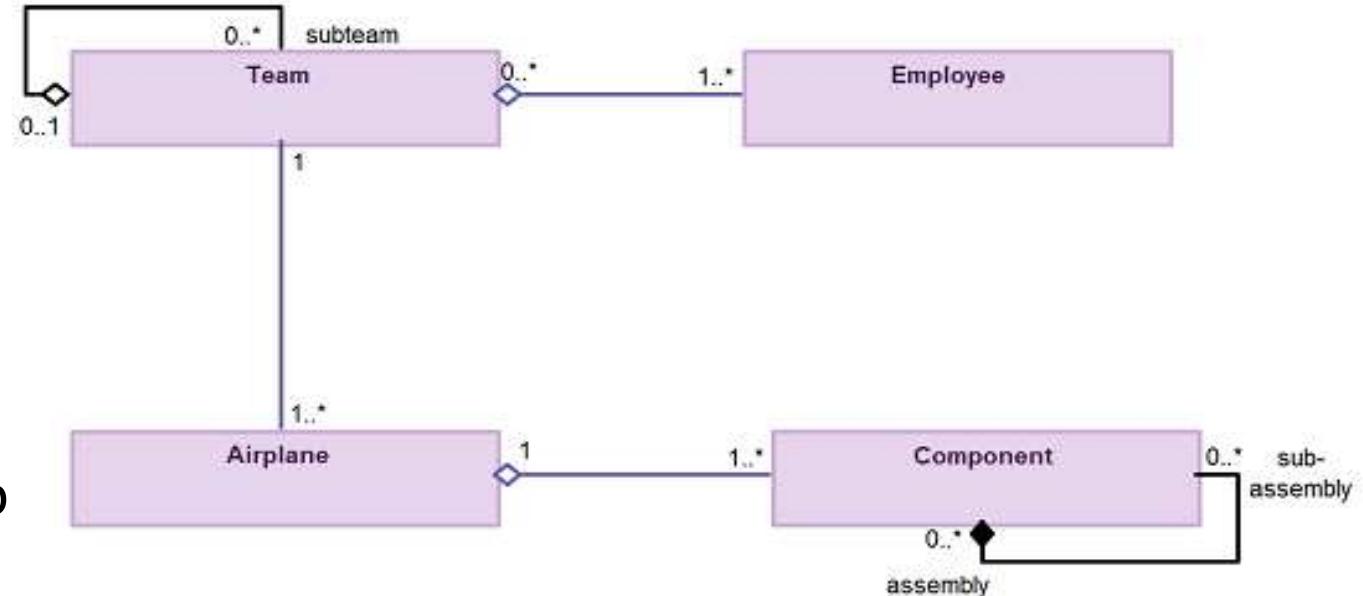
Harvinder S Jabbal



Relationship among Classes in Class Diagram

Aggregation and Composition

Aggregation is a specialization of association, highlighting an entire-part relationship that exists between two objects.



Composition is a much potent form of aggregation where the whole and parts have coincident lifetimes, and it is very common for the whole to manage the lifecycle of its parts.

Guideline for Aggregation and Composition

- You should be interested in both the whole and the part
- Depict the whole to the left of the part
- Apply composition to aggregates of physical items

Inheritance

- Inheritance models “is a” and “is like” relationships, enabling you to rather conveniently reuse data and code that already exist.
- When “A” inherits from “B” we say that “A” is the subclass of “B” and that “B” is the superclass of “A.”
- We have “pure inheritance” when “A” inherits all of the attributes and methods of “B”.
- The UML modeling notation for inheritance is usually depicted as a line that has a closed arrowhead, which points from the subclass right down to the superclass.
- Plus in the sentence rule for inheritance
- Put subclasses below superclasses
- Ensure that you are aware of data-based inheritance
- A subclass must inherit everything

Relationship

encompass all UML concepts such as

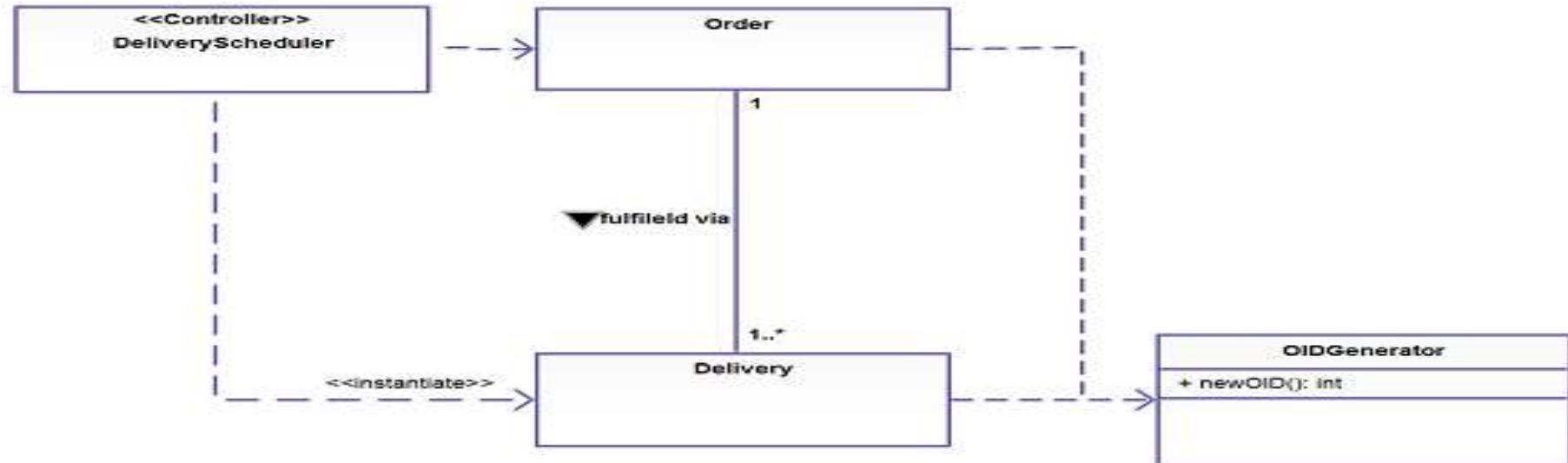
- aggregation,
- associations,
- dependencies,
- composition,
- realizations, and
- inheritance.

- if it's a line on a UML class diagram, it can be considered as a relationship.

Guideline for Relationship

- Ensure that you model relationships horizontally
 - Collaboration means a need for a relationship
 - Model a dependency when a relationship is in transition
 - As a rule it is best to always indicate the multiplicity
 - Avoid a multiplicity of “*” to avoid confusion
 - Never model implied relationships
-

Relationship (Figure A)

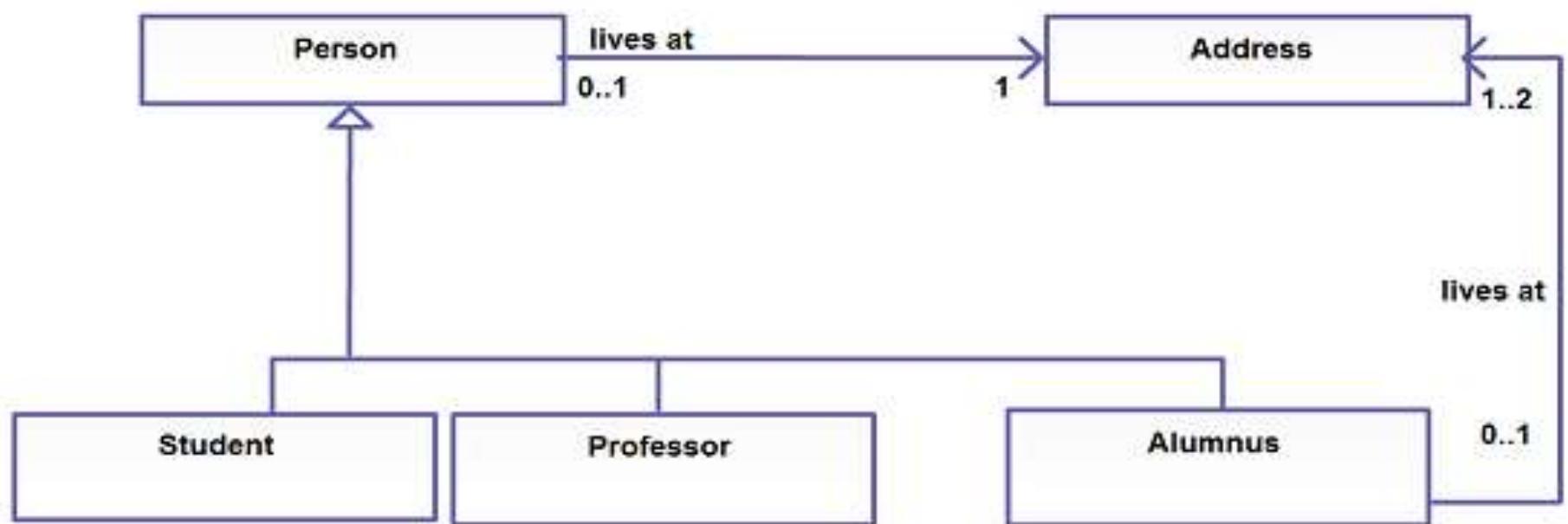


- Depict similar relationships involving a common class as a tree.
 - In **Figure A** you see that both **Delivery** and **Order** have a dependency on **OIDGenerator**. Note how the two dependencies are drawn in combination in “tree configuration”, instead of as two separate lines, to reduce clutter in the diagram.

Guideline for Relationship

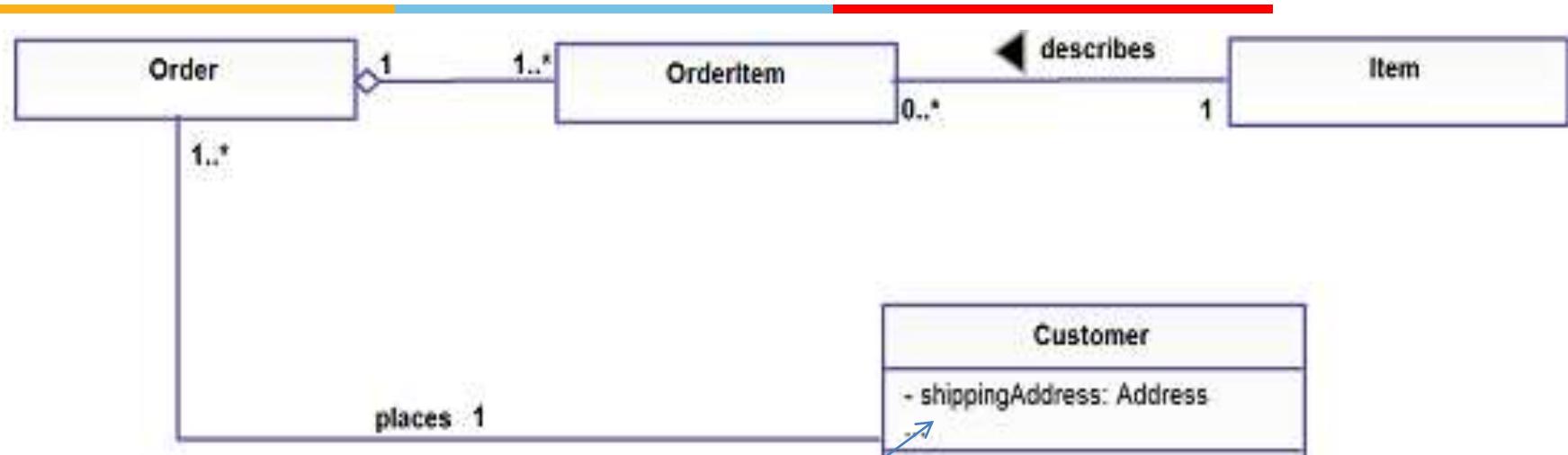
- Never model every single dependency
- Center names on associations
- Write concise association names in active voice
- Indicate directionality to clarify an association name
- Name unidirectional associations in the same direction
- Word association names left-to-right
- Indicate role names when multiple associations between two classes exist
- Indicate role names on recursive associations
- Redraw inherited associations only when something changes
- Question multiplicities involving minimums and maximums

Relationship (Figure B)



Make associations bi-directional only when collaboration occurs in both directions. The lives at association of **Figure B** is unidirectional.

Relationship (Figure C)



- Replace relationships by indicating attribute types.
- In **Figure** you see that the customer has a `shippingAddress` attribute of type `Address` – part of the scaffolding code to maintain the association between customer objects and address objects.

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
 - <http://creately.com/diagram-type/article/simple-guidelines-drawing-uml-class-diagrams>



Object Oriented Analysis & Design

Module-5 (RL 5.2.4)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



Guidelines to draw Class Diagram

Guideline for Classes

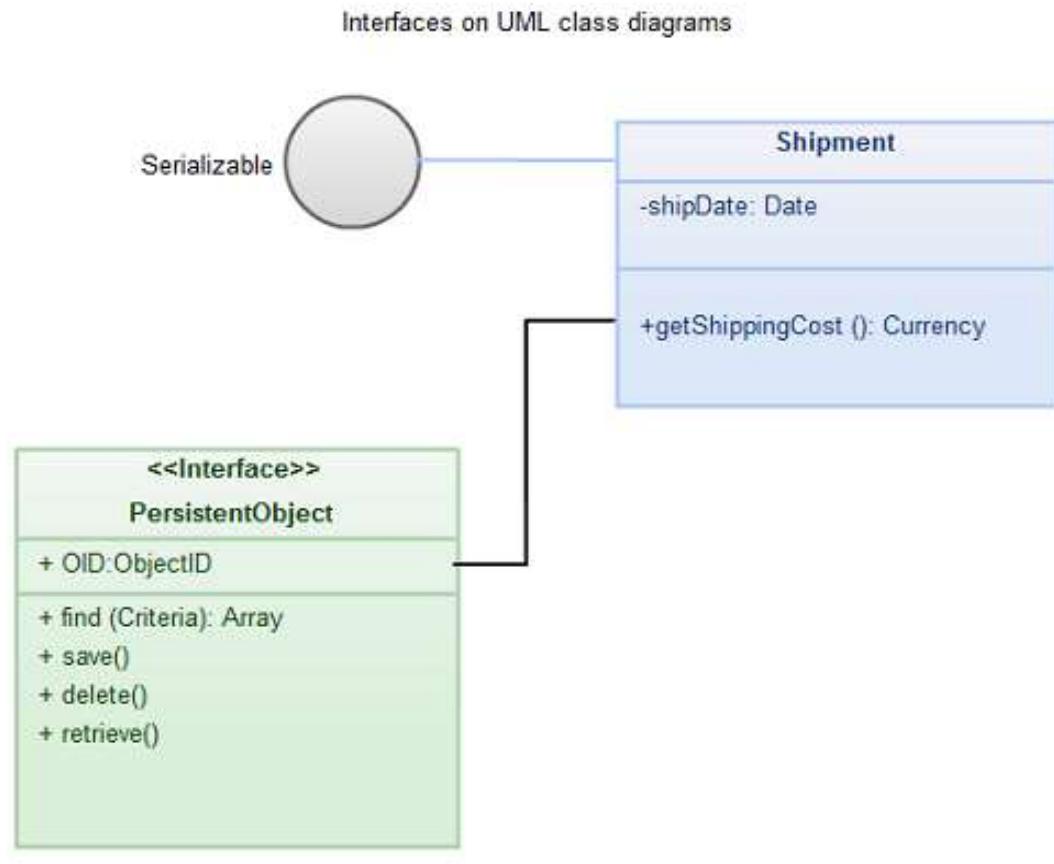
- Put common terminology for names
- Choose complete singular nouns over class names
- Name operations with a strong verb
- Name attributes with a domain-based noun
- Do not model scaffolding code.
- Never show classes with just two compartments
- Label uncommon class compartments
- Include an ellipsis (...) at the end of incomplete lists

Guidelines (cont.)

- List static operations/attributes before instance operations/attributes
- List operations/attributes in decreasing visibility
- For parameters that are objects, only list their type
- Develop consistent method signatures
- Avoid stereotypes implied by language naming conventions
- Indicate exceptions in an operation's property string. Exceptions can be indicated with a UML property string.

Interfaces

An interface can be defined as collection of operation signature and/or attribute definitions that ideally defines a cohesive set of behaviors.



Interfaces

- In order to realize an interface, a class or component should use the operations and attributes that are defined by the interface.
- Any given class or component may use zero or more interfaces and one or more classes or components can use the same interface.
- Interfaces are implemented, “realized” in UML parlance, by classes and components.

Guideline for Interfaces

- Interface definitions must reflect implementation language constraints.
- In the example, you see that a standard class box has been used to define the interface PersistentObject (note the use of the <<interface>> stereotype).
- Name interfaces according to language naming conventions
- Apply “Lollipop” notation to indicate that a class realizes an interface
- Define interfaces separately from your classes
- Do not model the operations and attributes of an interface in your classes. In the above image, you’ll notice that the shipment class does not include the attributes or operations defined by the interfaces that it realizes
- Consider an interface to be a contract

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
 - <http://creately.com/diagram-type/article/simple-guidelines-drawing-uml-class-diagrams>



Object Oriented Analysis & Design

Module-5 (RL 5.2.5)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

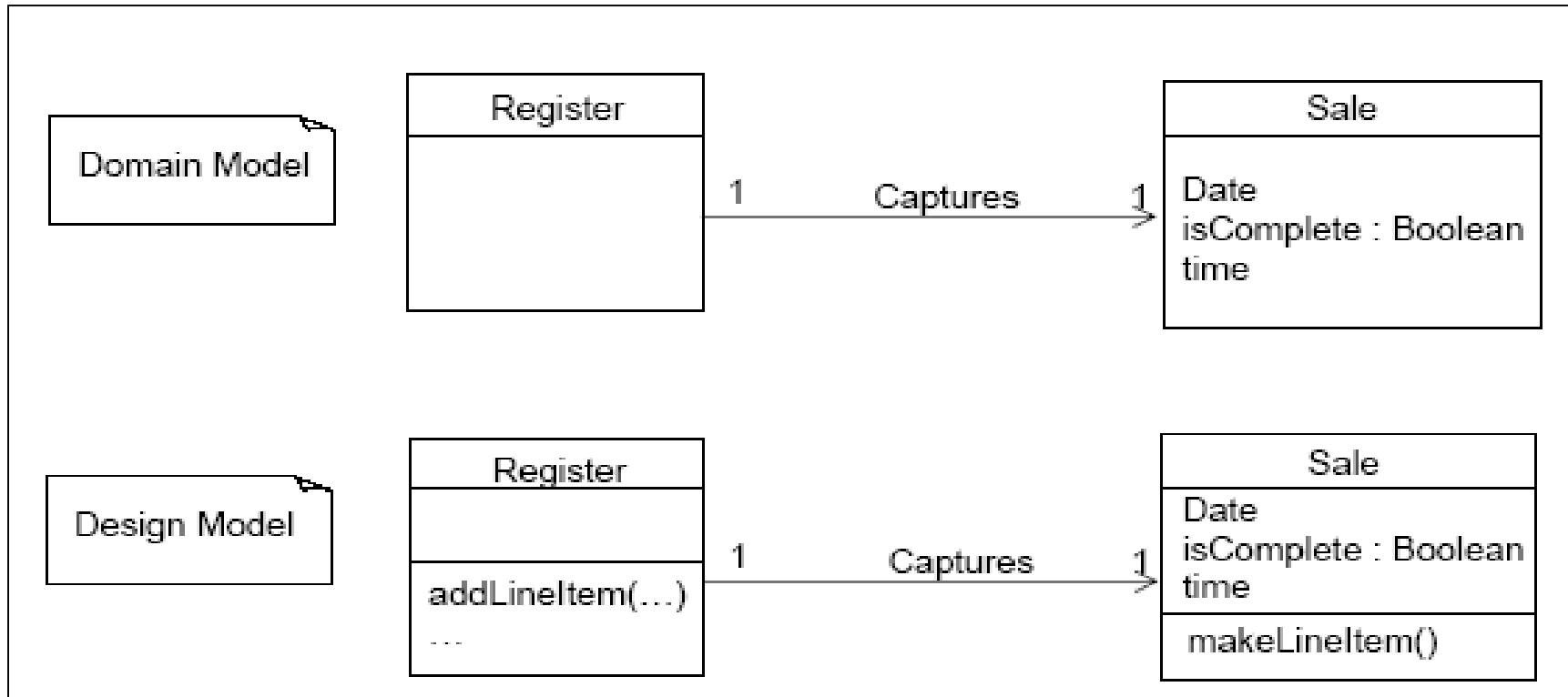
Harvinder S Jabbal



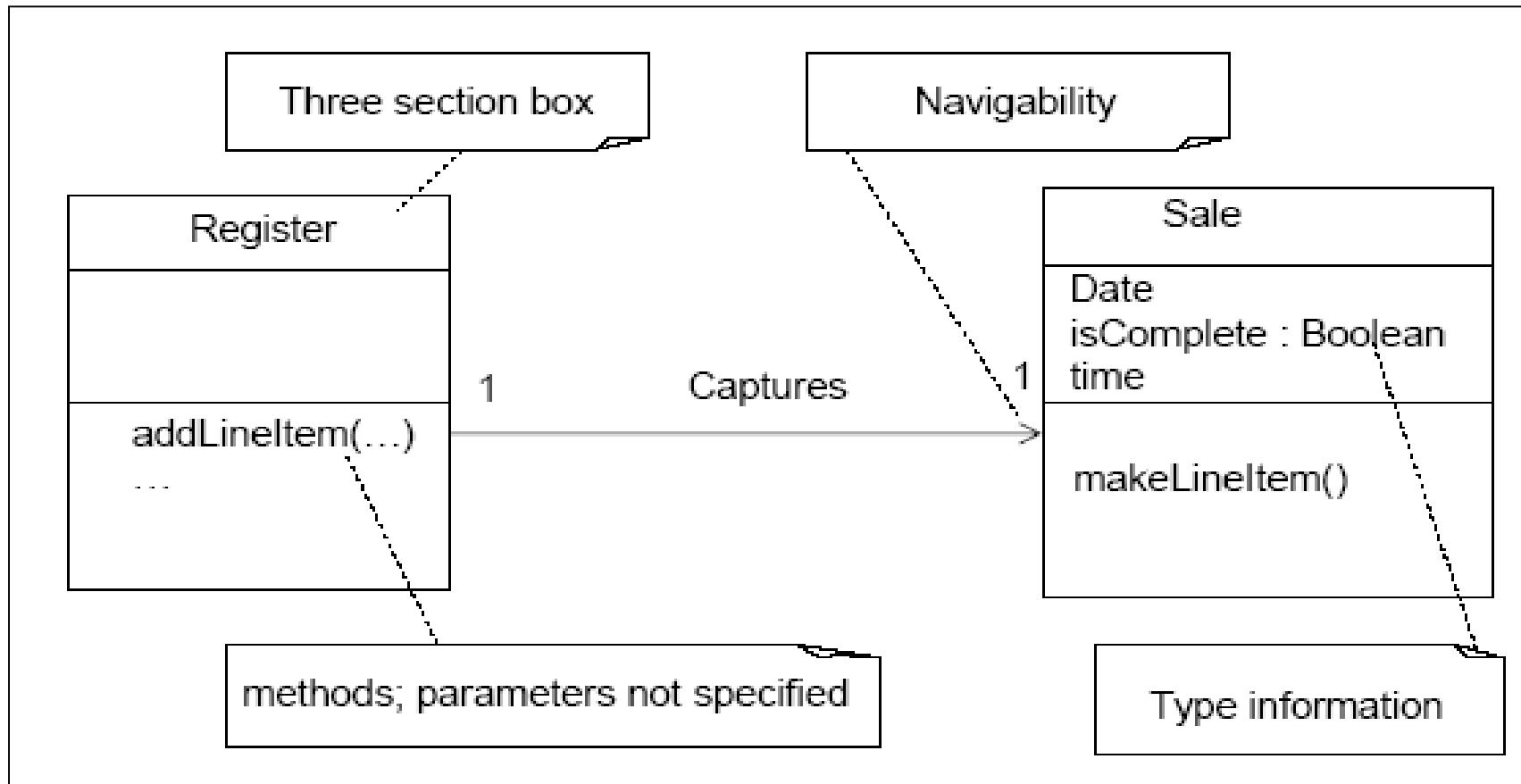
Draw Class Diagram for PoS System

Domain Model vs. Design Model

Classes

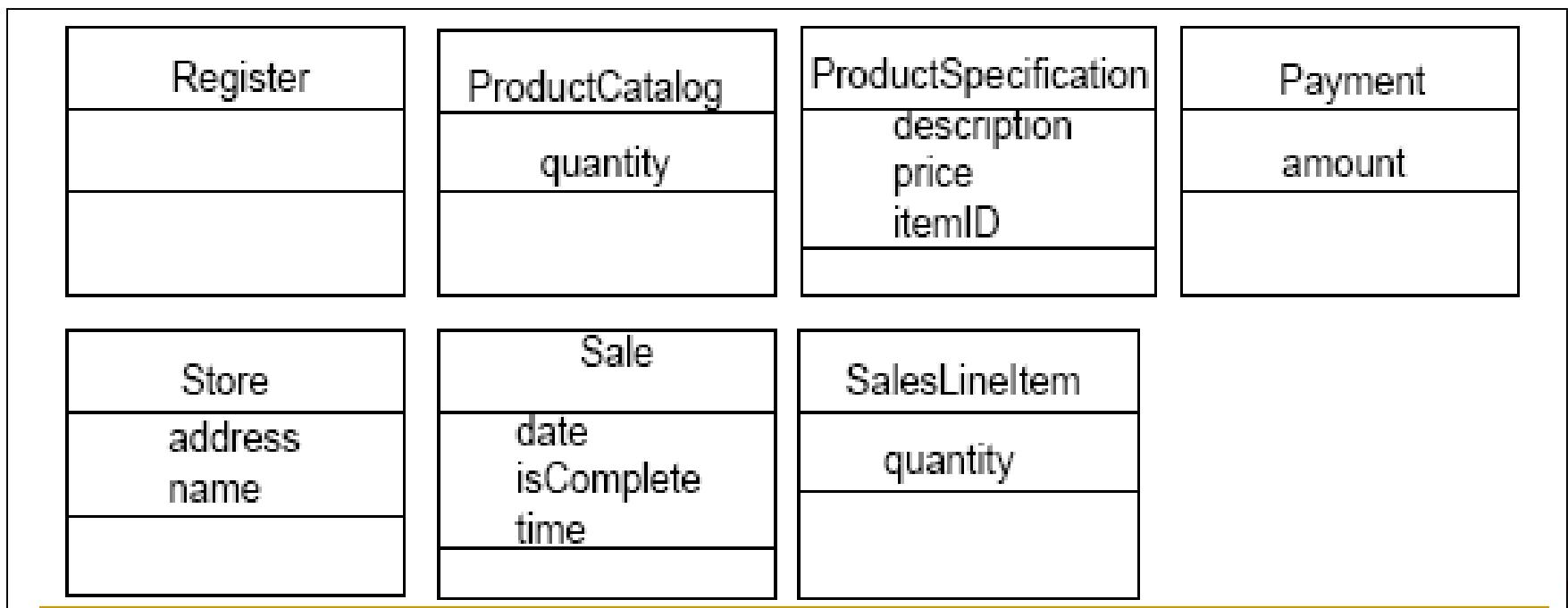


An Example DCD



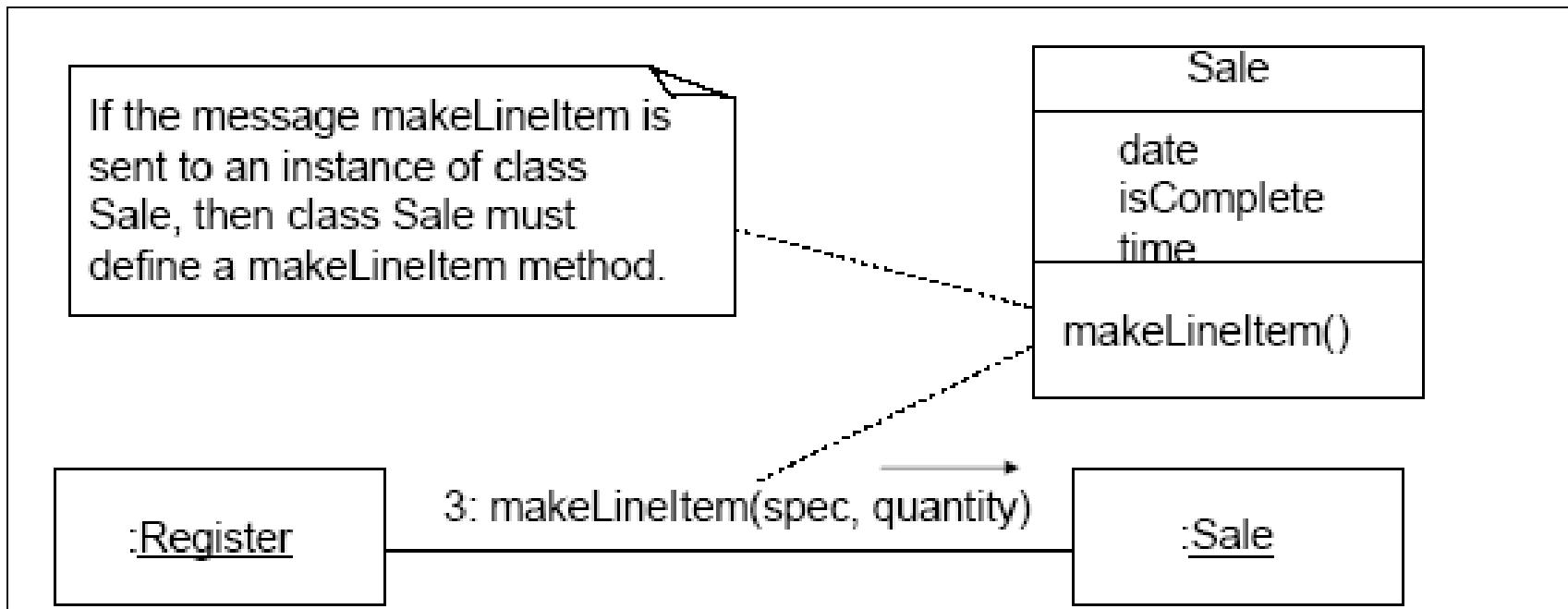
Creating a NextGen POS DCD

- Identify all the classes participating in the software solution. Do this by analyzing the interaction diagrams. Draw them in a class diagram.
- Duplicate the attributes from the associated concepts in the Domain Model.



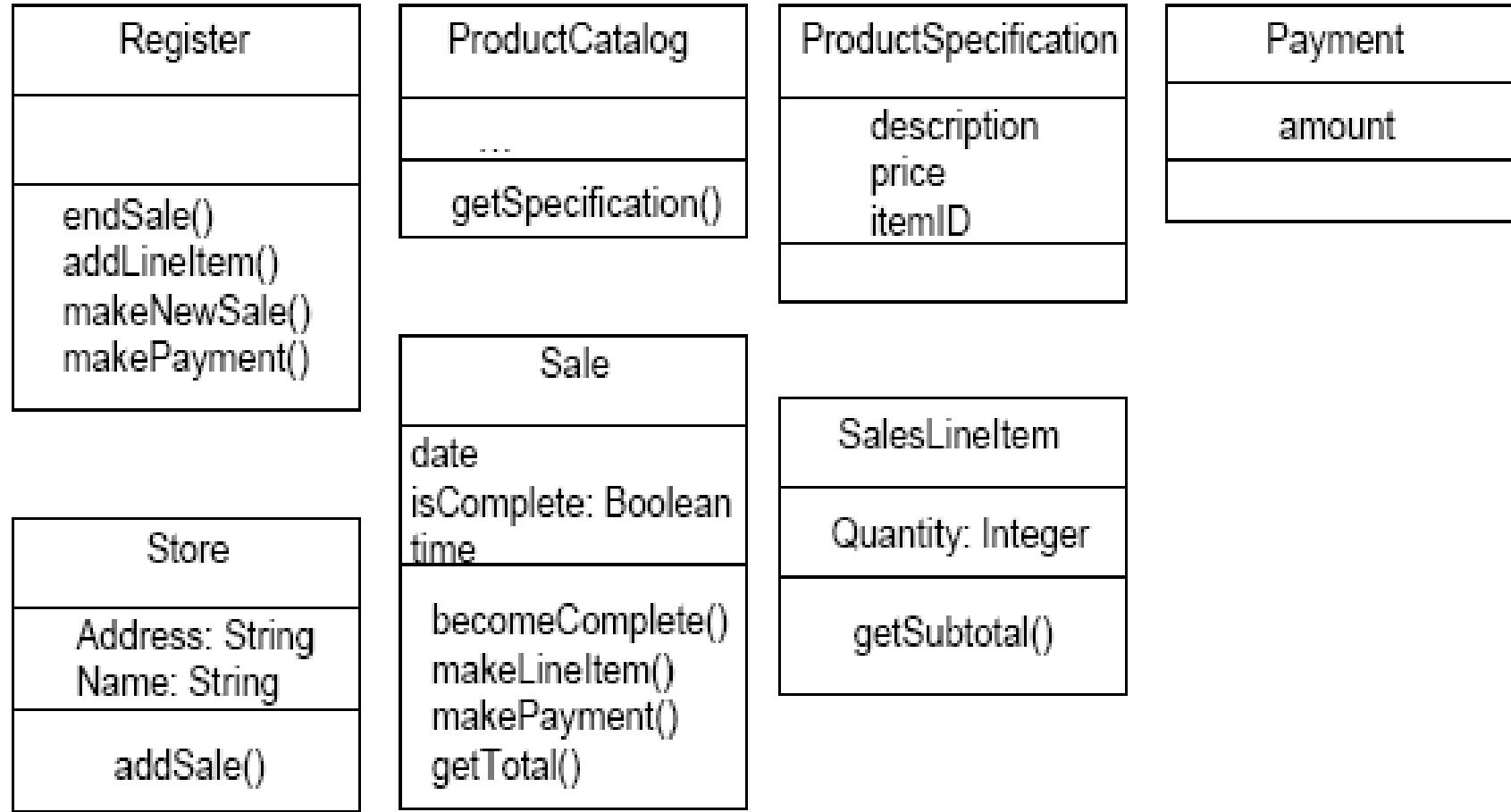
Creating a NextGen POS DCD

- Add method names by analyzing the interaction diagrams.
 - The methods for each class can be identified by analyzing the interaction diagrams.



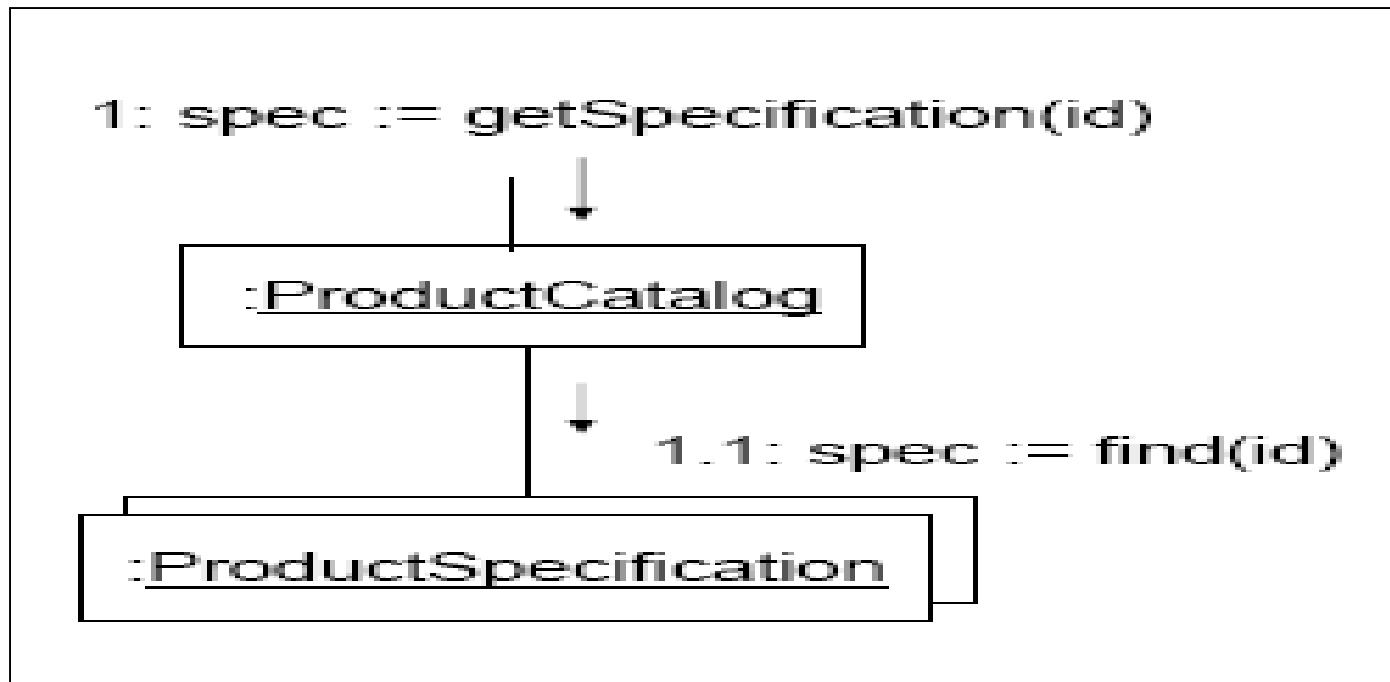
Creating a NextGen POS DCD

- Add type information to the attributes and methods.



Method Names -Multiobjects

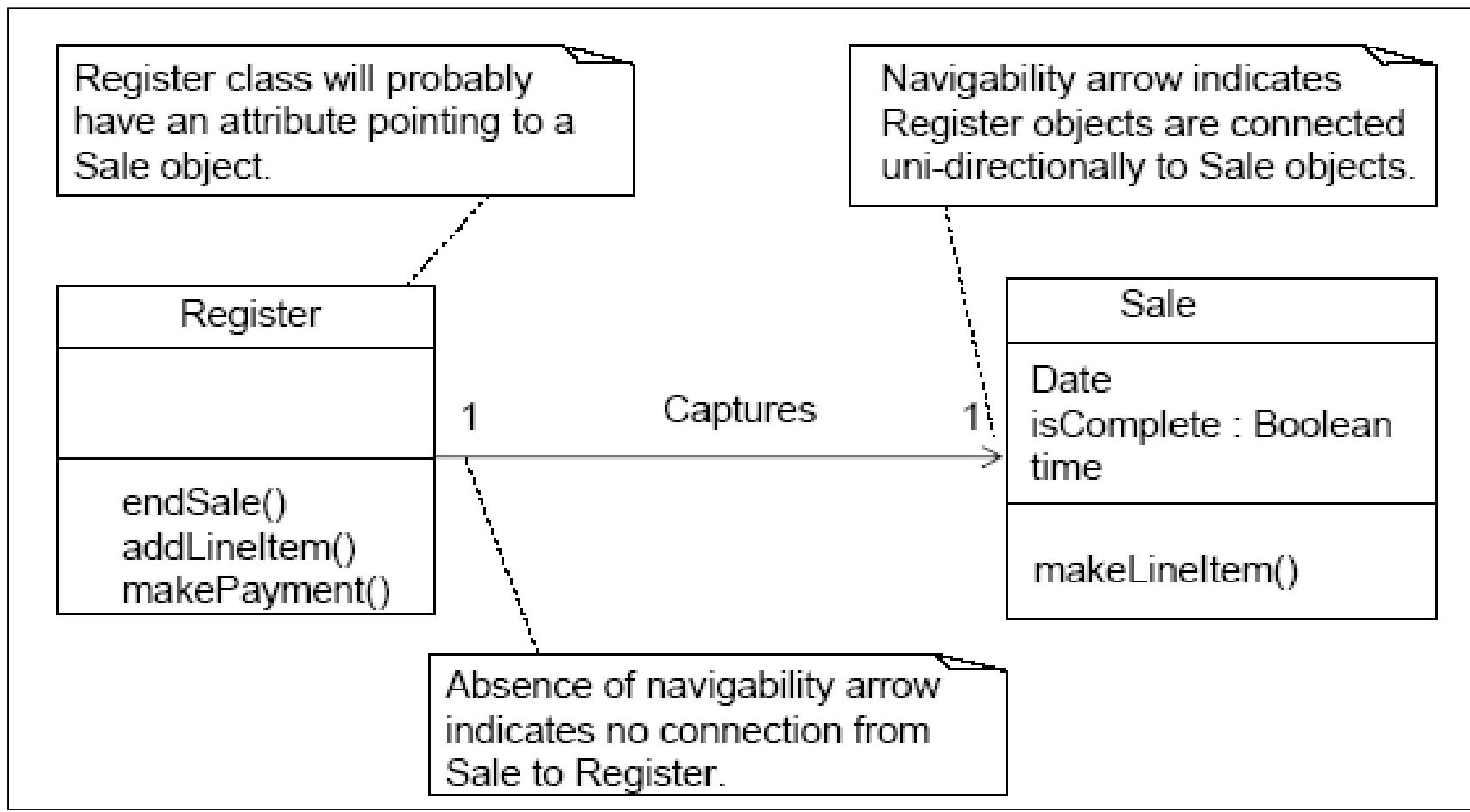
- The find message to the multiobject should be interpreted as a message to the container/ collection object.
- The find method is not part of the ProductSpecification class.



Associations, Navigability, and Dependency Relationships

- Add the associations necessary to support the required attribute visibility.
 - Each end of an association is called a role.
- Navigability is a property of the role implying visibility of the source to target class.
 - Attribute visibility is implied.
 - Add navigability arrows to the associations to indicate the direction of attribute visibility where applicable.
 - Common situations suggesting a need to define an association with navigability from A to B:
 - A sends a message to B.
 - A creates an instance of B.
 - A needs to maintain a connection to B
- Add dependency relationship lines to indicate non-attribute visibility.

Creating a NextGen POS DCD



Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
 - <http://creately.com/diagram-type/article/simple-guidelines-drawing-uml-class-diagrams>



Object Oriented Analysis & Design

Module-5 (RL 5.3.1)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



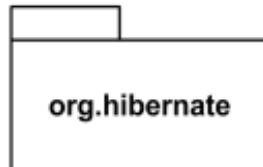
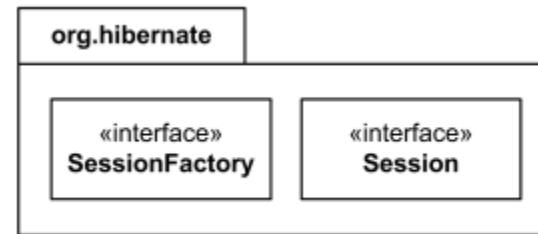
Grouping Classes in Package Diagram

Package

- **Package** is a **namespace** used to group together elements that are semantically related and might change together.
- It is a general purpose mechanism to organize elements into groups to provide better structure for system model.
- **Owned members** of a package should all be **packageable elements**.
- If a package is removed from a model, all the elements **owned** by the package will be removed.
- Package by itself is also a **packageable element**.
- Any package could be also a **member** of other packages.

Package as Namespace

- Because package is a namespace, elements of related or the same type should have unique names within the enclosing package. Different types of elements are allowed to have the same name.
- As a **namespace**, a package can **import** either individual members of other packages or all the members of other packages. Package can also be **merged** with other packages.
- A package is rendered as a tabbed folder - a rectangle with a small tab attached to the left side of the top of the rectangle.
- If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.
- Package org.hibernate*
- The members of the package may be shown within the boundaries of the package. In this case the name of the package should be placed on the tab.
- Package org.hibernate contains SessionFactory and Session*



Create a package diagram to:

- Depict a high-level overview of your requirements (overviewing a collection of UML Use Case diagrams)
- Depict a high-level overview of your architecture/design (overviewing a collection of UML Class diagrams).
- To logically modularize a complex diagram.
- To organize Java source code.

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
- Also: <http://agilemodeling.com/style/packageDiagram.htm>
- <http://www.uml-diagrams.org/package-diagrams.htm>



Object Oriented Analysis & Design

Module-5 (RL 5.3.2)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



Level & Partitions for Package Diagram

Levels for Package Diagram

Example of Subsystem Layers
(ISO Reference Model for OSI)



- In a layered architecture model, classes within each subsystem layer provide services to the layer above it.

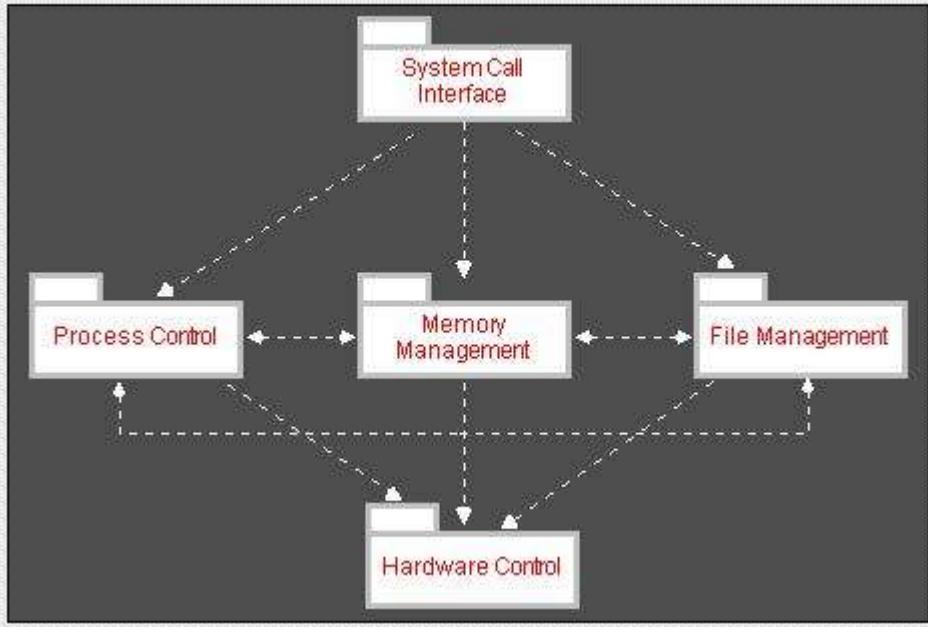
Source: Defining Layers & Partitions in Architecture
Topologies Craig Borysowich Dec 18, 2007 at
toolbox.com

- Ideally, this knowledge is one-way: each layer knows about the layer below it, but the converse is not true.

Partitions for Package Diagram



Example of Subsystem Partitions (OS Kernel Services)



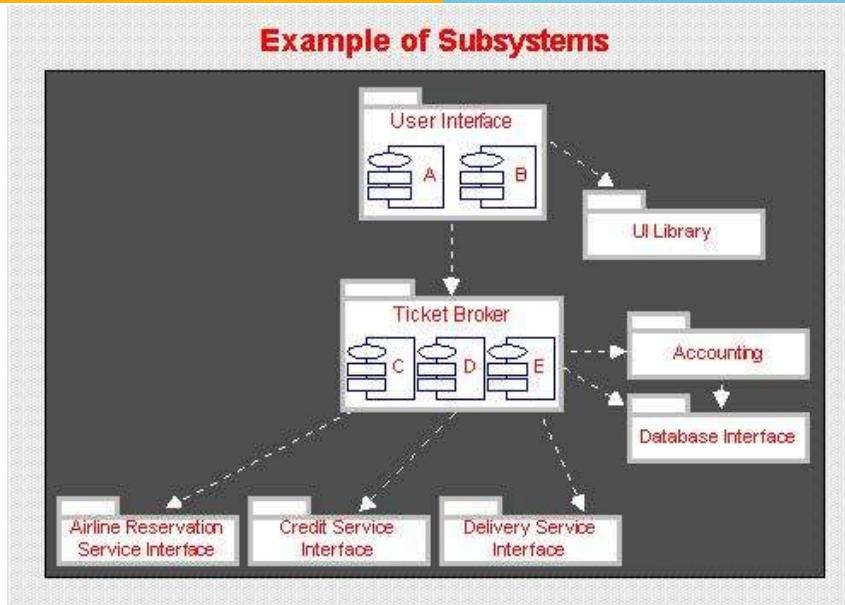
In a partitioned architecture model, subsystems collaborate with peer subsystems at the same level of abstraction.

Source: Defining Layers & Partitions in Architecture
Topologies Craig Borysowich Dec 18, 2007 at toolbox.com

Partitions for Package Diagram

- Classes within each subsystem partition can communicate with other classes in that partition.
- Selected subsystem classes (assuming weak coupling) can communicate with designated classes from subsystem partitions that are at the same level as the subject subsystem.
- Partitioning effects a vertical structuring of peer subsystems.
- Like layers, each subsystem partition represents a group of classes that share similar functionality at the same level of abstraction.

Levels and Partitions for Package Diagram



- Many large system architectures combine layers and partitions.

Source: Defining Layers & Partitions in Architecture
Topologies Craig Borysowich Dec 18, 2007 at toolbox.com

Levels and Partitions for Package Diagram

Example that combines layers and partitions

- Subsystems, or components, are shown by labeled file folder shapes, and interfaces among subsystems are illustrated via arrows.
- Several levels of abstraction are shown:

User Interface subsystem

- At the highest level of abstraction is the *User Interface* subsystem, which provides the human-computer interface for the system.
- The *User Interface* subsystem depends on the *User Interface Library* subsystem, which contains tools for implementing user interfaces on different platforms, to provide a user interface framework.
- In addition, it depends on the *Ticket Broker* subsystem for the business logic for brokering tickets.

Ticket Broker

- The *Ticket Broker* subsystem layer contains two partitions, one for *Ticket Broker* and one for the *Accounting* subsystem. The *Accounting* subsystem organizes the modules that manage customer accounts.
- Both of these depend on the *Database Interface* subsystem that provides persistent object services.
- The bottom layer contains partitions for the *Reservations Interface*, *Delivery Service* and *Credit Authorization Interface* subsystems, which interface with external reservation, delivery, and credit services, respectively.

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
- Also: <http://agilemodeling.com/style/packageDiagram.htm>
- <http://www.uml-diagrams.org/package-diagrams.htm>



Object Oriented Analysis & Design

Module-5 (RL 5.3.3)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

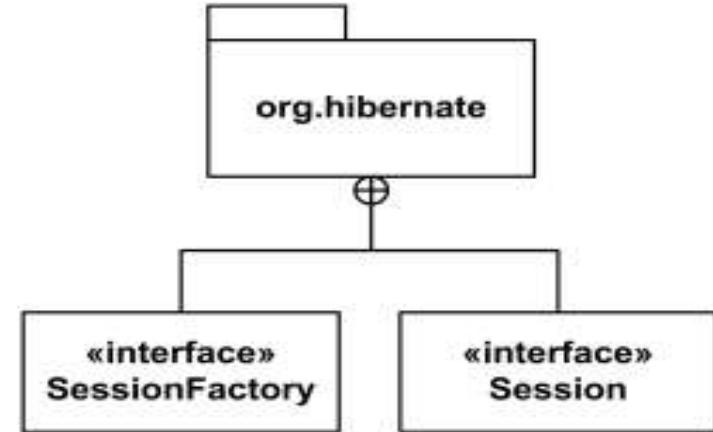
Harvinder S Jabbal



Showing Dependency in Package Diagram

Composition of Packages

- A diagram showing a package with content is allowed to show only a subset of the contained elements according to some criterion.
- Members of the package may be shown **outside** of the package by branching lines from the package to the members.
- A **plus sign (+) within a circle** is drawn at the end attached to the namespace (package).
- This notation for packages is semantically equivalent to **composition** (which is shown using solid diamond.)



Elements of a Package

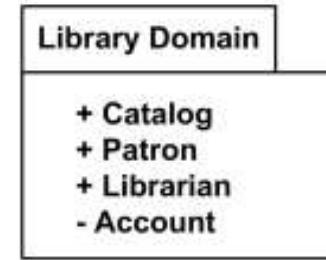
The elements that can be referred to within a package using **non-qualified** names are:

- owned elements,
- imported elements, and
- elements in enclosing (outer) namespaces.

Owned and imported elements may have a visibility that determines whether they are available outside the package.

- If an element that is owned by a package has visibility, it could be only public or private visibility.
- Protected or package visibility is not allowed.
- The visibility of a package element may be indicated by preceding the name of the element by a visibility symbol ("+" for public and "-" for private).
- The public elements of a package are always accessible outside the package through the use of qualified names.

- All elements of Library Domain package are public except for Account



Packageable element

Some examples of **packageable elements** are:

- Type
- **classifier** (--> type)
- **class** (--> classifier)
- **use case** (--> classifier)
- **component** (--> classifier)
- **package**
- **constraint**
- **dependency**
- **event**

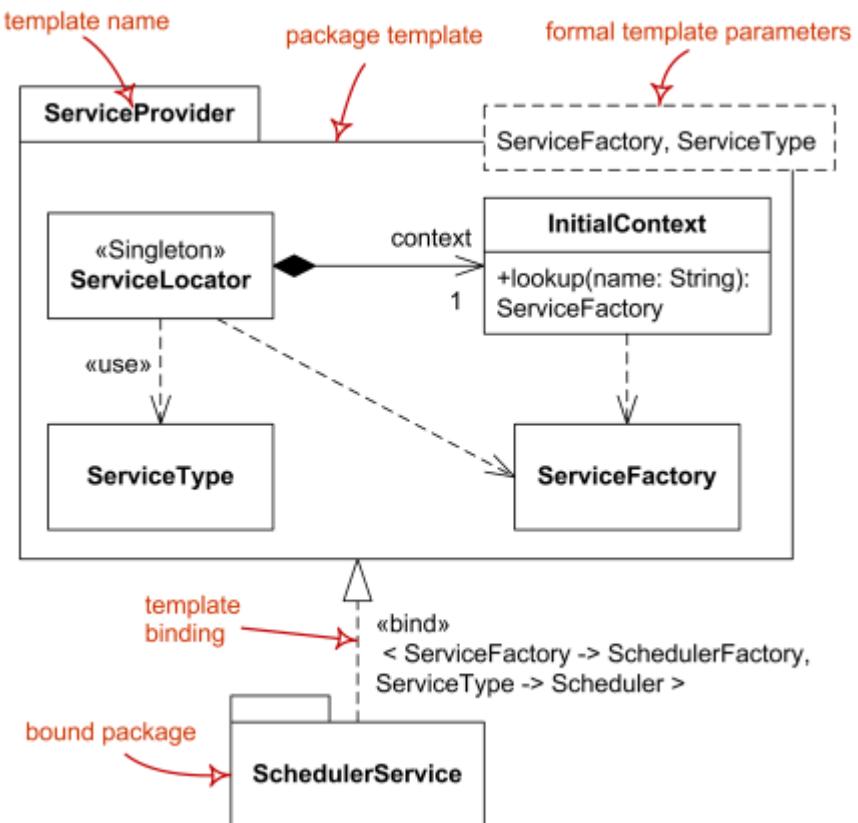
- **Packageable element** is a **named element** that may be **owned** directly by a **package**.

Package Template

Package can be used as a **template** for other packages. These are called **package template** and **template package**.

- **Packageable element** can be used as a **template parameter**.
- A package template parameter may refer to any element owned or used by the package template, or templates nested within it.
- A package may be **bound** to one or more template packages.
- When several bindings are applied the result of bindings is produced by taking the intermediate results and merging them into the combined result using **package merge**.

Package template Service Provider and bound package Scheduler Service.



Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
- Also: <http://agilemodeling.com/style/packageDiagram.htm>
- <http://www.uml-diagrams.org/package-diagrams.htm>



Object Oriented Analysis & Design

Module-5 (RL 5.3.4)



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal



BITS Pilani

Pilani|Dubai|Goa|Hyderabad



Guidelines for Package Diagram

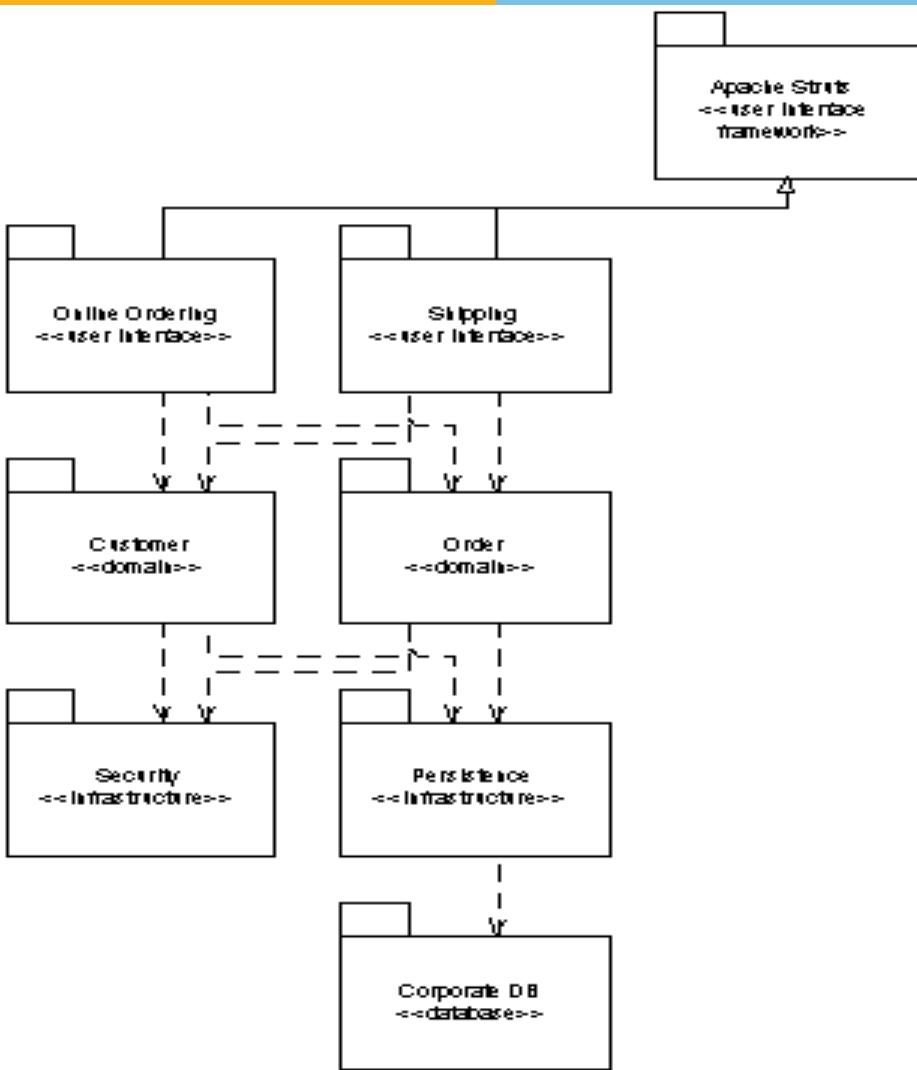
Guidelines for Package Diagrams

- Class Package Diagrams
- Use Case Package Diagrams
- Packages

Class Package Diagrams

- Create UML Component Diagrams to Physically Organize Your Design.
- Place Subpackages Below Parent Packages.
- Vertically Layer Class Package Diagrams.
- Create Class Package Diagrams to Logically Organize Your Design.

Package Diagram



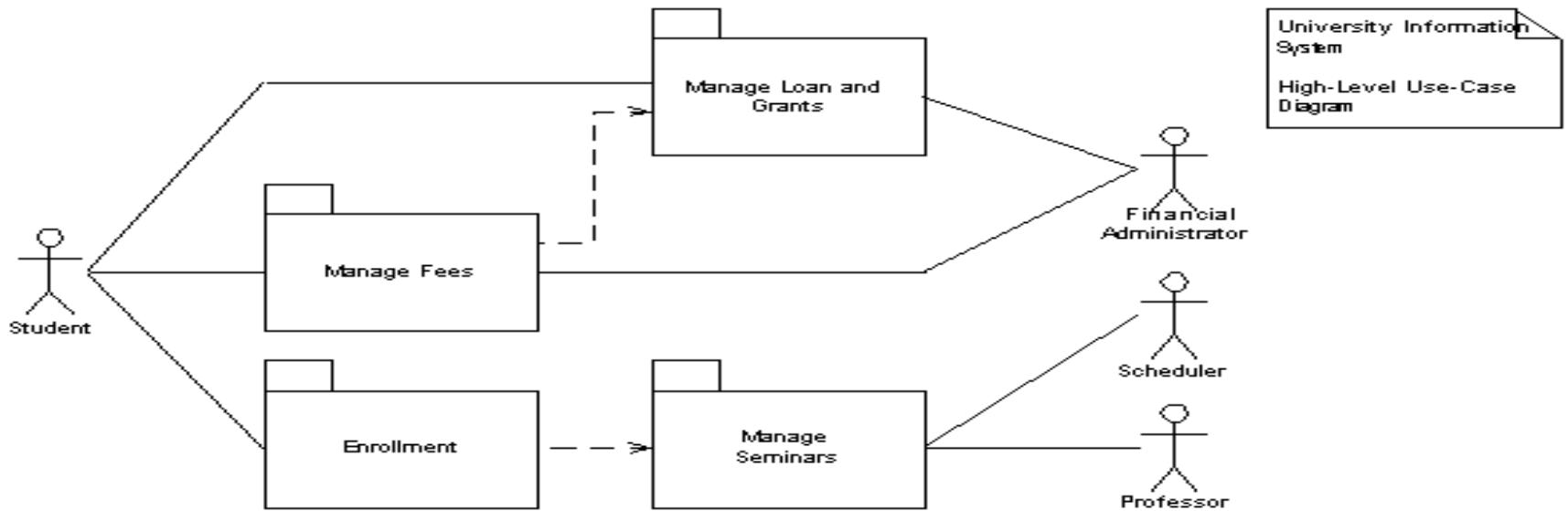
UML Class diagram
organized into
packages.

Guideline

Use the package guidelines heuristics given below to organize UML Class diagrams into package diagrams:

- Place the classes of a framework in the same package.
- Classes in the same inheritance hierarchy typically belong in the same package.
- Classes related to one another via aggregation or composition often belong in the same package.
- Classes that collaborate with each other a lot, information that is reflected by your UML Sequence diagrams and UML Collaboration diagrams, often belong in the same package.

Use Case Package Diagrams



- Create Use Case Package Diagrams to Organize Your Requirements
- Include Actors on Use Case Package Diagrams
- Horizontally Arrange Use Case Package Diagrams

Packages

Guideline applicable to the application of packages on any UML diagram, not just package diagrams. -

- Give Packages Simple, Descriptive Names
- Apply Packages to Simplify Diagrams
- Packages Should be Cohesive
- Indicate Architectural Layers With Stereotypes on Packages
- Avoid Cyclic Dependencies Between Packages
- Dependencies Should Reflect Internal Relationships

Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
- Also: <http://agilemodeling.com/style/packageDiagram.htm>
- <http://www.uml-diagrams.org/package-diagrams.htm>



Object Oriented Analysis & Design

Module-5 (RL 5.3.5)



BITS Pilani

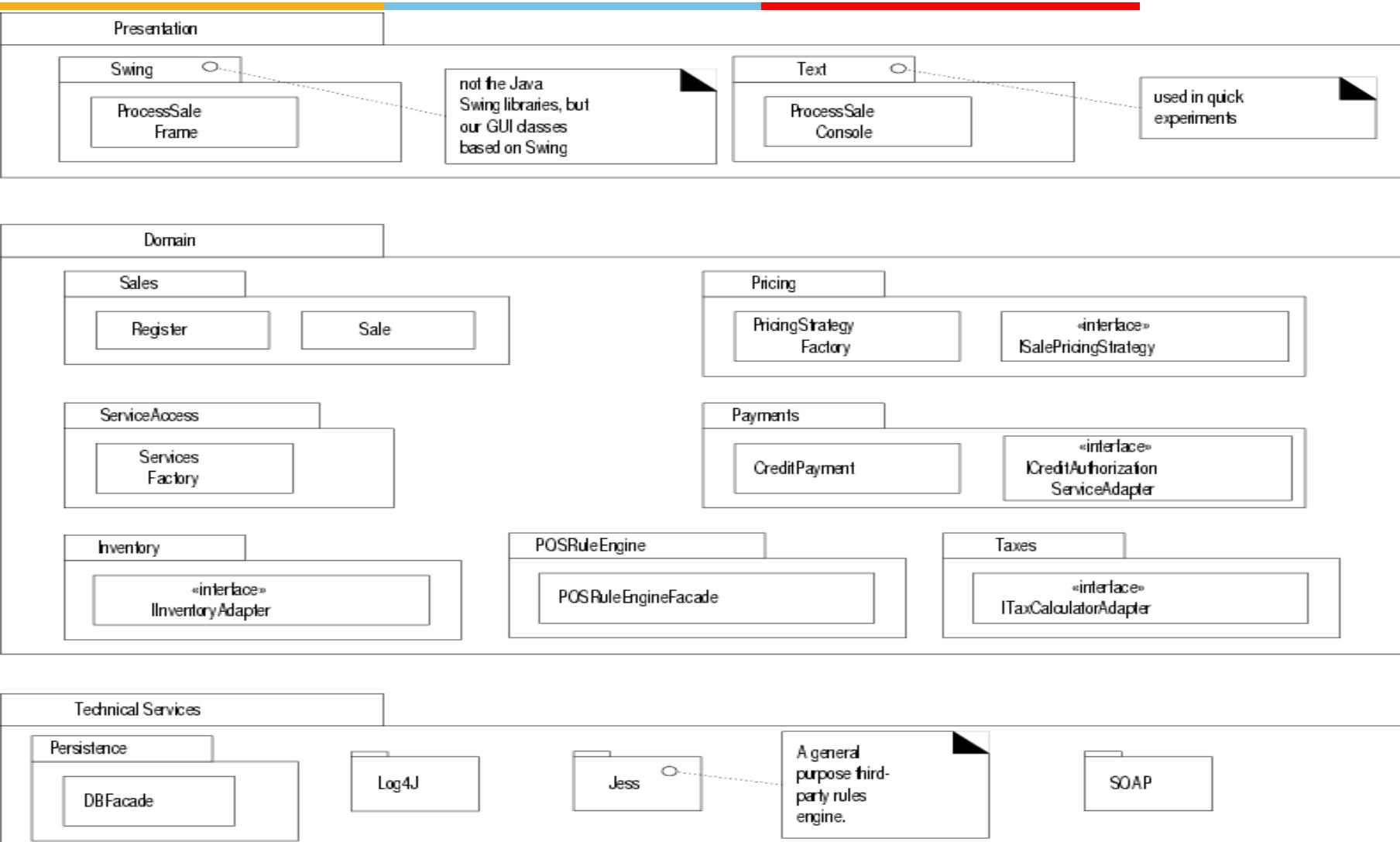
Pilani|Dubai|Goa|Hyderabad

Harvinder S Jabbal

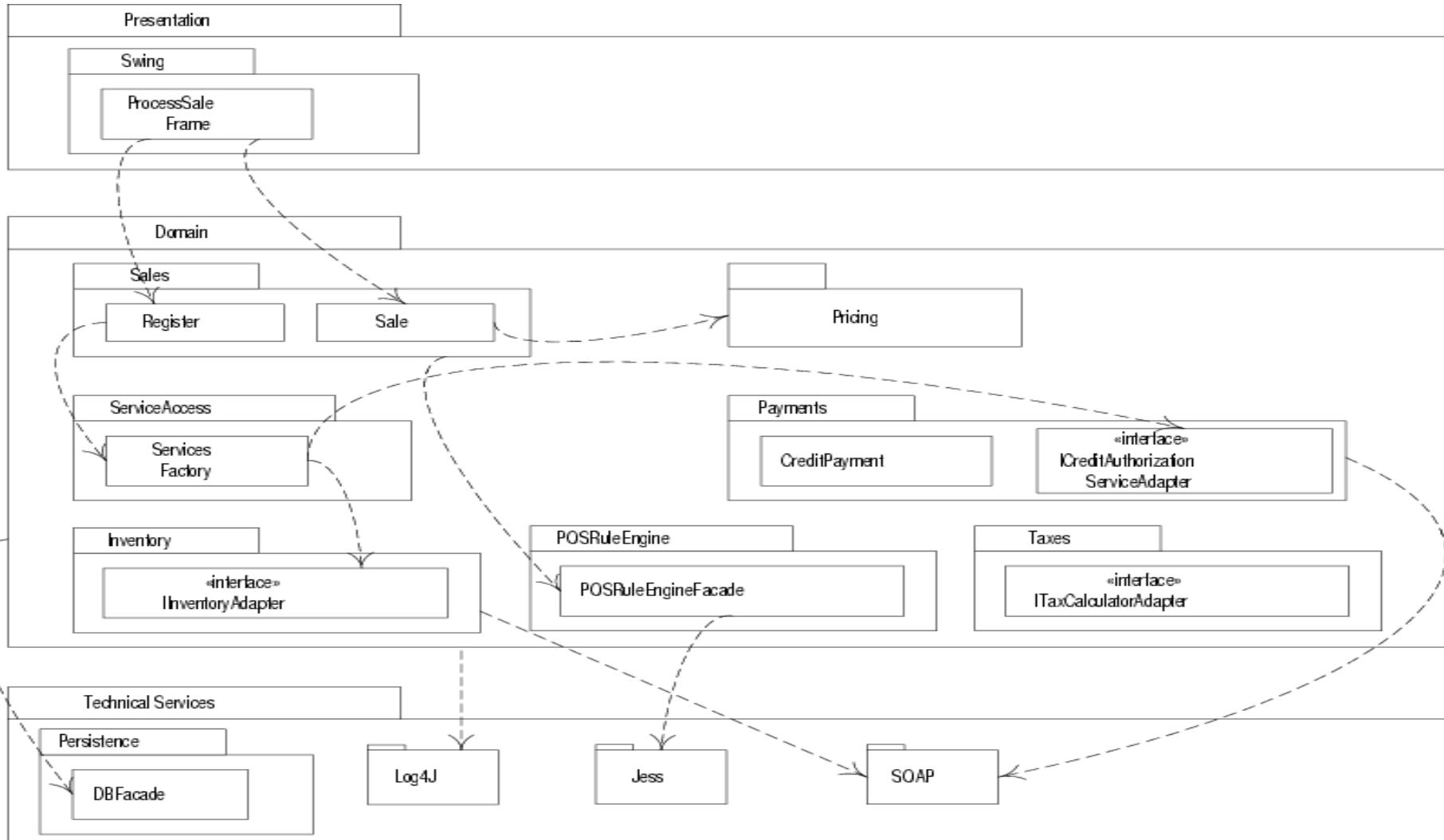


Drawing Package Diagram for PoS System

Partial logical view of layers in the NextGen application.



Partial coupling between packages



Acknowledgement

- Slides are based on Course Text Books:
 - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
 - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
 - Design Patterns (Elements of Reusable Object-Oriented Software) : Erich Gama | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
- Also: <http://agilemodeling.com/style/packageDiagram.htm>
- <http://www.uml-diagrams.org/package-diagrams.htm>