



**BITS** Pilani  
Pilani Campus

# BITS Pilani presentation

Paramananda Barik  
CS&IS Department



# **SSZG514/SEZG512, Object Oriented Analysis and Design Lecture No.1**

# Review of Object-Oriented Programming

---

## Key OOP Concepts

- Object, Class
- Instantiation, Constructors
- Encapsulation
- Inheritance and Subclasses
- Abstraction
- Reuse
- Polymorphism, Dynamic Binding

## Object-Oriented Design and Modeling

# Object

---

Definition: a thing that has identity, state, and behavior

- identity: a distinguished instance of a **class**
- state: collection of values for its **variables**
- behavior: capability to execute **methods**

\* variables and methods are defined in a class

# Class

---

Definition: a collection of data (fields/variables) and methods that operate on that data

- define the contents/capabilities of the instances (objects) of the class
- a class can be viewed as a *factory* for objects
- a class defines a *recipe* for its objects

# Instantiation

---

Object creation

Memory is allocated for the object's fields as defined in the class

Initialization is specified through a ***constructor***

- a special method invoked when objects are created

# Encapsulation

---

A key OO concept: “Information Hiding”

Key points

- The user of an object should have access only to those methods (or data) that are essential
- Unnecessary implementation details should be hidden from the user
- In Java/C++, use classes and access modifiers (public, private, protected)

# Inheritance

---

## Inheritance:

- programming language feature that allows for the implicit definition of variables/methods for a class through an existing class

## Subclass relationship

- B is a subclass of A
- B inherits all definitions (variables/methods) in A

# Abstraction

---

OOP is about ***abstraction***

Encapsulation and Inheritance are examples of abstraction

- What does the verb “abstract” mean?

# Reuse

---

Inheritance encourages software reuse  
Existing code need not be rewritten  
Successful reuse occurs only through careful planning and design  
– when defining classes, anticipate future modifications and extensions

---

# Polymorphism

---

“Many forms”

- allow several definitions under a single method name

Example:

- “move” means something for a person object but means something else for a car object

Dynamic binding:

- capability of an implementation to distinguish between the different forms during run-time

# Building Complex Systems

---

- From Software Engineering:  
complex systems are difficult to manage
- Proper use of OOP aids in managing this complexity
- The analysis and design of OO systems require corresponding modeling techniques

# Object-Oriented Modeling

---

UML: Unified Modeling Language

- OO Modeling Standard
- Booch, Jacobson, Rumbaugh

What is depicted?

- Class details and static relationships
- System functionality
- Object interaction
- State transition within an object

# Some UML Modeling Techniques

---

Class Diagrams

Use Cases/Use Case Diagrams

Interaction Diagrams

State Diagrams



# **SSZG514/SEZG512, Object Oriented Analysis and Design Lecture No.2**

# Difference Between Procedural and OOP

---

- Both Procedural Programming and Object Oriented Programming are high-level languages in programming world and are widely used in the development of applications.
- On the basis of nature of developing the code, both languages have different approaches on basis of which both are differentiate from each other.

# What is Procedural Programming?

---

- Procedural Programming is a programming language that follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions.
- In procedural oriented programming, each step is executed in a systematic manner so that the computer can understand what to do.

# What is Procedural Programming?

---

- The programming model of the procedural oriented programming is derived from structural programming.
- The concept followed in the procedural oriented programming is called the "procedure".
- These procedures consist several computational steps that are carried out during the execution of a program. Examples of procedural oriented programming language include – C, Pascal, ALGOL, COBOL, BASIC, etc.

# What is Object Oriented Programming?

---

- Object-oriented Programming is a programming language that uses classes and objects to create models based on the real world environment.
- These objects contain data in the form of attributes and program codes in the form of methods or functions.
- In OOP, the computer programs are designed by using the concept of objects that can interact with the real world entities.

# What is Object Oriented Programming?

---

- We have several types of object oriented programming languages, but the most popular is one among all is class-based language.
- In the class-based OOP languages, the objects are the instances of the classes that determine their types.
- Examples of some object oriented programming languages are – Java, C++, C#, Python, PHP, Swift, etc.

# Difference

Parameter	Object Oriented Programming	Procedural Programming
Definition	<p>Object-oriented Programming is a programming language that uses classes and objects to create models based on the real world environment.</p> <p>In OOPs, it makes it easy to maintain and modify existing code as new objects are created inheriting characteristics from existing ones.</p>	<p>Procedural Programming is a programming language that follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions.</p> <p>Each step is carried out in order in a systematic manner so that a computer can understand what to do.</p>
Approach	<p>In OOPs concept of objects and classes is introduced and hence the program is divided into small chunks called objects which are instances of classes.</p>	<p>In procedural programming, the main program is divided into small parts based on the functions and is treated as separate program for individual smaller program.</p>

# Difference

Parameter	Object Oriented Programming	Procedural Programming
Access modifiers	In OOPs access modifiers are introduced namely as Private, Public, and Protected.	No such modifiers are introduced in procedural programming.
Security	Due to abstraction in OOPs data hiding is possible and hence it is more secure than POP.	Procedural programming is less secure as compare to OOPs.
Complexity	OOPs due to modularity in its programs is less complex and hence new data objects can be created easily from existing objects making object-oriented programs easy to modify	There is no simple process to add data in procedural programming, at least not without revising the whole program.
Program division	OOP divides a program into small parts and these parts are referred to as objects.	Procedural programming divides a program into small programs and each small program is referred to as a function.

# Difference

Parameter	Object Oriented Programming	Procedural Programming
Importance	OOP gives importance to data rather than functions or procedures.	Procedural programming does not give importance to data. In POP, functions along with sequence of actions are followed.
Inheritance	OOP provides inheritance in three modes i.e. protected, private, and public	Procedural programming does not provide any inheritance.
Examples	C++, C#, Java, Python, etc. are the examples of OOP languages.	C, BASIC, COBOL, Pascal, etc. are the examples POP languages.



**BITS** Pilani  
Pilani Campus

# BITS Pilani presentation

Paramananda Barik  
CS&IS Department



# **SSZG514/SEZG512, Object Oriented Analysis and Design Lecture No.3**

# Unified Process (UP) Model

- A unified process (UP) is a software development process that uses the UML language to represent models of the software system to be developed.
- It is iterative, architecture centric, use case driven and risk confronting.

# Unified Process (UP) Model

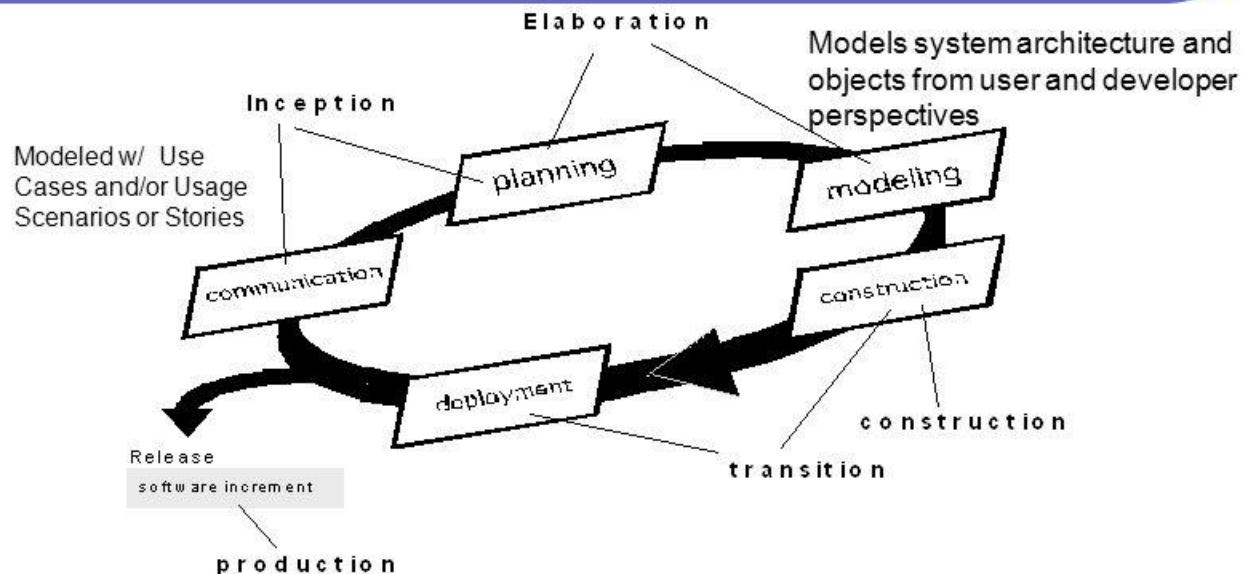
- Unified process (UP) is an architecture-centric, use-case driven, iterative and incremental development process that leverages unified modelling language and is compliant with the system process engineering metamodel.
- Unified process can be applied to different software systems with different levels of technical and managerial complexity across various domains and organizational cultures.
- UP is also referred to as the unified software development process.

# Unified Process (UP) Model

- Unified process is a refinement of rational unified process. It is an extensible framework that can be customized for specific projects.
- This process divides the development process into four phases:
  - Inception
  - Elaboration(milestone)
  - Conception(release)
  - Transition(final production release)

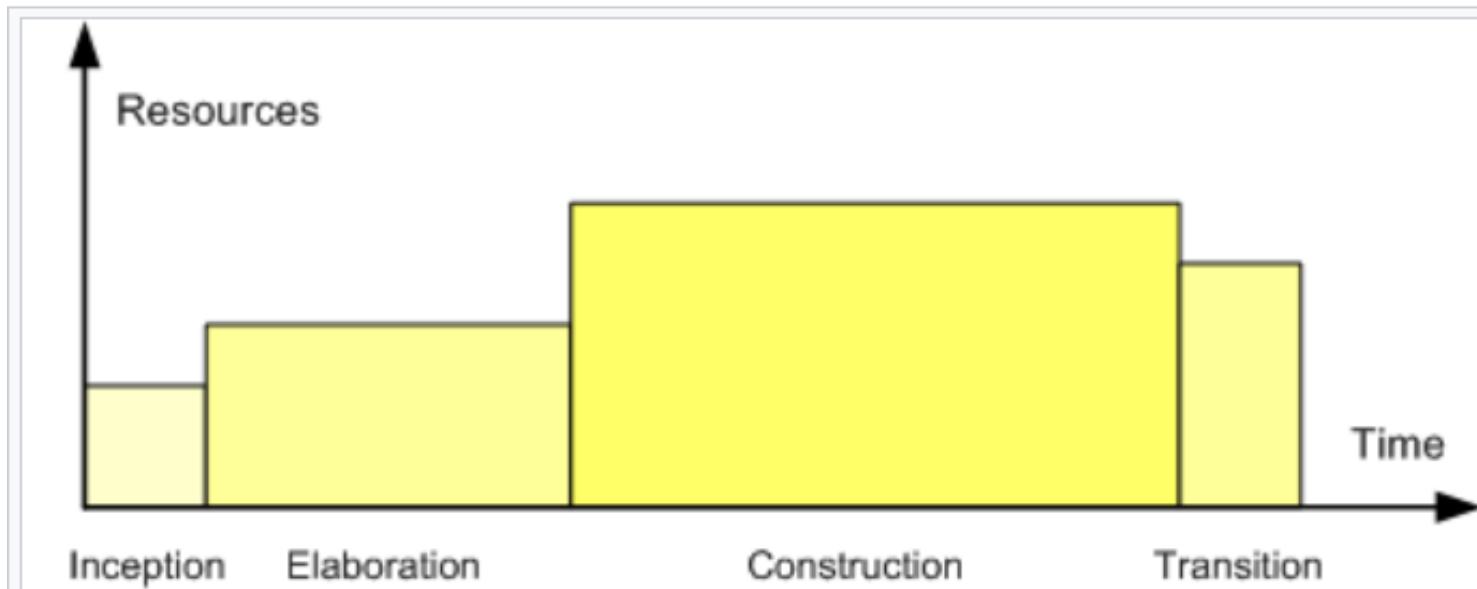
# Unified Process (UP) Model

## Unified Process Model



- A recommended model for CS4810/EGCP4810
- Similar to the incremental model with OO emphasis (cycles are also “pipelined”)
- Provides structure and notation (UML) which assists in organization and communication

# Unified Process (UP) Model



Profile of a typical project showing the relative sizes of the four phases of the Unified Process.

# Unified Process (UP) Model

- Inception Phase
  - ✓ Develop an approximation version of the system.
  - ✓ Make the business Case
  - ✓ Define Scope
  - ✓ Rough Cost and Schedule
- Elaboration Phase
  - ✓ System requirement
  - ✓ Address known risks
  - ✓ Validate system requirement
- Construction Phase
  - ✓ Largest phase, time-boxed iteration, executable release of s/w
- Transition Phase
  - ✓ Deployed to the target system
  - ✓ To work on the feedbacks received in initial phase

# Unified Process (UP) Model

- UP has the following major characteristics:
  - It is use-case driven
  - It is architecture-centric
  - It is risk focused
  - It is iterative and incremental

# Difference UP and Waterfall

UP	Waterfall
The disciplines (Analysis, Design, Coding and Testing ) are done iteratively and concurrently.	The disciplines are generally done sequentially (e.g. Coding only starts once Requirements have been completed and signed-off)
Large business applications	Classical model
provides a language for describing method content and processes	This model is used in governmental projects as well as many major companies associated with software engineering.

# Difference UP and Agile

UP	Waterfall
Unified Process is a detailed and well-defined process.	"Agile" is not a process at all, it is simply a way of saying that one follows the Agile manifesto which, in turn, is just a bunch of values and practices.
UP is more document (artifact) heavy than Agile is.	Less documentation
UP is better suited for larger teams on larger projects where you won't always have face to face communications.	Agile works better on smaller teams and smaller projects. But those are just generalizations, not hard and fast rules.

# Difference UP

UP	Waterfall
Unified Process is a detailed and well-defined process.	"Agile" is not a process at all, it is simply a way of saying that one follows the Agile manifesto which, in turn, is just a bunch of values and practices.
UP is more document (artifact) heavy than Agile is.	Less documentation
UP is better suited for larger teams on larger projects where you won't always have face to face communications.	Agile works better on smaller teams and smaller projects. But those are just generalizations, not hard and fast rules.

# AGILE MODEL

- The meaning of Agile is swift or versatile.
- "Agile process model" refers to a software development approach based on iterative development.
- Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning.
- The project scope and requirements are laid down at the beginning of the development process.
- Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.

# AGILE MODEL

- Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks.
- The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements.
- Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.

# AGILE MODEL



*Fig. Agile Model*

# PHASES OF AGILE MODEL

- **Phases of Agile Model:**

- Requirements gathering
- Design the requirements
- Construction/ iteration
- Testing/ Quality assurance
- Deployment
- Feedback

# AGILE PRINCIPLES

- #1 Satisfy Customers Through Early & Continuous Delivery
- #2 Welcome Changing Requirements Even Late in the Project
- #3 Deliver Value Frequently
- #4 Break the Silos of Your Project
- #5 Build Projects Around Motivated Individuals
- #6 The Most Effective Way of Communication is Face-to-face
- #7 Working Software is the Primary Measure of Progress



# AGILE PRINCIPLES

- #8 Maintain a Sustainable Working Pace
- #9 Continuous Excellence Enhances Agility
- #10 Simplicity is Essential
- #11 Self-organizing Teams Generate Most Value
- #12 Regularly Reflect and Adjust Your Way of Work to Boost Effectiveness



# AGILE MENIFESTO

- The Agile Manifesto is a document that sets out the key values and principles behind the Agile philosophy and serves to help development teams work more efficiently and sustainably.



# AGILE MENIFESTO

## ■ Key Values and Principles

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

# AGILE TESTING METHODS

- Agile Testing Methods / Approaches:
  - Scrum
  - Crystal
  - Dynamic Software Development Method(DSDM)
  - Feature Driven Development(FDD)
  - Lean Software Development
  - eXtreme Programming(XP)

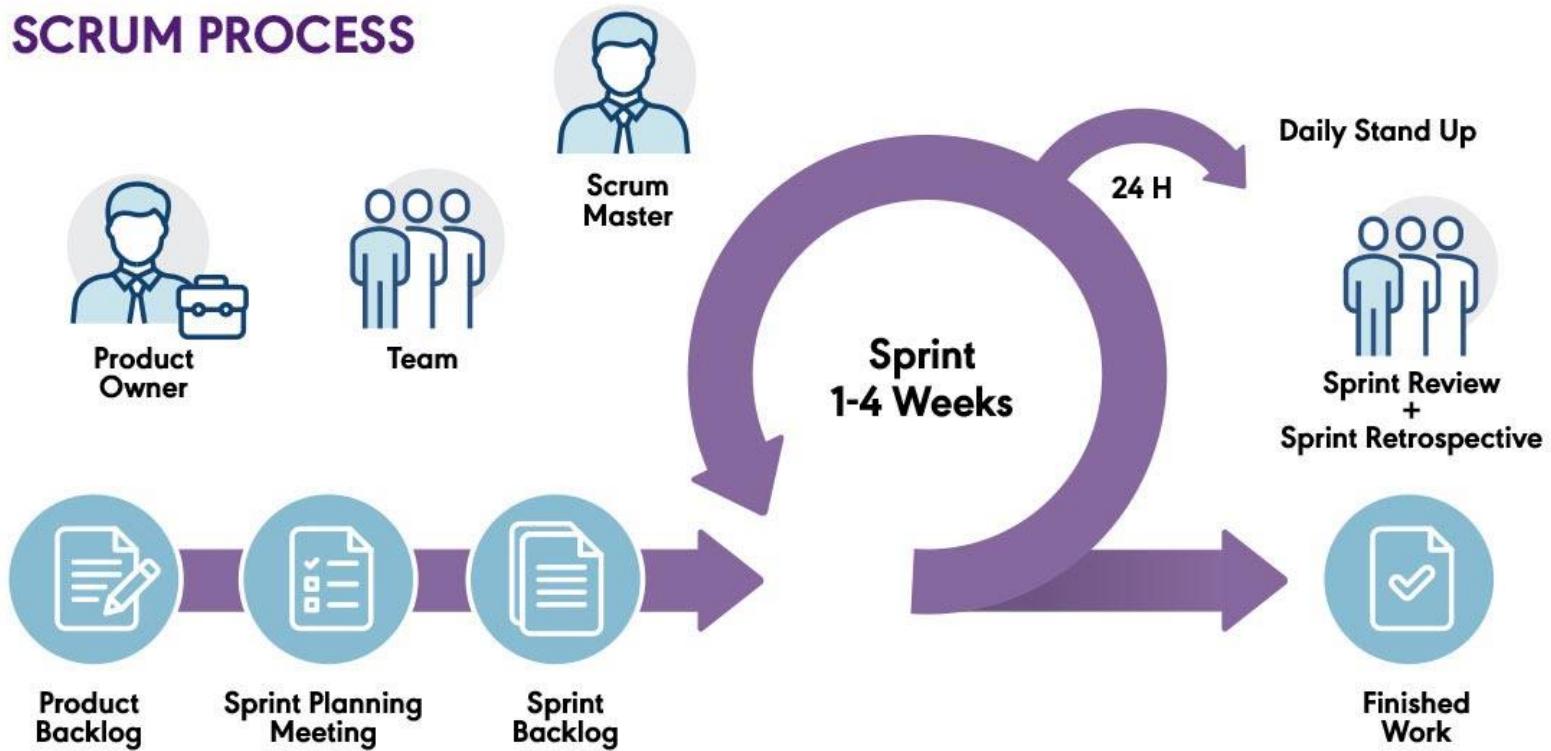
# SCRUM

- SCRUM is an agile development process focused primarily on ways to manage tasks in team-based development conditions.
- There are three roles in it, and their responsibilities are:
  - **Scrum Master:** The scrum can set up the master team, arrange the meeting and remove obstacles for the process
  - **Product owner:** The product owner makes the product backlog, prioritizes the delay and is responsible for the distribution of functionality on each repetition.
  - **Scrum Team:** The team manages its work and organizes the work to complete the sprint or cycle.

# SCRUM



## SCRUM PROCESS



# eXtreme Programming(XP)

---

- Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team.
  - XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development.
  - This type of methodology is used when customers are constantly changing demands or requirements, or when they are not sure about the system's performance.
-

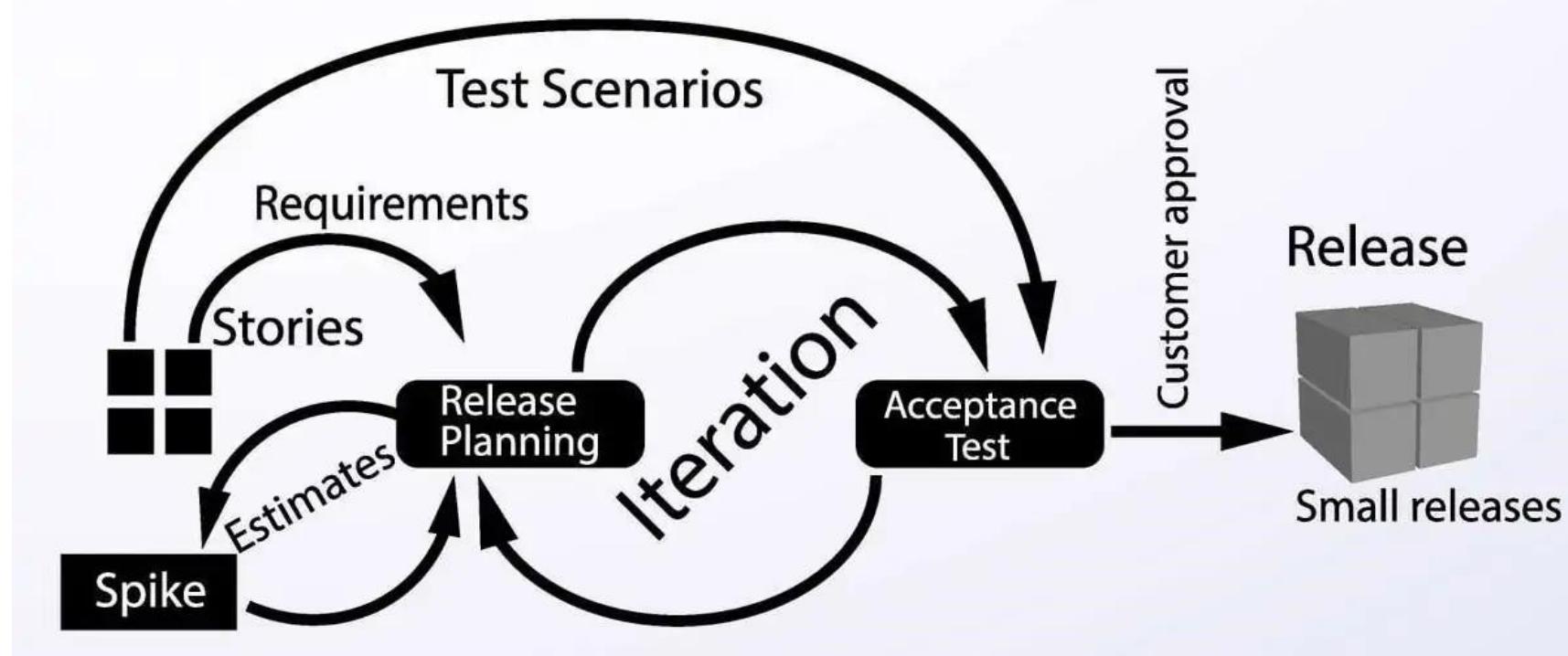
# eXtreme Programming(XP)

---

- When to use extreme programming?
  - Expect their system's functionality to change every few months.
  - Experience constantly changing requirements or work with customers who aren't sure what they want the system to do.
  - Want to mitigate project risk, especially around tight deadlines.
- Example
  - A system might have small releases every three weeks. When many little steps are made, the customer has more control over the development process and the system that is being developed.

# eXtreme Programming(XP)

## Extreme Programming

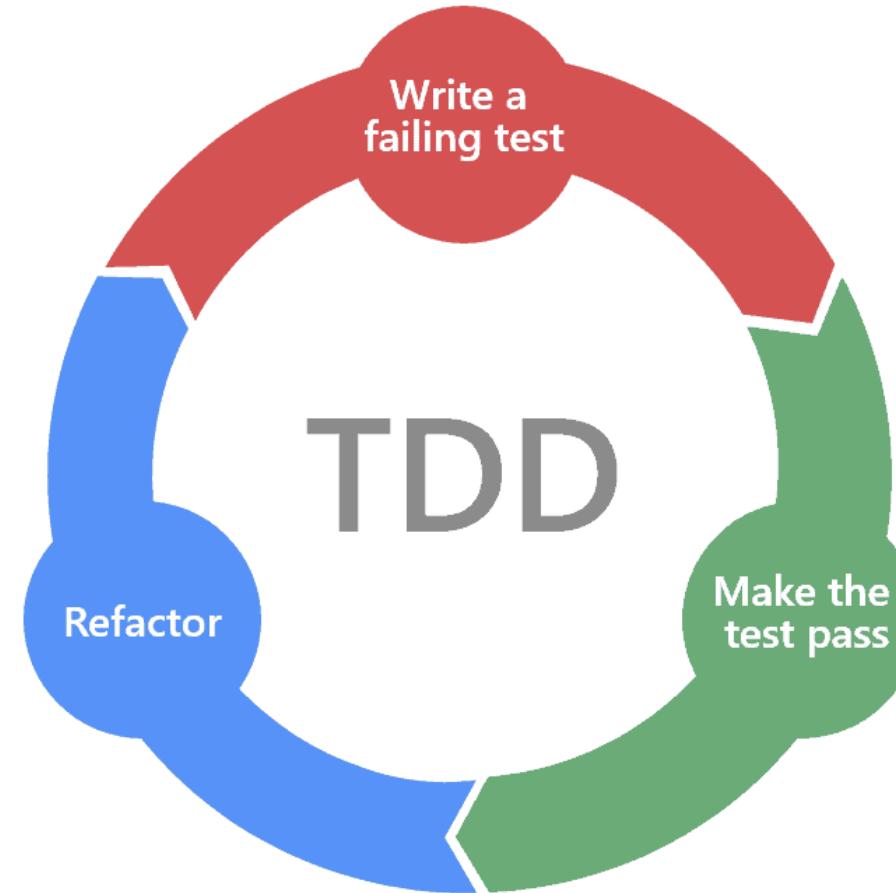


# TDD- Test Driven Development

---

- Test-driven development (TDD), also called test-driven design, is a method of implementing software programming that interlaces unit testing, programming and refactoring on source code.
- There are 5 steps in the TDD flow:
  - Read, understand, and process the feature or bug request.
  - Translate the requirement by writing a unit test. ...
  - Write and implement the code that fulfills the requirement.
  - ...
  - Clean up your code by refactoring.
  - Rinse, lather and repeat.

# TDD- Test Driven Development



# Refactoring

---

- Refactoring is the process of restructuring code, while not changing its original functionality.
  - The goal of refactoring is to improve internal code by making many small changes without altering the code's external behaviour.
  - Refactoring is the process of changing a software system in such a way that it does not alter the function of the code yet improves its internal structure.
-

# Refactoring

---

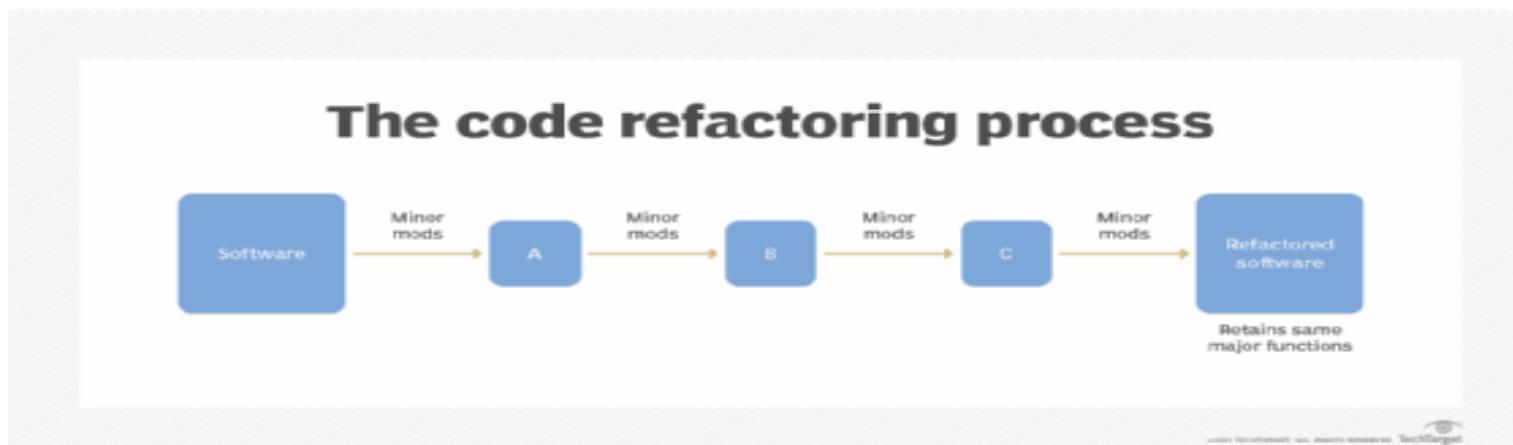
- Refactoring consists of changing the internal structure of the code in a way that doesn't modify its behaviour.
  - This makes the code more maintainable and easier to understand.
  - It enables the developers in the team to keep complexity under control.
-

# Refactoring

---

- Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behaviour.
- The noun “refactoring” refers to one particular behaviour-preserving transformation, such as “Extract Method” or “Introduce Parameter.”

# Refactoring

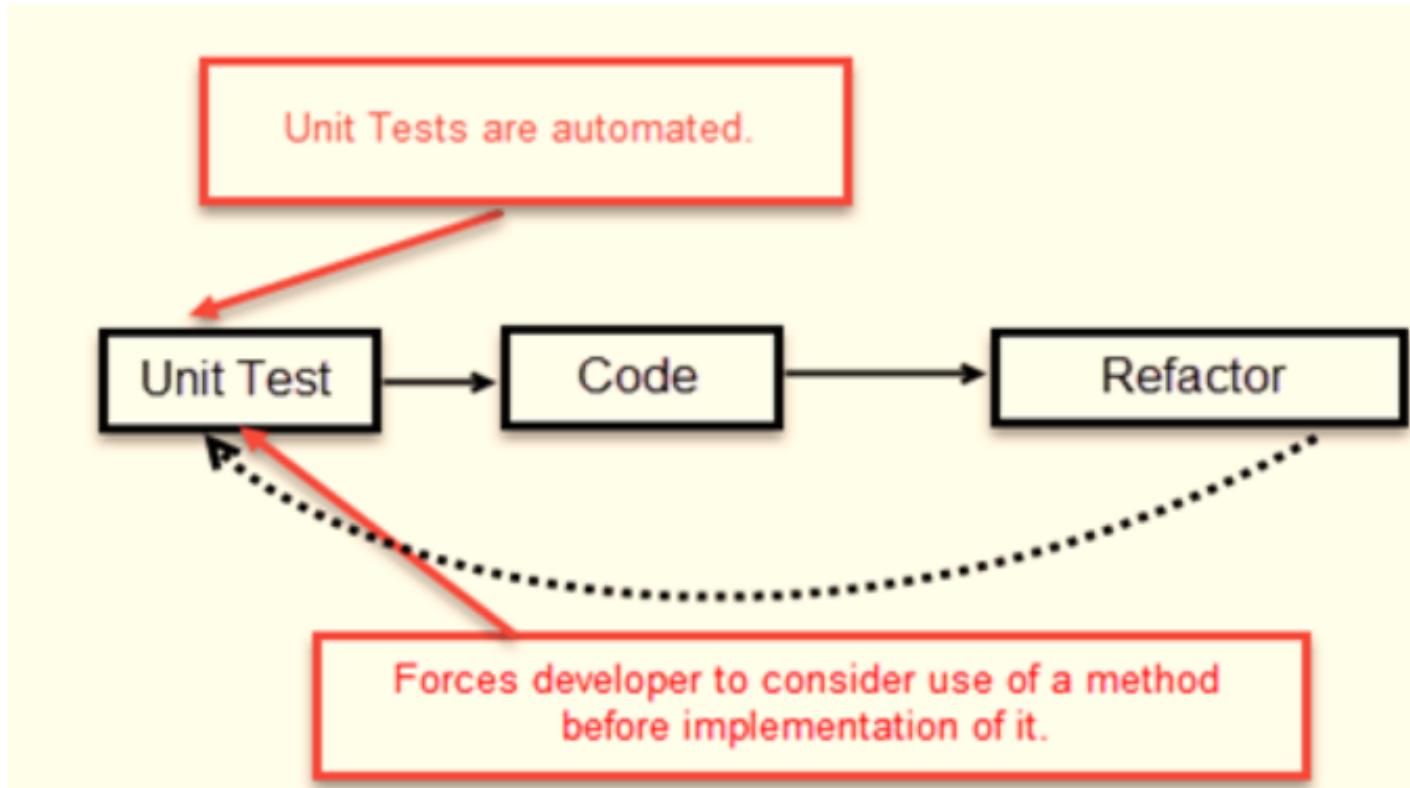


# Example TDD

---

- Test Driven Development (TDD) is software development approach in which test cases are developed to specify and validate what the code will do.
  - In simple terms, test cases for each functionality are created and tested first and if the test fails then the new code is written in order to pass the test and making code simple and bug-free.
  - Test-Driven Development starts with designing and developing tests for every small functionality of an application.
  - TDD framework instructs developers to write new code only if an automated test has failed. This avoids duplication of code. The TDD full form is Test-driven development.
-

# Example TDD



# Example TDD

---

- The simple concept of TDD is to write and correct the failed tests before writing new code (before development). This helps to avoid duplication of code as we write a small amount of code at a time in order to pass tests. (Tests are nothing but requirement conditions that we need to test to fulfil them).
  - Test-Driven development is a process of developing and running automated test before actual development of the application. Hence, TDD sometimes also called as Test First Development.
-



# **SSZG514/SEZG512, Object Oriented Analysis and Design Lecture No.2**

# Software Development Life Cycle (SDLC)

# Capability Maturity Model (CMM)

- A bench-mark for measuring the maturity of an organization's software process
- CMM defines 5 levels of process maturity based on certain Key Process Areas (KPA)

# CMM Levels

## **Level 5 – Optimizing (< 1%)**

- process change management
- technology change management
- defect prevention

## **Level 4 – Managed (< 5%)**

- software quality management
- quantitative process management

## **Level 3 – Defined (< 10%)**

- peer reviews
- intergroup coordination
- software product engineering
- integrated software management
- training program
- organization process definition
- organization process focus

## **Level 2 – Repeatable (~ 15%)**

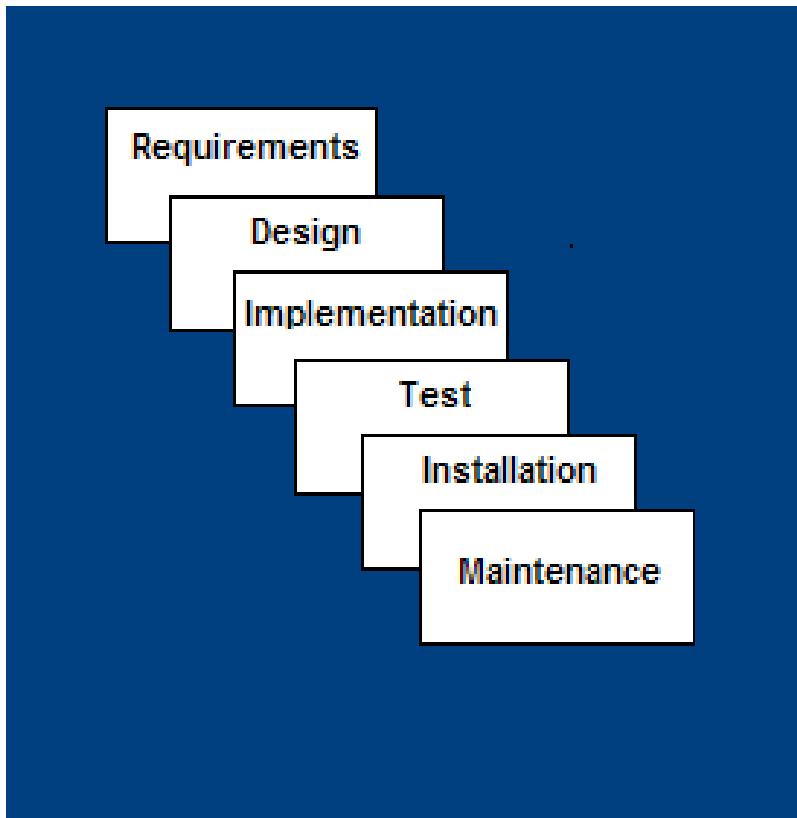
- software configuration management
- software quality assurance
- software project tracking and oversight
- software project planning
- requirements management

## **Level 1 – Initial (~ 70%)**

# SDLC Model

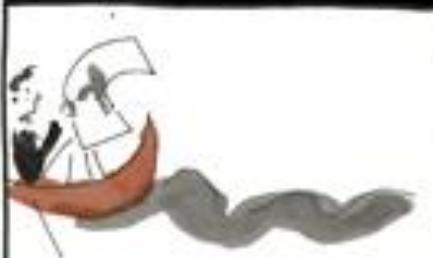
A framework that describes the activities performed at each stage of a software development project.

# Waterfall Model



- **Requirements** – defines needed information, function, behavior, performance and interfaces.
- **Design** – data structures, software architecture, interface representations, algorithmic details.
- **Implementation** – source code, database, user documentation, testing.

## THE NEW PRODUCT WATERFALL

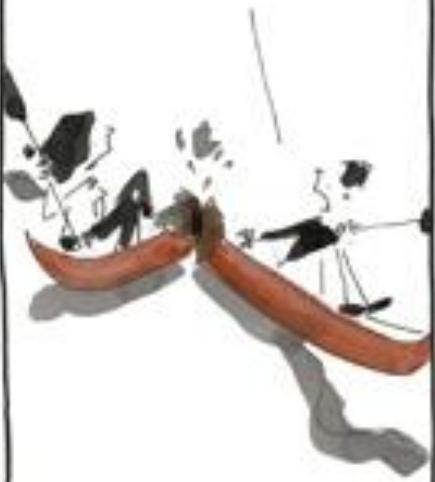


HOW DO WE  
CHART OUR  
ENTIRE COURSE  
IF WE DON'T  
KNOW WHAT'S  
AHEAD?



WHATEVER  
HAPPENS, JUST  
KEEP PADDLING!

I WISH WE'D  
DESIGNED FOR  
THIS SCENARIO  
UPFRONT



TEST

PATCH IT AS  
BEST WE CAN.  
NO TIME TO  
CHANGE COURSE  
NOW



LAUNCH

# Waterfall Strengths

- Easy to understand, easy to use
- Provides structure to inexperienced staff
- Milestones are well understood
- Sets requirements stability
- Good for management control (plan, staff, track)
- Works well when quality is more important than cost or schedule

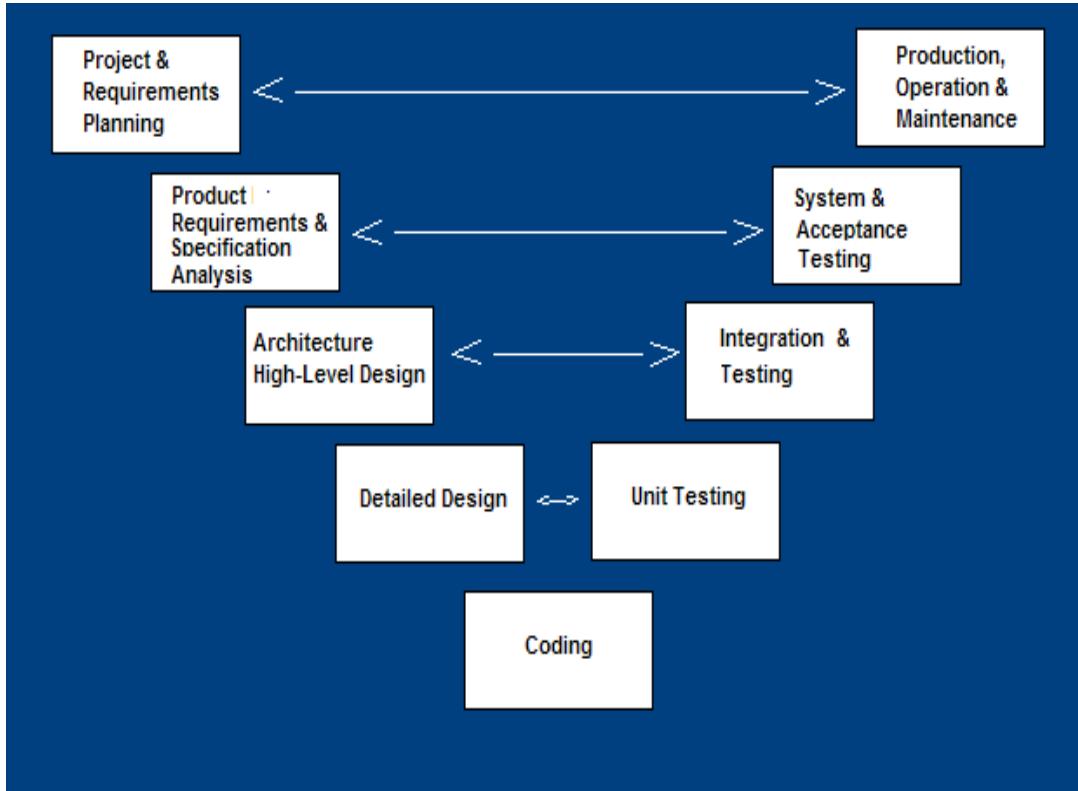
# Waterfall Deficiencies

- All requirements must be known upfront
- Deliverables created for each phase are considered frozen – inhibits flexibility
- Can give a false impression of progress
- Does not reflect problem-solving nature of software development – iterations of phases
- Integration is one big bang at the end
- Little opportunity for customer to preview the system (until it may be too late)

# When to use the Waterfall Model

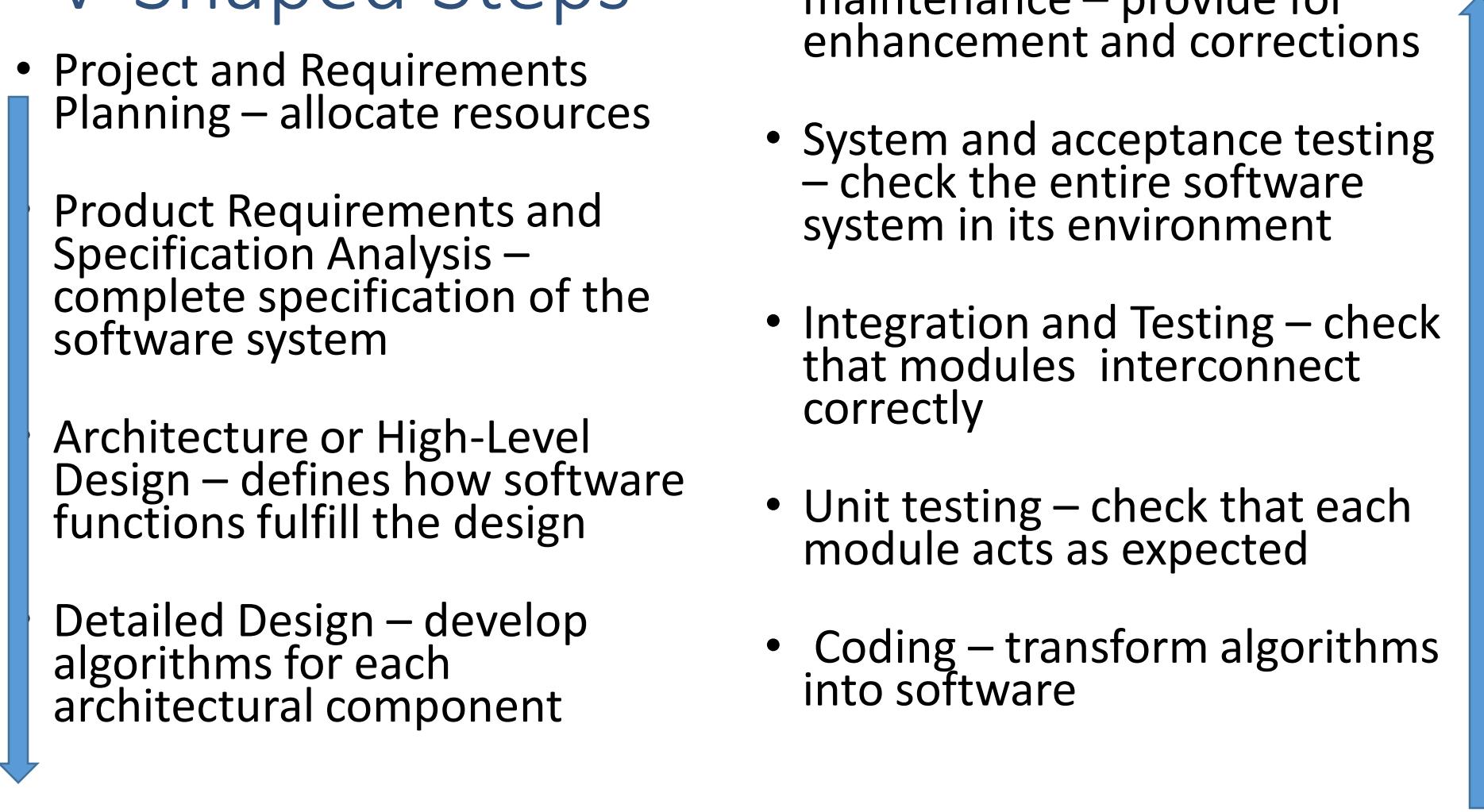
- Requirements are very well known
  - Product definition is stable
  - Technology is understood
  - New version of an existing product
  - Porting an existing product to a new platform.
- 
- High risk for new systems because of specification and design problems.
  - Low risk for well-understood developments using familiar technology.

# V-Shaped SDLC Model



- A variant of the Waterfall that emphasizes the verification and validation of the product.
- Testing of the product is planned in parallel with a corresponding phase of development

# V-Shaped Steps

- 
- The diagram illustrates the V-Shaped software development process. It features a large blue downward-pointing arrow on the left and a large blue upward-pointing arrow on the right, forming a wide V-shape. Inside the V, there are two columns of text. The left column contains four items: 'Project and Requirements Planning – allocate resources', 'Product Requirements and Specification Analysis – complete specification of the software system', 'Architecture or High-Level Design – defines how software functions fulfill the design', and 'Detailed Design – develop algorithms for each architectural component'. The right column contains five items: 'Production, operation and maintenance – provide for enhancement and corrections', 'System and acceptance testing – check the entire software system in its environment', 'Integration and Testing – check that modules interconnect correctly', 'Unit testing – check that each module acts as expected', and 'Coding – transform algorithms into software'.
- Project and Requirements Planning – allocate resources
  - Product Requirements and Specification Analysis – complete specification of the software system
  - Architecture or High-Level Design – defines how software functions fulfill the design
  - Detailed Design – develop algorithms for each architectural component
  - Production, operation and maintenance – provide for enhancement and corrections
  - System and acceptance testing – check the entire software system in its environment
  - Integration and Testing – check that modules interconnect correctly
  - Unit testing – check that each module acts as expected
  - Coding – transform algorithms into software

# V-Shaped Strengths

- Emphasize planning for verification and validation of the product in early stages of product development
- Each deliverable must be testable
- Project management can track progress by milestones
- Easy to use

# V-Shaped Weaknesses

- Does not easily handle concurrent events
- Does not handle iterations or phases
- Does not easily handle dynamic changes in requirements
- Does not contain risk analysis activities

# When to use the V-Shaped Model

- Excellent choice for systems requiring high reliability – hospital patient control applications
- All requirements are known up-front
- When it can be modified to handle changing requirements beyond analysis phase
- Solution and technology are known

# Prototyping: Basic Steps

- Identify basic requirements
  - Including input and output info
  - Details (e.g., security) generally ignored
- Develop initial prototype
  - UI first
- Review
  - Customers/end –users review and give feedback
- Revise and enhance the prototype & specs
  - Negotiation about scope of contract may be necessary

# Dimensions of prototyping

- Horizontal prototype
  - Broad view of entire system/sub-system
  - Focus is on user interaction more than low-level system functionality (e.g., database access)
  - Useful for:
    - Confirmation of UI requirements and system scope
    - Demonstration version of the system to obtain buy-in from business/customers
    - Develop preliminary estimates of development time, cost, effort

# Dimensions of Prototyping

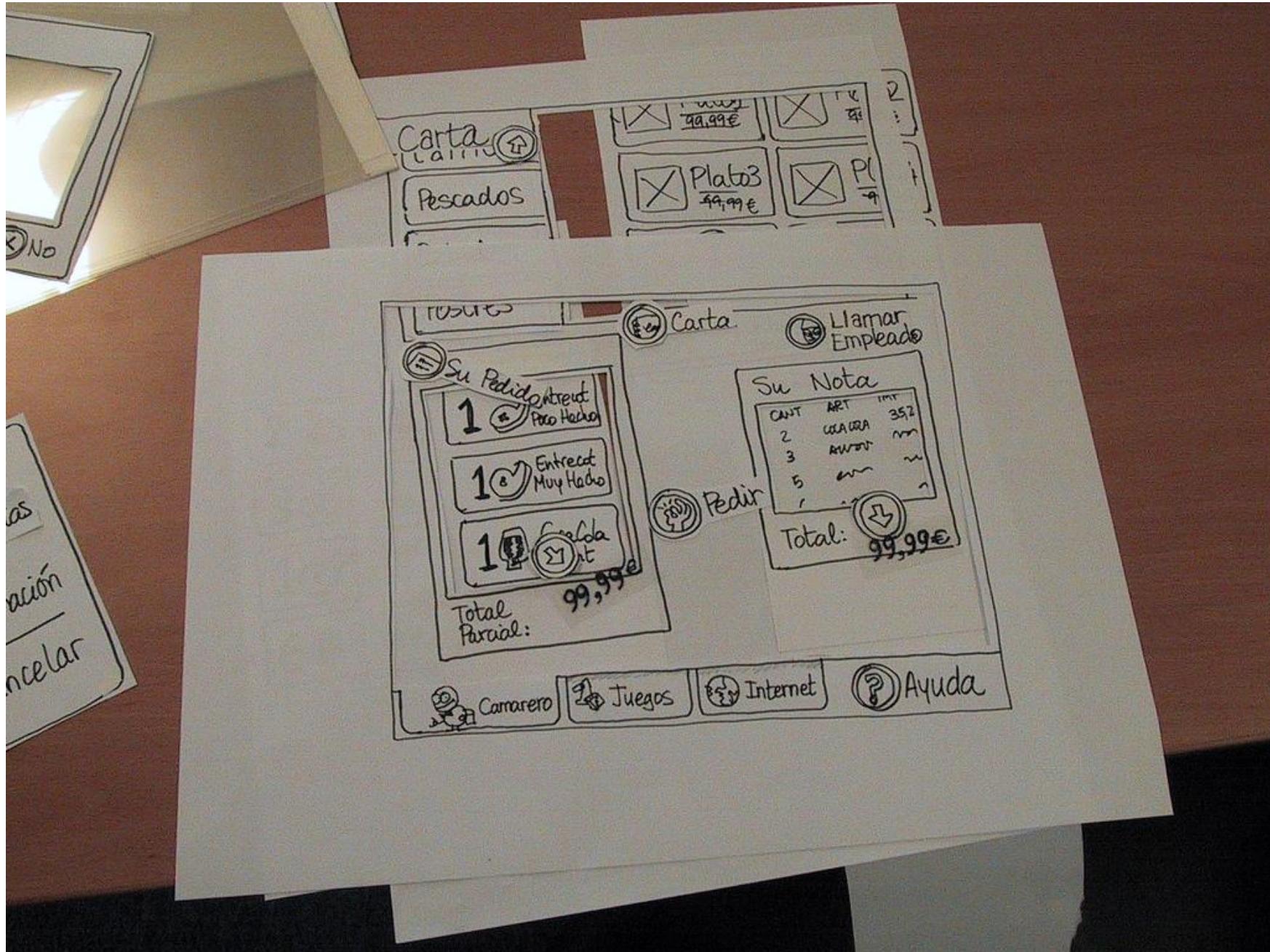
- Vertical prototype
  - More complete elaboration of a single sub-system or function
  - Useful for:
    - Obtaining detailed requirements for a given function
    - Refining database design
    - Obtaining info on system interface needs
    - Clarifying complex requirements by drilling down to actual system functionality

# Types of prototyping

- Throwaway /rapid/close-ended prototyping
  - Creation of a model that will be discarded rather than becoming part of the final delivered software
  - After preliminary requirements gathering, used to visually show the users what their requirements may look like when implemented
- Focus is on quickly developing the model
  - not on good programming practices
  - Can Wizard of Oz things

# Fidelity of Prototype

- Low-fidelity
  - Paper/pencil
    - Mimics the functionality, but does not look like it



# Fidelity of Prototype

- Medium to High-fidelity
  - GUI builder
  - “Click dummy” prototype – looks like the system, but does not provide the functionality
  - Or provide functionality, but have it be general and not linked to specific data
  - <http://www.youtube.com/watch?v=VGjcFouSlpk>
  - <http://www.youtube.com/watch?v=5oLlmNbxaP4&feature=related>

# Throwaway Prototyping steps

- Write preliminary requirements
- Design the prototype
- User experiences/uses the prototype, specifies new requirements
- Repeat if necessary
- Write the final requirements
- Develop the real products

# Evolutionary Prototyping

- Aka breadboard prototyping
- Goal is to build a very robust prototype in a structured manner and constantly refine it
- The evolutionary prototype forms the heart of the new system and is added to and refined
- Allow the development team to add features or make changes that were not conceived in the initial requirements

# Evolutionary Prototyping Model

- Developers build a prototype during the requirements phase
- Prototype is evaluated by end users
- Users give corrective feedback
- Developers further refine the prototype
- When the user is satisfied, the prototype code is brought up to the standards needed for a final product.

# EP Steps

- A preliminary project plan is developed
- An partial high-level paper model is created
- The model is source for a partial requirements specification
- A prototype is built with basic and critical attributes
- The designer builds
  - the database
  - user interface
  - algorithmic functions
- The designer demonstrates the prototype, the user evaluates for problems and suggests improvements.
- This loop continues until the user is satisfied

# EP Strengths

- Customers can “see” the system requirements as they are being gathered
- Developers learn from customers
- A more accurate end product
- Unexpected requirements accommodated
- Allows for flexible design and development
- Steady, visible signs of progress produced
- Interaction with the prototype stimulates awareness of additional needed functionality

# Incremental prototyping

- Final product built as separate prototypes
- At the end, the prototypes are merged into a final design

# Extreme Prototyping

- Often used for web applications
- Development broken down into 3 phases, each based on the preceding 1
  1. Static prototype consisting of HTML pages
  2. Screens are programmed and fully functional using a simulated services layer
    - Fully functional UI is developed with little regard to the services, other than their contract
  3. Services are implemented

# Prototyping advantages

- Reduced time and cost
  - Can improve the quality of requirements and specifications provided to developers
    - Early determination of what the user really wants can result in faster and less expensive software
- Improved/increased user involvement
  - User can see and interact with the prototype, allowing them to provide better/more complete feedback and specs
  - Misunderstandings/miscommunications revealed
  - Final product more likely to satisfy their desired look/feel/performance

# Disadvantages of prototyping 1

- Insufficient analysis
  - Focus on limited prototype can distract developers from analyzing complete project
  - May overlook better solutions
  - Conversion of limited prototypes into poorly engineered final projects that are hard to maintain
  - Limited functionality may not scale well if used as the basis of a final deliverable
    - May not be noticed if developers too focused on building prototype as a model

# Disadvantages of prototyping 2

- User confusion of prototype and finished system
  - Users can think that a prototype (intended to be thrown away) is actually a final system that needs to be polished
    - Unaware of the scope of programming needed to give prototype robust functionality
  - Users can become attached to features included in prototype for consideration and then removed from final specification

# Disadvantages of prototyping 3

- Developer attachment to prototype
  - If spend a great deal of time/effort to produce, may become attached
  - Might try to attempt to convert a limited prototype into a final system
    - Bad if the prototype does not have an appropriate underlying architecture

# Disadvantages of prototyping 4

- Excessive development time of the prototype
  - Prototyping supposed to be done quickly
  - If developers lose sight of this, can try to build a prototype that is too complex
  - For throw away prototypes, the benefits realized from the prototype (precise requirements) may not offset the time spent in developing the prototype – expected productivity reduced
  - Users can be stuck in debates over prototype details and hold up development process

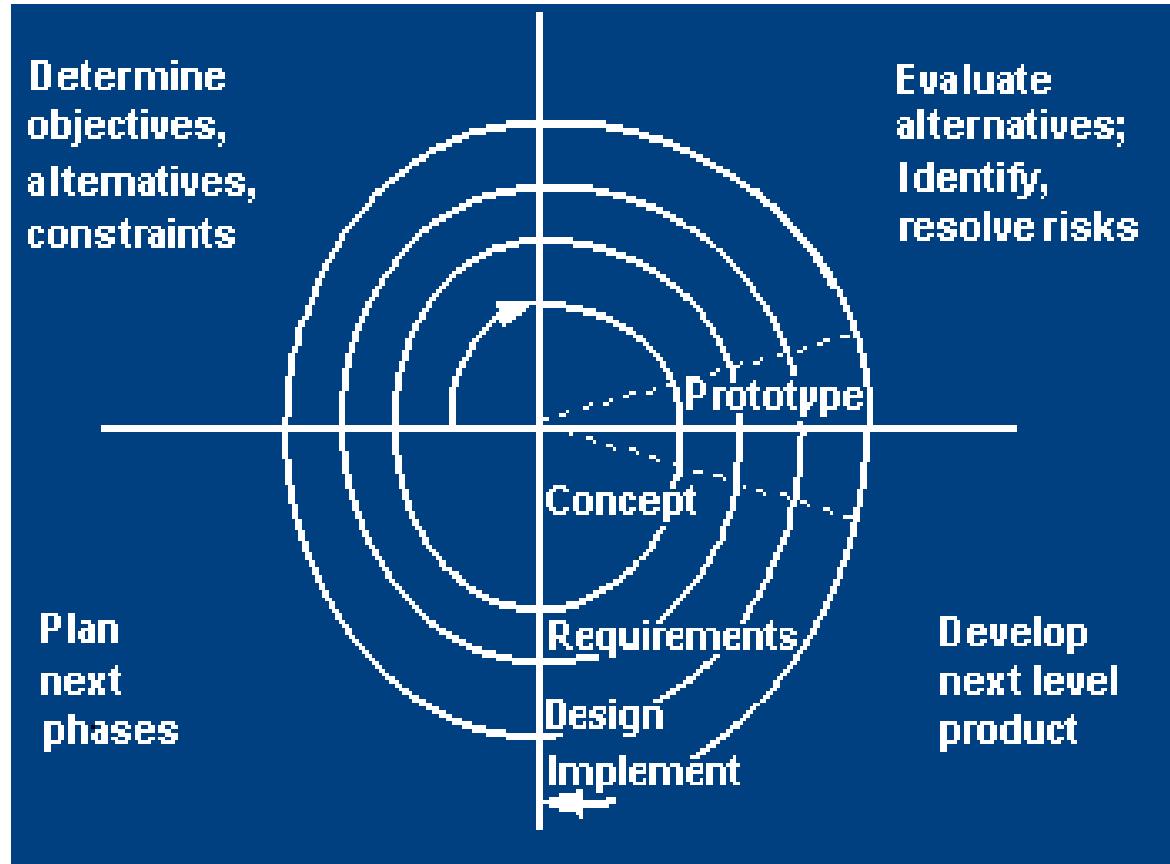
# Disadvantages of prototyping 5

- Expense of implementing prototyping
  - Start up costs of prototyping may be high
  - Expensive to change development methodologies in place (re-training, re-tooling)
  - Slow development if proper training not in place
    - High expectations for productivity unrealistic if insufficient recognition of the learning curve
  - Lower productivity can result if overlook the need to develop corporate and project specific underlying structure to support the technology

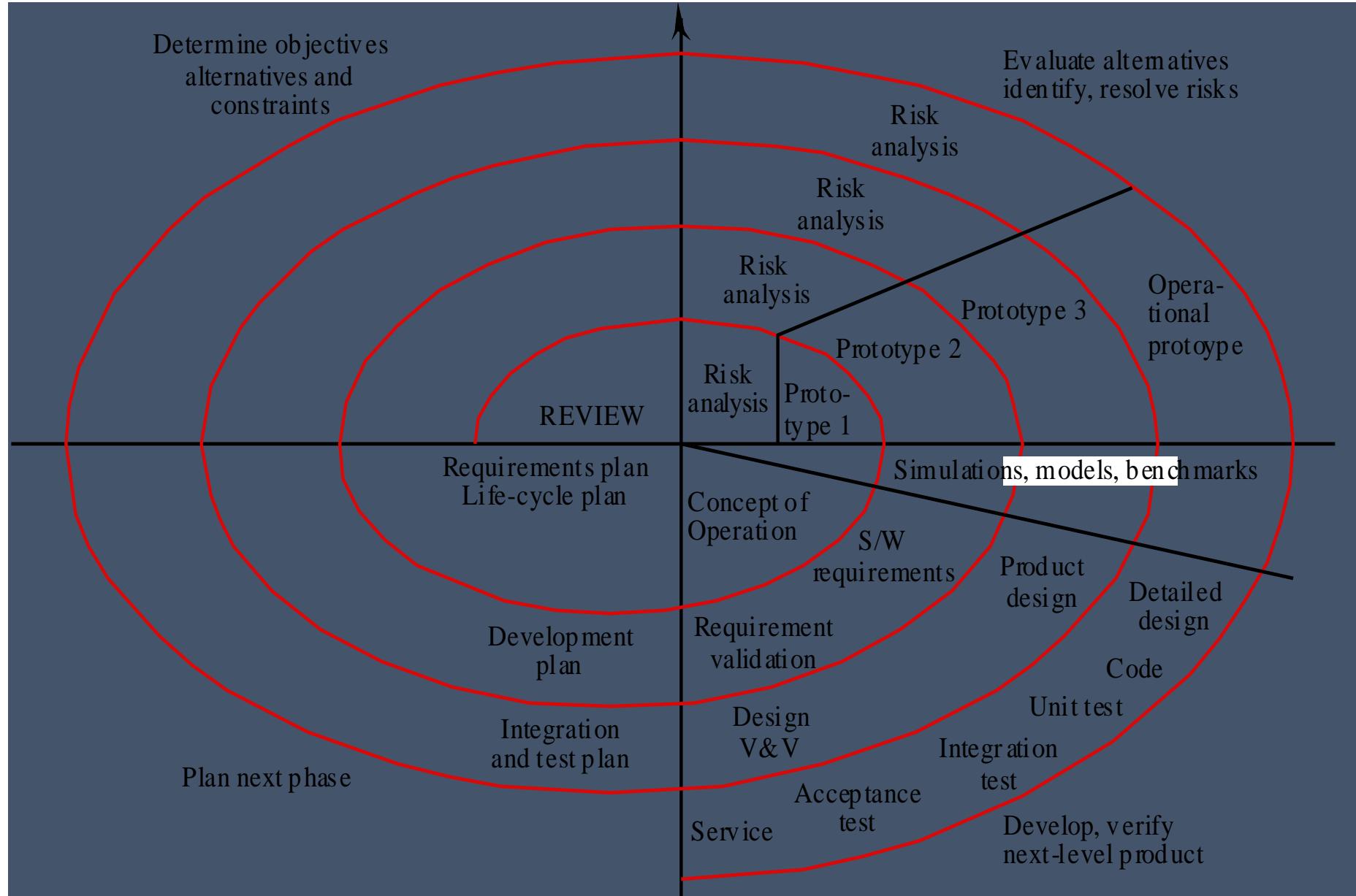
# Best uses of prototyping

- Most beneficial for systems that will have many interactions with end users
- The greater the interaction between the computer and the user, the greater the benefit of building a quick system for the user to play with
- Especially good for designing good human-computer interfaces

# Spiral SDLC Model



- Adds risk analysis, and 4gl RAD prototyping to the waterfall model
- Each cycle involves the same sequence of steps as the waterfall process model



Spiral Quadrant: Determine objectives, alternatives and constraints

- Objectives: functionality, performance, hardware/software interface, critical success factors, etc.
- Alternatives: build, reuse, buy, sub-contract, etc.
- Constraints: cost, schedule, interface, etc.

Spiral Quadrant: Evaluate alternatives, identify and resolve risks

- Study alternatives relative to objectives and constraints
- Identify risks (lack of experience, new technology, tight schedules, poor process, etc.)
- Resolve risks (evaluate if money could be lost by continuing system development)

## Spiral Quadrant: Develop next-level product

- Typical activites:
  - Create a design
  - Review design
  - Develop code
  - Inspect code
  - Test product

## Spiral Quadrant: Plan next phase

- Typical activities
  - Develop project plan
  - Develop configuration management plan
  - Develop a test plan
  - Develop an installation plan

# Spiral Model Strengths

- Provides early indication of insurmountable risks, without much cost
- Users see the system early because of rapid prototyping tools
- Critical high-risk functions are developed first
- The design does not have to be perfect
- Users can be closely tied to all lifecycle steps
- Early and frequent feedback from users
- Cumulative costs assessed frequently

# Spiral Model Weaknesses

- Time spent for evaluating risks too large for small or low-risk projects
- Time spent planning, resetting objectives, doing risk analysis and prototyping may be excessive
- The model is complex
- Risk assessment expertise is required
- Spiral may continue indefinitely
- Developers must be reassigned during non-development phase activities
- May be hard to define objective, verifiable milestones that indicate readiness to proceed through the next iteration

# When to use Spiral Model

- When creation of a prototype is appropriate
- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

# Role Playing Game for SE's

- <http://www.youtube.com/watch?v=kkkl3LucxTY&feature=related>

# Housekeeping

- Individual Assignment:
  - Post mortem + peer review
- Final presentations/demos
  - July 26/28 - 25 minutes per
    - ~8 minute presentation
    - ~10 minute demo
    - ~7 minutes questions
- Course evaluations this Thursday (4:05 pm)

# The Rise and Fall of Waterfall

- <http://www.youtube.com/watch?v=X1c2--sP3o0&NR=1&feature=fvwp>
- Warning: bad language at 3:50! (hands over ears if easily offended!)

# Agile software development life cycles

# Agile SDLC's

- Speed up or bypass one or more life cycle phases
- Usually less formal and reduced scope
- Used for time-critical applications
- Used in organizations that employ disciplined methods

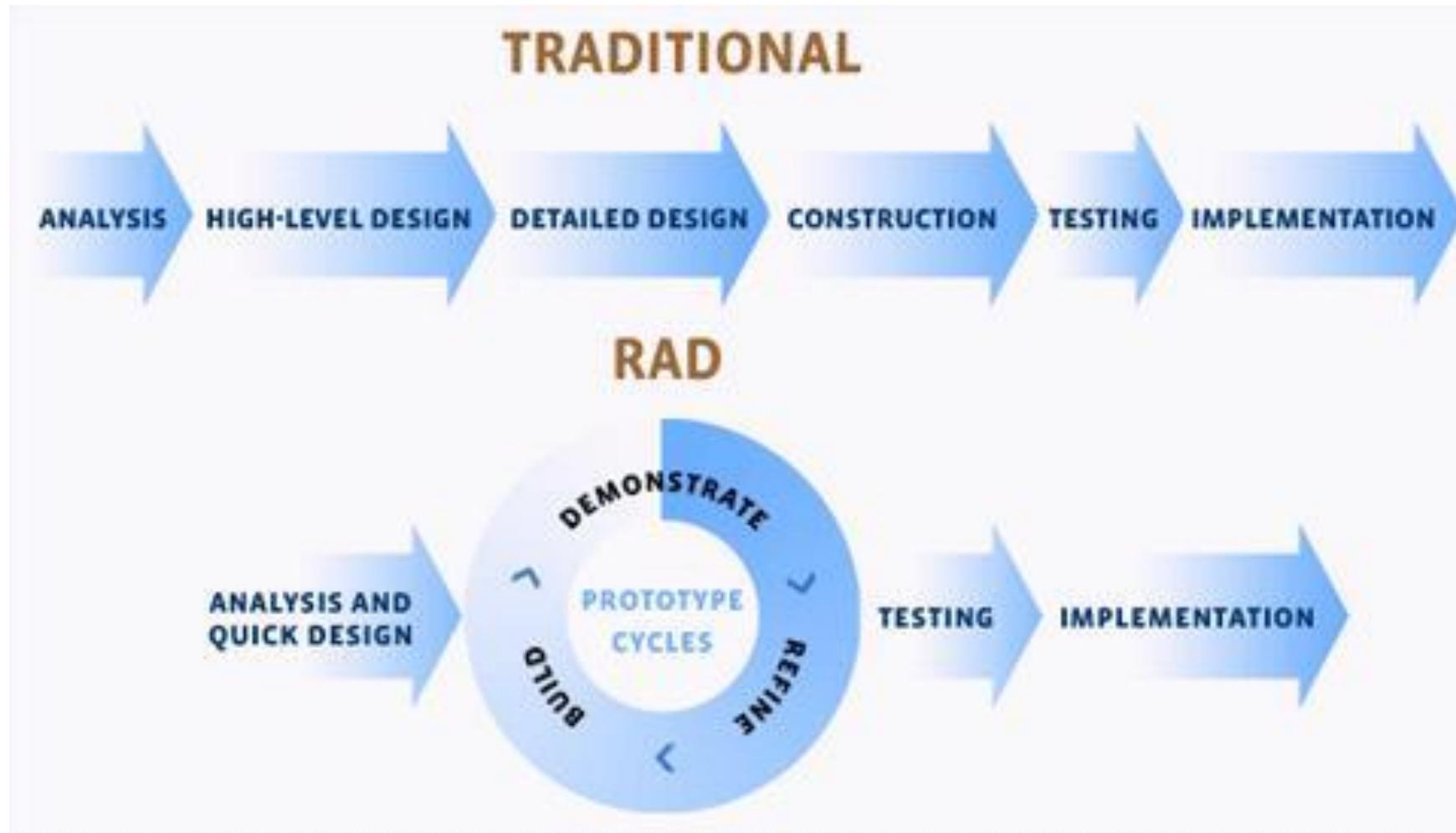
# Some Agile Methods

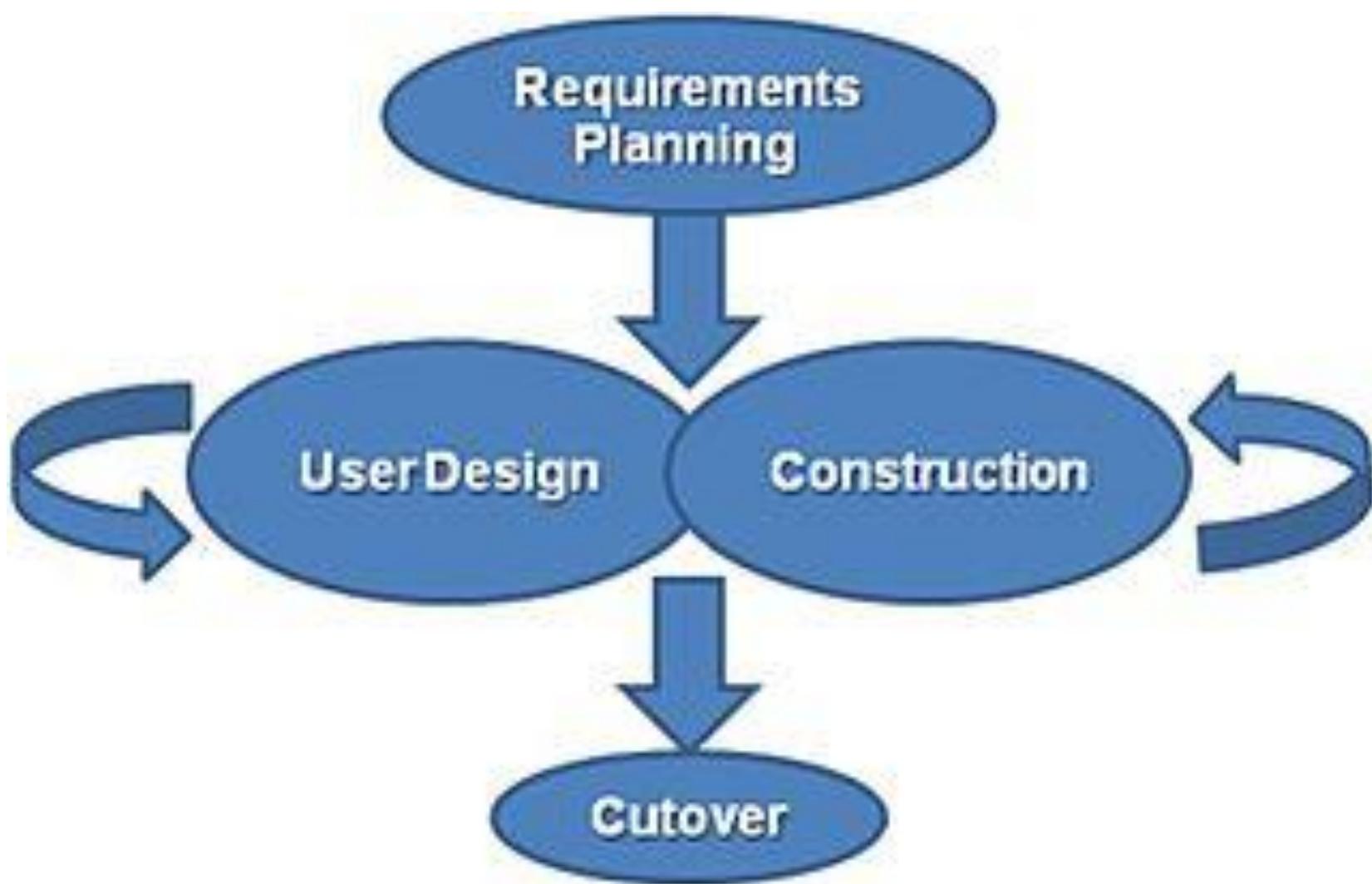
- Rapid Application Development (RAD)
- Incremental SDLC
- Scrum
- Extreme Programming (XP)
- Adaptive Software Development (ASD)
- Feature Driven Development (FDD)
- Crystal Clear
- Dynamic Software Development Method (DSDM)
- Rational Unify Process (RUP)

# Agile vs Waterfall Propaganda

- <http://www.youtube.com/watch?v=gDDO3ob-4ZY&feature=related>

# Rapid application development (RAD) Model





# Rapid Application Model (RAD)

- Requirements planning phase (a workshop utilizing structured discussion of business problems)
- User description phase – automated tools capture information from users
- Construction phase – productivity tools, such as code generators, screen generators, etc. inside a time-box. (“Do until done”)
- Cutover phase -- installation of the system, user acceptance testing and user training

# Requirements Planning Phase

- Combines elements of the system planning and systems analysis phases of the System Development Life Cycle (SDLC).
- Users, managers, and IT staff members discuss and agree on business needs, project scope, constraints, and system requirements.
- It ends when the team agrees on the key issues and obtains management authorization to continue.

# User Design Phase

- Users interact with systems analysts and develop models and prototypes that represent all system processes, inputs, and outputs.
- Typically use a combination of Joint Application Development (JAD) techniques and CASE tools to translate user needs into working models.
- A continuous interactive process that allows users to understand, modify, and eventually approve a working model of the system that meets their needs.

# JAD Techniques

- [http://en.wikipedia.org/wiki/Joint application design](http://en.wikipedia.org/wiki/Joint_application_design)

## CASE Tools

- [http://en.wikipedia.org/wiki/Computer-aided software engineering](http://en.wikipedia.org/wiki/Computer-aided_software_engineering)

# Construction Phase

- Focuses on program and application development task similar to the SDLC.
- However, users continue to participate and can still suggest changes or improvements as actual screens or reports are developed.
- Its tasks are programming and application development, coding, unit-integration, and system testing.

# Cutover Phase

- Resembles the final tasks in the SDLC implementation phase.
- Compared with traditional methods, the entire process is compressed. As a result, the new system is built, delivered, and placed in operation much sooner.
- Tasks are data conversion, full-scale testing, system changeover, user training.

# RAD Strengths

- Reduced cycle time and improved productivity with fewer people means lower costs
- Time-box approach mitigates cost and schedule risk
- Customer involved throughout the complete cycle minimizes risk of not achieving customer satisfaction and business needs
- Focus moves from documentation to code (WYSIWYG).
- Uses modeling concepts to capture information about business, data, and processes.

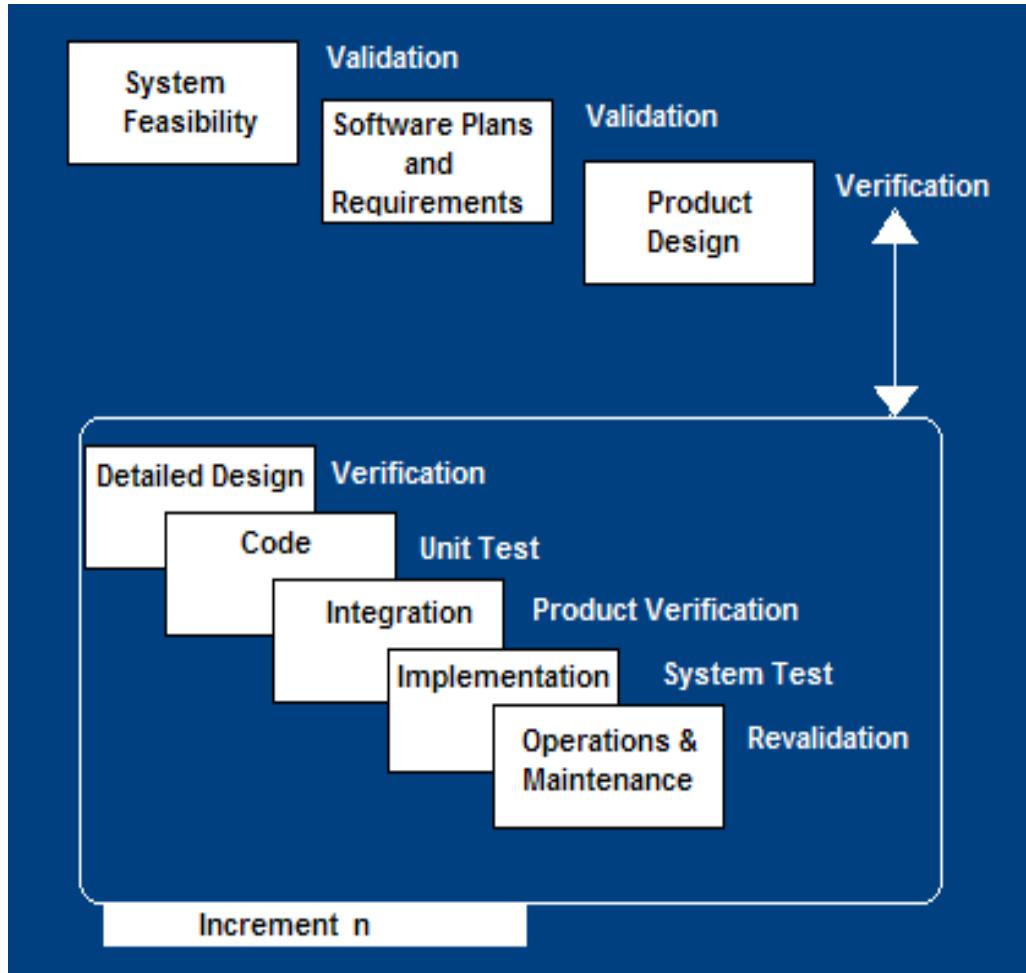
# RAD Weaknesses

- Accelerated development process must give quick responses to the user
- Risk of never achieving closure
- Hard to use with legacy systems
- Requires a system that can be modularized
- Developers and customers must be committed to rapid-fire activities in an abbreviated time frame.

# When to use RAD

- Reasonably well-known requirements
- User involved throughout the life cycle
- Project can be time-boxed
- Functionality delivered in increments
- High performance not required
- Low technical risks
- System can be modularized

# Incremental SDLC Model



- Construct a partial implementation of a total system
- Then slowly add increased functionality
- The incremental model prioritizes requirements of the system and then implements them in groups.
- Each subsequent release of the system adds function to the previous release, until all designed functionality has been implemented.

# Incremental Model Strengths

- Develop high-risk or major functions first
- Each release delivers an operational product
- Customer can respond to each build
- Uses “divide and conquer” breakdown of tasks
- Lowers initial delivery cost
- Initial product delivery is faster
- Customers get important functionality early
- Risk of changing requirements is reduced

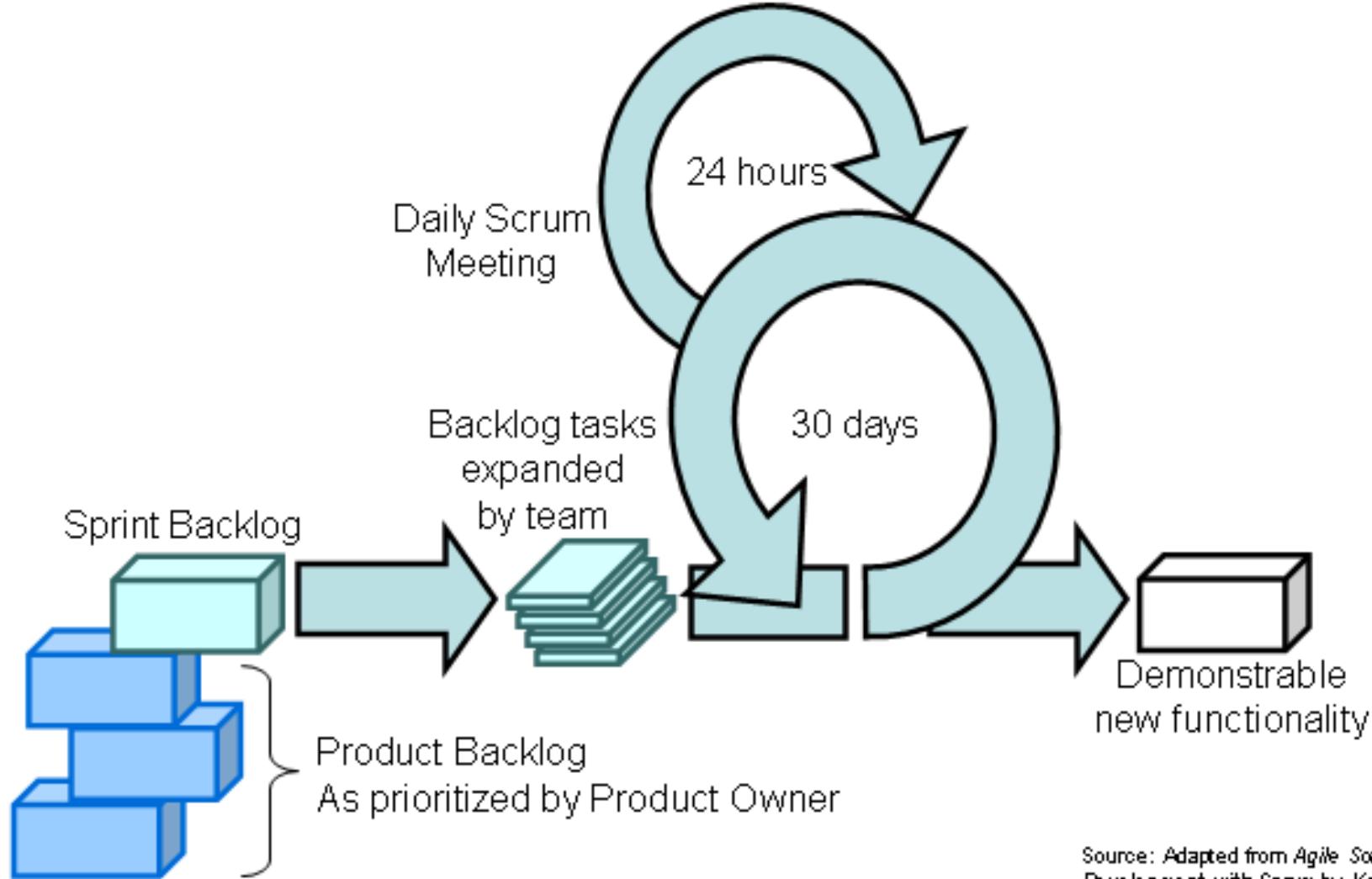
# Incremental Model Weaknesses

- Requires good planning and design
- Requires early definition of a complete and fully functional system to allow for the definition of increments
- Well-defined module interfaces are required (some will be developed long before others)
- Total cost of the complete system is not lower

# When to use the Incremental Model

- Risk, funding, schedule, program complexity, or need for early realization of benefits.
- Most of the requirements are known up-front but are expected to evolve over time
- A need to get basic functionality to the market early
- On projects which have lengthy development schedules
- On a project with new technology

scrum



Source: Adapted from *Agile Software Development with Scrum* by Ken Schwaber and Mike Beedle.

- Scrum in 13 seconds:
  - <http://www.youtube.com/watch?v=9DKM9HcRnZ8&feature=related>
- Scrum in 10 minutes:
  - <http://www.youtube.com/watch?v=Q5k7a9YEoUI>
- More scrum slides:
  - <http://www.mountaingoatsoftware.com/system/presentation/file/129/Getting-Agile-With-Scrum-Cohn-NDC2010.pdf?1276712017>
  - Scalability of scrum addressed on slides 33-35

# Scrum advantages

- Agile scrum helps the company in saving time and money.
- Scrum methodology enables projects where the business requirements documentation is hard to quantify to be successfully developed.
- Fast moving, cutting edge developments can be quickly coded and tested using this method, as a mistake can be easily rectified.

# Scrum advantages

- It is a lightly controlled method which insists on frequent updating of the progress in work through regular meetings. Thus there is clear visibility of the project development.
- Like any other agile methodology, this is also iterative in nature. It requires continuous feedback from the user.
- Due to short sprints and constant feedback, it becomes easier to cope with the changes.

# Scrum advantages

- Daily meetings make it possible to measure individual productivity. This leads to the improvement in the productivity of each of the team members.
- Issues are identified well in advance through the daily meetings and hence can be resolved in speedily
- It is easier to deliver a quality product in a scheduled time.

# Scrum advantages

- Agile Scrum can work with any technology/ programming language but is particularly useful for fast moving web 2.0 or new media projects.
- The overhead cost in terms of process and management is minimal thus leading to a quicker, cheaper result.

# Scrum disadvantages

- Agile Scrum is one of the leading causes of scope creep because unless there is a definite end date, the project management stakeholders will be tempted to keep demanding new functionality is delivered.
- If a task is not well defined, estimating project costs and time will not be accurate. In such a case, the task can be spread over several sprints.
- If the team members are not committed, the project will either never complete or fail.

# Scrum disadvantages

- It is good for small, fast moving projects as it works well **only** with small team.
- This methodology needs experienced team members only. If the team consists of people who are novices, the project cannot be completed in time.
- Scrum works well when the Scrum Master trusts the team they are managing. If they practice too strict control over the team members, it can be extremely frustrating for them, leading to demoralisation and the failure of the project.

# Scrum disadvantages

- If any of the team members leave during a development it can have a huge inverse effect on the project development
- Project quality management is hard to implement and quantify unless the test team are able to conduct regression testing after each sprint.

Paramananda Barik



## **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.

# 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.

# What is Use Case Diagram (UCD)

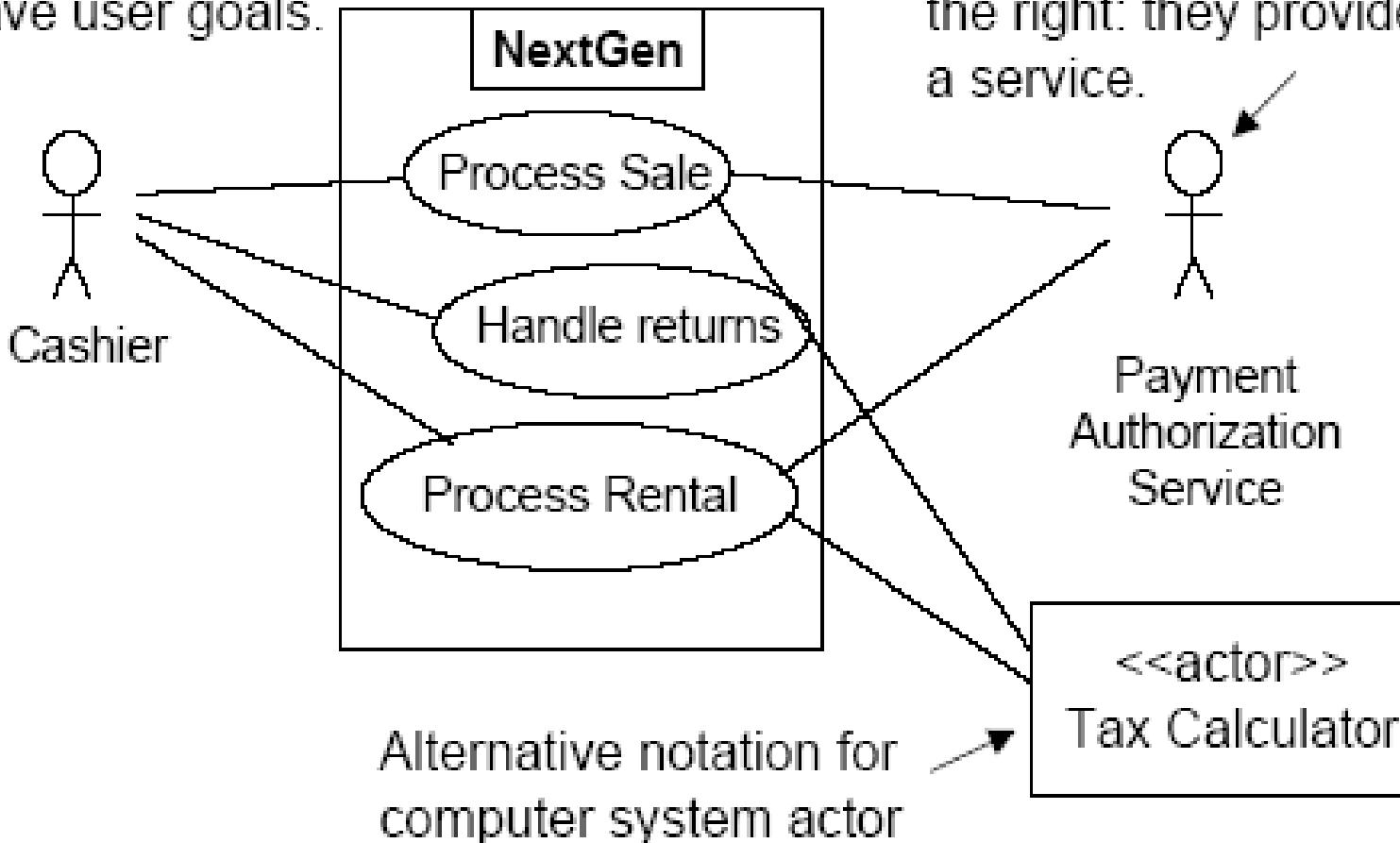
- It shows interaction among actors (external parties) and Use Cases (Modules) of the System
- It is not Data Flow Diagram !!
- Significance of Use Case Diagram

# 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

# Use Case Diagram for PoS

Primary actors to the left: have user goals.



Supporting actors to the right: they provide a service.

# 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.

# 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.



## Styles of Use Cases

# Styles of Use Cases

---

- 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.



**BITS** Pilani  
Pilani Campus

# BITS Pilani presentation

Paramananda Barik  
CS&IS Department



# **SSZG514/SEZG512, Object Oriented Analysis and Design Lecture No.3 and 4**



# **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.



# **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

# 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.

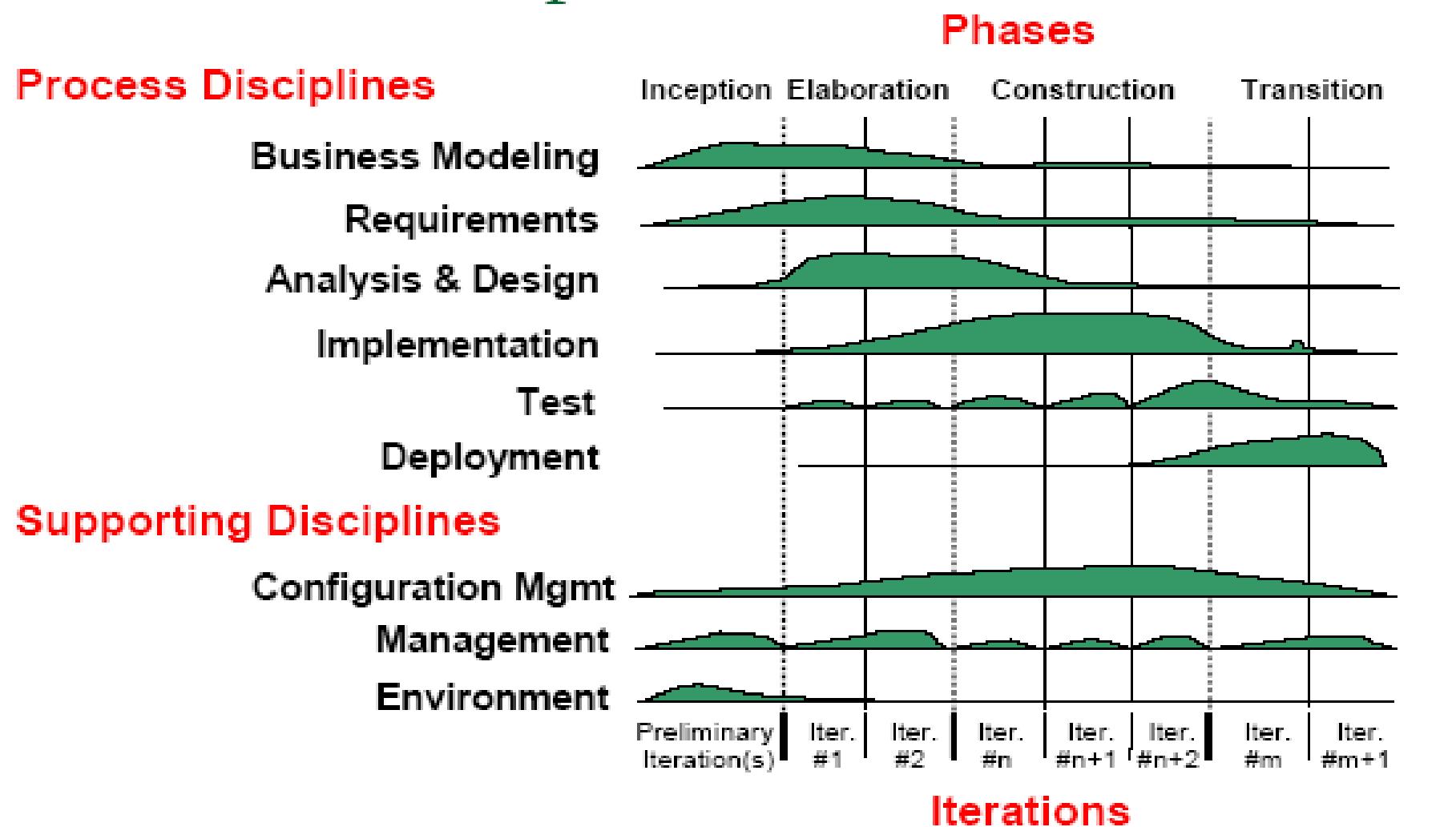


## UP : Phases & Disciplines

# 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





# **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
  - Responding to change over following a plan



**BITS**  
**Pilani**

Pilani|Dubai|Goa|Hyderabad



# eXtreme Programming (XP)

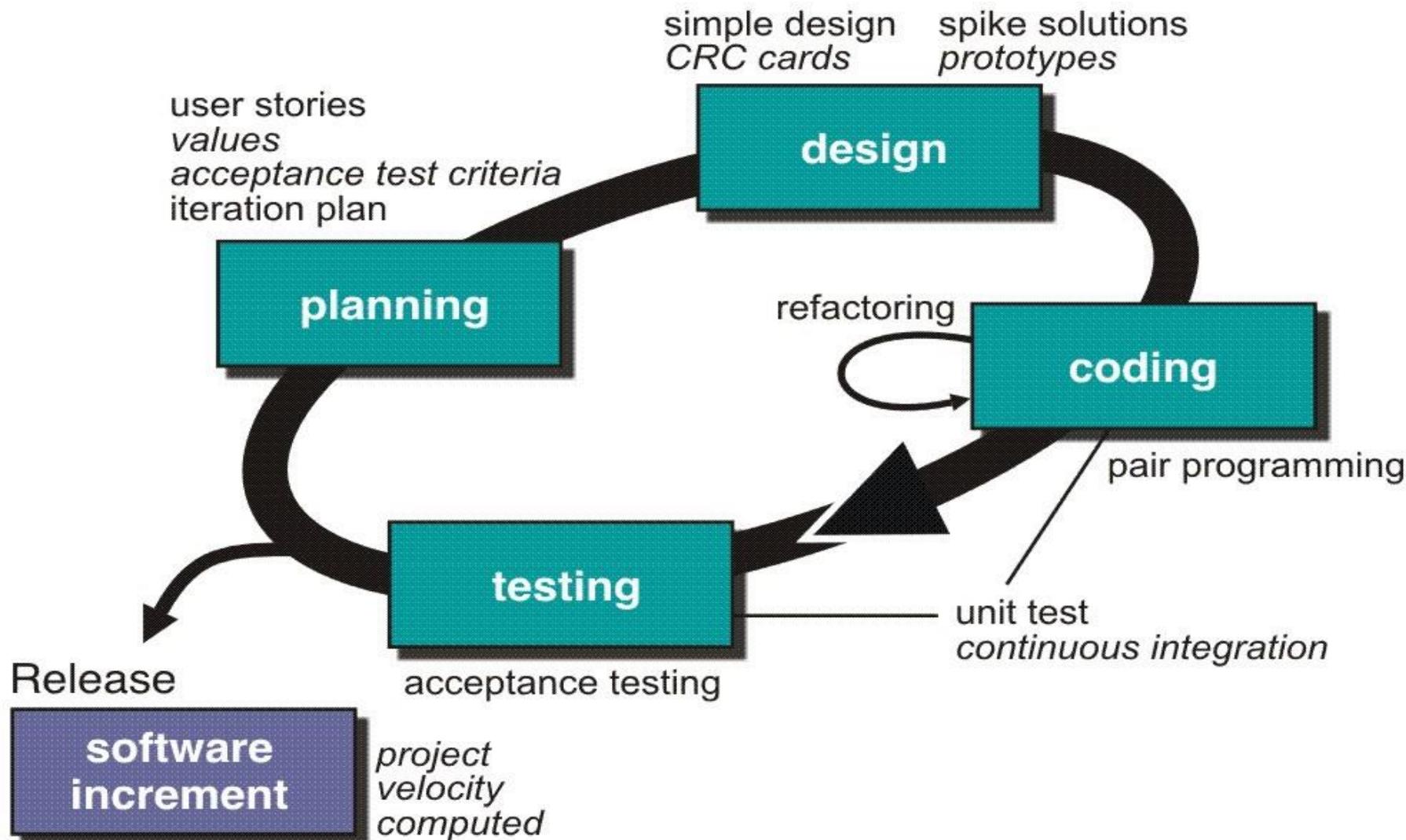
# eXtreme Programming (XP)

- 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)





**BITS**  
**Pilani**

Pilani|Dubai|Goa|Hyderabad



**SCRUM**

# SCRU

## M

- 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



# **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



# 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

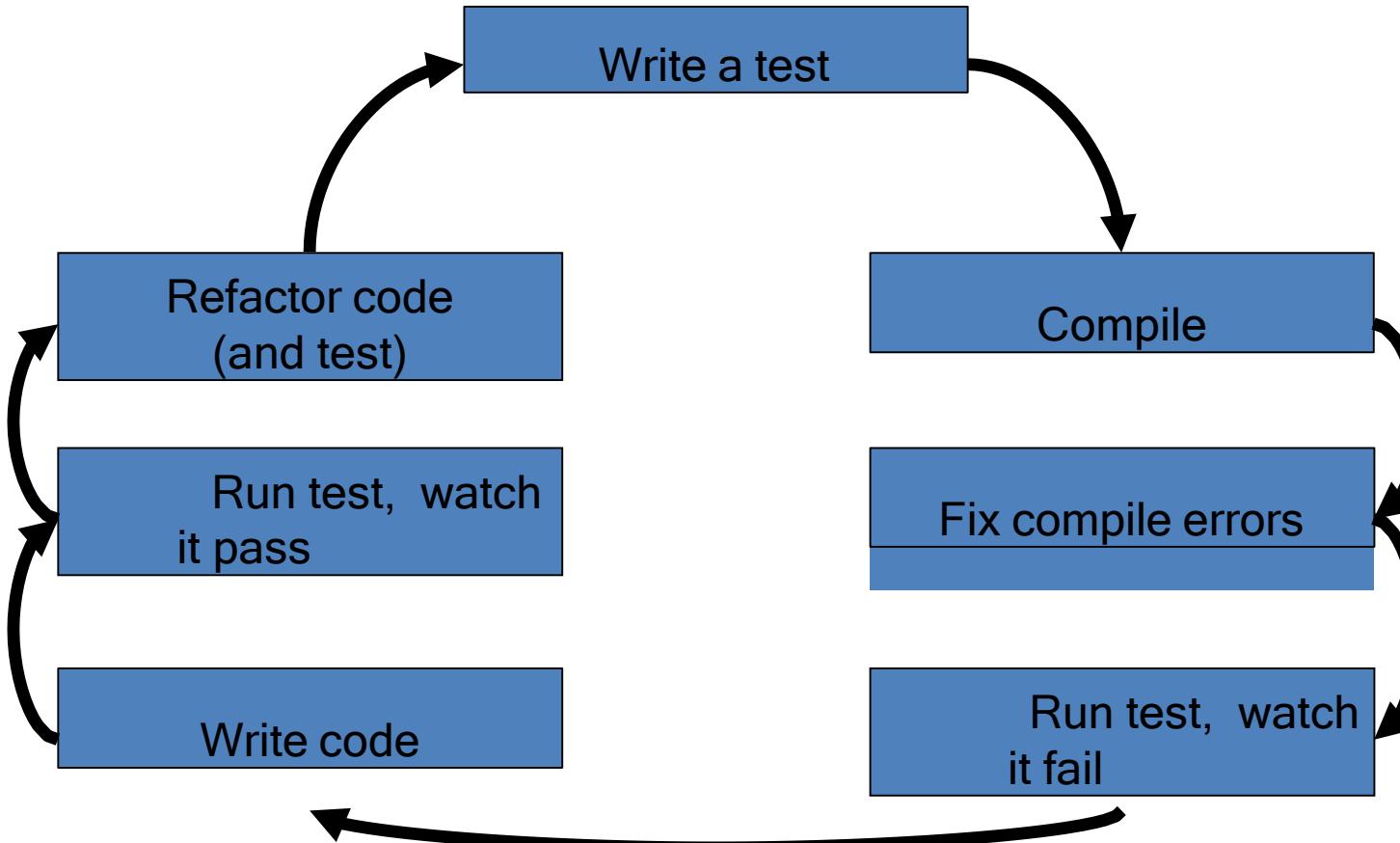
- Some might argue that “tests are part of the documentation”

# 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



# Refactoring & Continuous Integration

# Refactorin g

- 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



# **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 fulfils 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.



## Overview of UML

# What is UML?

---

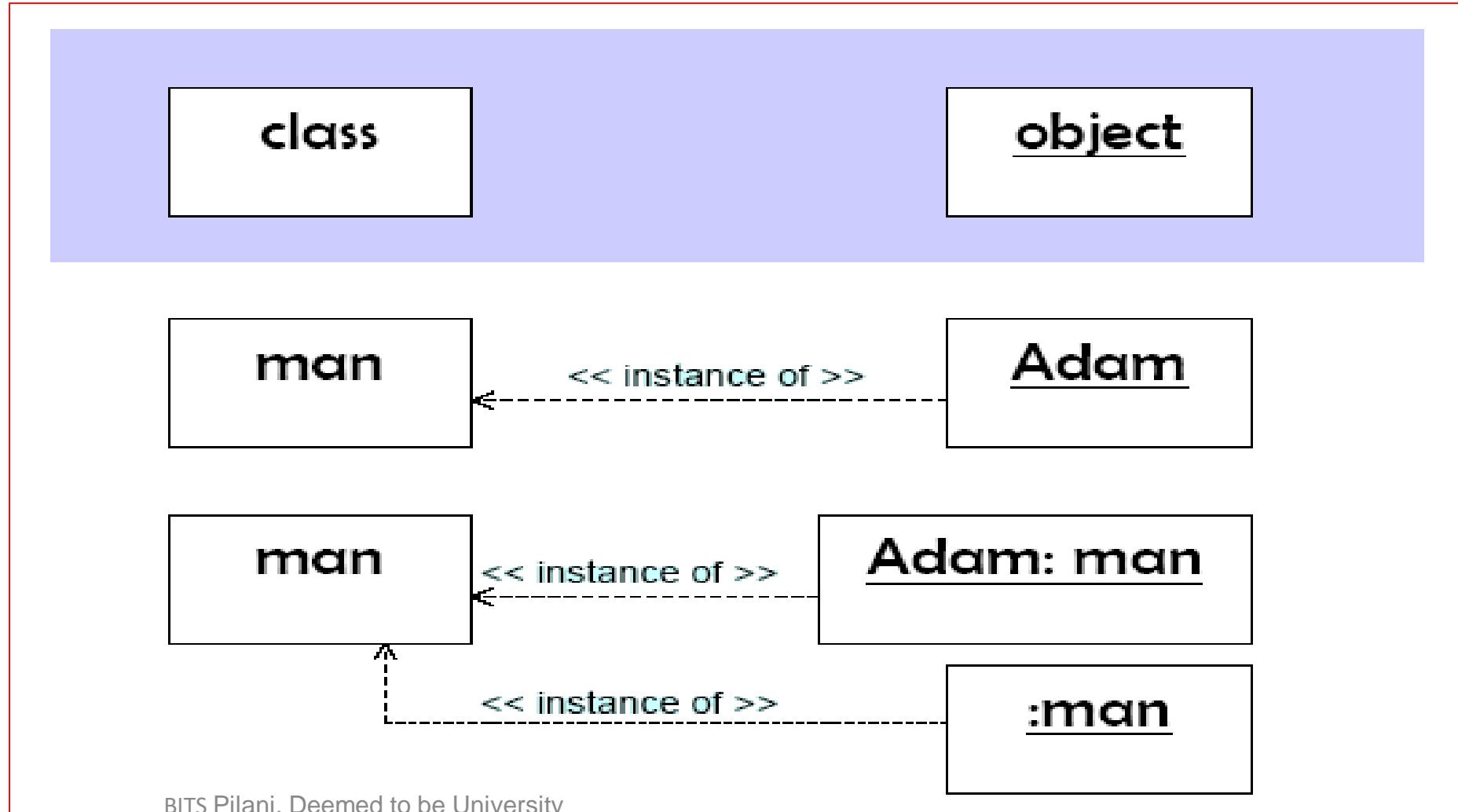


- Unified Modeling Language
    - modeling language to draw diagrams in Uniform way Object Oriented Analysis & Design
  - Need of UML
  - Who uses UML?
-



## Concept of Class & Object

# Class and Object: in UML



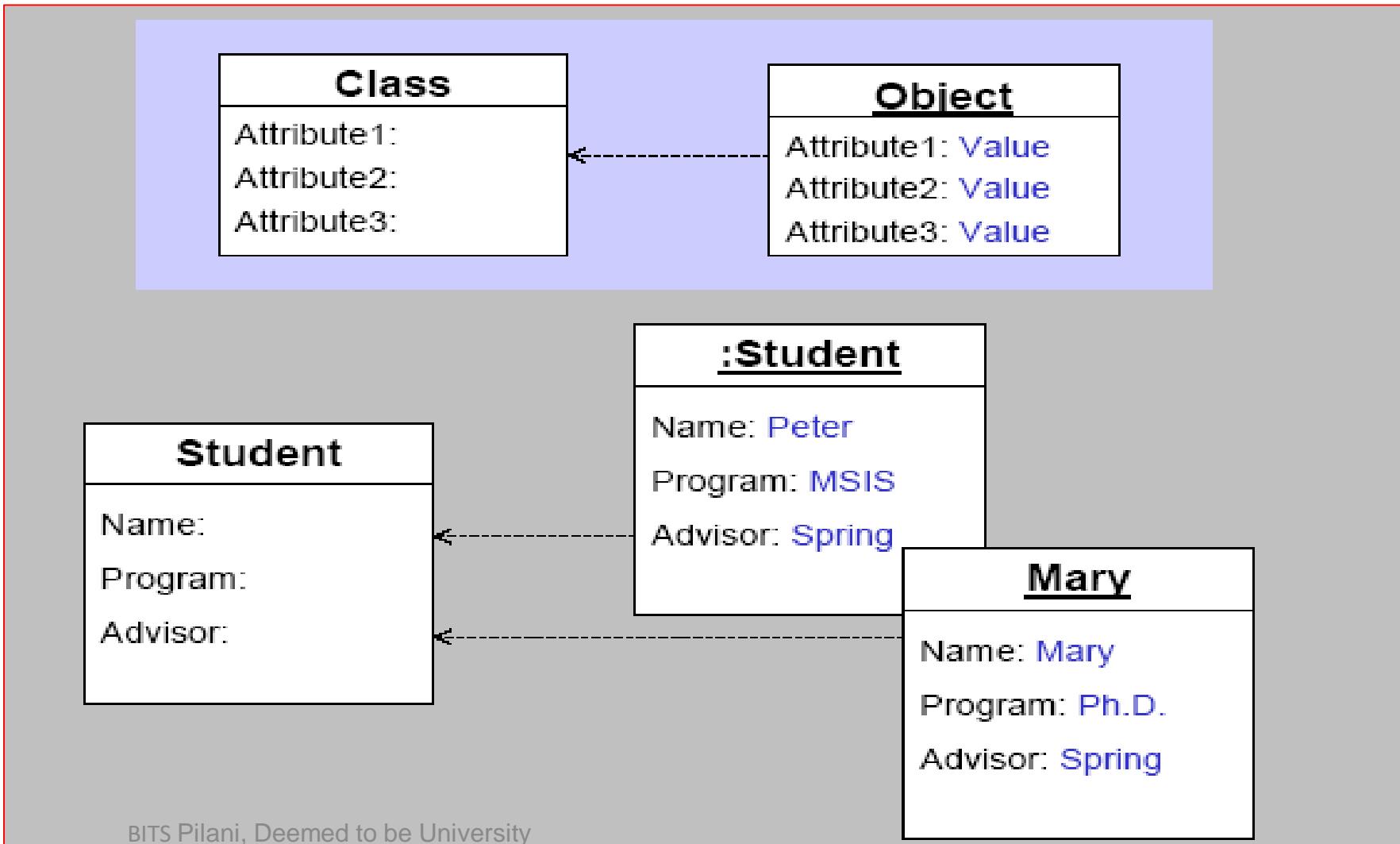
# Attribute

S

---

- 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**.

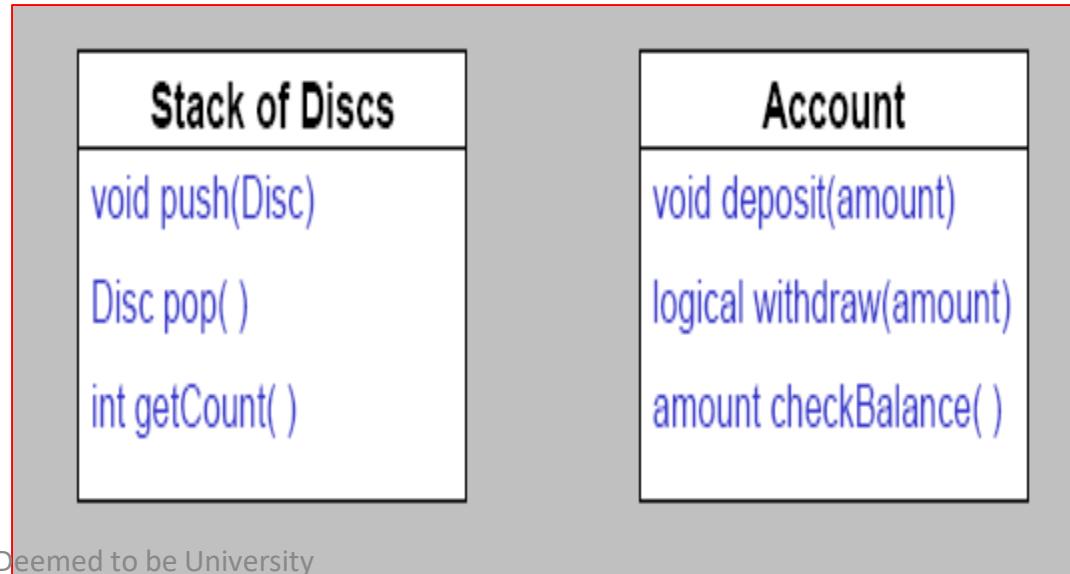
# Attributes: in UML



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

**deposit(\$1000)** →

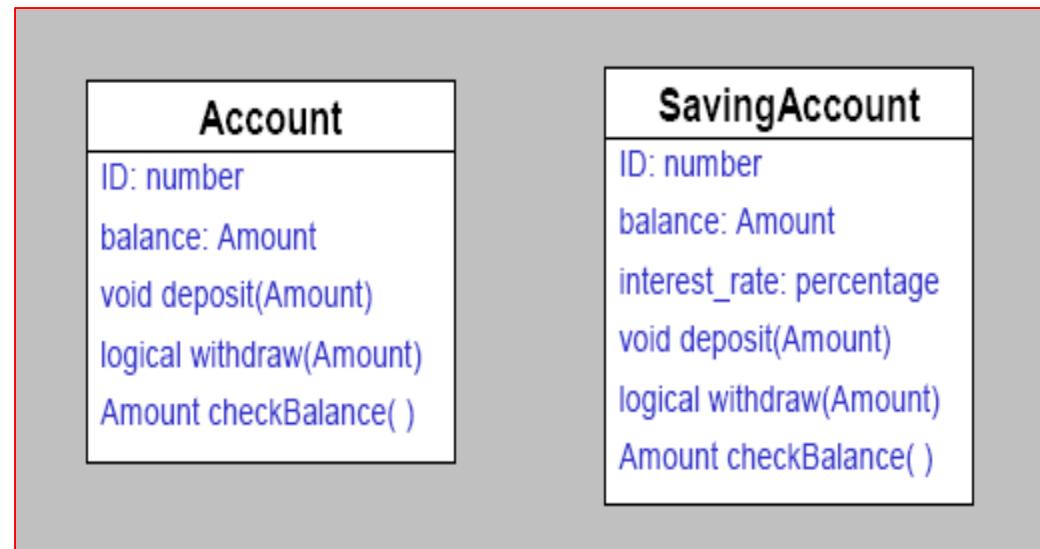


# **Class Relationships in UML**

# Inheritance: superclass & subclass

---

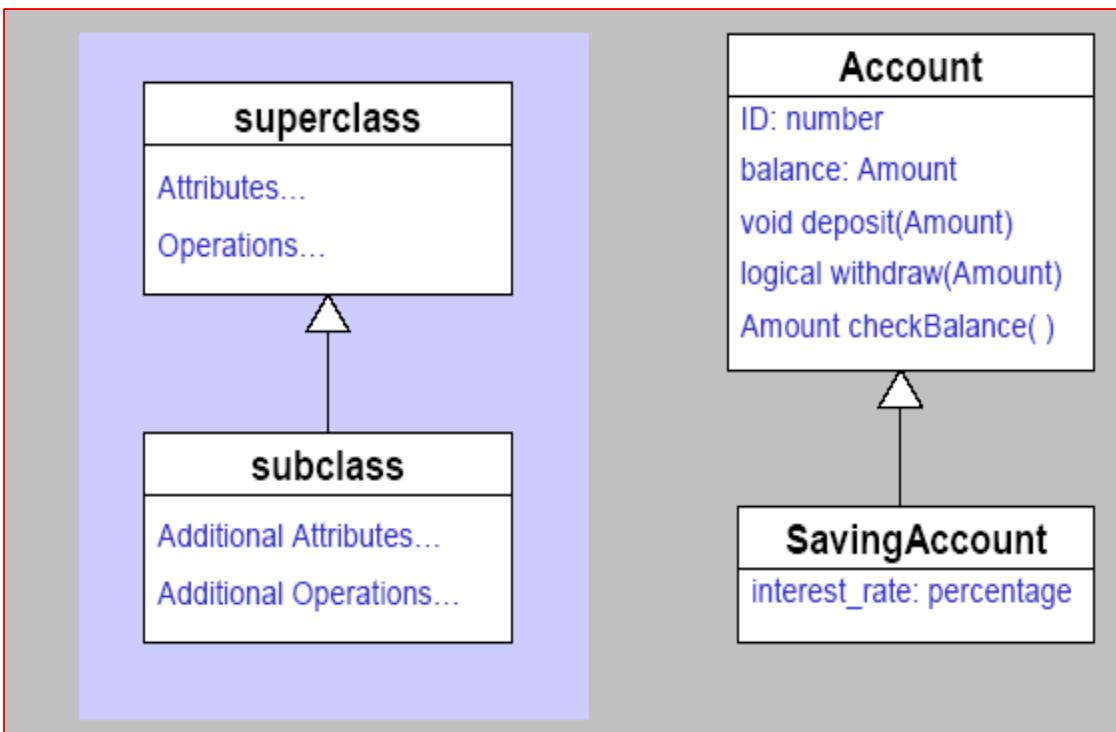
- A class may be the subset of another class...
- A saving account is also an account.
- Not all accounts are saving accounts.



# Inheritance in UML

---

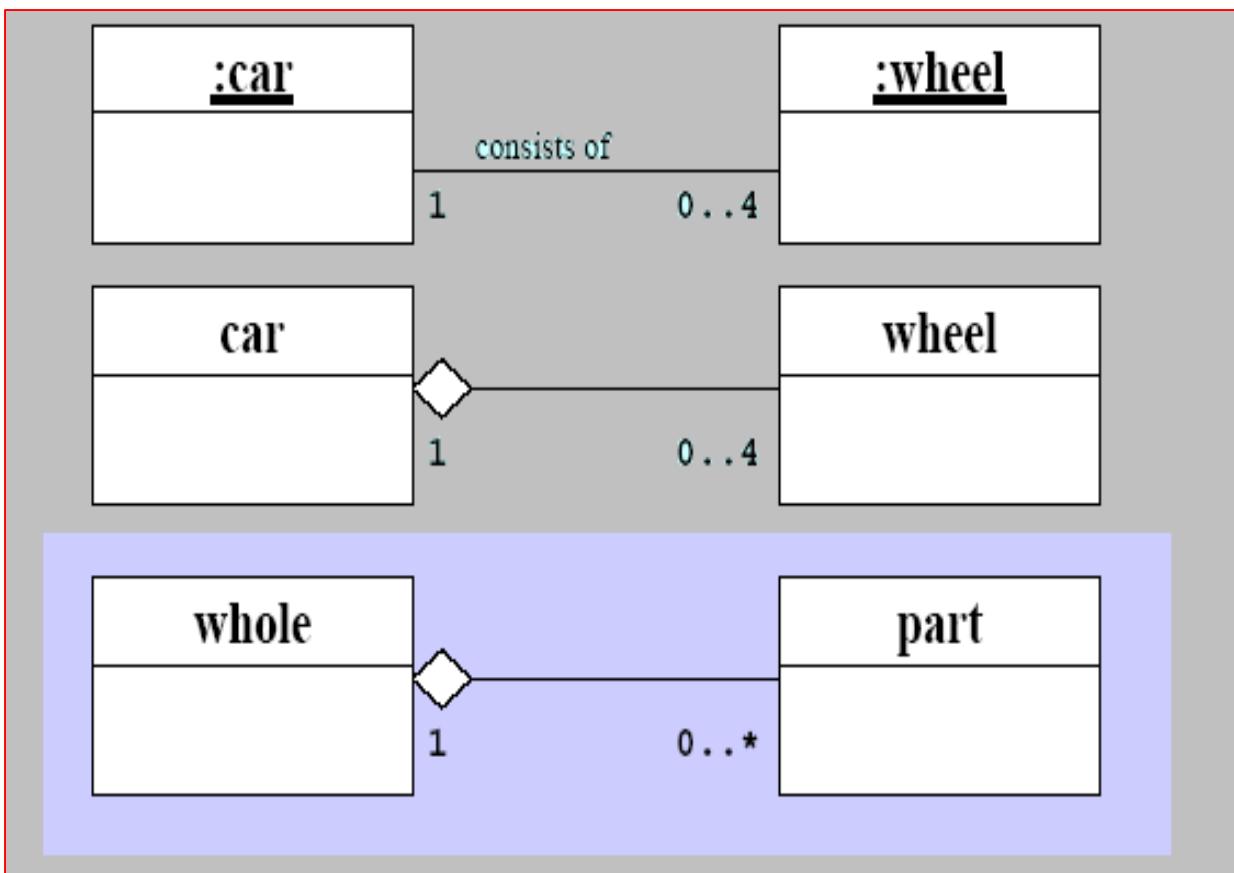
- The subclass inherits from the superclass the attributes as well as the operations.



# Aggregation: in UML

---

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



Paramananda Barik



## **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.

# 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.

# What is Use Case Diagram (UCD)

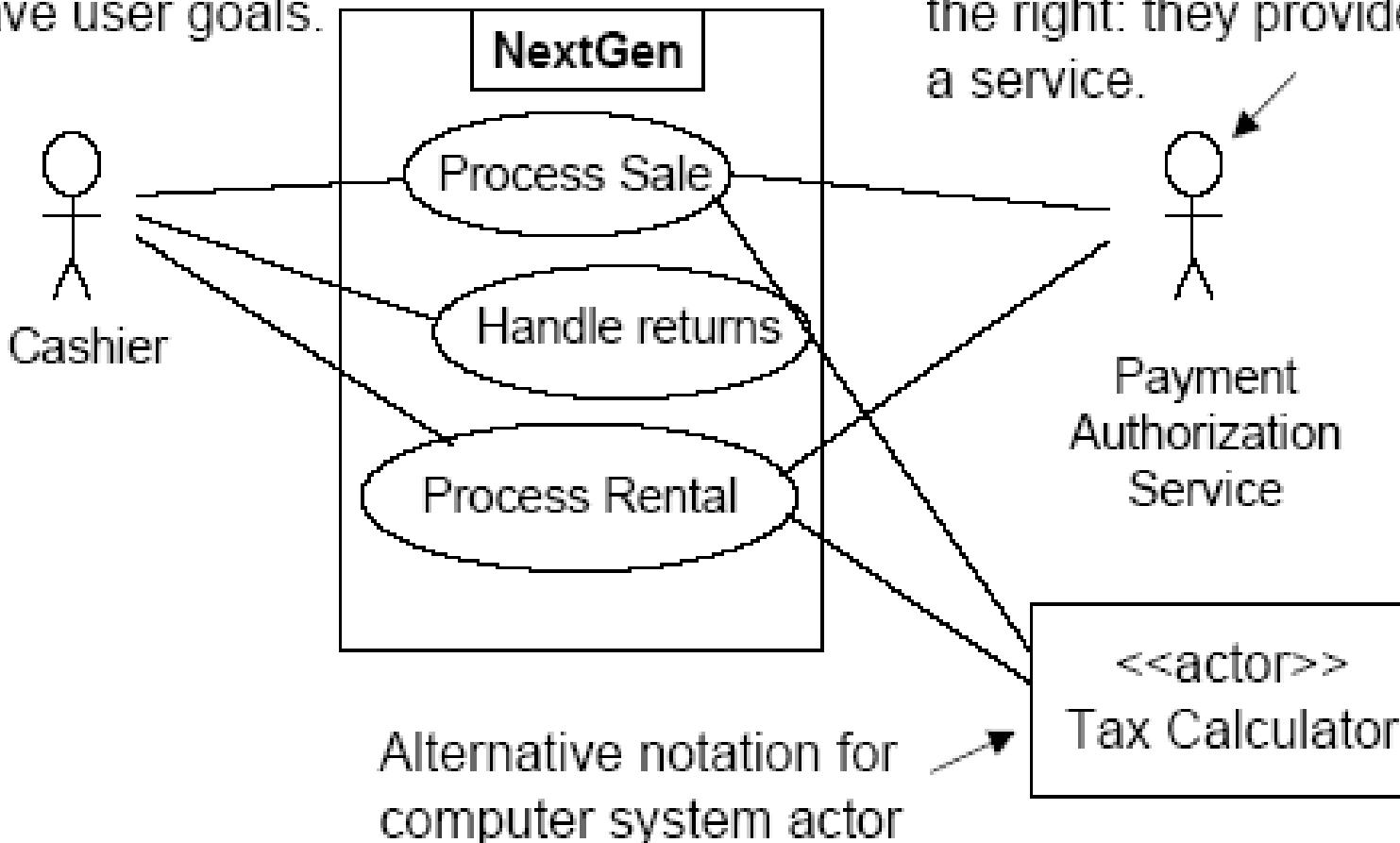
- It shows interaction among actors (external parties) and Use Cases (Modules) of the System
- It is not Data Flow Diagram !!
- Significance of Use Case Diagram

# 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

# Use Case Diagram for PoS

Primary actors to the left: have user goals.



Supporting actors to the right: they provide a service.

# 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.

# 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.



## Styles of Use Cases

# Styles of Use Cases

---

- 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.

Paramananda Barik



## What is Domain Model?



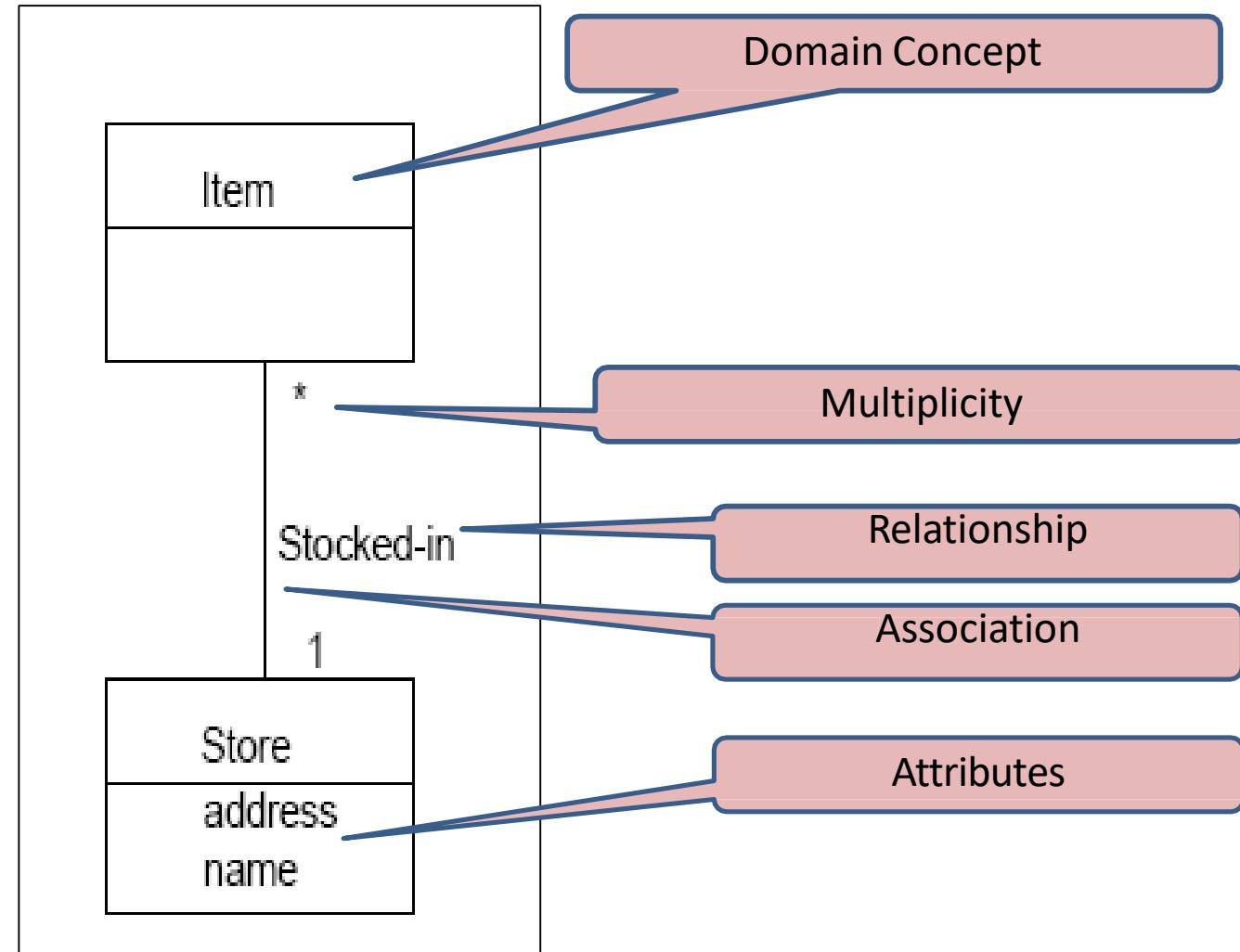
# Domain Model

- 
- 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



## **How Domain Model is represented in UML?**

# Domain Model in UML



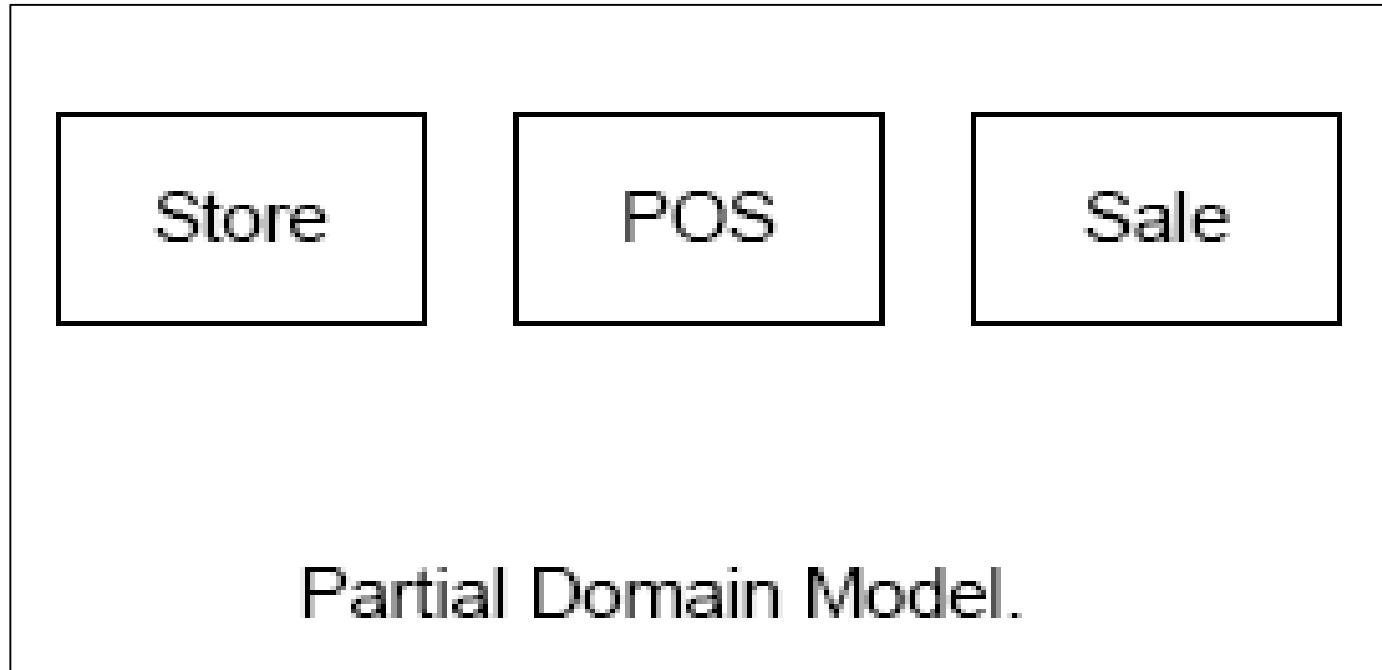


## **Identification of Domain Concepts from Use Case**

# 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



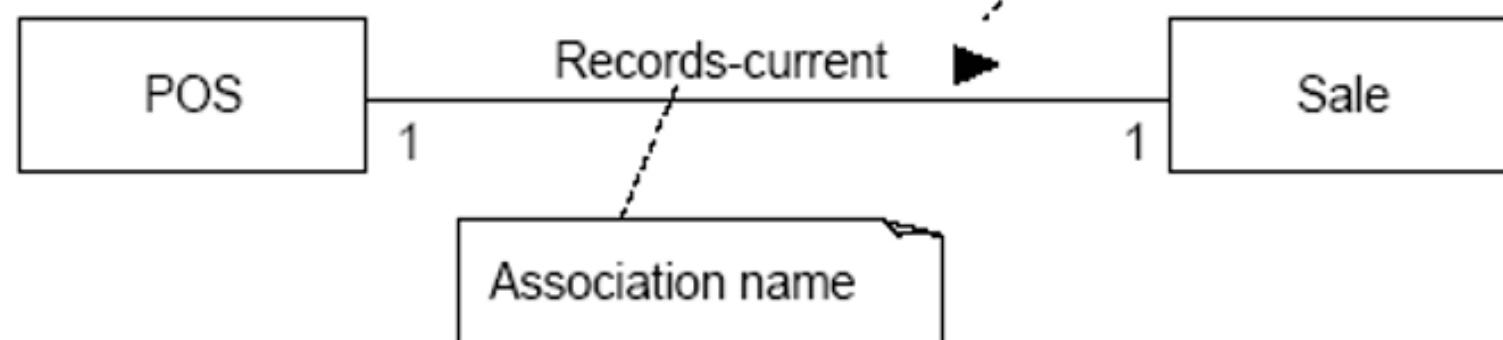


## **Identification of relationship among domain concepts**

# Adding Associations

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

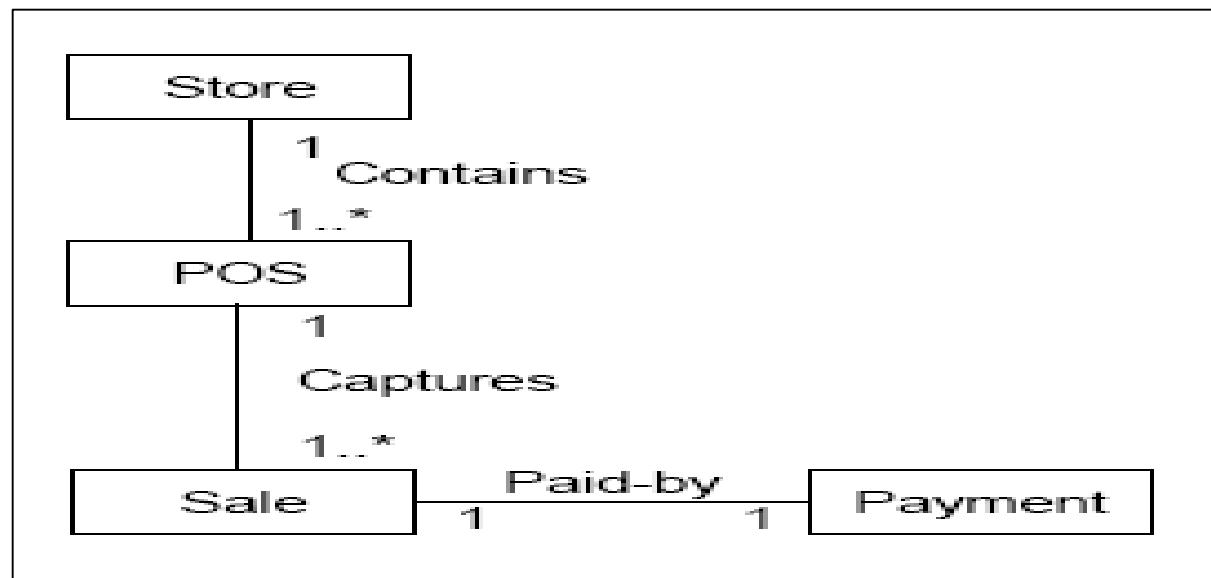
“Direction reading arrow” has no meaning other than to indicate direction of reading the association label.  
Optional (often excluded)



# Naming Associations



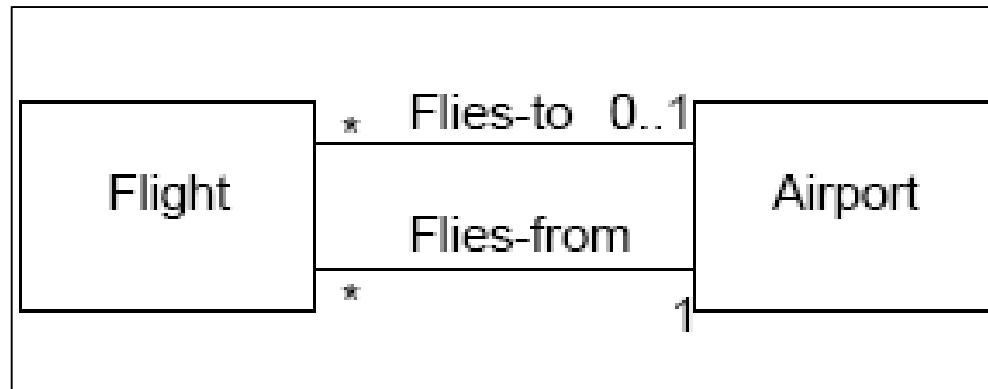
- Name an association based on a TypeName-VerbPhrase-TypeName format.
- Association names should start with a capital letter.
- A verb phrase should be constructed with hyphens.
- The default direction to read an association name is left to right, or top to bottom.



# 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.



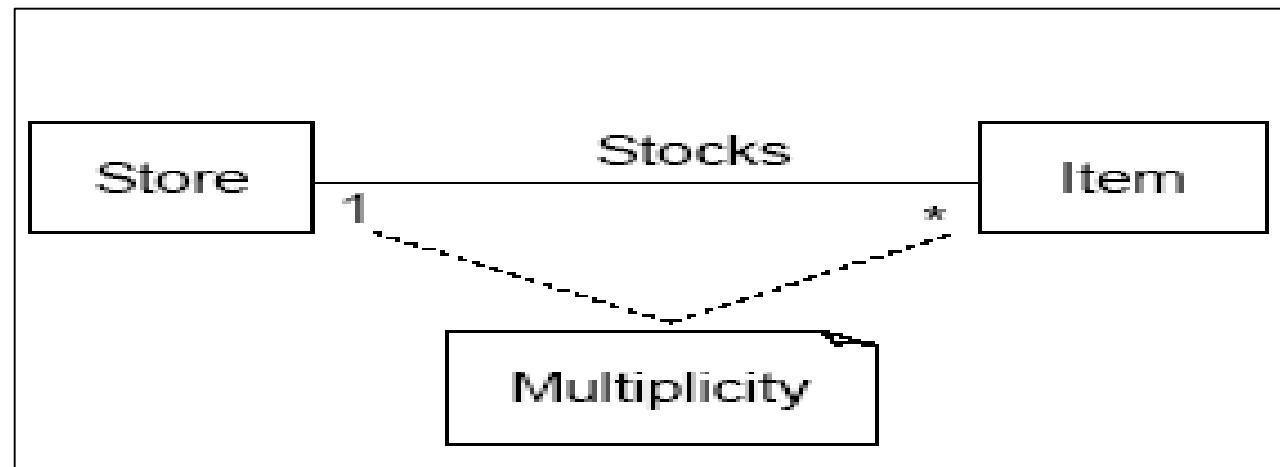


## Finding multiplicity among Domain Concepts

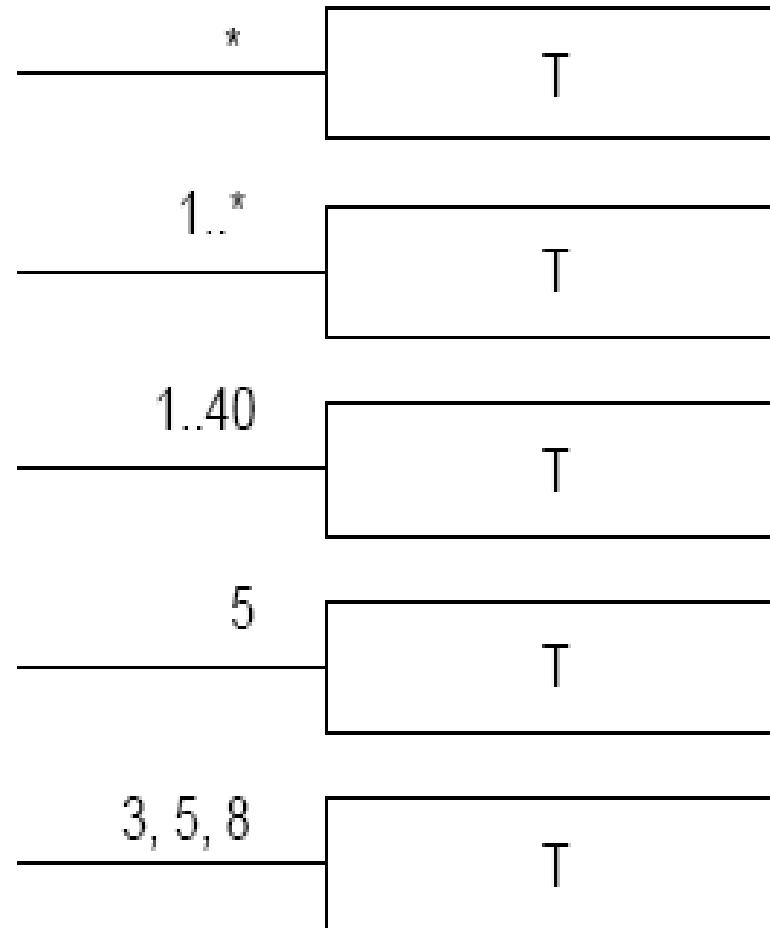
# Multiplicity



- 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.

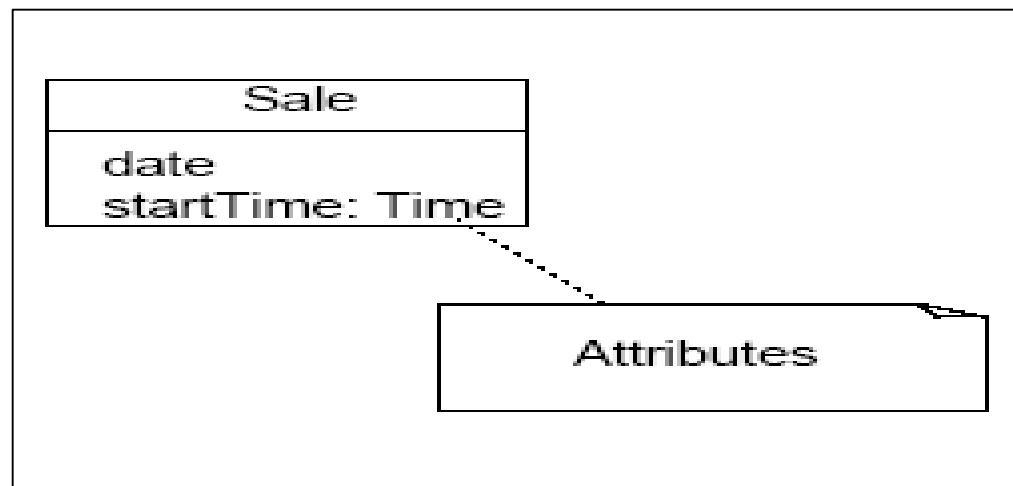


## **Adding attributes to Domain Model**

# Adding Attributes



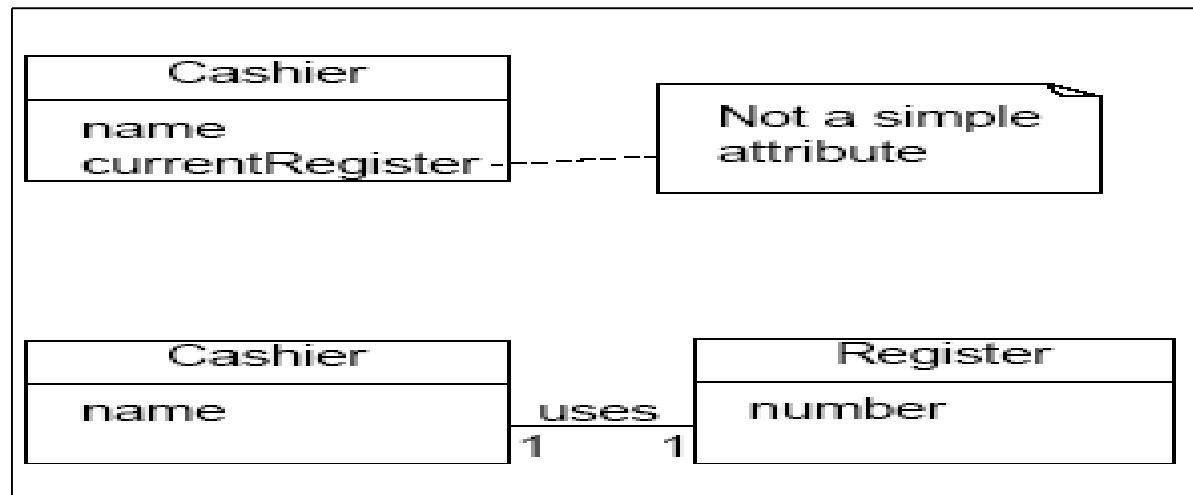
- An attribute is a logical data value of an object.
- Include the following attributes: those for which the requirements suggest or imply a need to remember information.
- For example, a Sales receipt normally includes a date and time.
- The Sale concept would need a date and time attribute.



# Valid Attribute Types



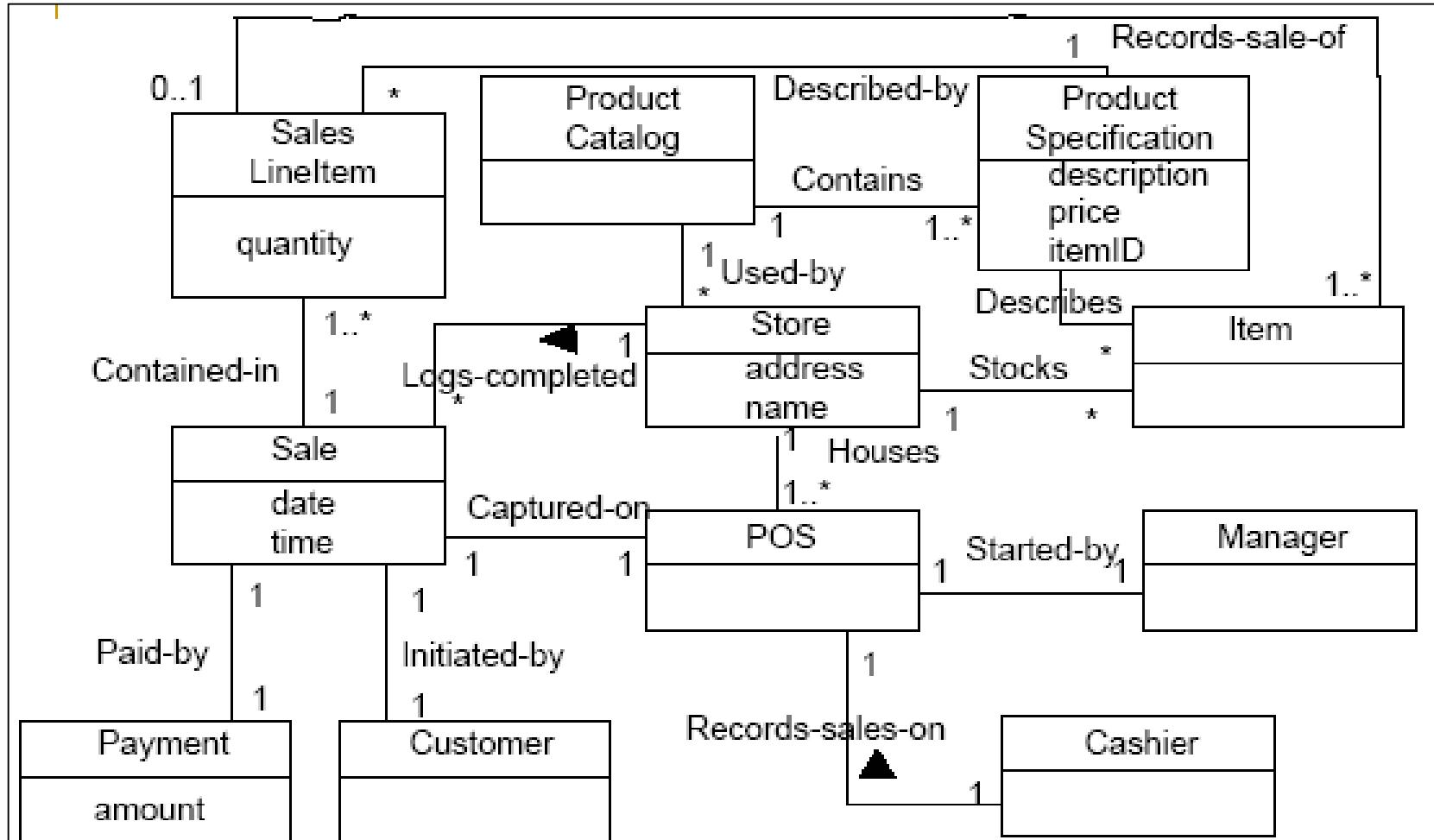
- Keep attributes simple.
- The type of an attribute should not normally be a complex domain concept, such as Sale or Airport.
- Attributes in a Domain Model should preferably be
  - Pure data values: Boolean, Date, Number, String, ...
  - Simple attributes: color, phone number, zip code, universal product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





## **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



## **What is SSD (System Sequence Diagram)**



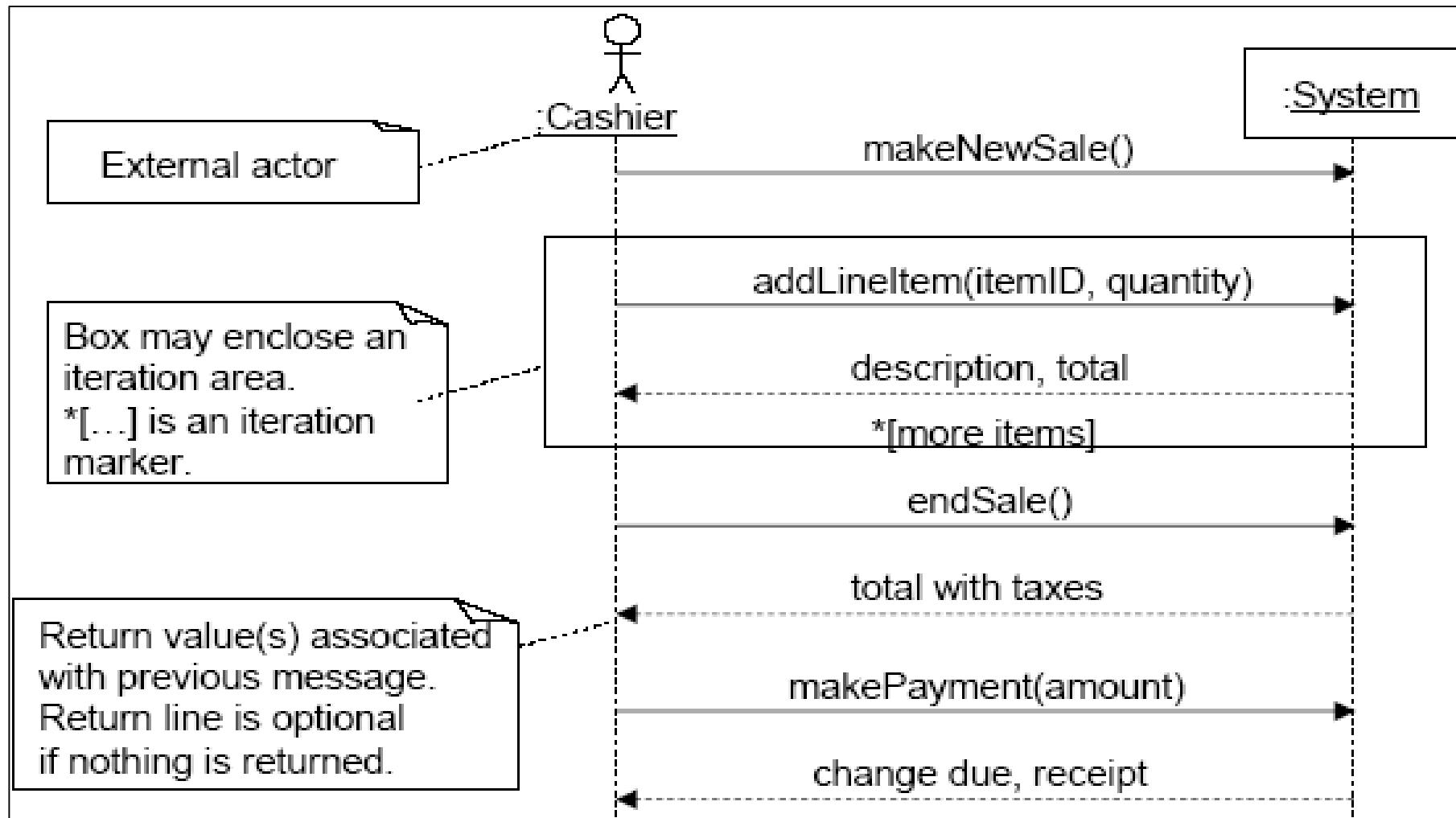
# What is SSD (System Sequence Diagram) ?

- 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.



## Drawing SSD for PoS

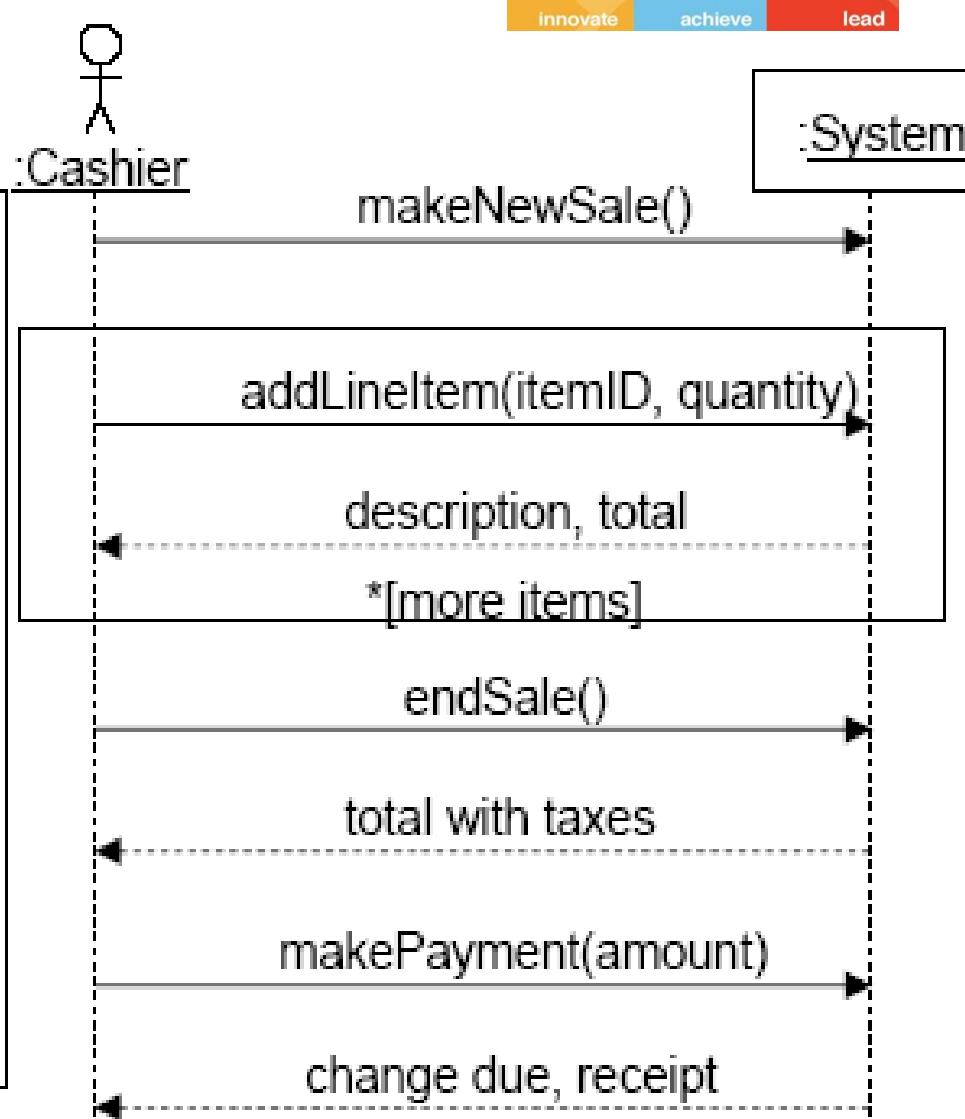
# SSDs for Process Sale Scenario



# SSD and Use Cases

## Simple cash-only Process Sale Scenario

1. Customer arrives at a POS checkout with goods to purchase.
  2. Cashier starts a new sale.
  3. Cashier enters item identifier.
  4. System records sale line item, and presents item description, price and running total.  
cashier repeats steps 3-4 until indicates done.
  5. System presents total with taxes calculated.
- ...



# 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)



## **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



## What is Operation Contracts

# Why Operation Contracts?



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



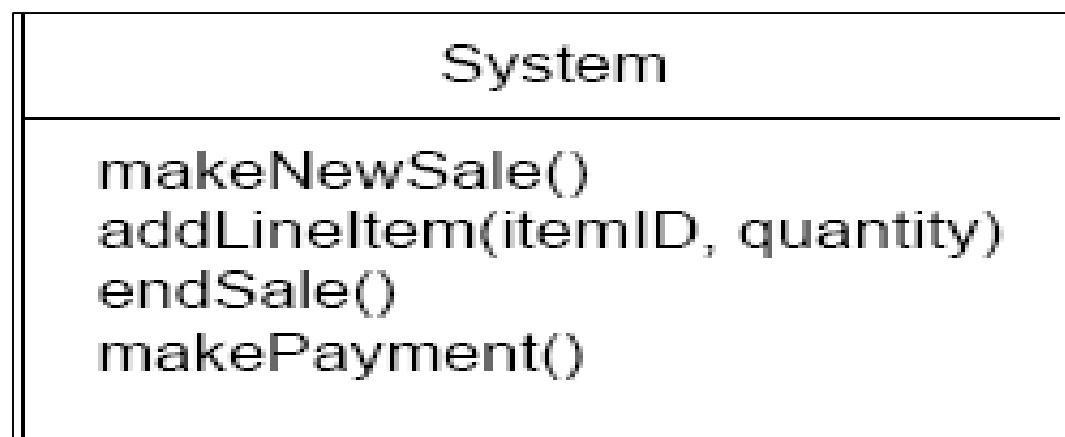
# What is Operation Contract?

- 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



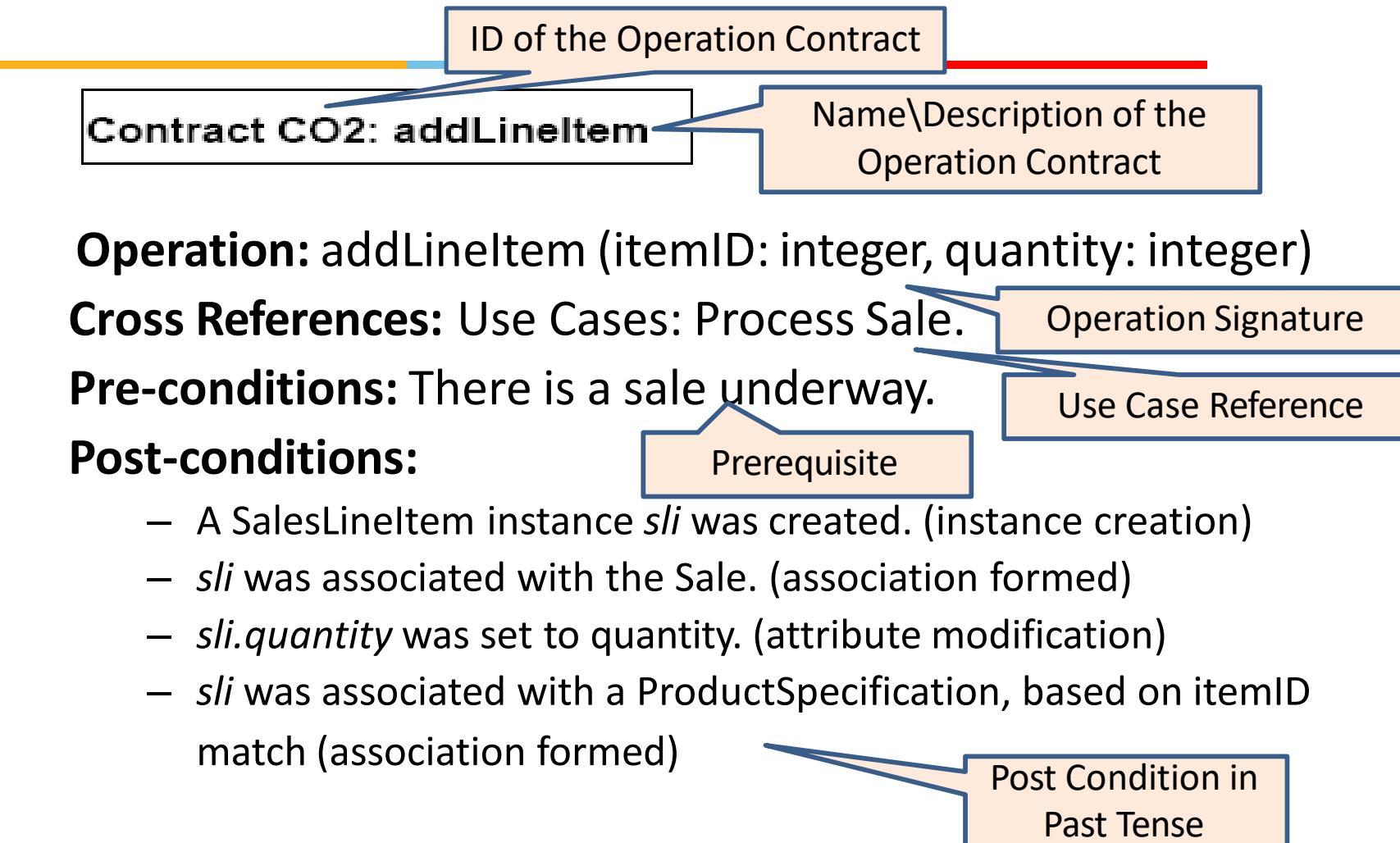
- In the UML the system as a whole can be represented as a class.
- Contracts are written for each system operation to describe its behavior.





## Represent Operation Contract in UML

# Operation Contract in UML





# **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



**BITS** Pilani

Pilani|Dubai|Goa|Hyderabad

Paramananda Barik



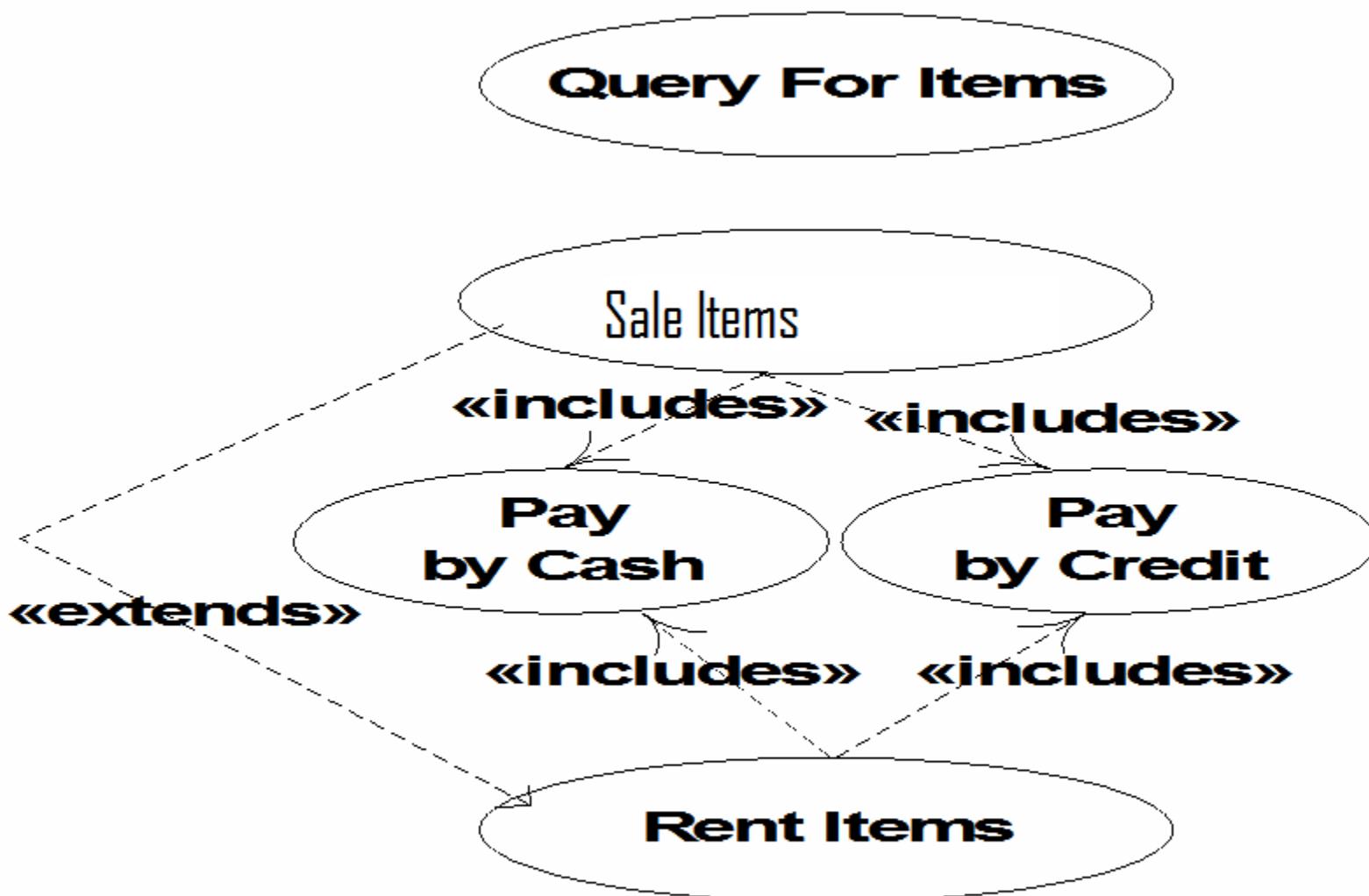
**Relating Use Cases : includes, extends relationships**



# 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**

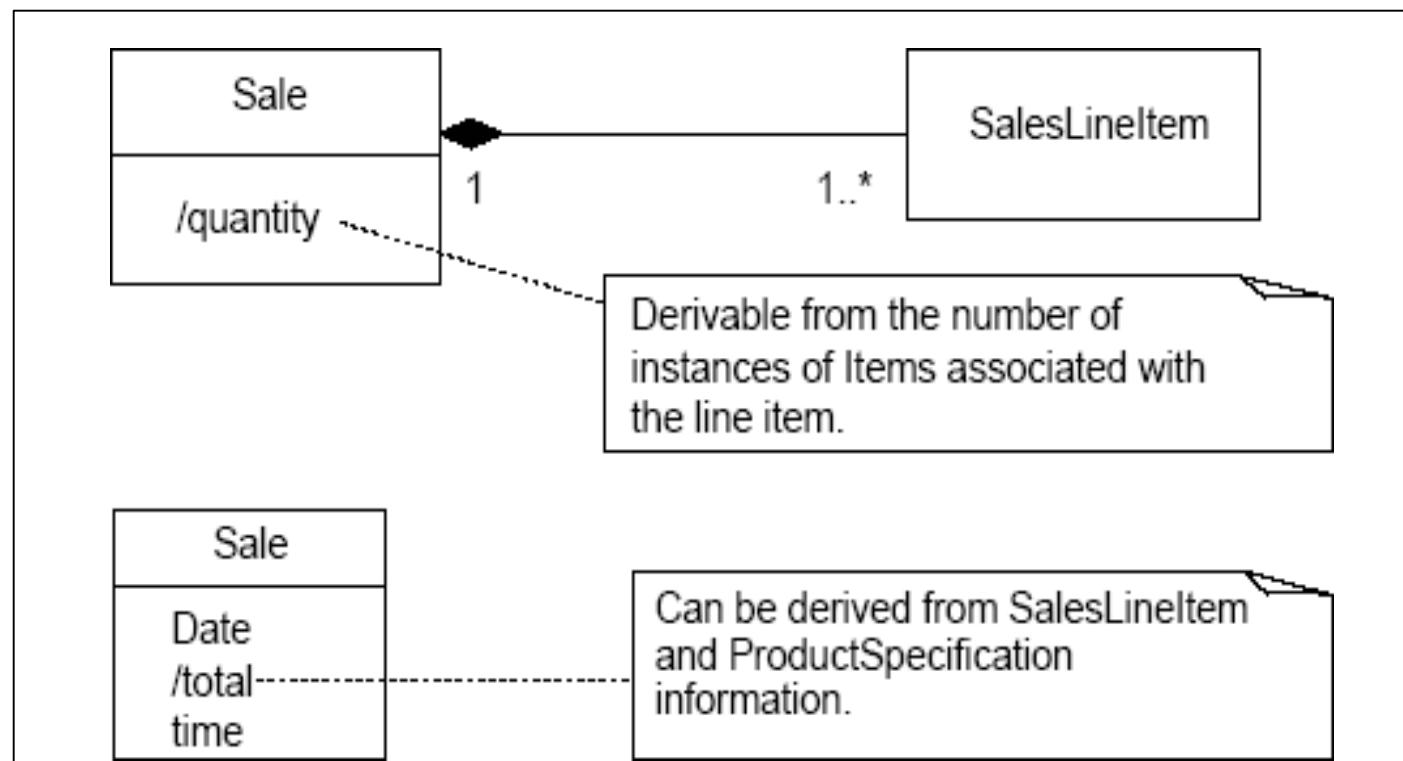




## **Refining Domain Model : Derived Attributes**

# Derived Elements

- A derived element can be determined from others.





## **Refining Domain Model : Association Classes**

# Association Classes

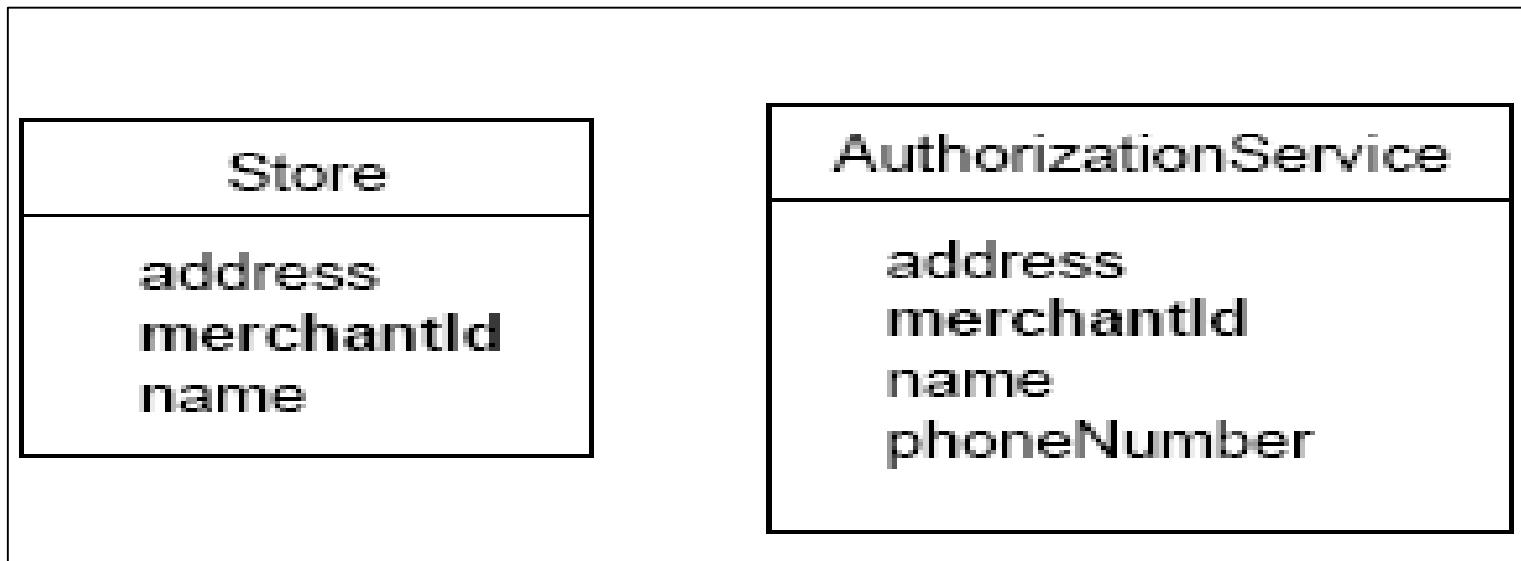
---

- 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

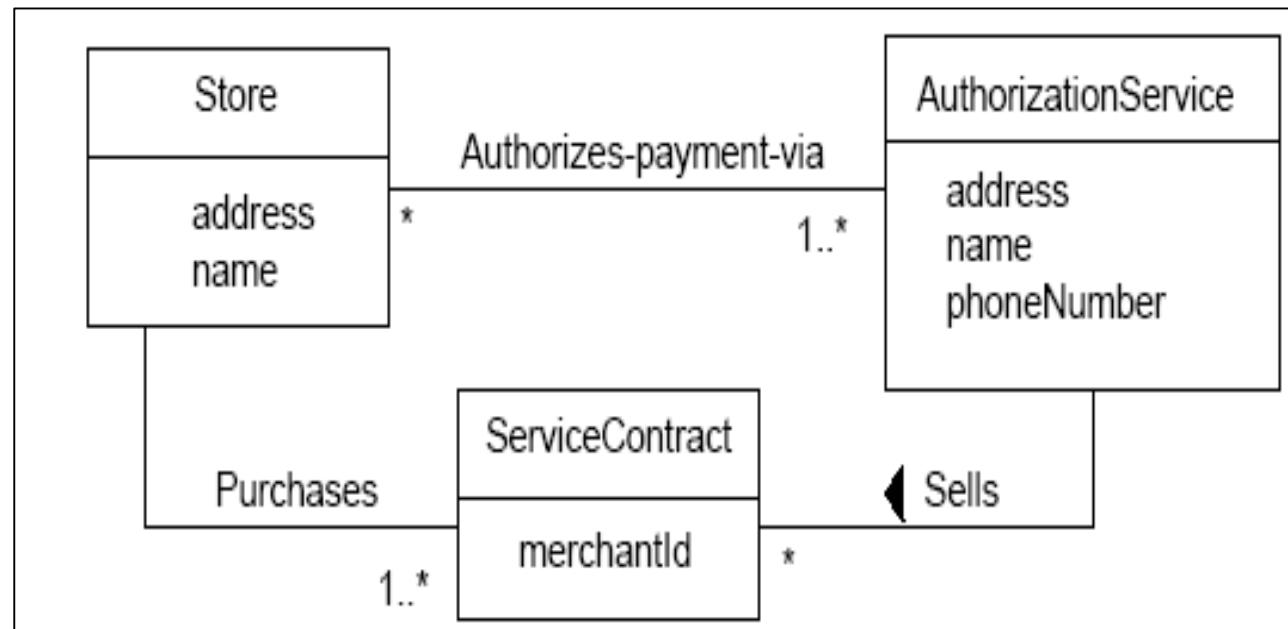
---

- 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



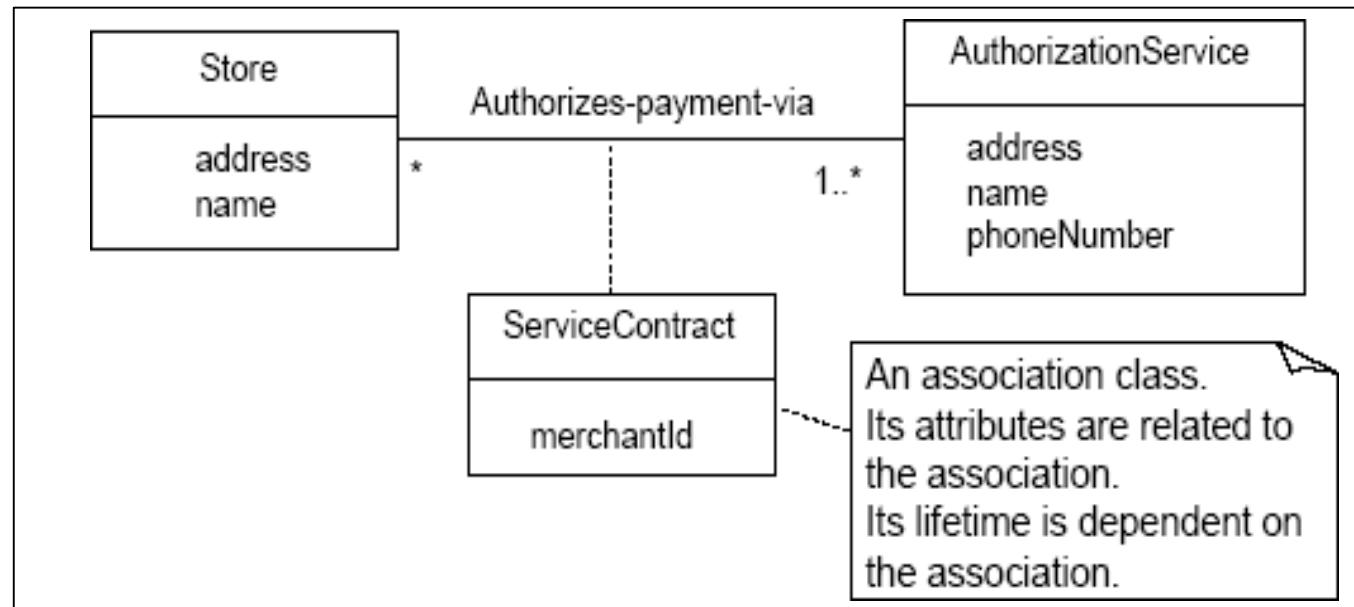
# Association Classes

- 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.



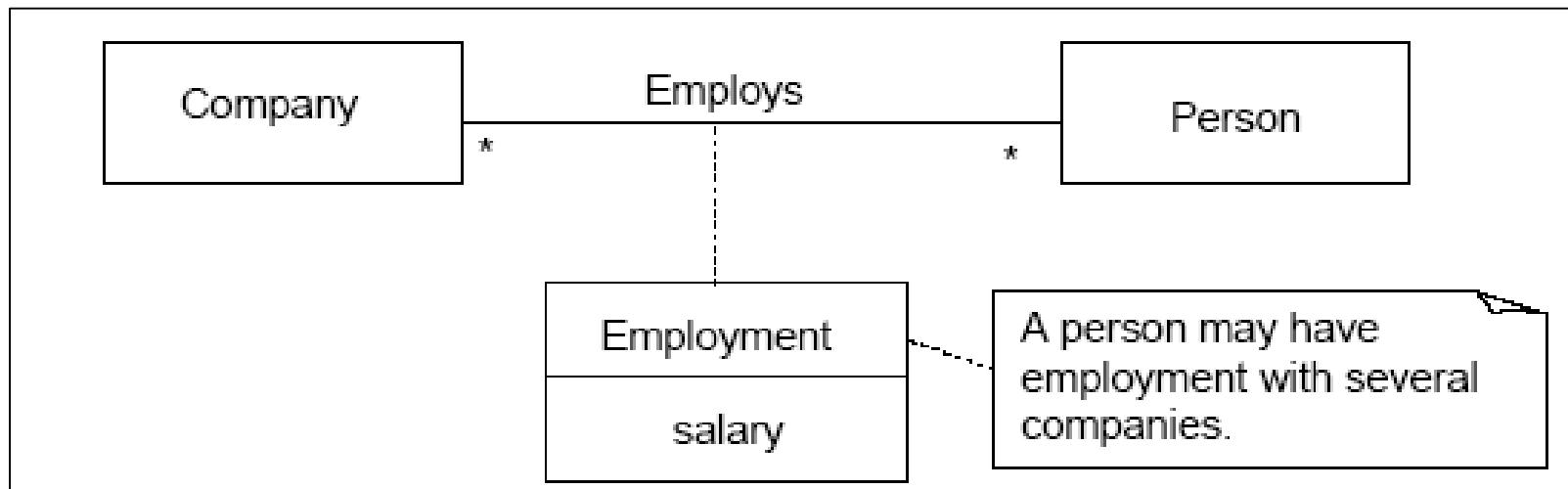
# Association Classes

- 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.





## 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)?*



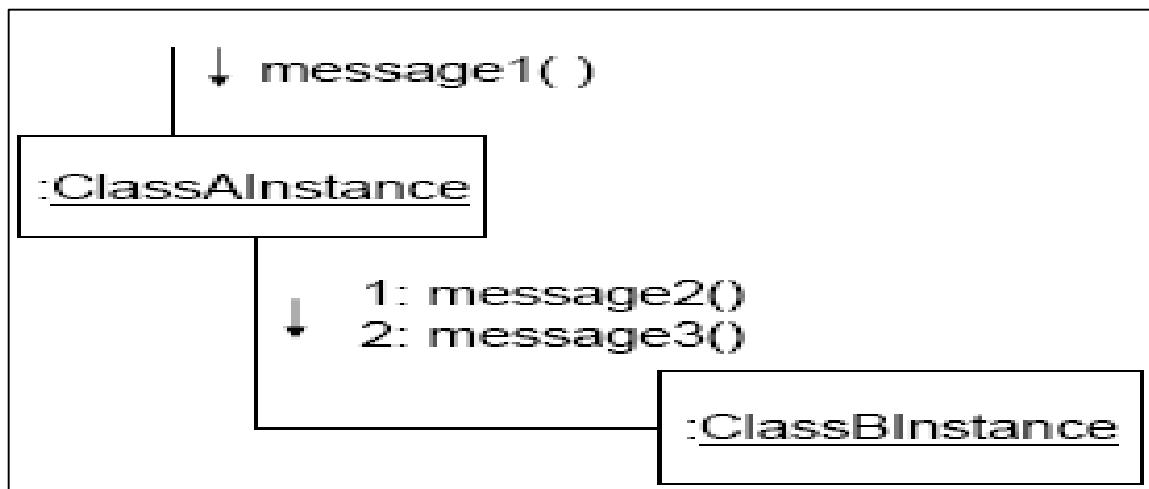
## Types of Interactions Diagram

# Types of Interaction Diagrams



- *Two Types : Collaboration Diagram & Sequence Diagram*

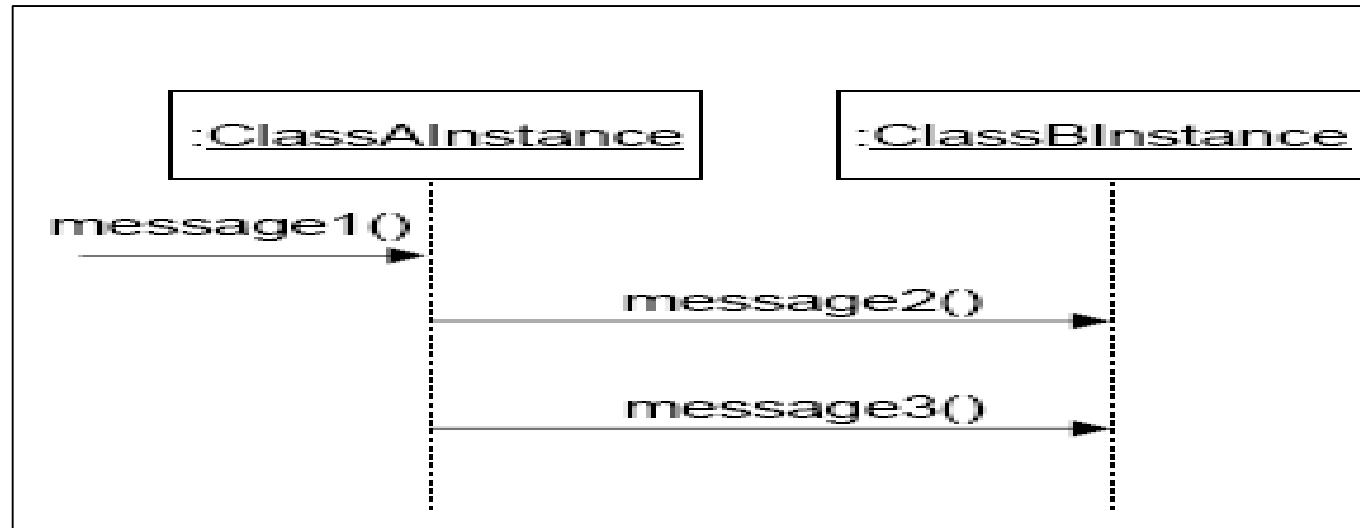
*Collaboration diagrams* illustrate object interactions in a graph or network format.



# Types of Interaction Diagrams



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





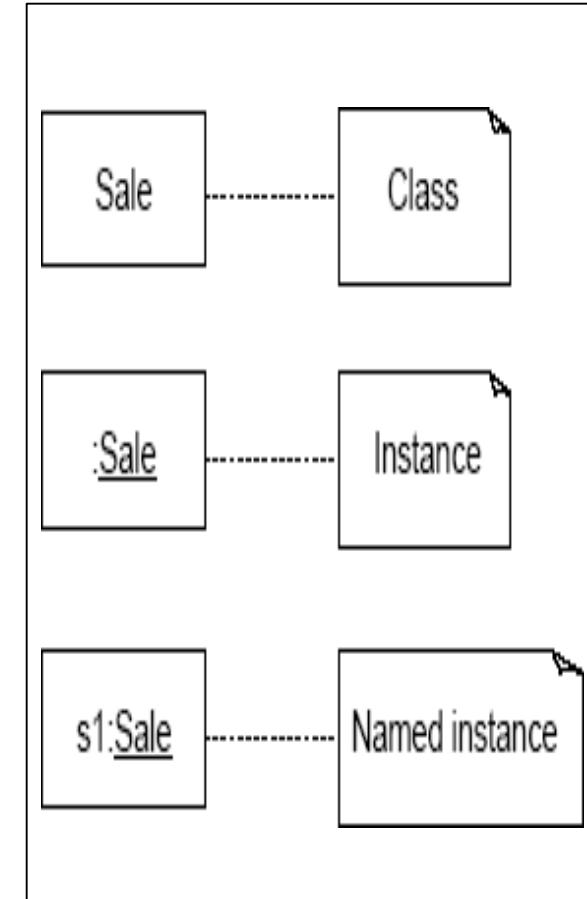
# Representation of Interaction Diagrams in UML

# Illustrating Classes and Instances

---

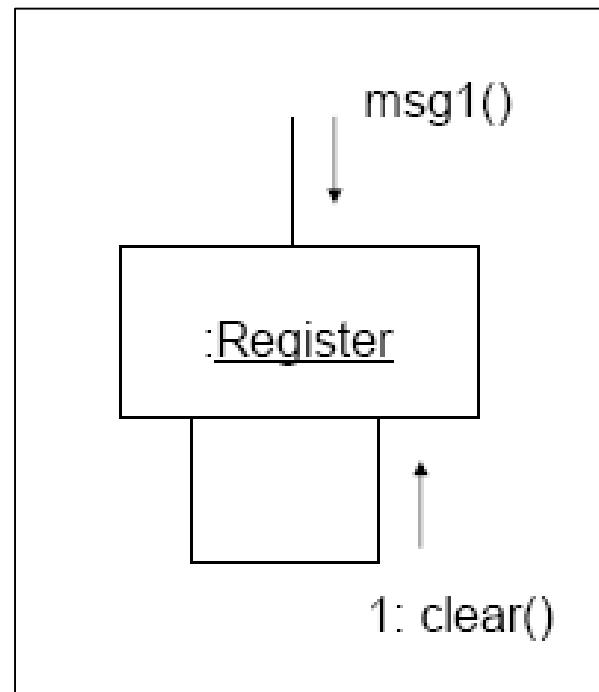


- To show an instance of a class, the regular class box graphic symbol is used, but the name is underlined.
- Additionally a class name should be preceded by a colon.
- An instance name can be used to uniquely identify the instance.



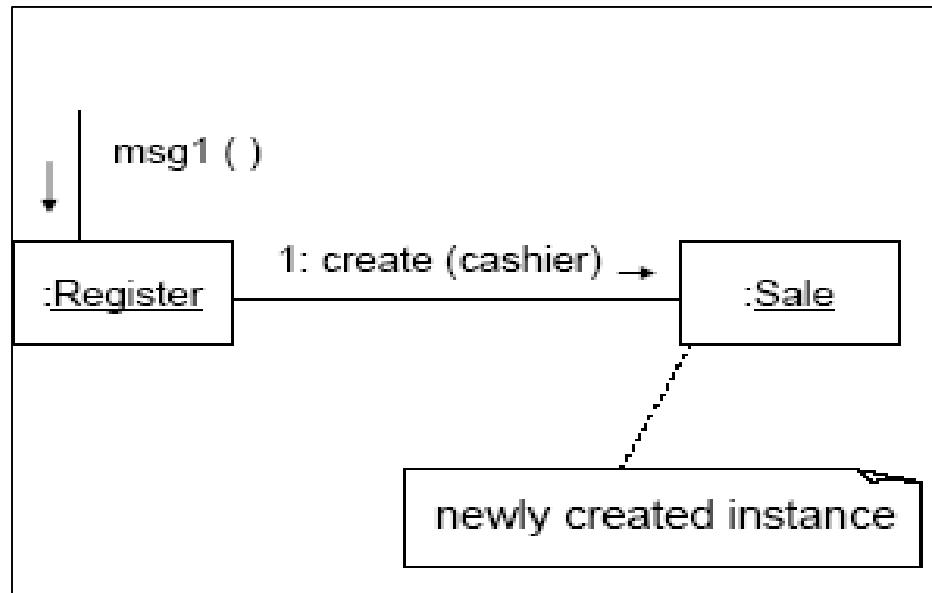
# Messages to “self” or “this”

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



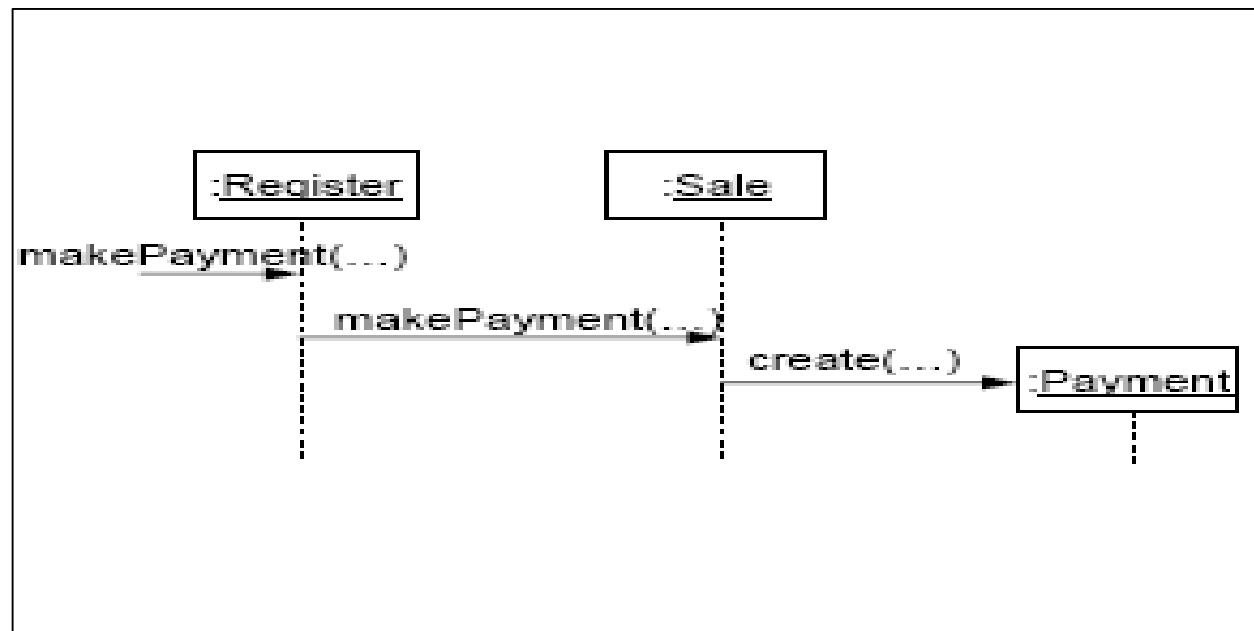
# Creation of Instances

- 
- 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.



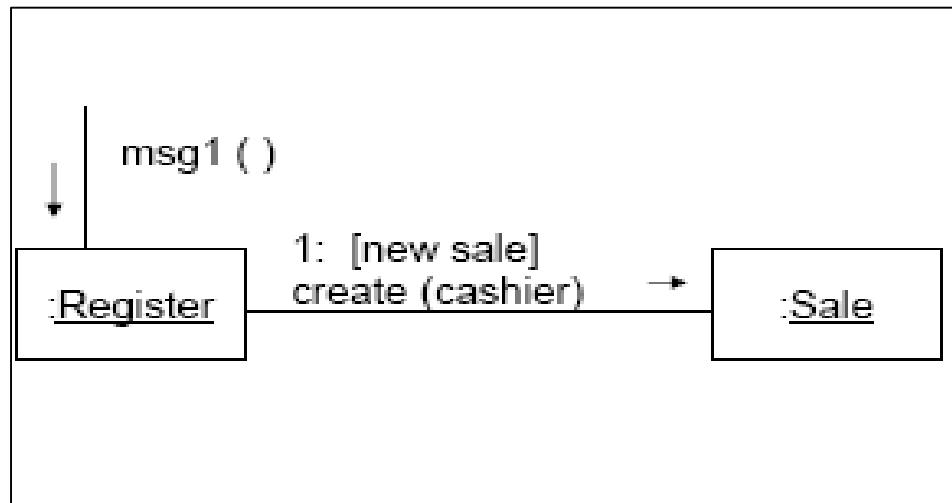
# Creation of Instances

- 
- 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.

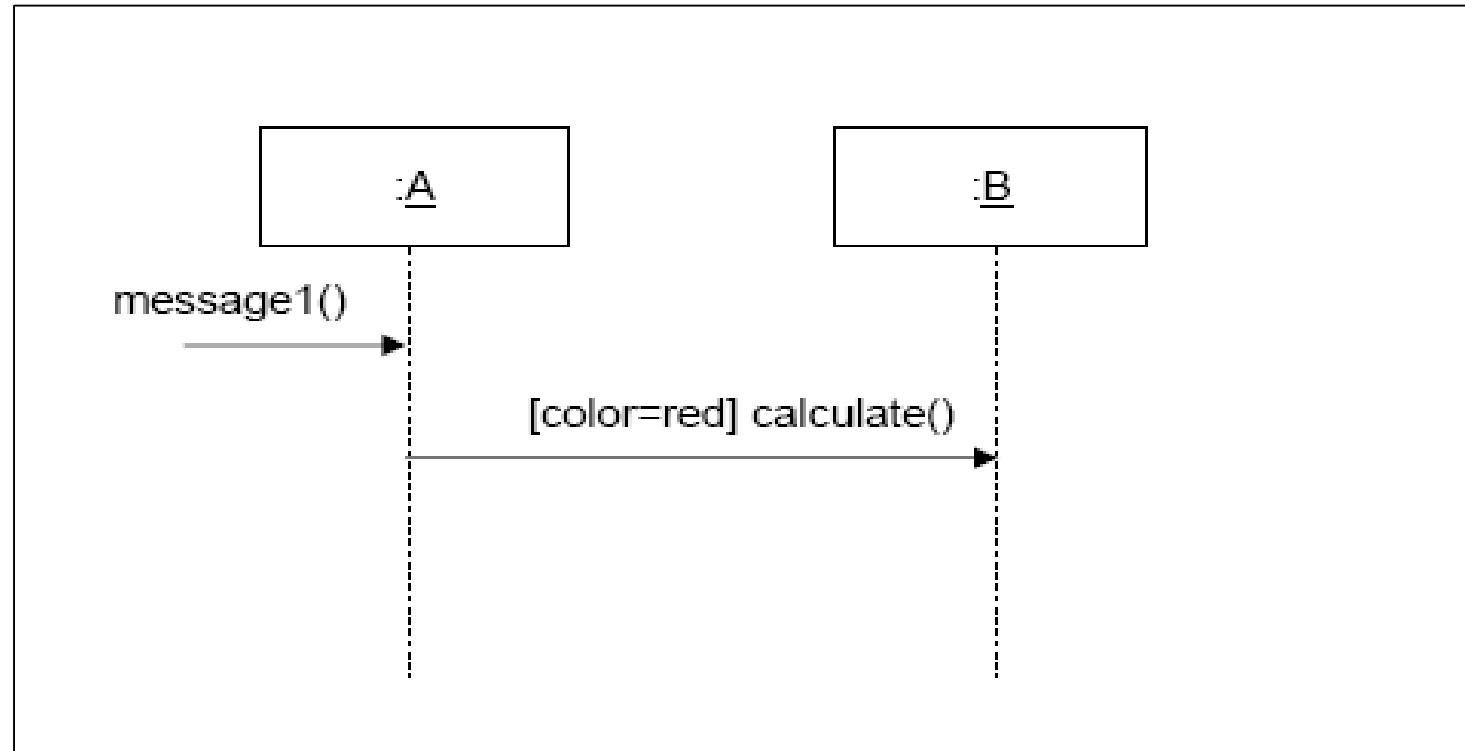


# Conditional Messages

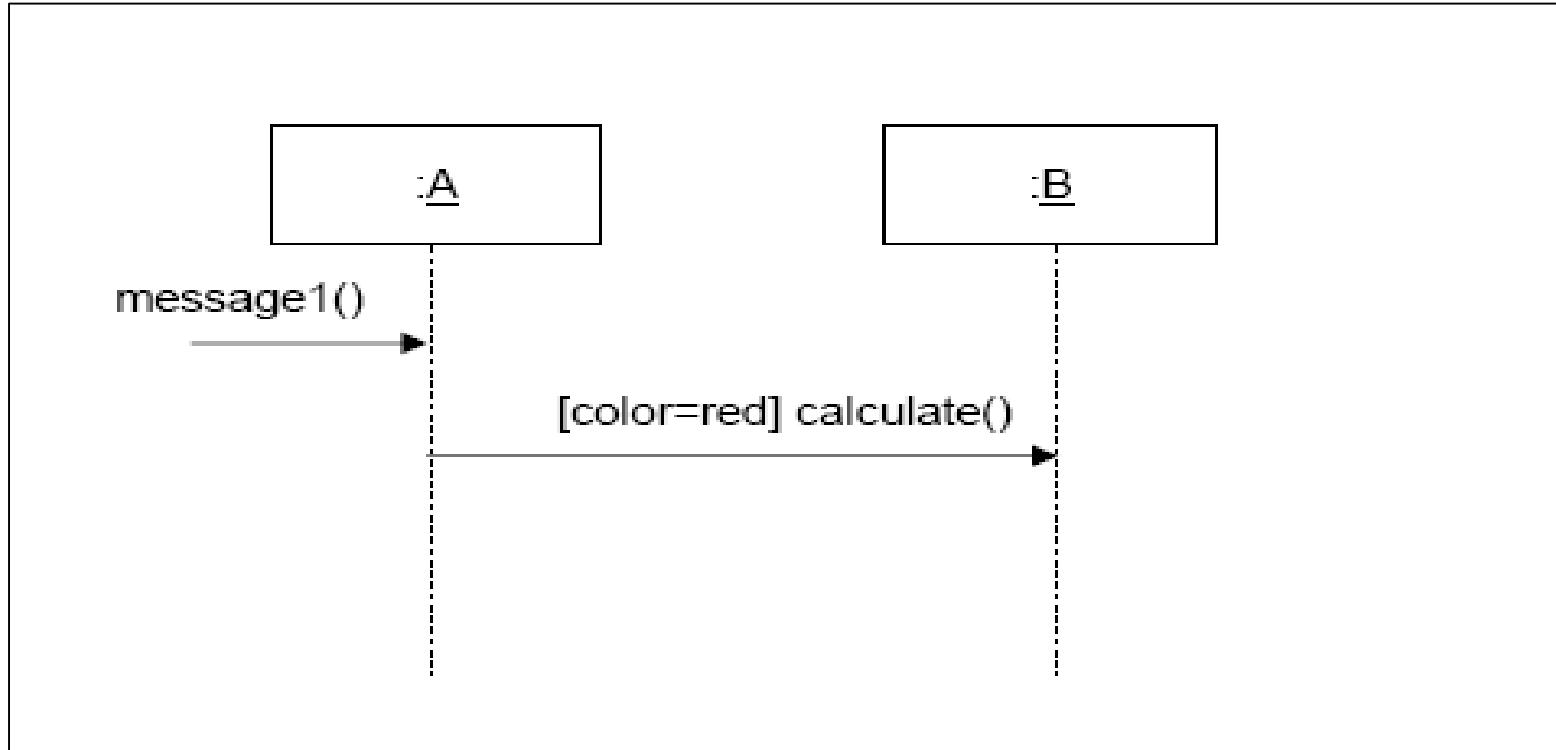
- 
- A conditional message is shown by following a sequence number with a conditional clause in square brackets, similar to the iteration clause.
  - The message is sent only if the clause evaluates to true.



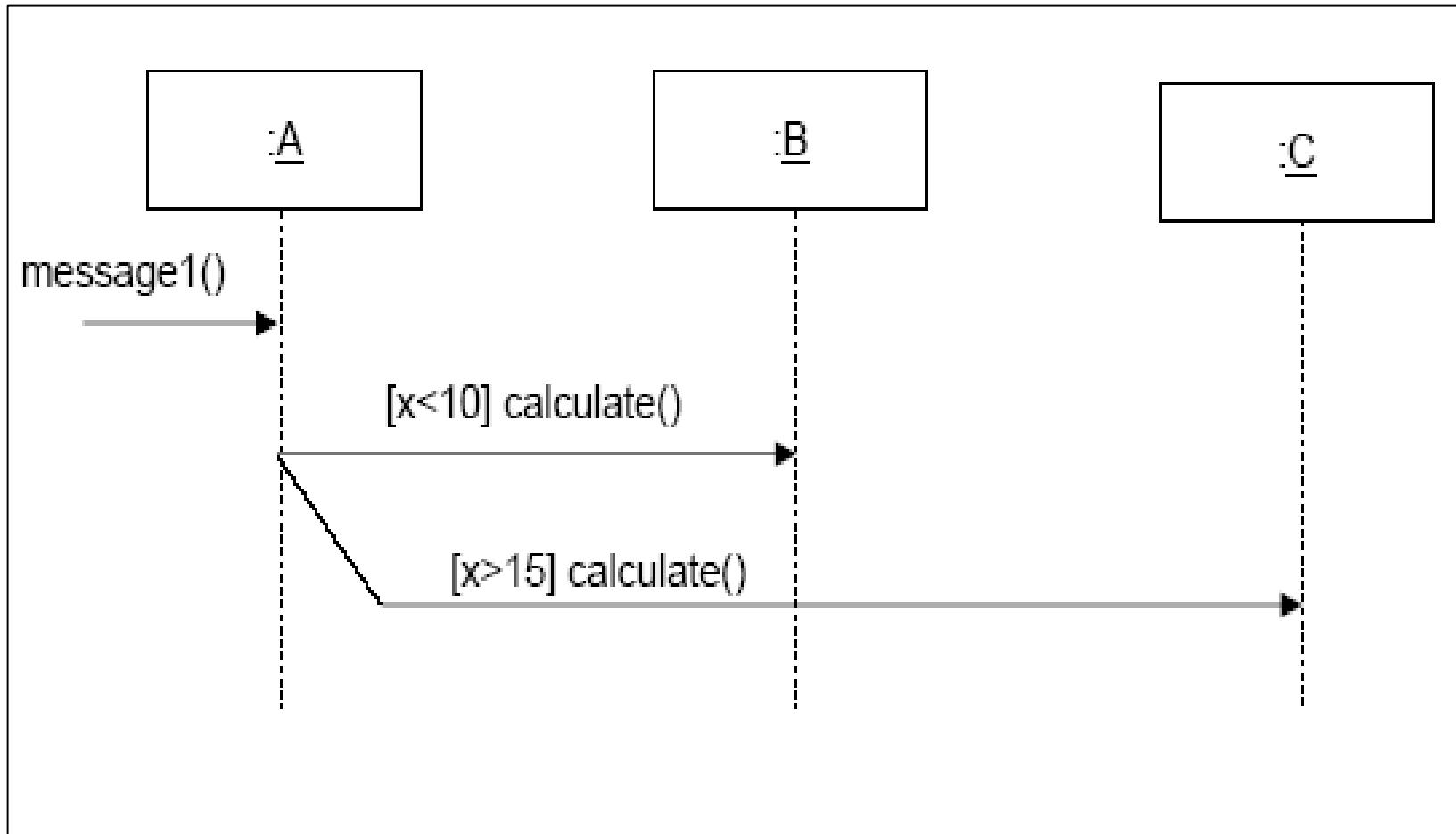
# Conditional Messages



# Mutually Exclusive Conditional Paths

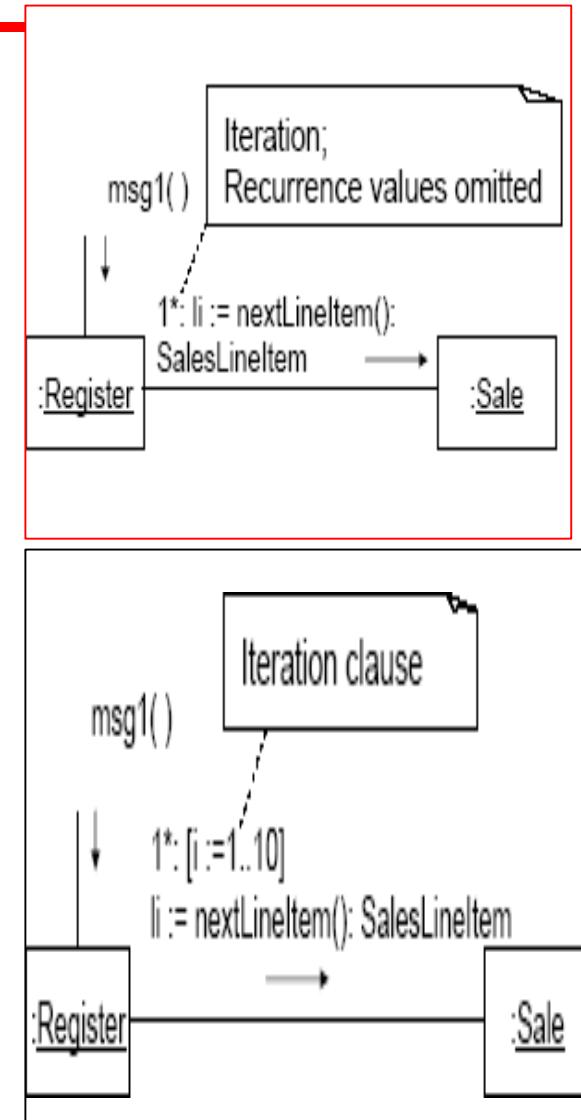


# Mutually Exclusive Conditional Messages



# Iteration or Looping

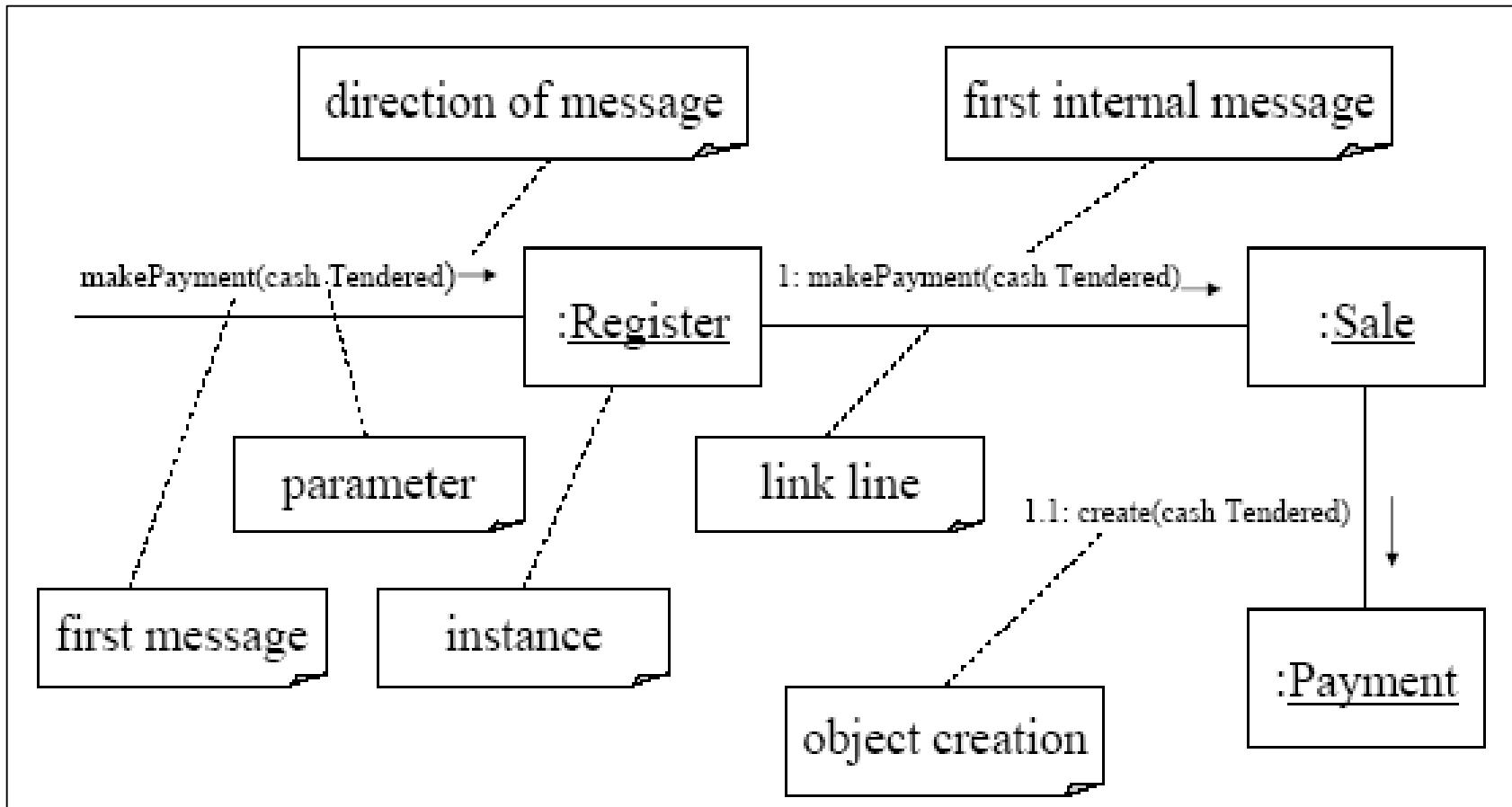
- Iteration is indicated by following the sequence number with a star \*
- This expresses that the message is being sent repeatedly, in a loop, to the receiver.
- It is also possible to include an iteration clause indicating the recurrence values.



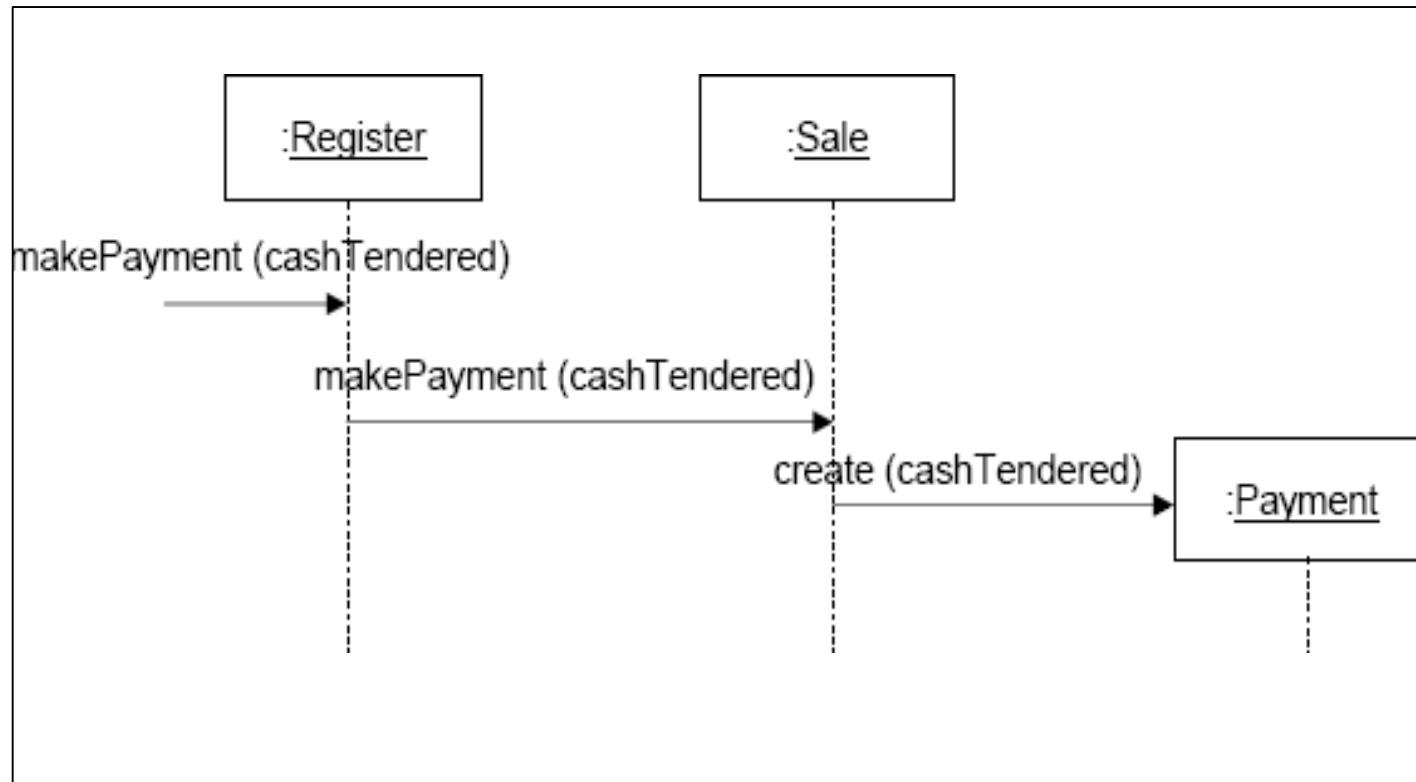


# Drawing Interaction Diagrams for PoS

# Collaboration Diagram: makePayment



# Sequence Diagram : makePayment





# 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

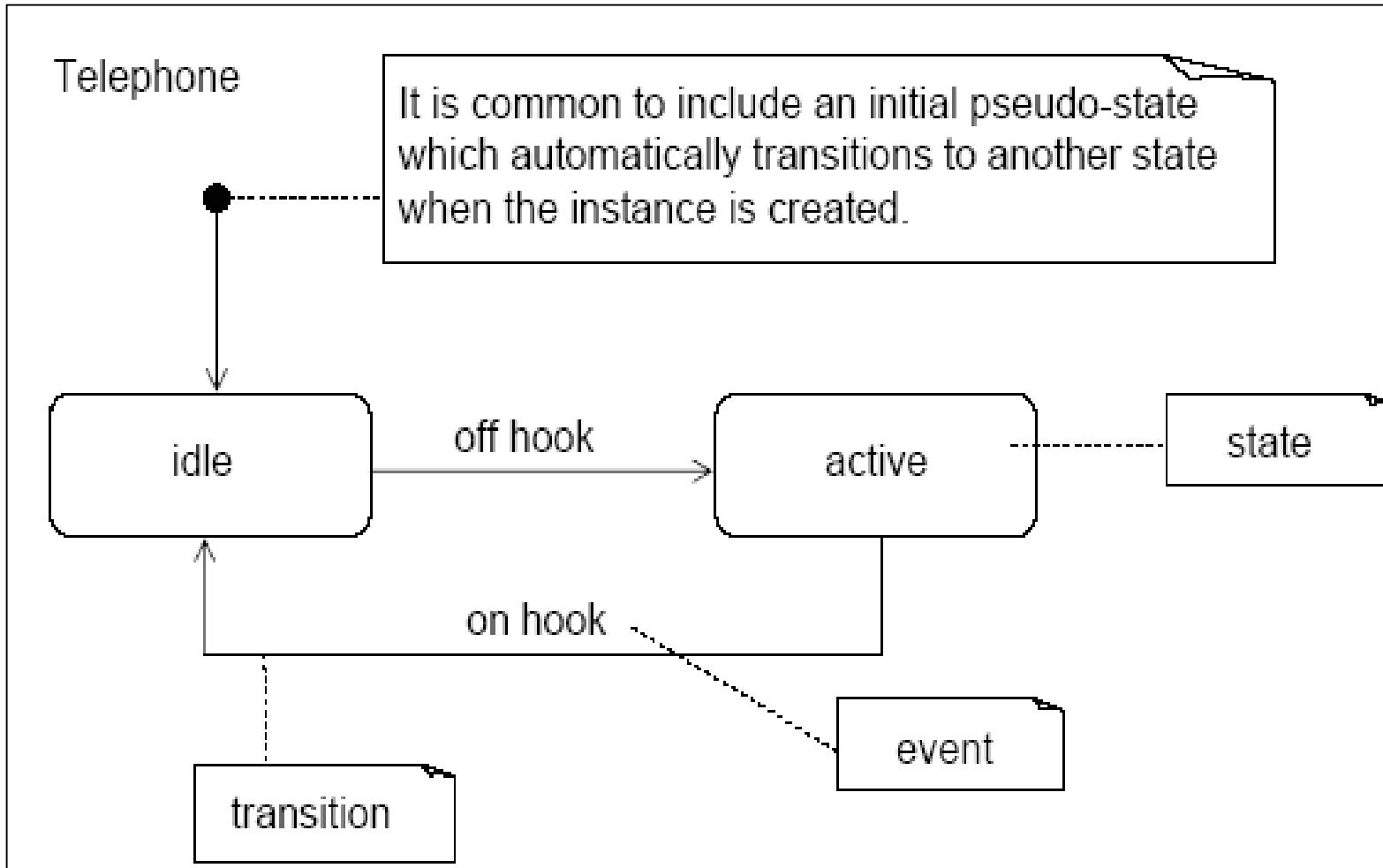


- A statechart diagram shows the life-cycle of an object; what events it experiences, its transitions and the states it is in between events.
- A state diagram need not illustrate every possible event; if an event arises that is not represented in the diagram, the event is ignored as far as the state 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.

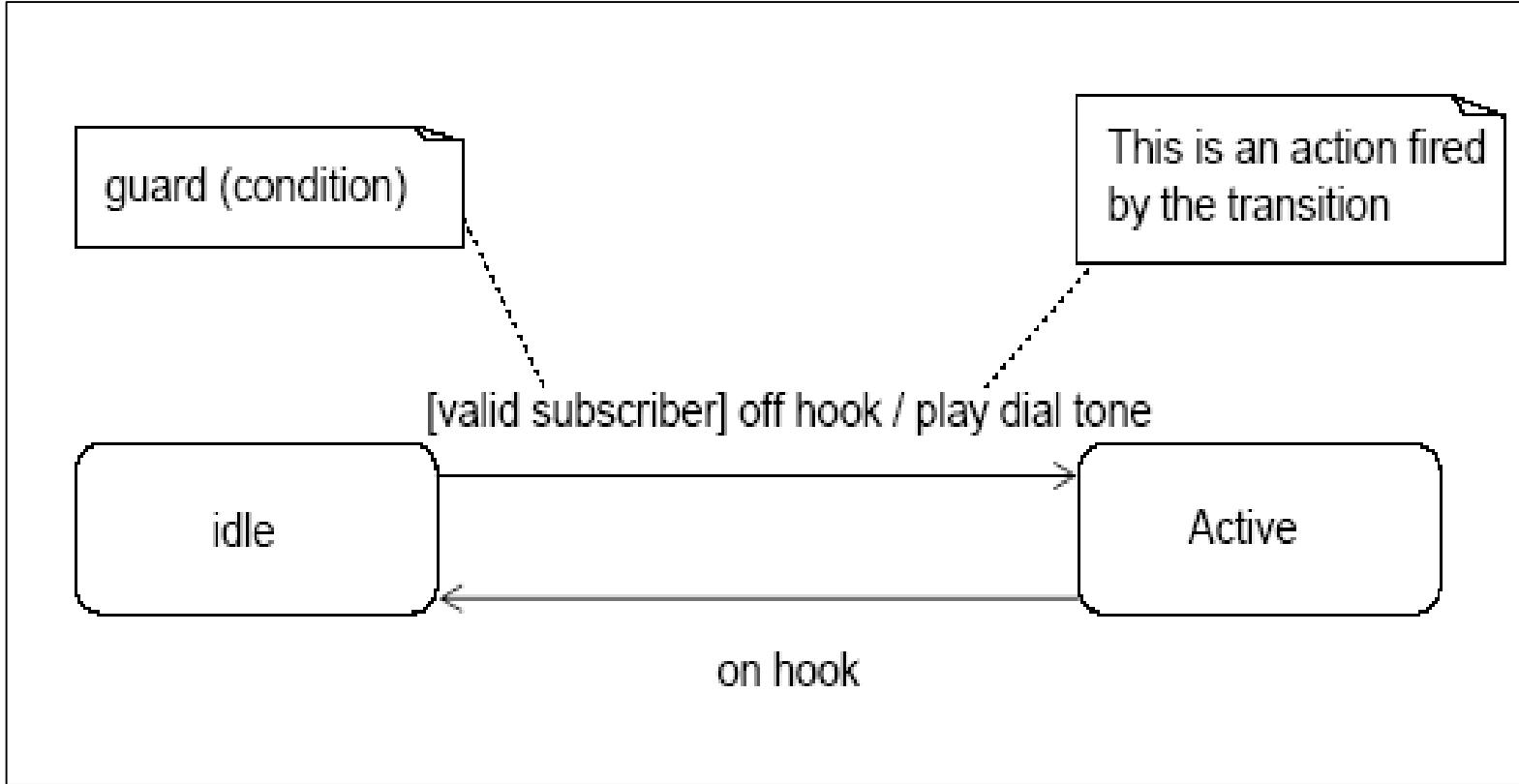


# Representing State Transition Diagram in UML

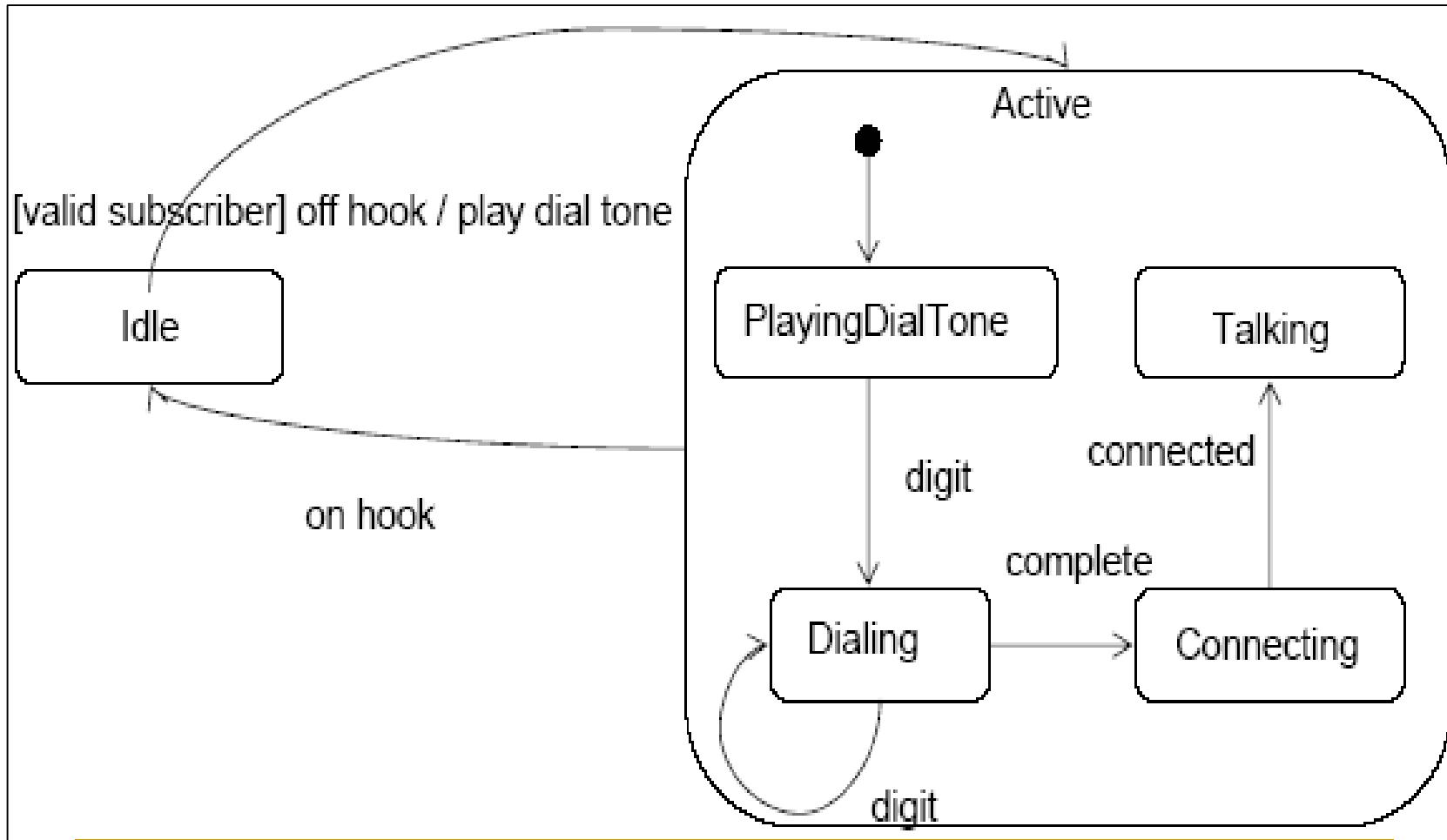
# State Transition Diagram in UML



# Additional State Transition Diagram Notation



# Additional State Transition Diagram Notation



# Additional State Transition Diagram Notation

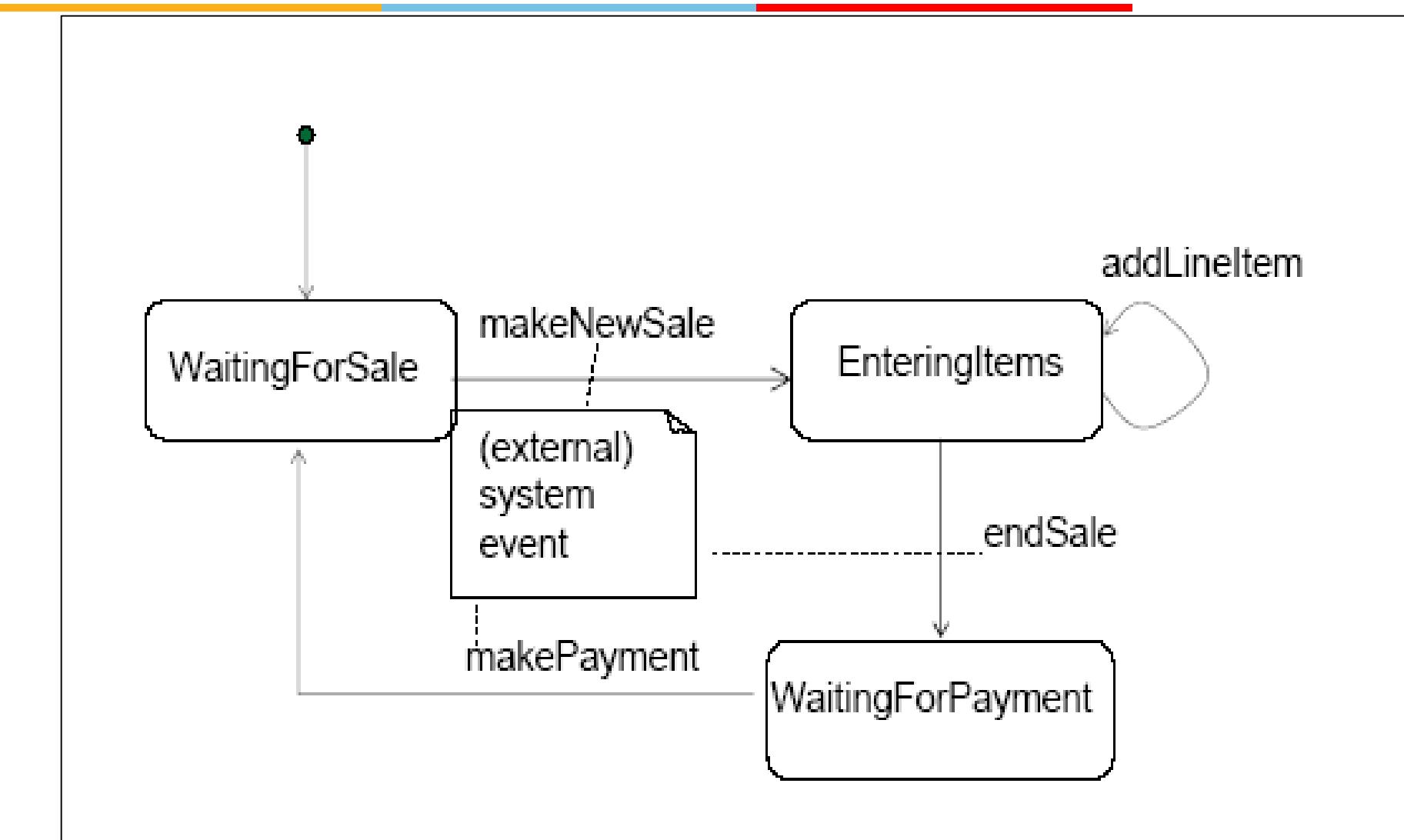


- A state allows nesting to contain substates. A substate inherits the transitions of its superstate (the enclosing state).
- Within the Active state, and no matter what substate the object is in, if the on hook event occurs, a transition to the idle state occurs.



## Drawing State Transition diagram for PoS

# State Transition Diagram for PoS





## 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.



## Representing Activity Diagram in UML

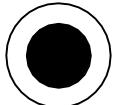
# Activity Diagram in UML



- Initial state

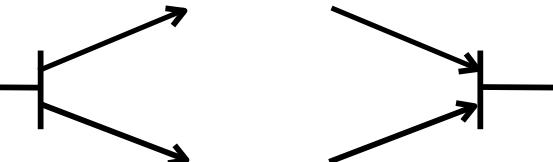


- Final state



- Fork and join

  - model concurrent flows



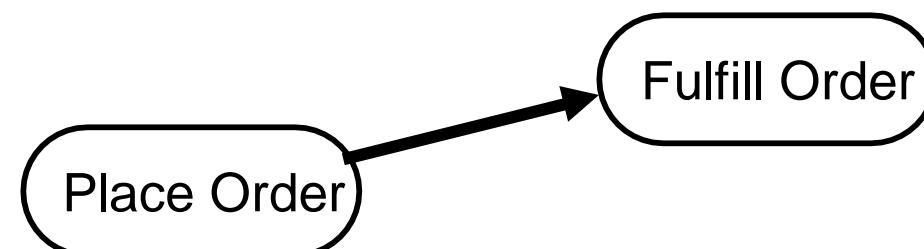
- Activities

  - Step in overall Process



- Transitions

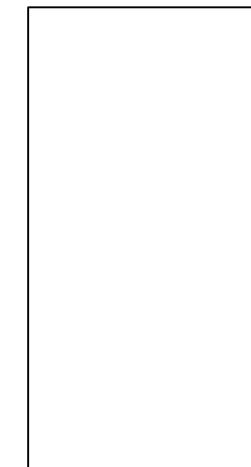
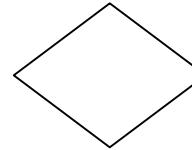
  - Triggered by end of previous activity and initiates next activity



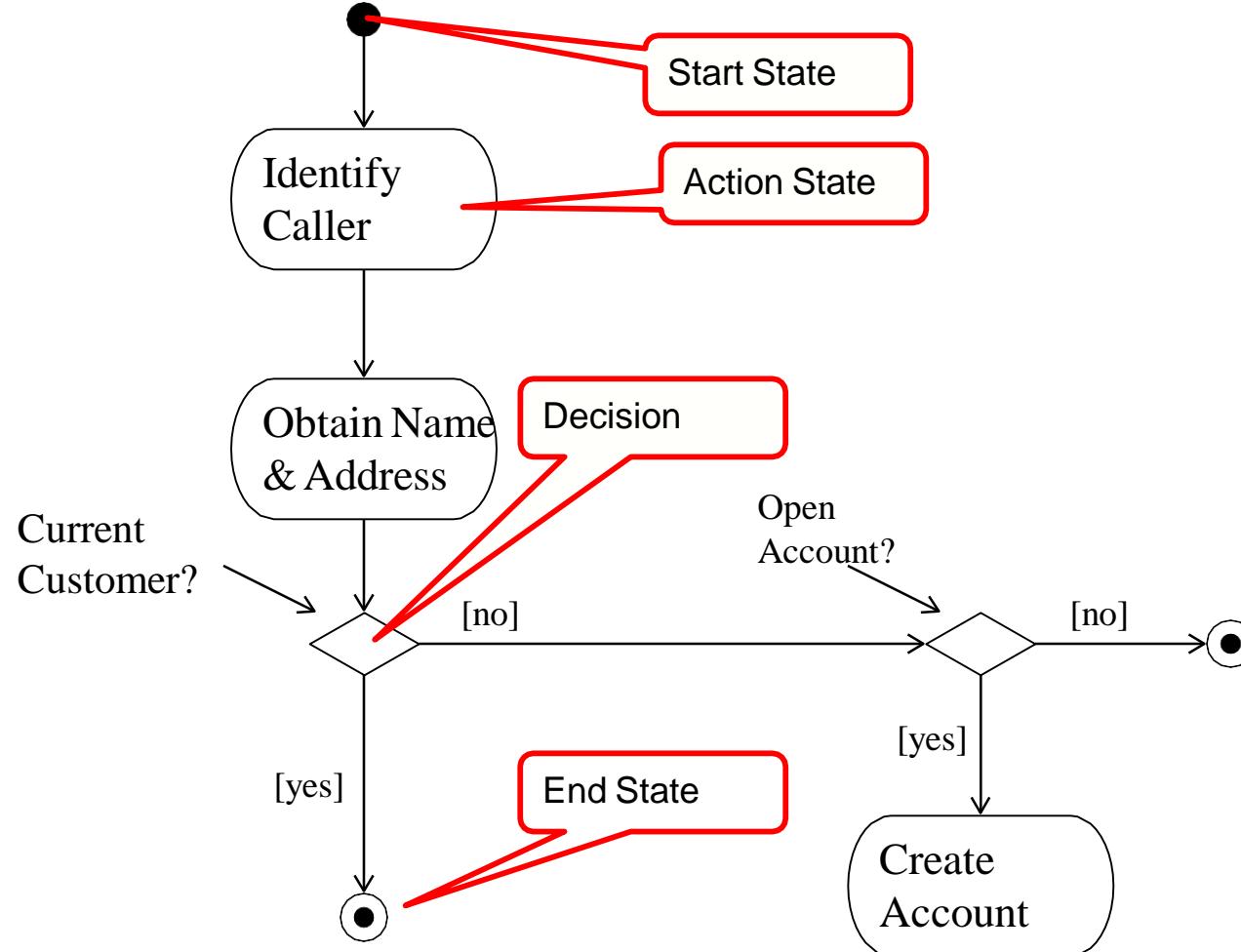
# Activity Diagram in UML



- 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)

Paramananda Barik

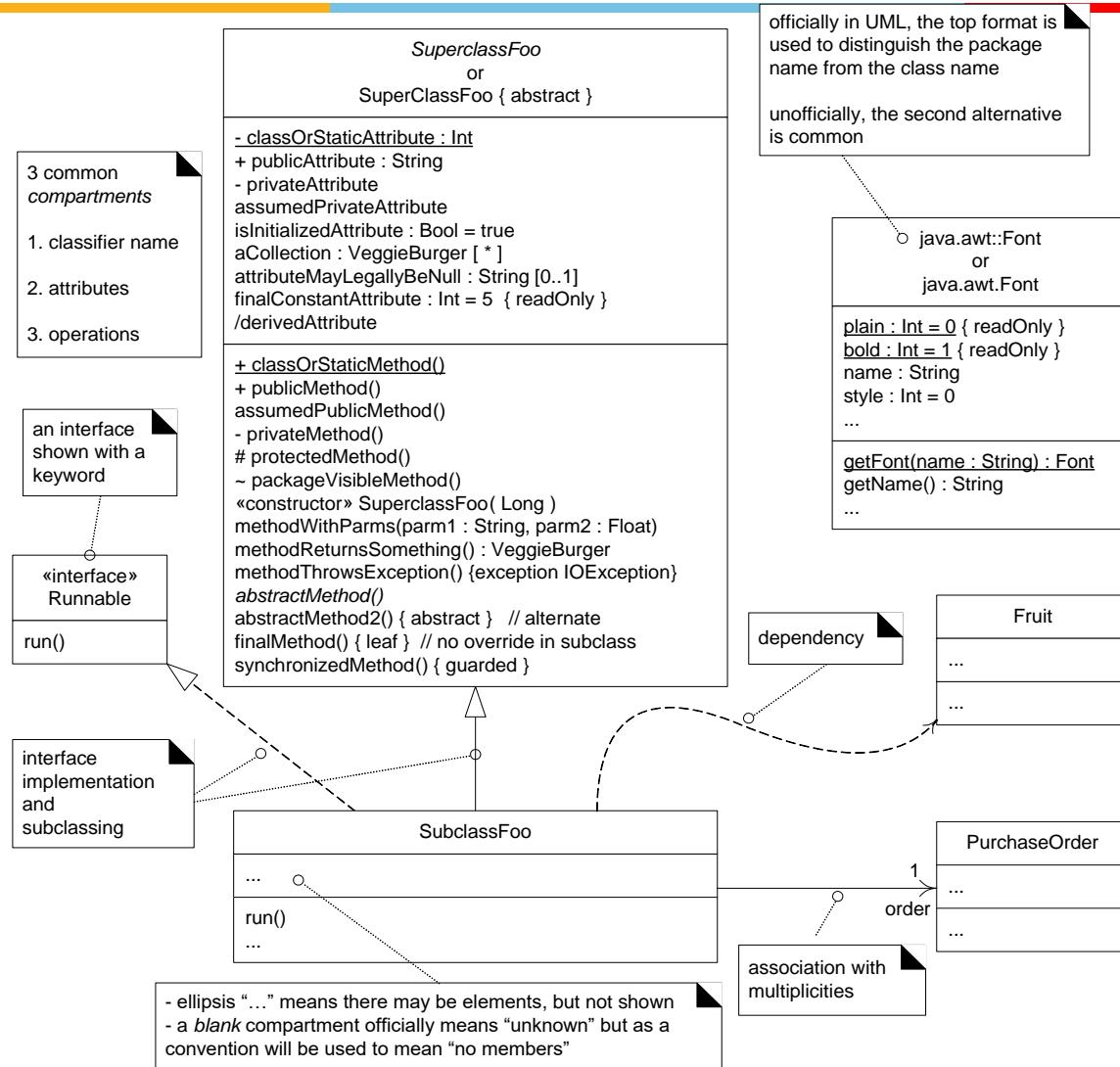


**BITS** Pilani  
Pilani Campus



## What is Visibility among Objects

# Visibility in UML Class Diagrams



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



## **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:

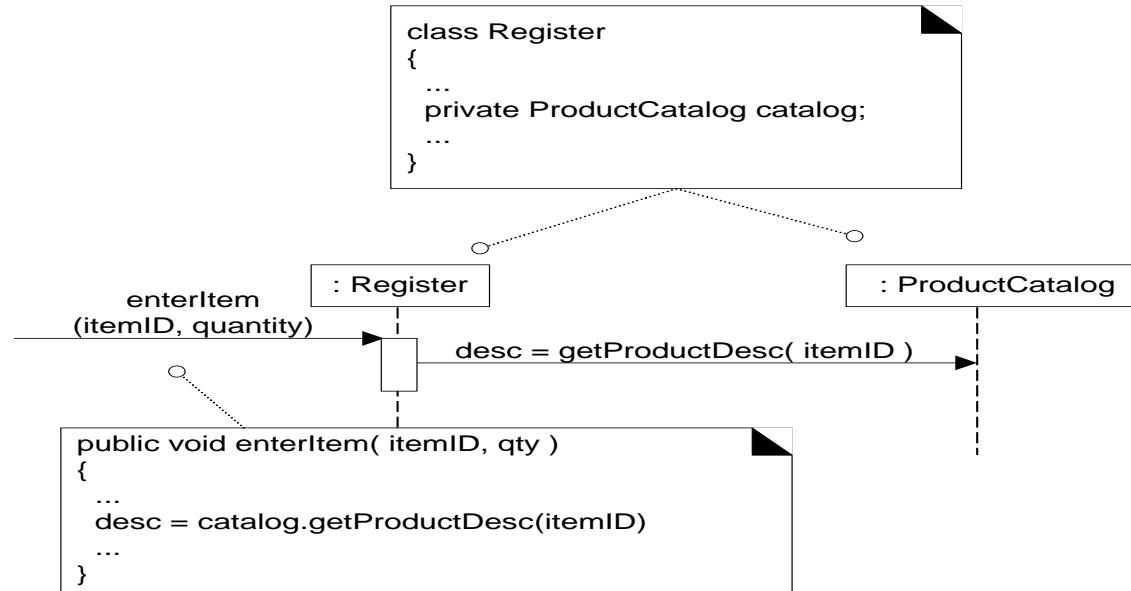
- + (public),
- - (private),
- ~ (package), and
- # (protected)

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.



## **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.



## **Attribute Visibility**

# Attribute Visibility

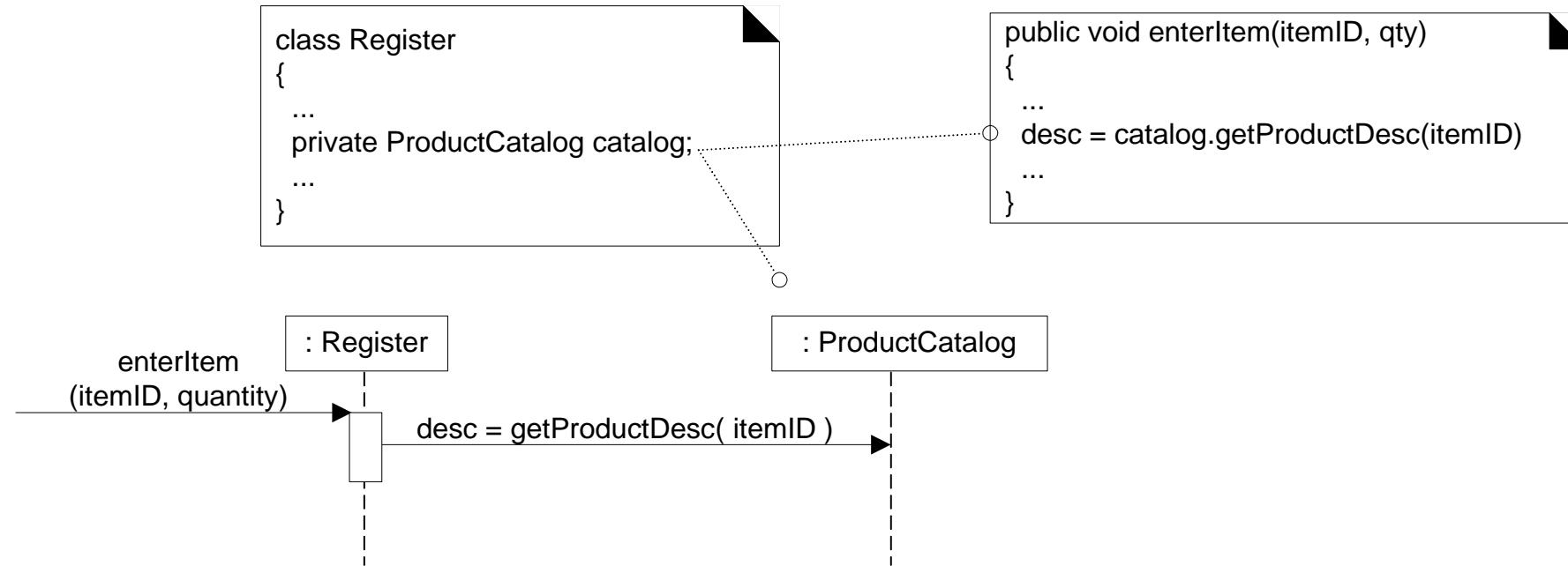
Object B will be visible to Object A

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  
`getProductDesc` is to be sent



## Parameter Visibility

# Parameter Visibility

Object B will be visible to Object A

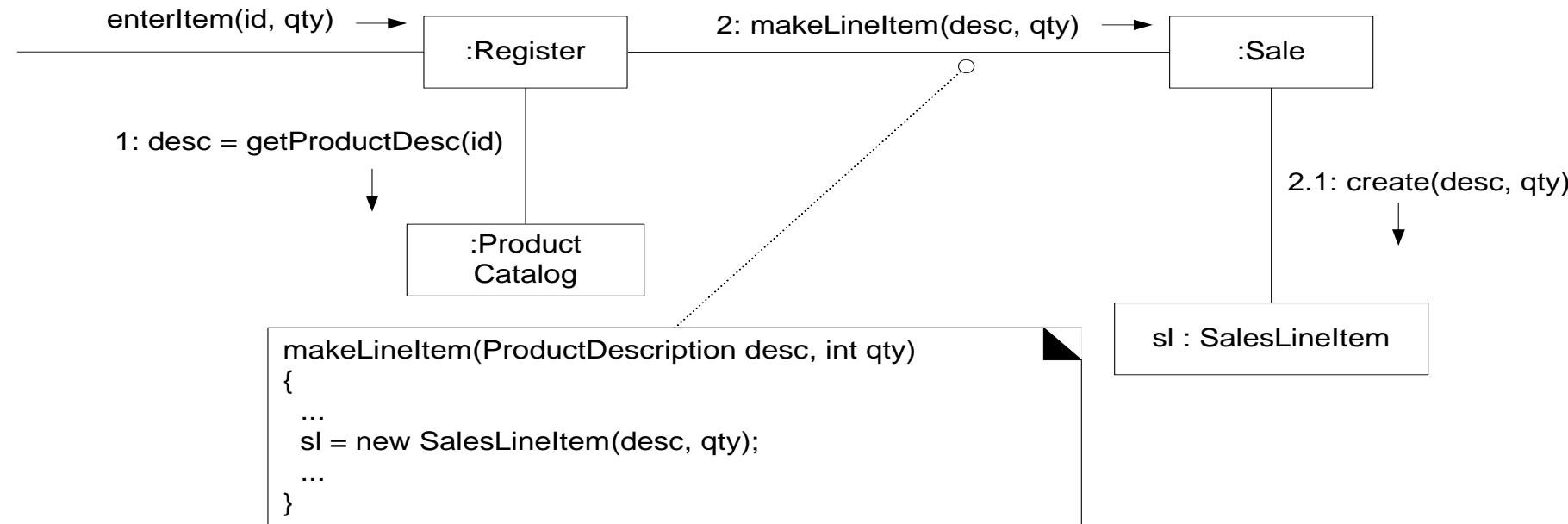
If

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

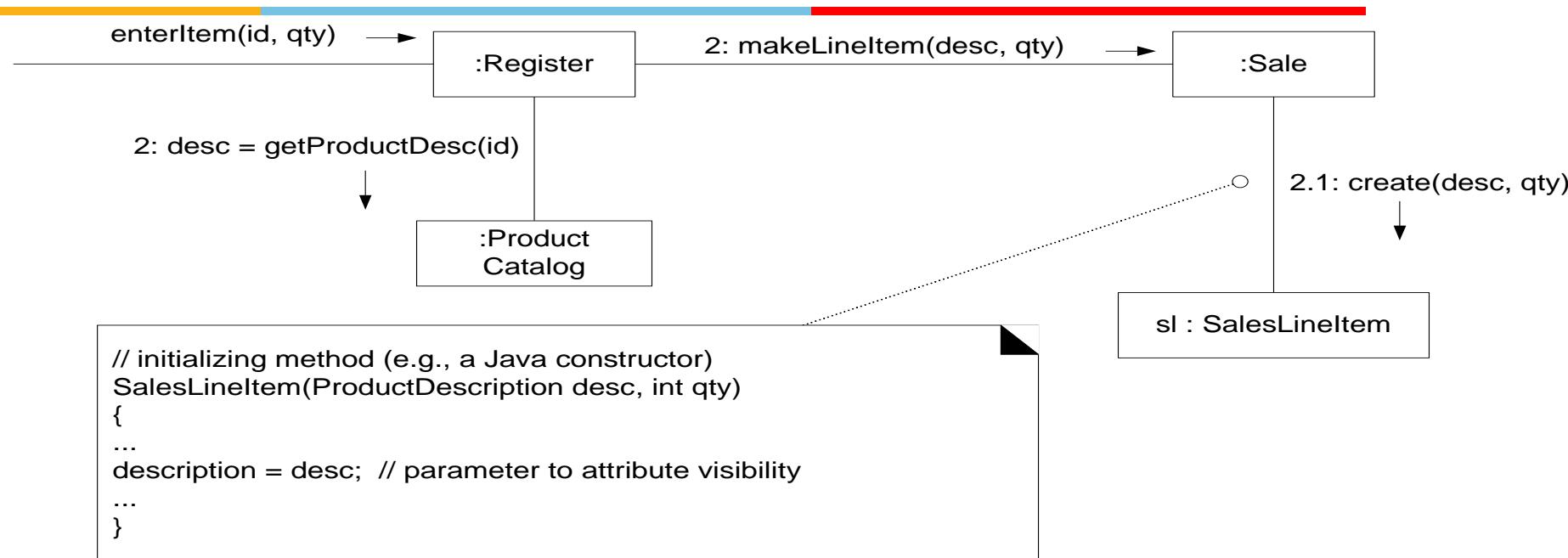
## Persists only within the scope of the method



**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.



## 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);
...
}
```



As with parameter visibility  
It is often transformed into Attribute visibility

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);



## 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.



## 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

Order
Placement Date
Delivery Date
Order Number
Calculate Total
Calculate Taxes

### Design

Order
- deliveryDate: Date
- orderNumber: int
- placementDate: Date
- taxes: Currency
- 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

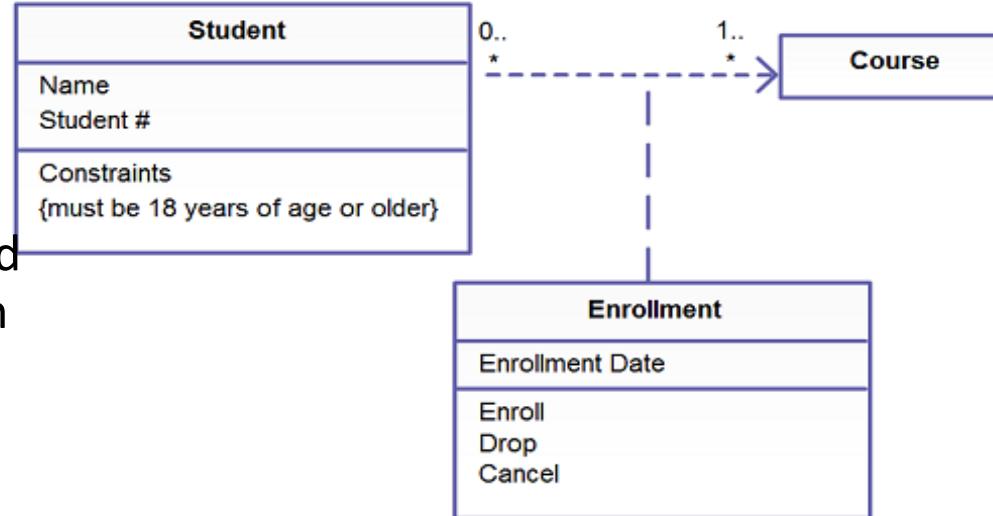
# Association Classes

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

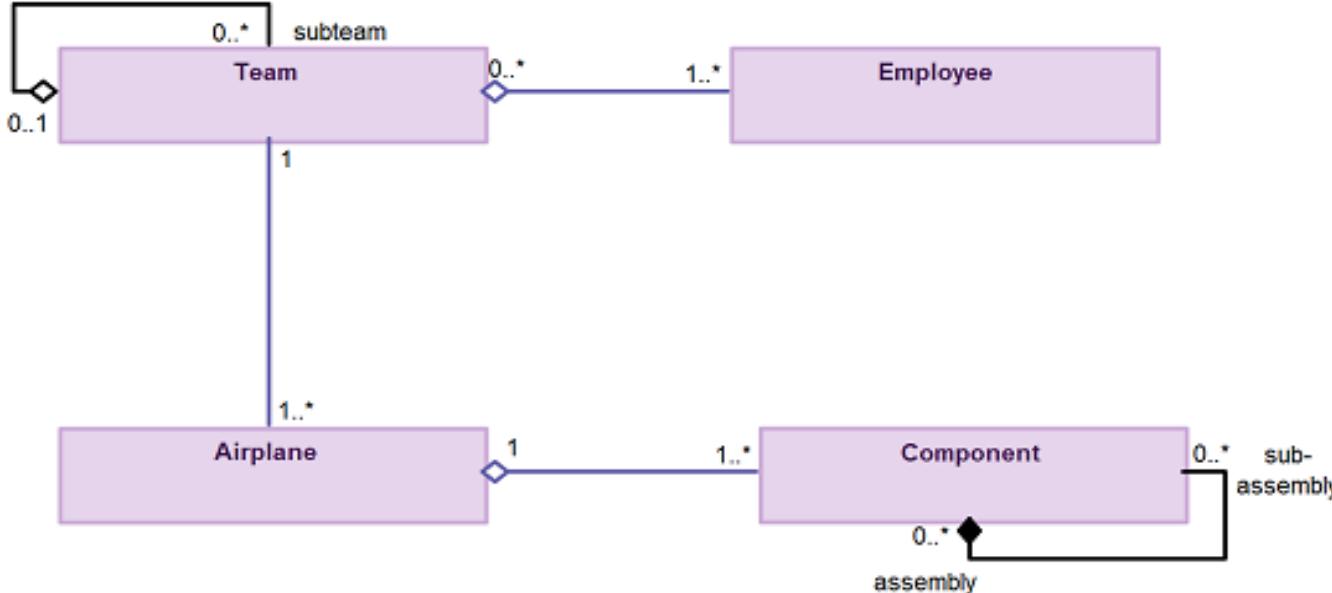
OrderItem
#numberOrdered:int
+findForItem(Item): Vector
+findForOrder(Order): Vector
#calculateTaxes(): Currency
#calculateTotal(): Currency
-getTaxEngine()



## **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

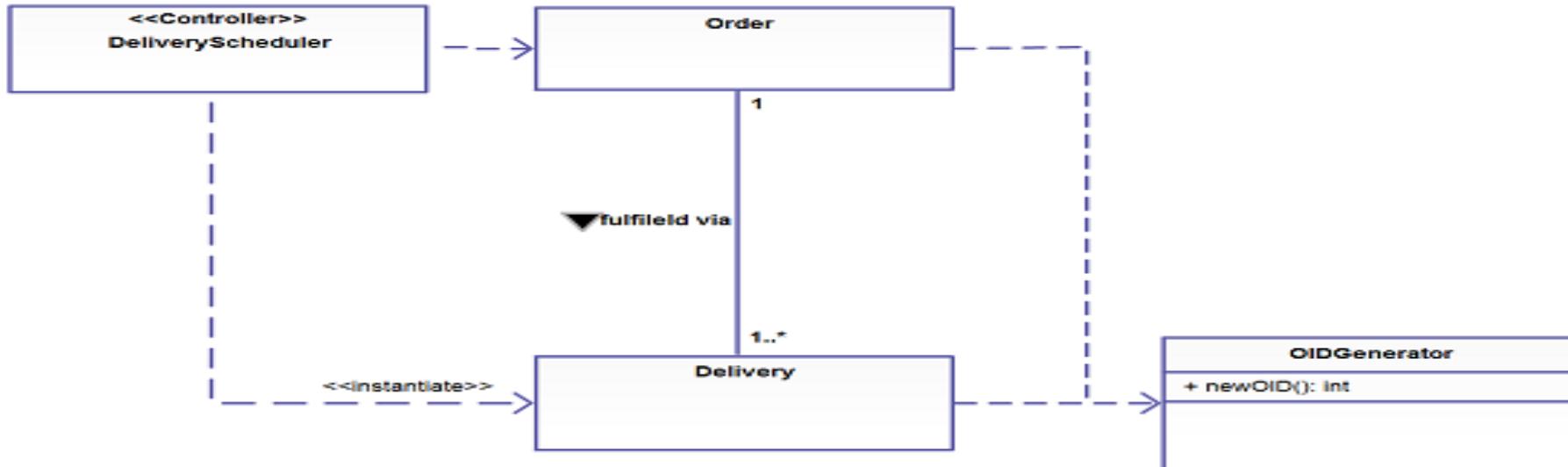
- aggregation,
- associations,
- dependencies,
- composition,
- realizations, and
- inheritance.

- 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)

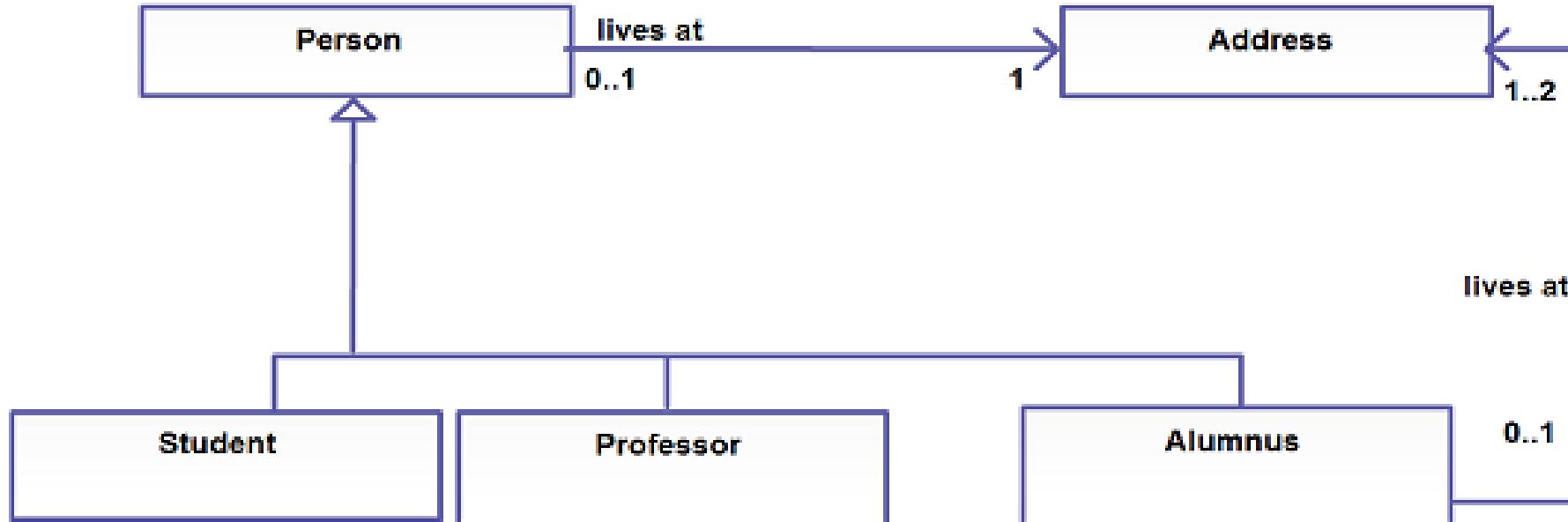


- Depict similar relationships involving a common class as a tree.
  - In **Figure A** you see that both **Delivery** and **Order** have a dependency on **OIDGenerator**. Note how the two dependencies are drawn in combination in “tree configuration”, instead of as two separate lines, to reduce clutter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

# Guideline for Relationship

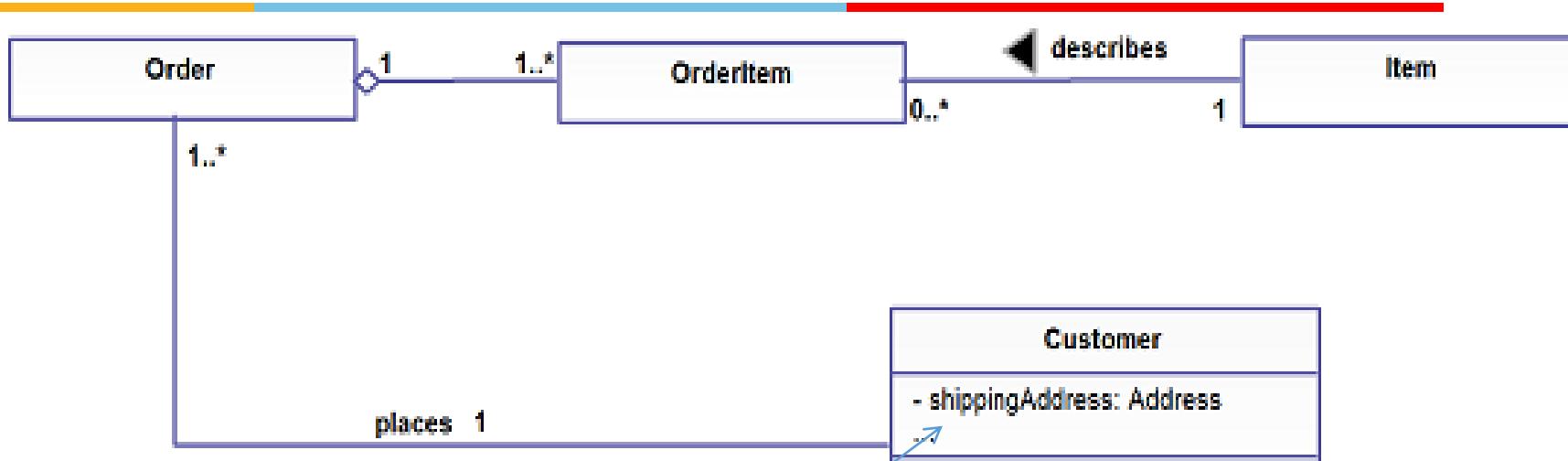
- Never model every single dependency
- Center names on associations
- Write concise association names in active voice
- Indicate directionality to clarify an association name
- Name unidirectional associations in the same direction
- Word association names left-to-right
- Indicate role names when multiple 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 uni-directional.

# 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.



## **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

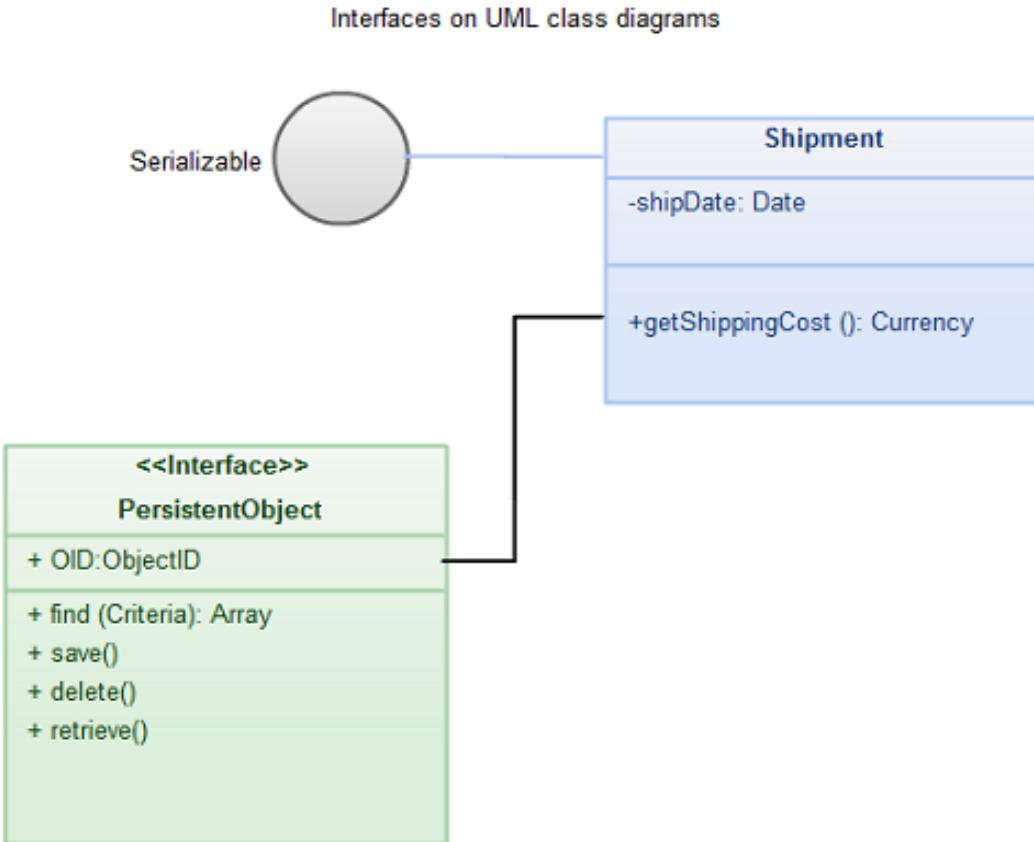
# Guidelines (cont.)

---

- 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

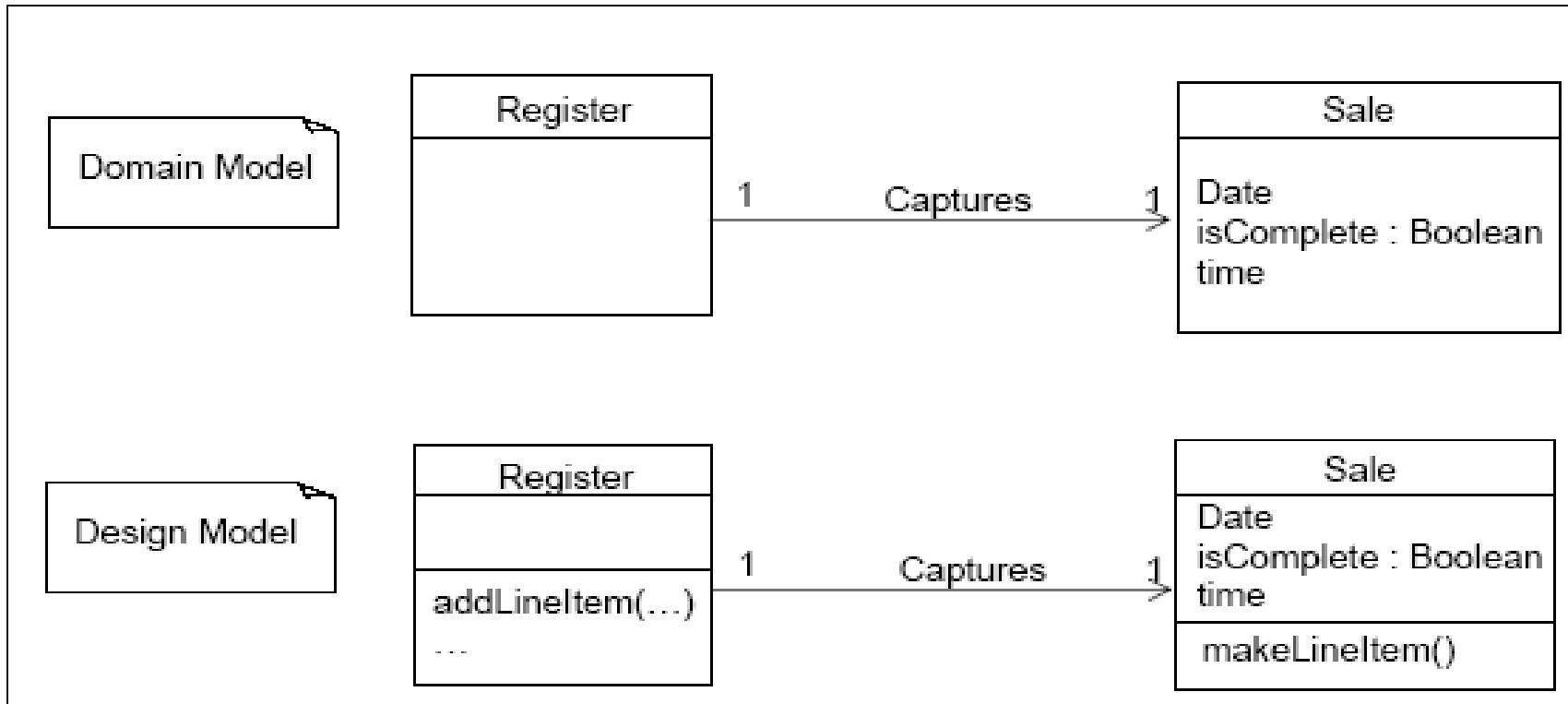
- Interface definitions must reflect implementation language constraints.
- In the example, you see that a standard class box has been used to define the interface PersistentObject (note the use of the <>interface>> stereotype).
- Name interfaces according to language naming conventions
- Apply “Lollipop” 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



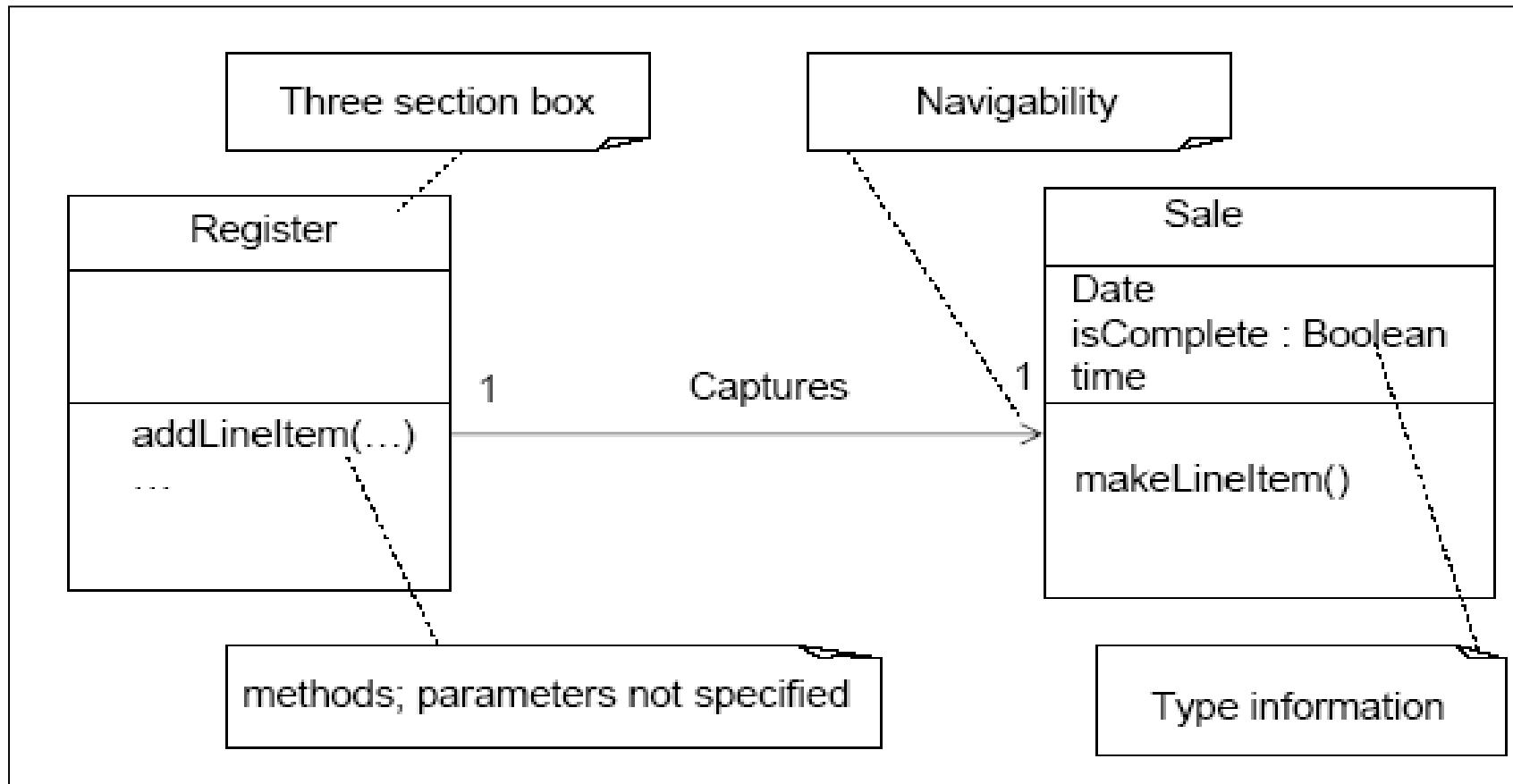
# Draw Class Diagram for PoS System

# Domain Model vs. Design Model

## Classes

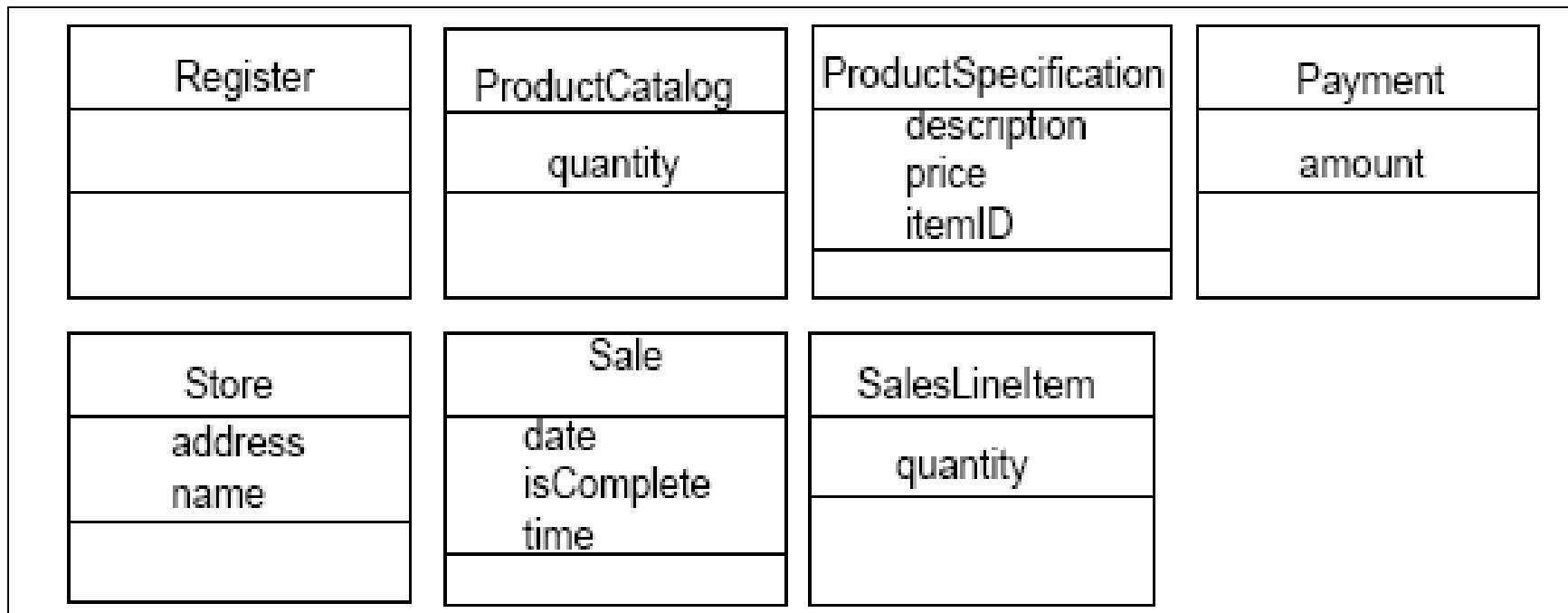


# An Example DCD



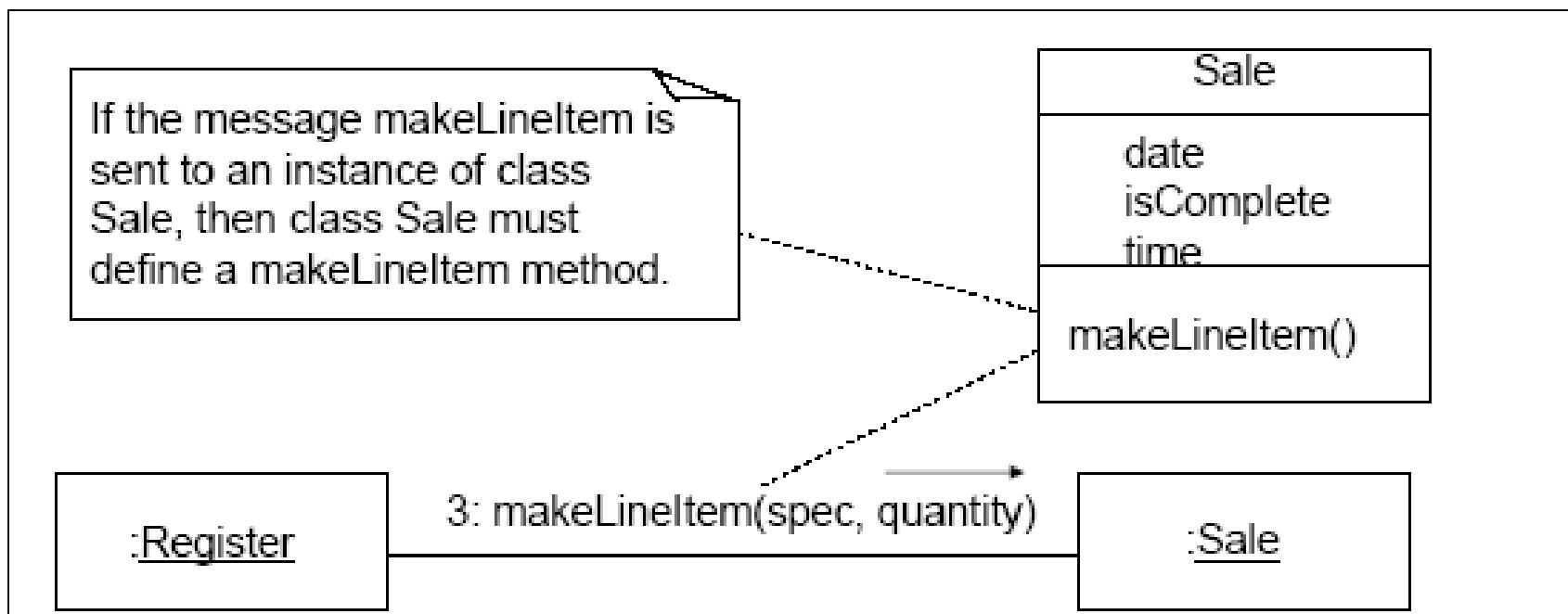
# Creating a NextGen POS DCD

- 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.



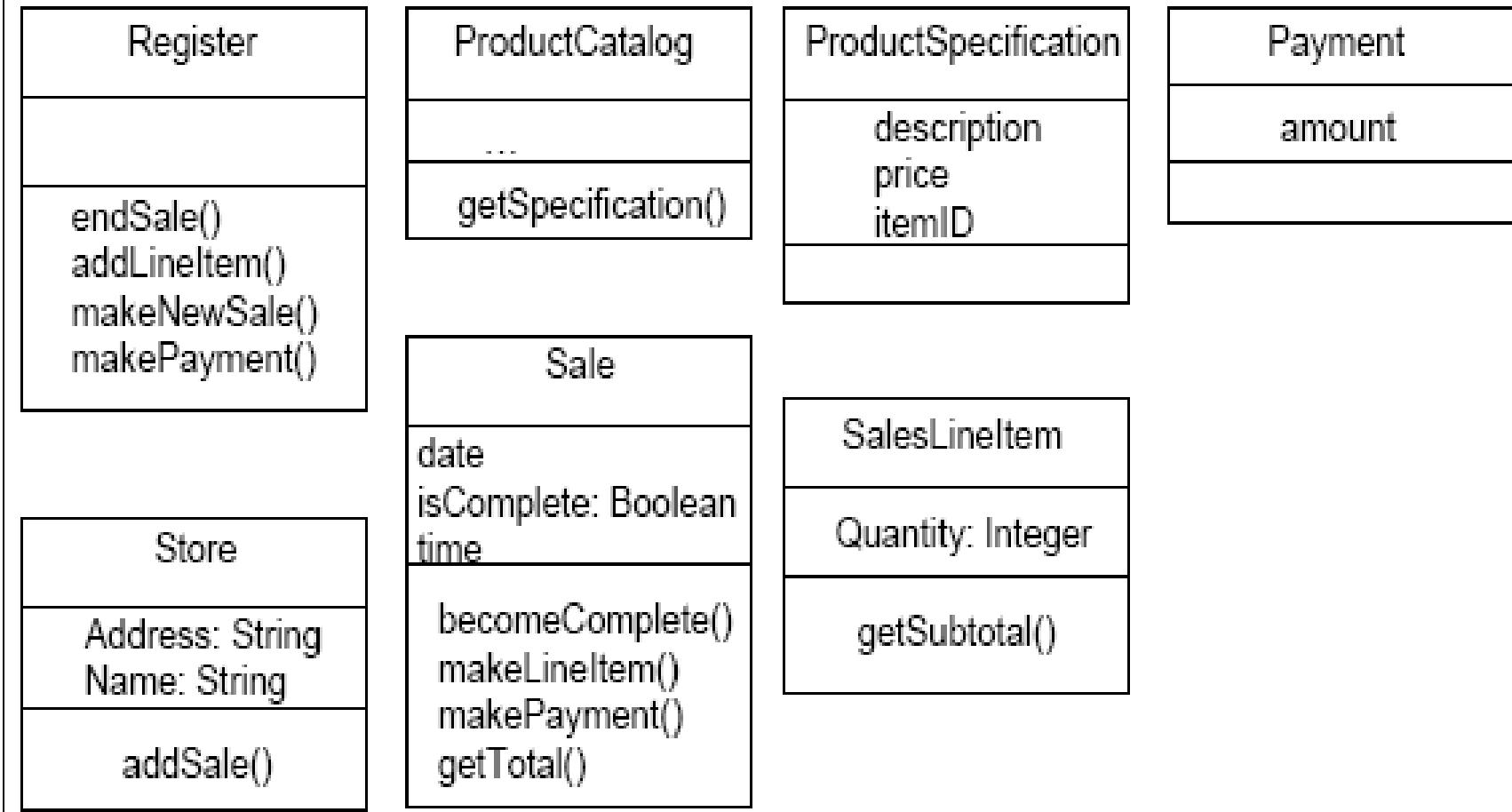
# Creating a NextGen POS DCD

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



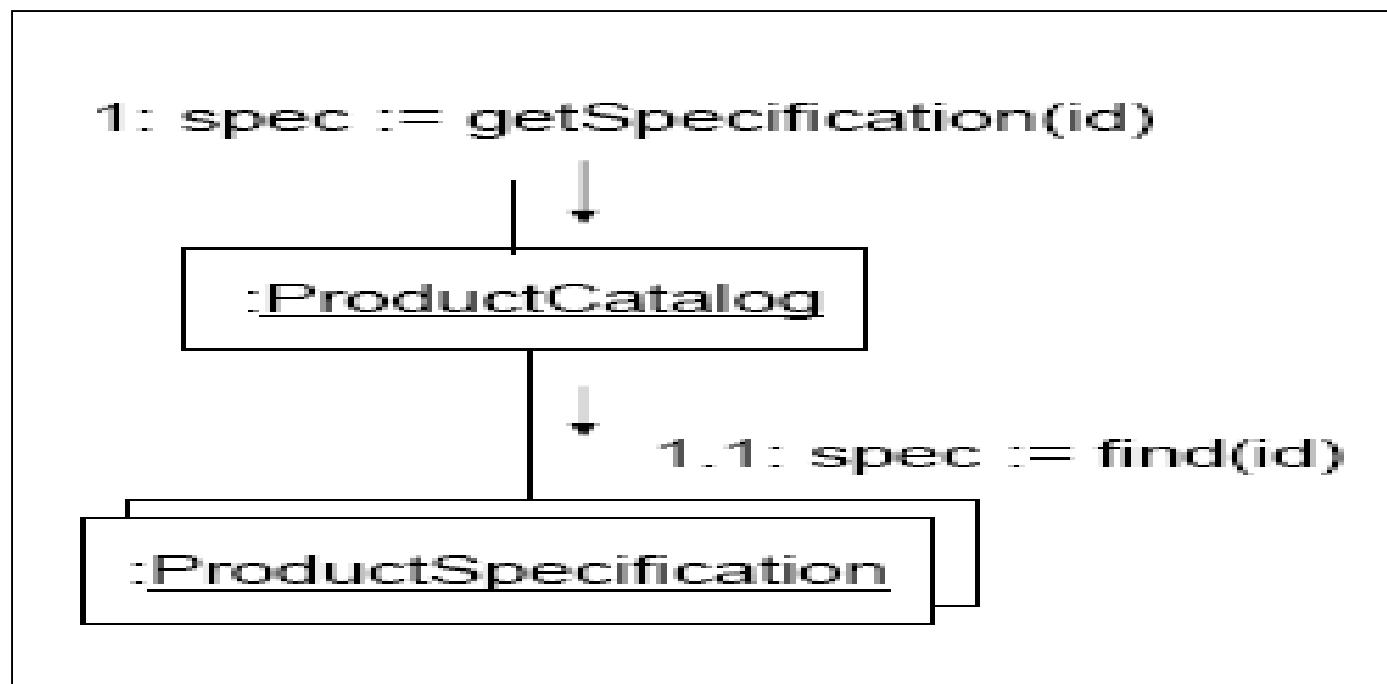
# Creating a NextGen POS DCD

- Add type information to the attributes and methods.



# Method Names -Multiobjects

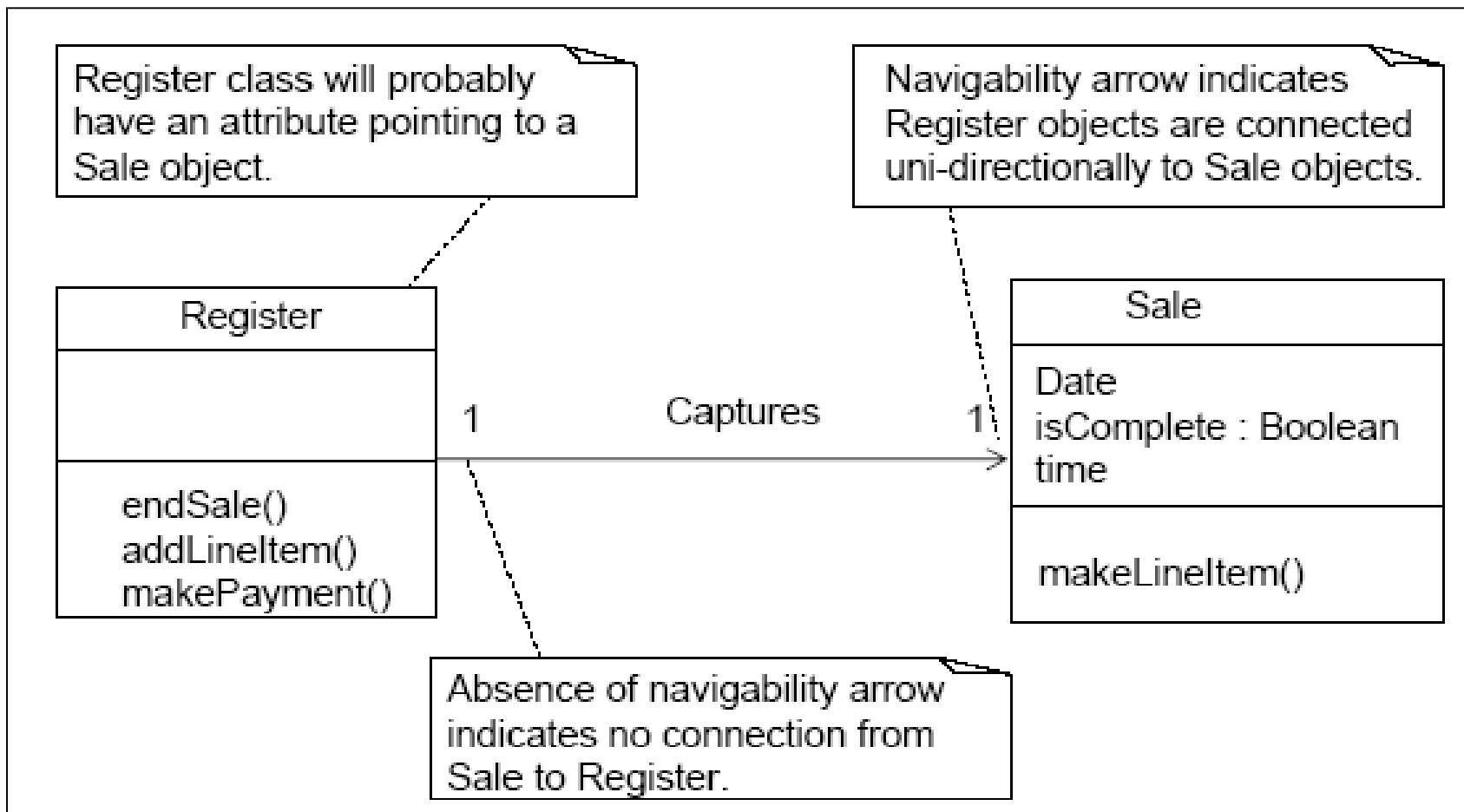
- The find message to the multiobject should be interpreted as a message to the container/ collection object.
- The find method is not part of the ProductSpecification class.



# 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





# Grouping Classes in Package Diagram

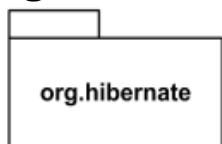
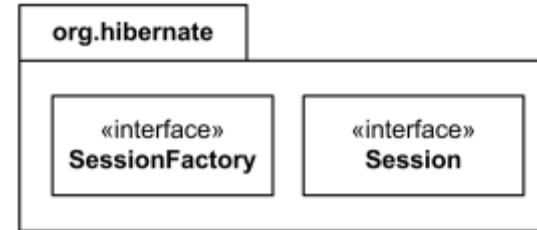
# Package

---

- **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 **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.

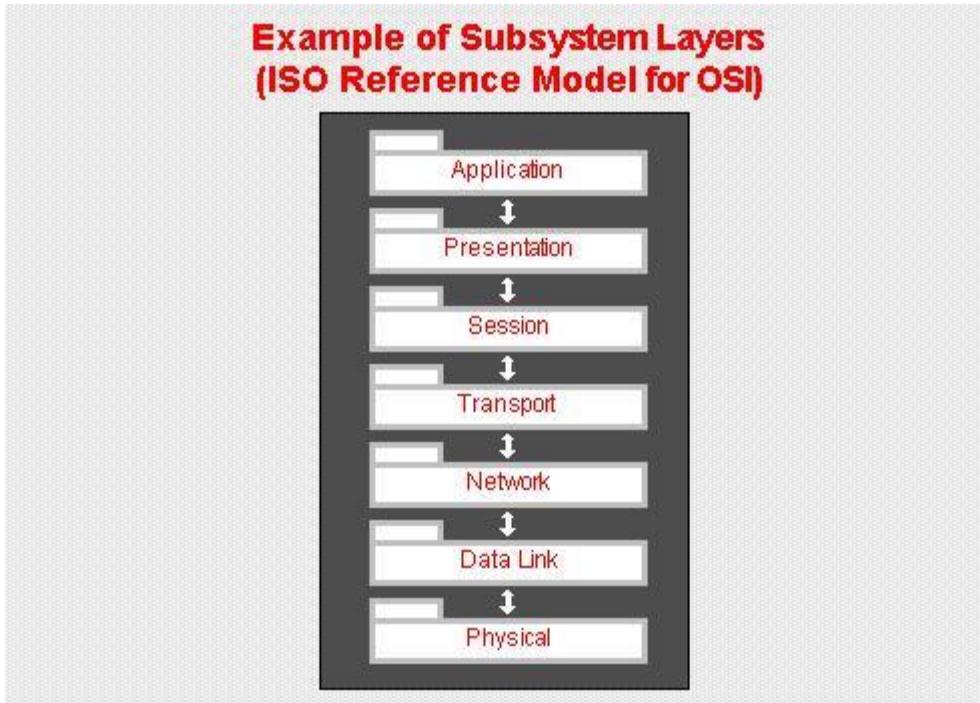
# Acknowledgement

- 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 Gamma | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
- Also: <http://agilemodeling.com/style/packageDiagram.htm>
- <http://www.uml-diagrams.org/package-diagrams.htm>



## Level & Partitions for Package Diagram

# Levels for Package Diagram

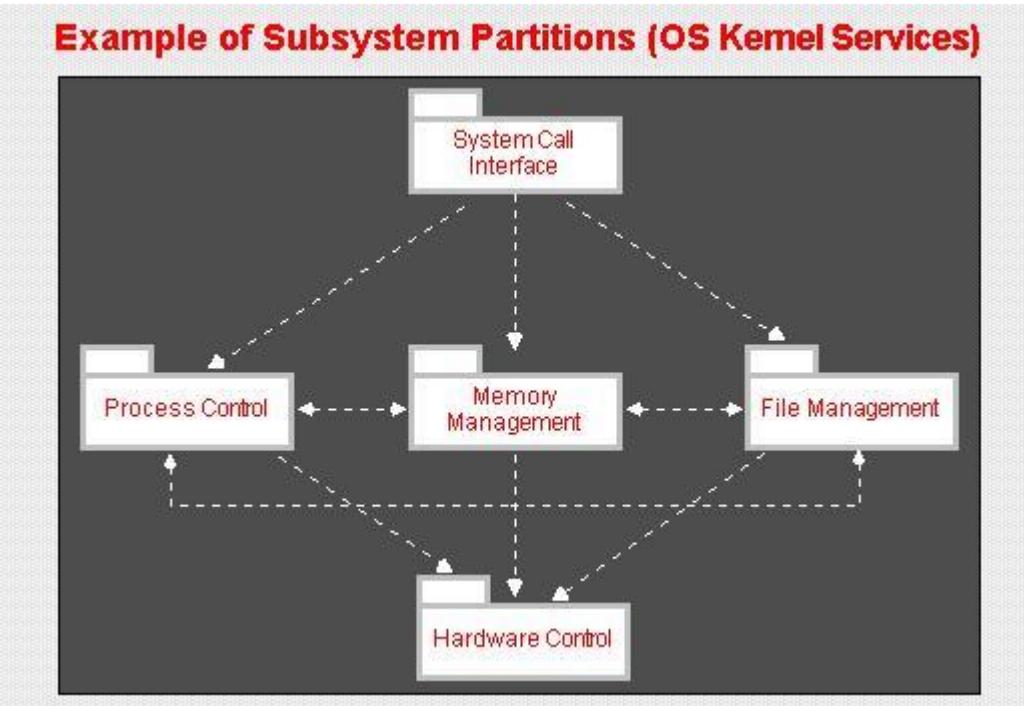


- In a layered architecture model, classes within each subsystem layer provide services to the layer above it.

Source: Defining Layers & Partitions in Architecture  
Topologies Craig Borysowich Dec 18, 2007 at  
[toolbox.com](http://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



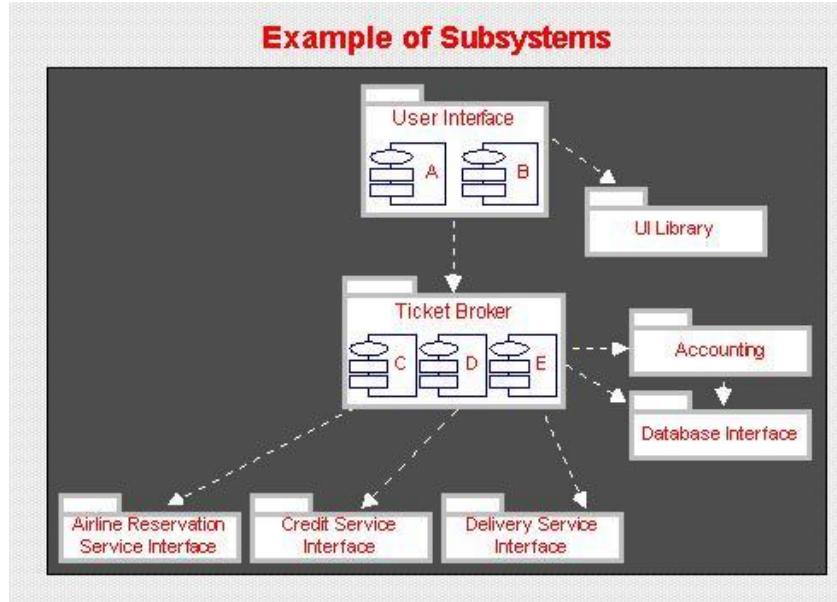
- In a partitioned architecture model, subsystems collaborate with peer subsystems at the same level of abstraction.

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

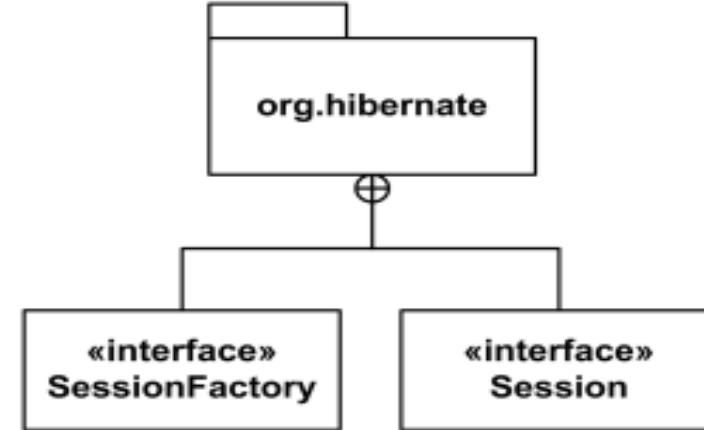
- Slides are based on Course Text Books:
  - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
  - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
  - Design Patterns (Elements of Reusable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



## 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

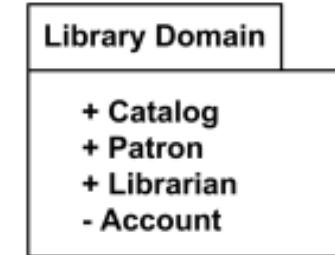
The elements that can be referred to within a package using **non-qualified** names are:

- 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**

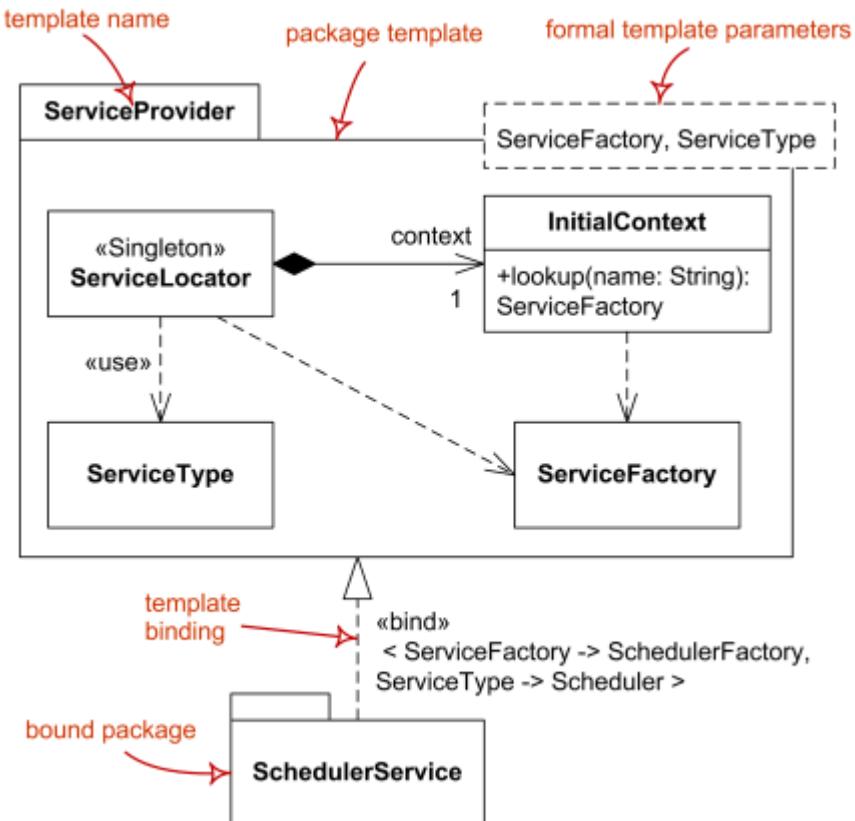
- **Packageable element** is a **named element** that may be **owned** directly by a **package**.

# 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 Gamma | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
- Also: <http://agilemodeling.com/style/packageDiagram.htm>
- <http://www.uml-diagrams.org/package-diagrams.htm>



# Guidelines for Package Diagram

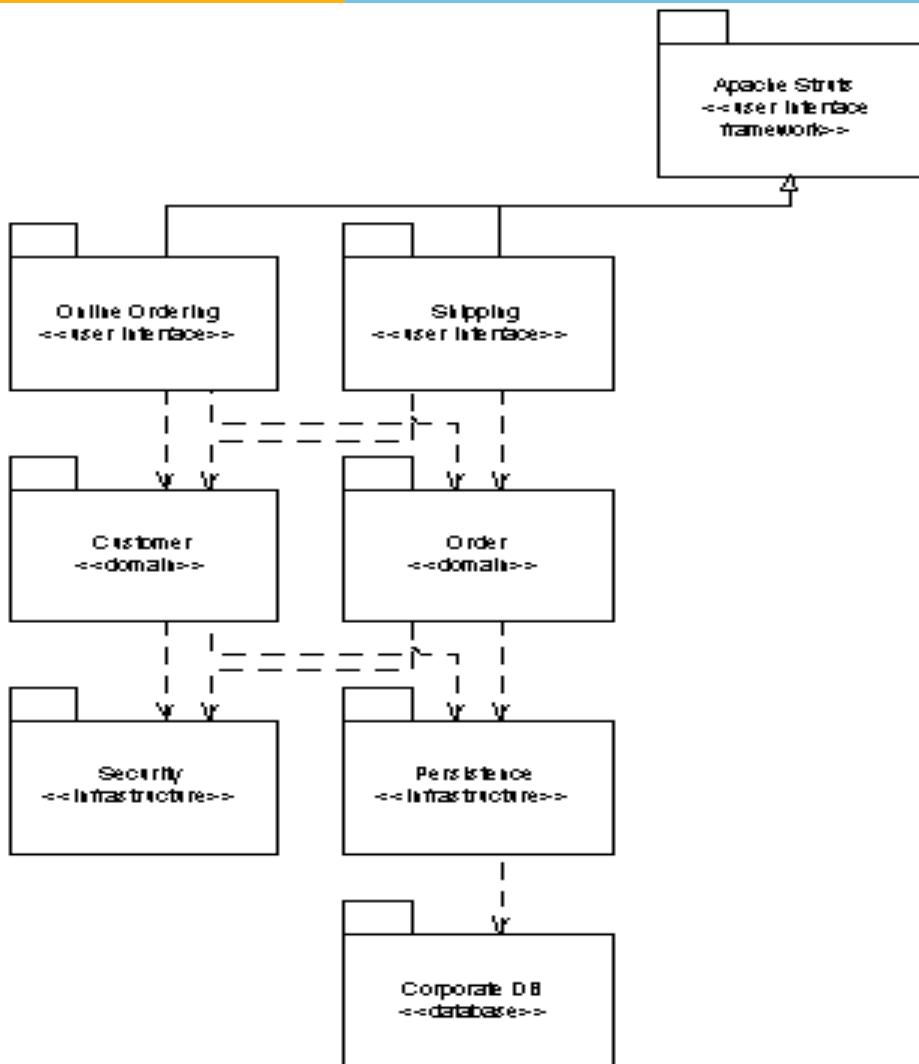
# Guidelines for Package Diagrams

- 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



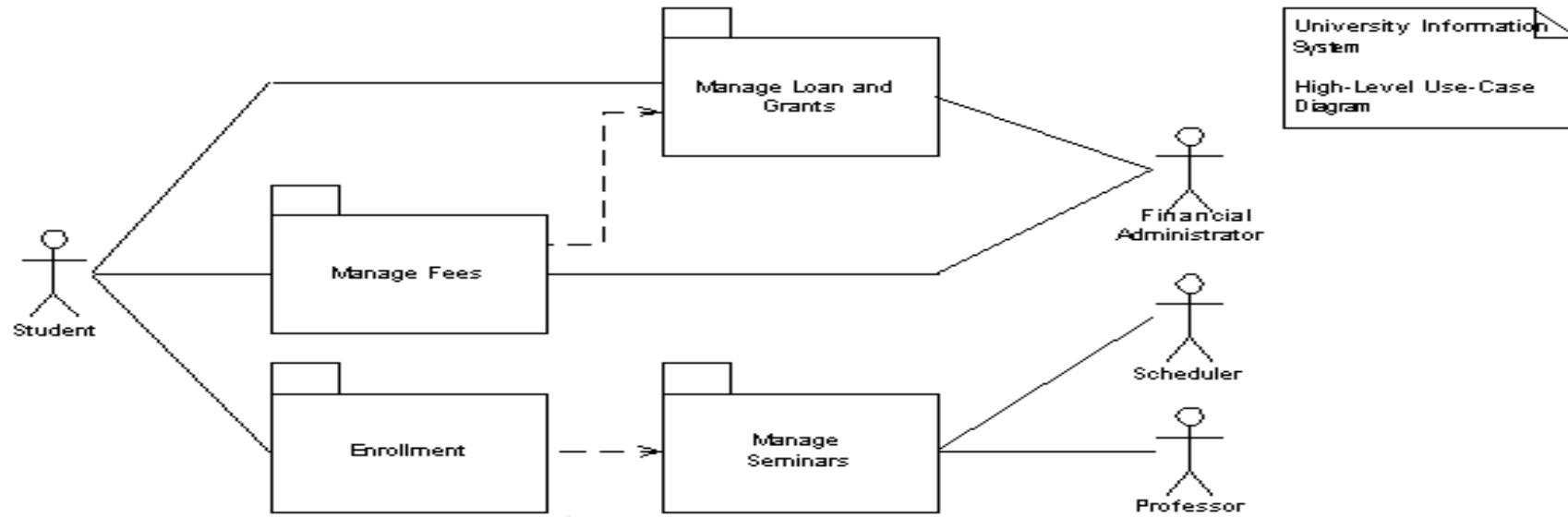
UML Class diagram organized  
into packages.

# 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

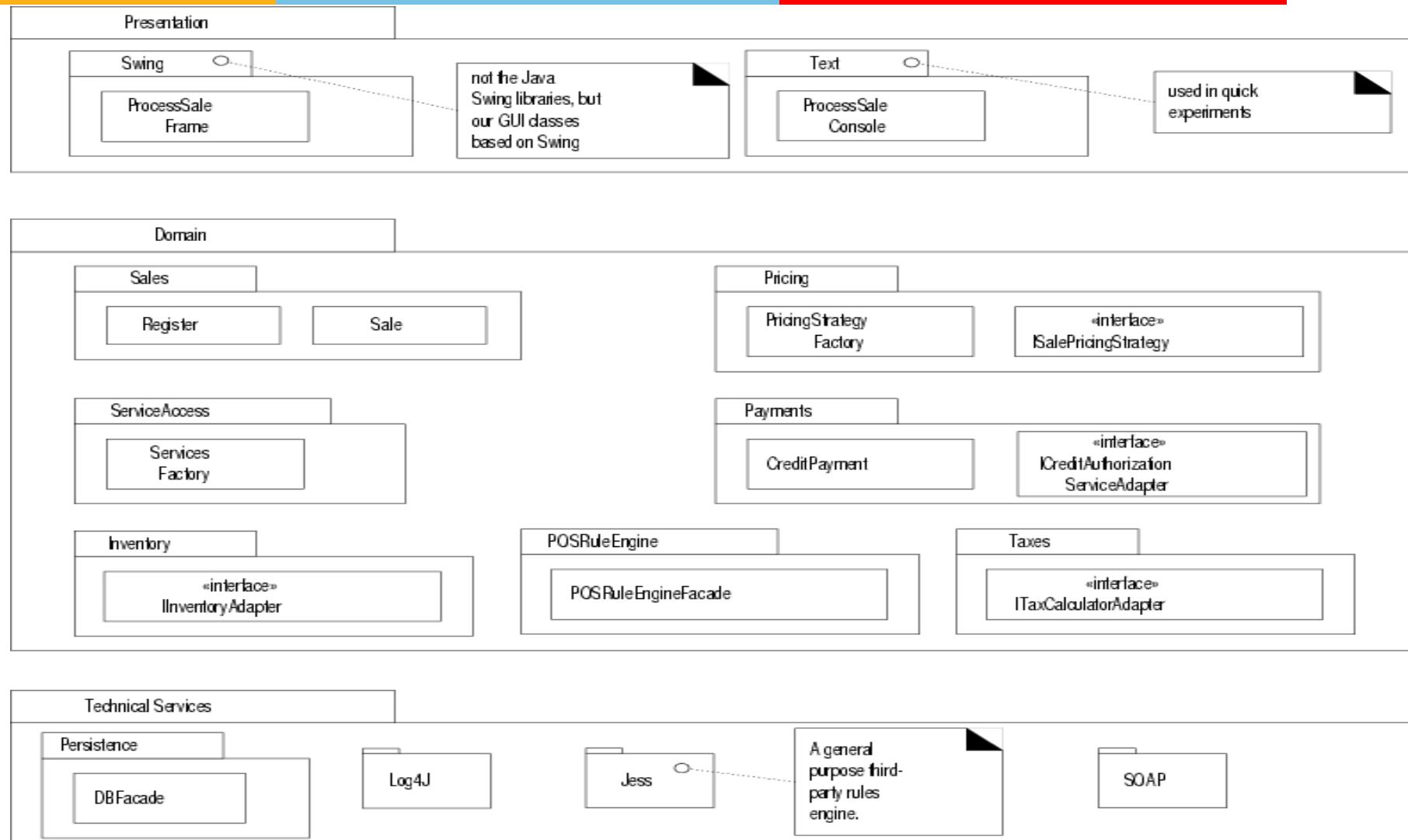
# Acknowledgement

- 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 Gamma | Richard Helm | Rapph Johnson | John Vlissides alias GoF.
- Also: <http://agilemodeling.com/style/packageDiagram.htm>
- <http://www.uml-diagrams.org/package-diagrams.htm>

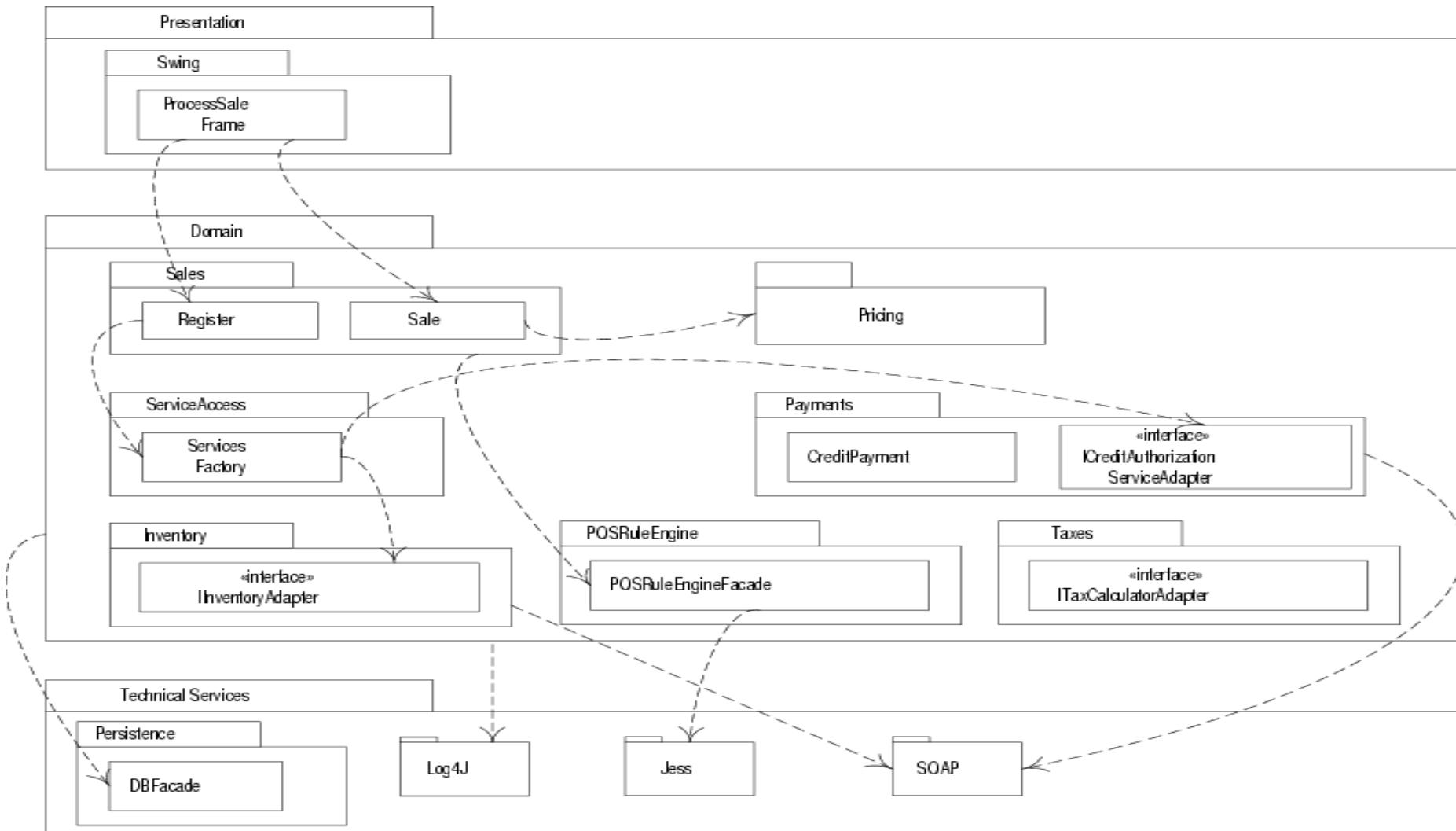


## Drawing Package Diagram for PoS System

# 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 Gamma | 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



**BITS** Pilani  
Pilani Campus

Paramananda Barik



## **What is Pattern? What is Design Pattern?**

# Design Pattern

- In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design.
- A design pattern isn't a finished design that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.

# Design Patterns

- We will emphasize principles (expressed in patterns) to guide choices in where to assign responsibilities.
- A pattern is a named description of a problem and a solution that can be applied to new contexts; it provides advice in how to apply it in varying circumstances. For example,
  - Pattern name: Information Expert
  - Problem: What is the most basic principle by which to assign responsibilities to objects?
  - Solution: Assign a responsibility to the class that has the information needed to fulfill it

# Types of Design Patterns

---

## GRASP PATTERNS

Is an approach to understanding and using design principals based on patterns of assigning responsibilities.

## GoF PATTERNS

- According to The Gang of Four design patterns are primarily based on the following principles of object orientated design.

Program to an interface not an implementation

Favour object composition over inheritance

# GRASP (General Responsibility Assignment Software Patterns and Principles.)



- Information Expert
  - Creator
  - Controller
  - Low Coupling
  - High Cohesion
  - Polymorphism
  - Pure Fabrication
  - Indirection
  - Protected Variation
- Apply Design Reasoning in a methodical, rational, explainable way.

# What is Gang of Four (GoF)?

---

- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.
- These authors are collectively known as **Gang of Four (GoF)**.
- **The Original GoF**  
They came to prominence during the Cultural Revolution (1966–76) and were later charged with a series of treasonous crimes. The gang's leading figure was **Mao Zedong**'s last wife **Jiang Qing**. The other members were **Zhang Chunqiao**, **Yao Wenyuan**, and **Wang Hongwen**.

# GoF Patterns

## Structural design patterns

This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

## Behavioral design patterns

This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

## Creational design patterns

This design patterns is all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

# Adv - Uses of Design Patterns

- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Effective software design requires considering issues that may not become visible until later in the implementation.
- Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

# Uses of Design Patterns

- Often, people only understand how to apply certain software design techniques to certain problems.
- These techniques are difficult to apply to a broader range of problems.
- Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.
- In addition, patterns allow developers to communicate using well-known, well understood names for software interactions.
- Common design patterns can be improved over time, making them more robust than ad-hoc designs.

# Design Patterns – Designer's & Programmer's perspective



**Communication, Learning and Enhanced Insight:** Over the last two decade design patterns have become part of every systems vocabulary. Reference to patterns in communication helps developers with better insight about parts of the application or 3rd party frameworks.

**Decomposing System into Objects :** Design Patterns really helps identify less obvious abstractions. These objects are seldom found during analysis or even the early design, they're discovered later in the course of making a design more flexible and reusable.

**Determining Object Granularity:** Design patterns helps in coming up with objects with different levels of granularity that makes sense.

**Specifying Object Interface :** Laying down method signatures for all classes can also be quite tricky. Use of Interfaces as method signatures solves this problem. Forget interfaces, most of the times, Design Patterns really helps in this area.

# Design Patterns – Designer's & Programmer's perspective



## Specifying Object

**Implementation :** Design Patterns provide guidance so that one can Program to an interface (type) not to an implementation (concrete class) which can result in really good OO code.

## Ensuring right reuse mechanism :

When to use Inheritance, when to use Composition, when to use Parameterized Types? Is delegation the right design decision in this context?

Knowledge of design patterns can really come handy when making such decisions.

## Relating run-time and compile

**time structures :** Knowledge of design patterns can make some of the hidden structure obvious.

**Designing for change :** Each design pattern lets some aspect of the system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.



## **What is Responsibility Assignment Problem?**

# Responsibilities and Methods

- The focus of object design is to identify classes and objects, decide what methods belong where and how these objects should interact.
- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - Doing:
    - Doing something itself (e.g. creating an object, doing a calculation)
    - Initiating action in other objects.
    - Controlling and coordinating activities in other objects.
  - Knowing:
    - Knowing about private encapsulated data.
    - Knowing about related objects.
    - Knowing about things it can derive or calculate.

# Responsibilities and Methods

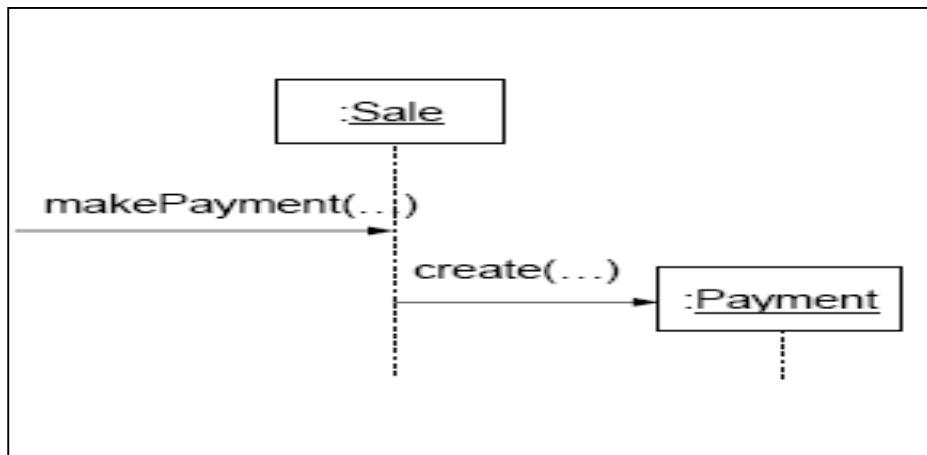
- Responsibilities are assigned to classes during object design. For example, we may declare the following:
  - “*a Sale is responsible for creating SalesLineItems*” (doing)
  - “*a Sale is responsible for knowing its total*” (knowing)
- Responsibilities related to “knowing” are often inferable from the Domain Model (because of the attributes and associations it illustrates)

# Responsibilities and Methods

- The translation of responsibilities into classes and methods is influenced by the granularity of responsibility.
  - For example, “*provide access to relational databases*” may involve dozens of classes and hundreds of methods, whereas “*create a Sale*” may involve only one or few methods.
- A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities.
- Methods either act alone, or collaborate with other methods and objects.

# Responsibilities and Interaction Diagrams

- Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams.
- Sale objects have been given the responsibility to create Payments, handled with the makePayment method.





GRASP Patterns – Information Expert, Controller, Creator,  
Low Coupling & High Cohesion

# GRASP (General Responsibility Assignment Software Patterns and Principles.)



- Information Expert
  - Creator
  - Controller
  - Low Coupling
  - High Cohesion
  - Polymorphism
  - Pure Fabrication
  - Indirection
  - Protected Variation
- Apply Design Reasoning in a methodical, rational, explainable way.

## • Information Expert

---

- Assign a responsibility to the information expert.
- The class that has the information necessary to fulfil the responsibility.

A General principle of object design and responsibility assignment.

## • Creator

---

Assign class B the responsibility to create an instance of class A if one of the following is true:

1. B contains A
2. B aggregates A
3. B has the initialisation data for A
4. B records A
5. B closely uses A.

Who Creates?

(Factory is a common alternate solution)

# • Controller

---

Assign the responsibility to an object representing one of these choices:

1. Represents the overall “system” a “root object”, a device the software is running within, or a major subsystem (all variants of façade controller)
2. Represent a use case scenario within which a system operations occurs (a use case or a session controller)

What first object beyond the UI layer receives and coordinates (“controls”) a system operation?

## • Low Coupling

---

- Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

How to reduce the impact of change?

# High Cohesion

---

- Assign Responsibilities so that the cohesion remains high. Use this to evaluate alternatives.

How to keep objects focused, understandable, and manageable, and as a side-effect, support Low Coupling?

# Polymorphism

---

- When related alternatives or behaviours vary by type (class) assign responsibility for the behaviour- using polymorphism – to the types for which the behaviour varies.

Who is responsible when behaviour varies with type?

# Pure Fabrication

---

- Assign a higher cohesive set of responsibilities to an artificial or convenient “behaviour” class that does not represent a problem domain concept;
- This may sometimes be made up, in order to support high cohesion, low coupling and reuse.

Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?

# Indirection

---

- Assign the responsibility to an intermediate object to mediate between the components or services, so that they are not directly coupled.

How is assigned responsibility to avoid direct coupling?

# Protected Variation

- Identify points of predicted variation or instability;
- Assign responsibility to create a stable “interface” around them.

How to assign responsibility to objects, subsystems and systems so that the variations or services, so that they are not directly coupled.



## Information Expert Pattern – Problem & Solution

# Information Expert (or Expert)

- Problem: what is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.
- In the NextGen POS application, who should be responsible for knowing the grand total of a sale?
- By Information Expert we should look for that class that has the information needed to determine the total.

## • Information Expert

---

- Assign a responsibility to the information expert.
- The class that has the information necessary to fulfil the responsibility.

A General principle of object design and responsibility assignment.

# Information Expert (or Expert)

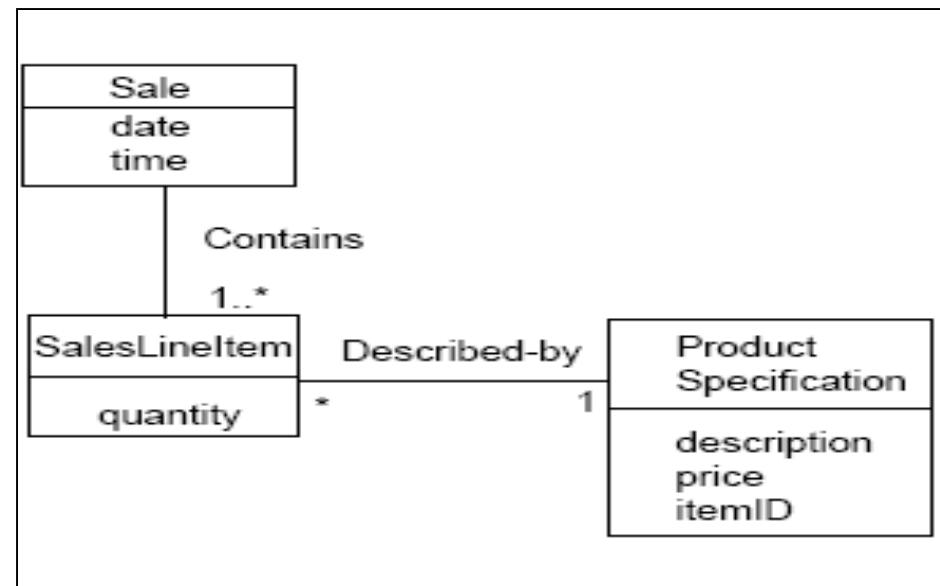
- Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed?
- A: Both. Assume there is no or minimal Design Model. Look to the Domain Model for information experts.



## Application of Information Expert in PoS System

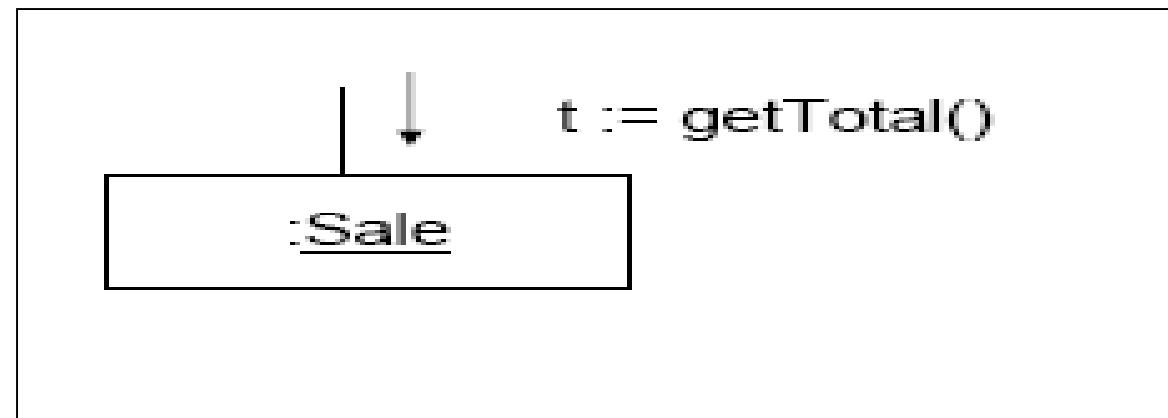
# Information Expert (or Expert)

- It is necessary to know about all the SalesLineItem instances of a sale and the sum of the subtotals.
- A Sale instance contains these, i.e. it is an information expert for this responsibility.



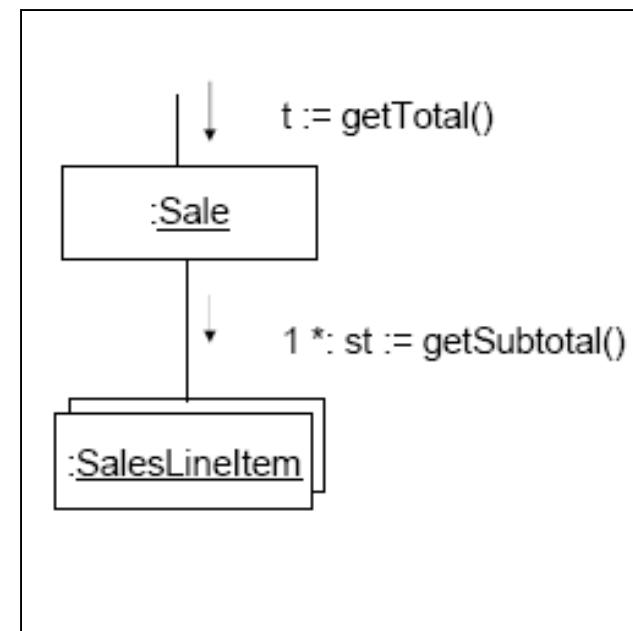
# Information Expert (or Expert)

- This is a partial interaction diagram.



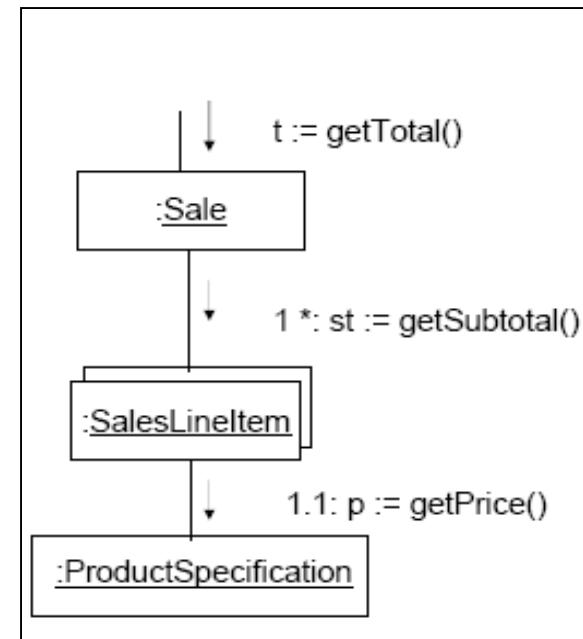
# Information Expert (or Expert)

- What information is needed to determine the line item subtotal?
  - quantity and price.
- SalesLineItem should determine the subtotal.
- This means that Sale needs to send `getSubtotal()` messages to each of the SalesLineItems and sum the results.



# Information Expert (or Expert)

- To fulfil the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.
- The ProductSpecification is the information expert on answering its price.



# Information Expert (or Expert)

- To fulfil the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes
- The fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many “partial experts” who will collaborate in the task.

Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item total
ProductSpecification	Knows product price



## Controller Pattern – Problem & Solution

# Controller

Solution:

- Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:
  - Represents the overall system.
  - Represents a use case scenario.
- A Controller is a non-user interface object that defines the method for the system operation.  
Note that windows, applets, etc. typically receive events and delegate them to a controller.

Problem:

Who should be responsible for handling an input system event?

# • Controller

Assign the responsibility to an object representing one of these choices:

1. Represents the overall “system” a “root object”, a device the software is running within, or a major subsystem (all variants of façade controller)
2. Represent a use case scenario within which a system operations occurs (a use case or a session controller)

What first object beyond the UI layer receives and coordinates (“controls”) a system operation?



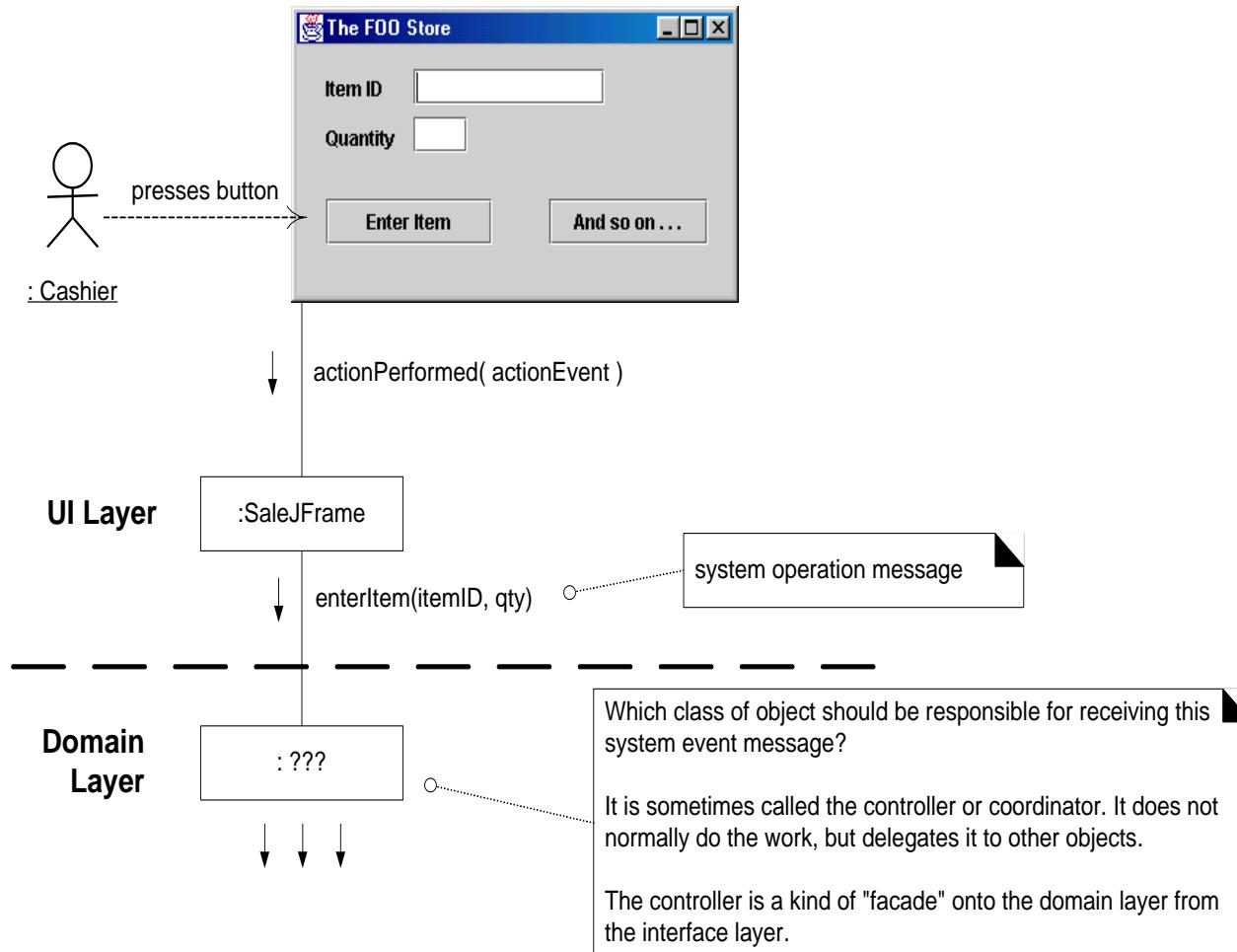
## Application of Controller Pattern in PoS System

# Controller

- The NextGen application contains several operations.
- You can model the system itself as a class.
- During analysis , the system operations may be assigned to the class System in some analysis model, to indicate that these are system operations. This does not mean that a software class named System fulfils them during design. During design a controller class is assigned the responsibility for system operations. (Next Slide)

System
endSale()
enterItem()
makeNewSale()
makePayment()
...

# Controller



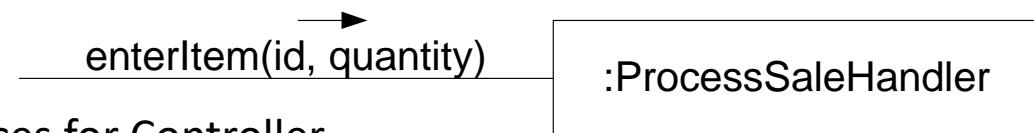
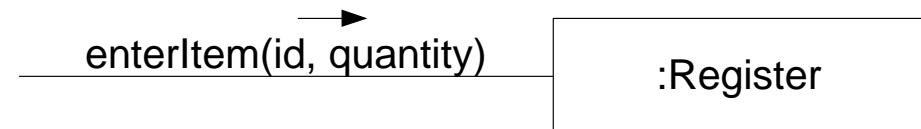
- Who should be the controller for system events such as `enterItem` and `endSale`?

# Controller

By the Controller Pattern, here are some choices:

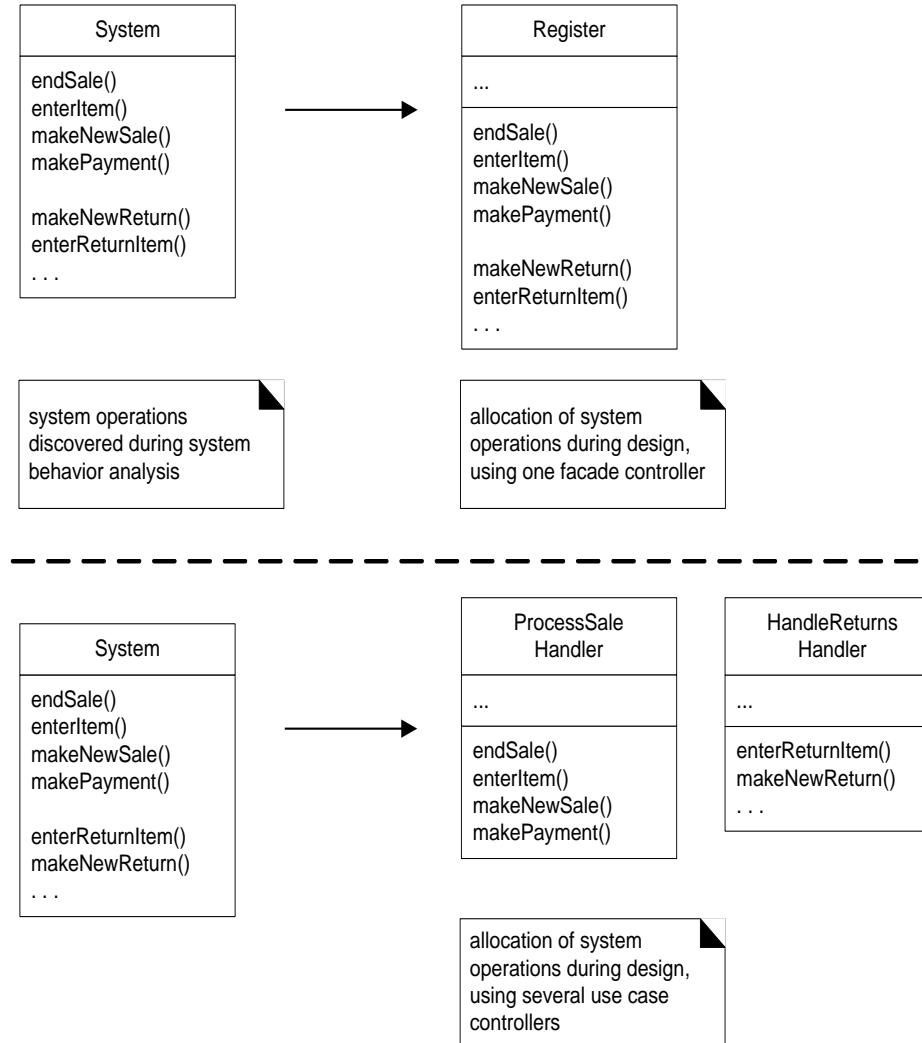
- Represent the overall “system”, “root object”, device or subsystem: Register, POSSystem.
- Represent a receiver or handler of all system events of a use case scenario: ProcessSaleHandler, ProcessSaleSession.

Note: domain of POS, aRegister (POS terminal) is a specialized device with software running on it.



Choices for Controller

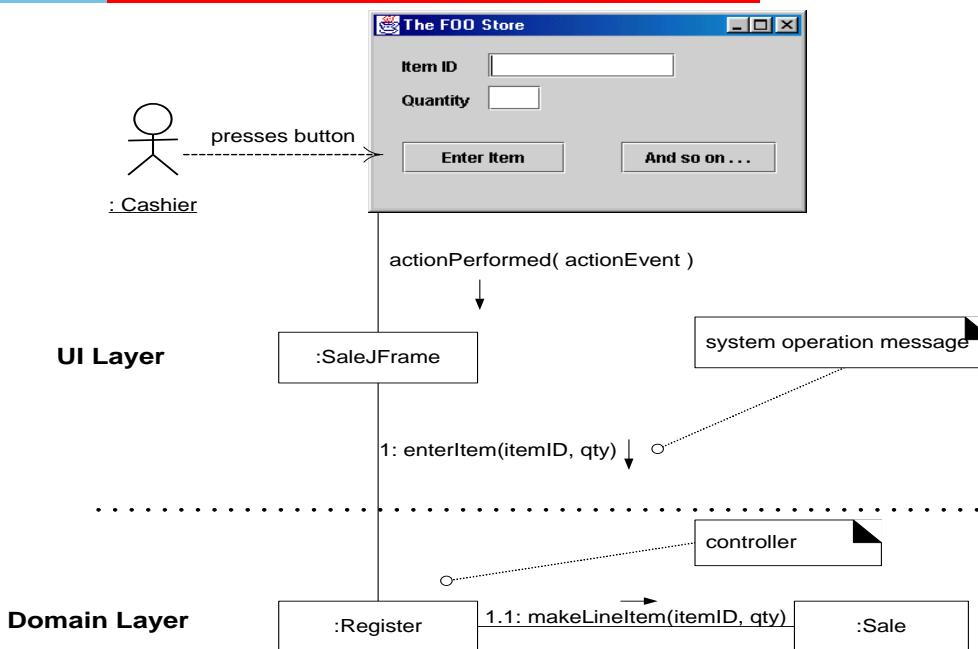
# System Behaviors Analysis



The assignment of responsibility for systems operation may be assigned to one or more classes. See examples on left.

# Assignment by UI Layer

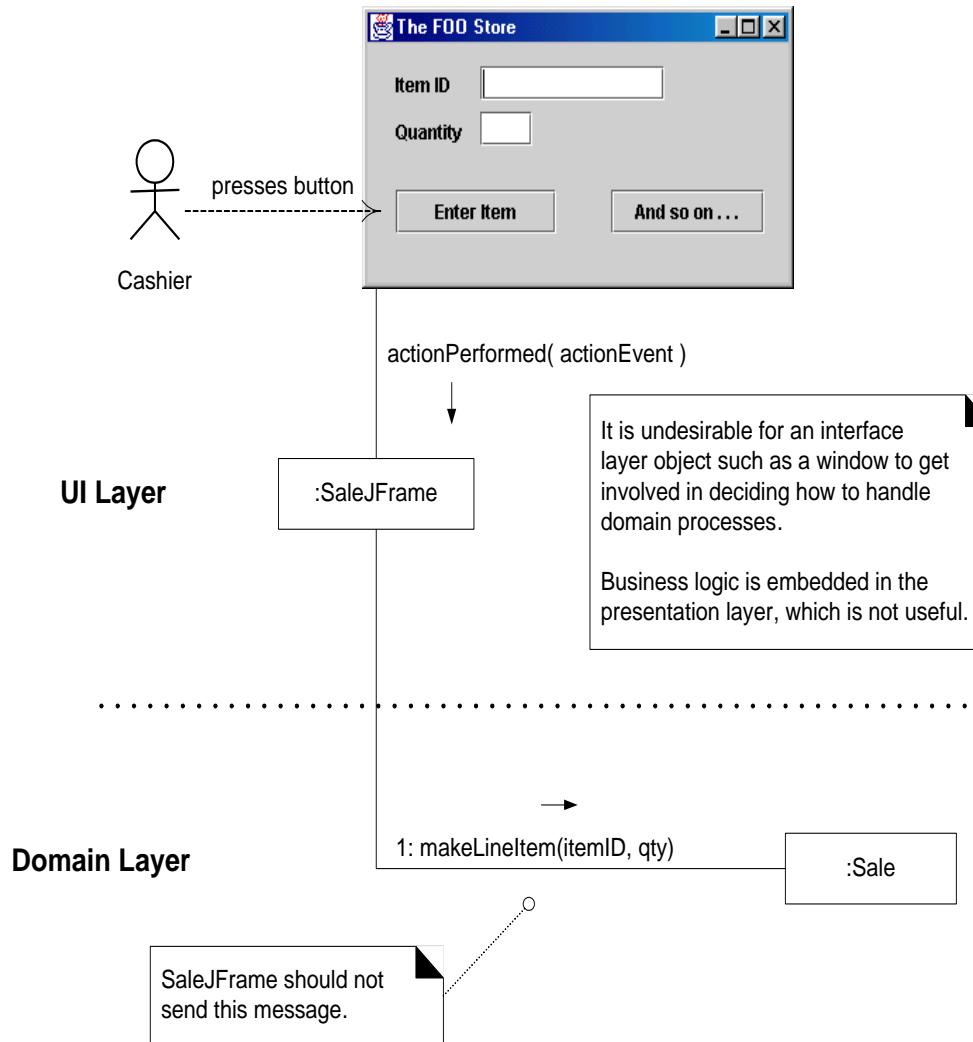
- Controller is basically a delegation pattern.
- It does not itself do work.
- UI Layer should not get involved with business logic.
- Controller pattern common choices of developer with respect to domain object delegate that receive the work request.
- Controller is a façade for the domain layer from UI layer.



# Some Concepts

- Boundary objects – abstraction of interfaces
- Entity objects- application-independent (typically persistent) domain software objects.
- Control objects – case handlers
- Façade controllers – suitable when there are not many system events.
- Use case controllers are specialised controllers for each use case. All requests in one use case should go to one controller to maintain sequence.

# Undesirable situations



- UI Layer not to handle domain layer (as in the sketch on left)

## Bloated Controllers to be avoided

- A single controller receiving all systems requests and there may be many.
- Controller handling systems events
- Controller maintains attributes



## Creator Pattern – Problem & Solution

# Creator

- Problem: Who should be responsible for creating a new instance of some class?
- Solution: Assign class B the responsibility to create
  - an instance of class A if one or more of the following is true:
    1. B *aggregates* A objects.
    2. B *contains* A objects.
    3. B *records* instances of A objects.
    4. B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A).

# • Creator

---

Assign class B the responsibility to create an instance of class A if one of the following is true:

1. B contains A
2. B aggregates A
3. B has the initialisation data for A
4. B records A
5. B closely uses A.

## Who Creates?

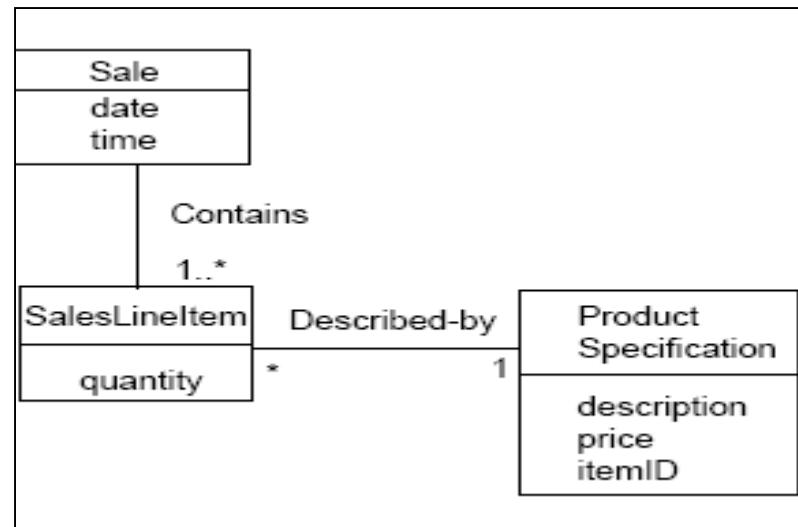
(Factory is a common alternate solution)



## Application of Creator Pattern in PoS System

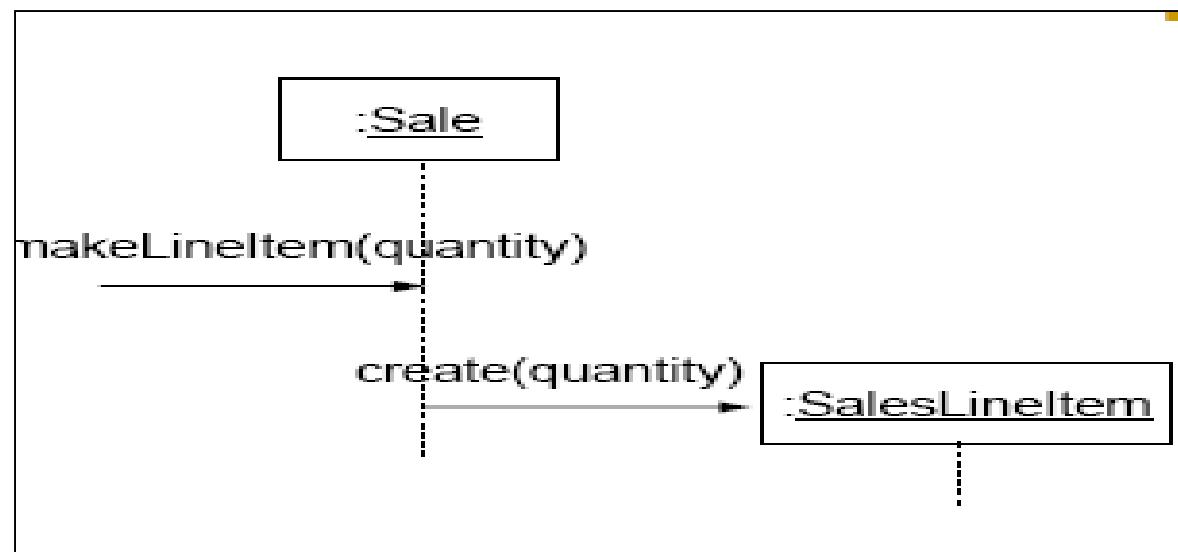
# Creator

- In the POS application, who should be responsible for creating a SalesLineItem instance?
- Since a Sale contains many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate.



# Creator

- This assignment of responsibilities requires that a makeLineItem method be defined in Sale.





## Low Coupling Pattern – Problem & Solution

# Low Coupling

- Coupling: it is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements.
- A class with high coupling depends on many other classes (libraries, tools).
- Problems because of a design with high coupling:
  - Changes in related classes force local changes.
  - Harder to understand in isolation; need to understand other classes.
  - Harder to reuse because it requires additional presence of other classes.
- Problem: How to support low dependency, low change impact and increased reuse?
- Solution: Assign a responsibility so that coupling remains low.

## • Low Coupling

---

- Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

How to reduce the impact of change?



Application of Low Coupling to optimize the design

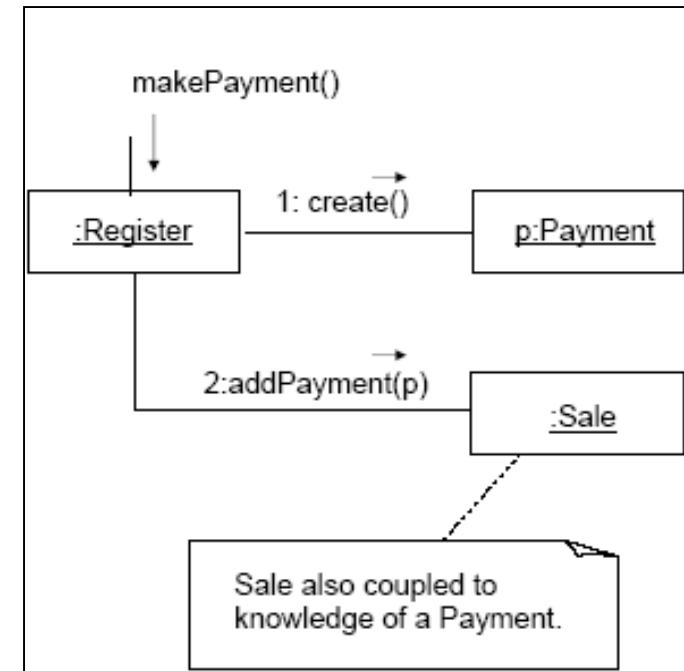
# Low Coupling

- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- By Creator, Register is a candidate.



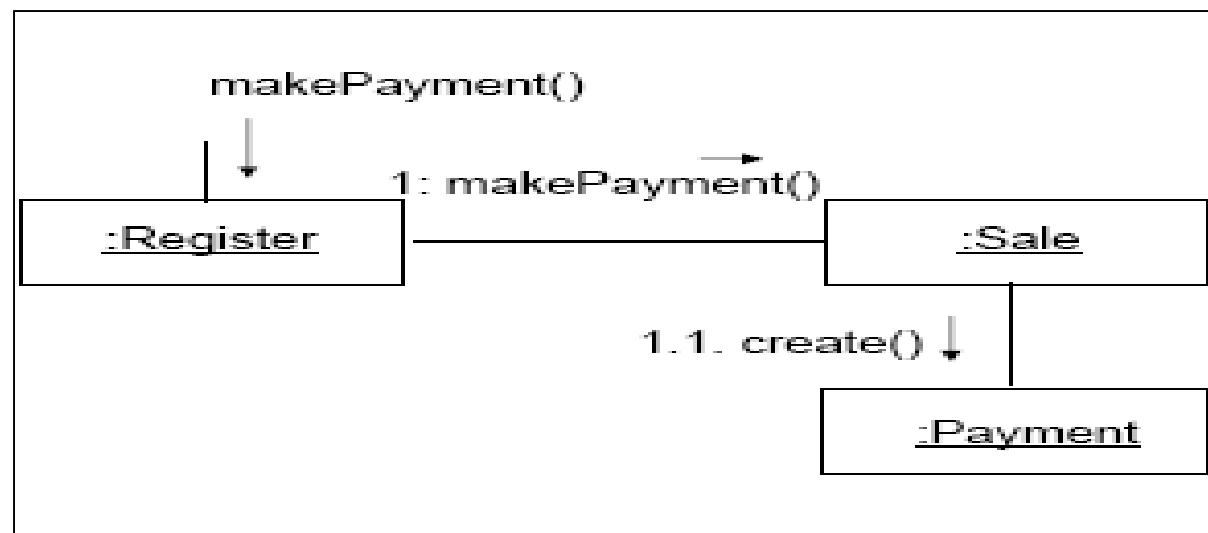
# Low Coupling

- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- The assignment of responsibilities couples the Register class to knowledge of the Payment class.



# Low Coupling

- An alternative solution is to create Payment and associate it with the Sale.
- No coupling between Register and Payment.



# Low Coupling

- Some of the places where coupling occurs:
  - Attributes: X has an attribute that refers to a Y instance.
  - Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
  - Subclasses: X is a subclass of Y.
  - Types: X implements interface Y.
- There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling.
- There will always be some coupling among objects, otherwise, there would be no collaboration.



## High Cohesion Pattern – Problem & Solution

# High Cohesion

- Cohesion: it is a measure of how strongly related and focused the responsibilities of an element are.
- A class with low cohesion does many unrelated activities or does too much work.
- Problems because of a design with low cohesion:
  - Hard to understand.
  - Hard to reuse.
  - Hard to maintain.
  - Delicate, affected by change.
- Problem: How to keep complexity manageable?
- Solution: Assign a responsibility so that cohesion remains high.

# High Cohesion

---

- Assign Responsibilities so that the cohesion remains high. Use this to evaluate alternatives.

How to keep objects focused, understandable, and manageable, and as a side-effect, support Low Coupling?

# High Cohesion

- Scenarios that illustrate varying degrees of functional cohesion
  1. Very low cohesion: class responsible for many things in many different areas.
    - e.g.: a class responsible for interfacing with a data base and remote-procedure-calls.
  2. Low cohesion: class responsible for complex task in a functional area.
    - e.g.: a class responsible for interacting with a relational database.

# High Cohesion

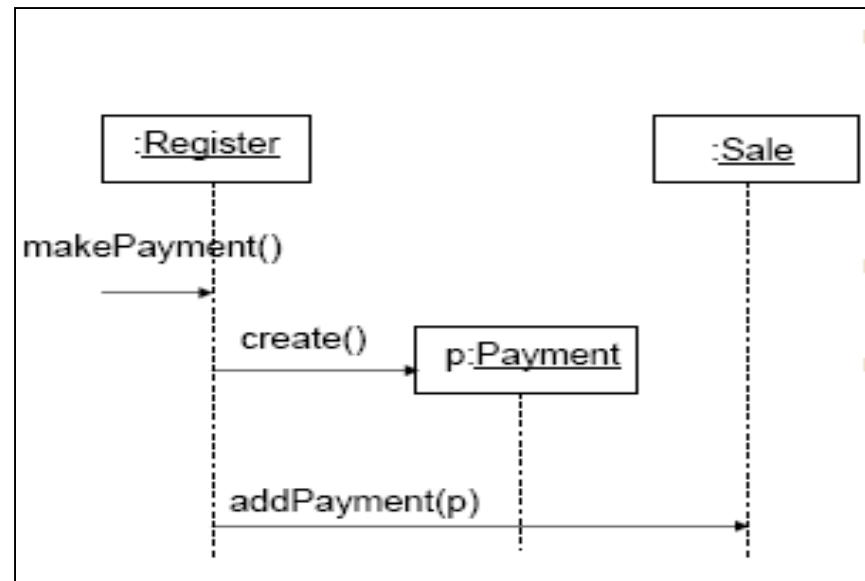
- 3. High cohesion: class has moderate responsibility in one functional area and it collaborates with other classes to fulfill a task.
  - e.g.: a class responsible for one section of interfacing with a data base.
  - Rule of thumb: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn't do much work. It collaborates and delegates.



Application of High Cohesion Pattern to optimize the design

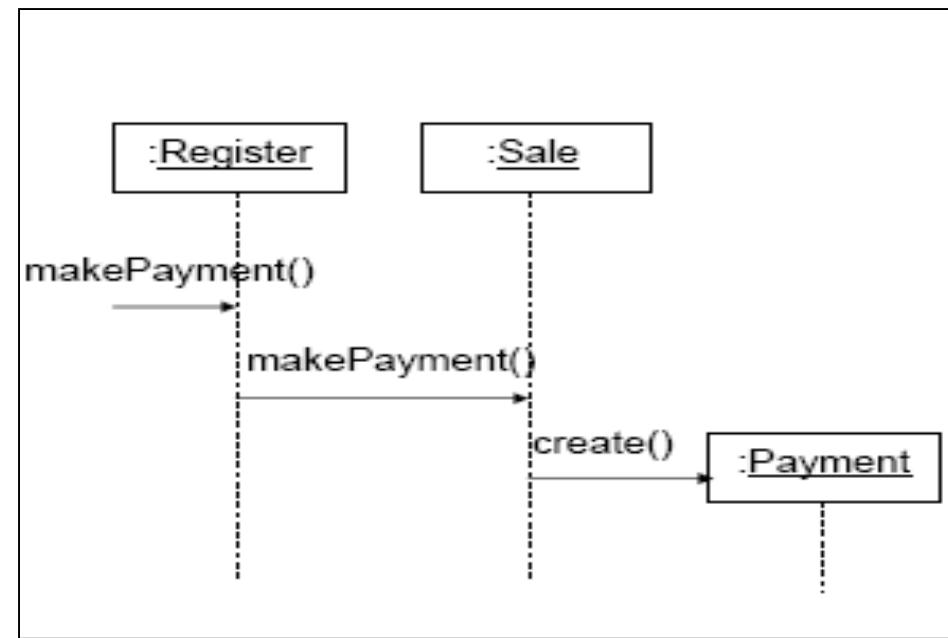
# High Cohesion

- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- By Creator, Register is a candidate.
- Register may become bloated if it is assigned more and more system operations.



# High Cohesion

- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion and low coupling.





Additional Patterns : Polymorphism, Pure Fabrication,  
Indirection & Protected Variation

# GRASP(Additional Design Patterns)

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variation

# Polymorphism

---

- When related alternatives or behaviours vary by type (class) assign responsibility for the behaviour- using polymorphism – to the types for which the behaviour varies.

Who is responsible when behaviour varies with type?

# Pure Fabrication

---

- Assign a higher cohesive set of responsibilities to an artificial or convenient “behaviour” class that does not represent a problem domain concept;
- This may sometimes be made up, in order to support high cohesion, low coupling and reuse.

Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?

# Indirection

---

- Assign the responsibility to an intermediate object to mediate between the components or services, so that they are not directly coupled.

How is assigned responsibility to avoid direct coupling?

# Protected Variation

- Identify points of predicted variation or instability;
- Assign responsibility to create a stable “interface” around them.

How to assign responsibility to objects, subsystems and systems so that the variations or services, so that they are not directly coupled.



## Polymorphism Pattern – Problem & Solution

# Polymorphism

- Solution: When related alternatives or behaviours vary by type (class), assign responsibilities for the behaviour (using polymorphic operations) to the types for which the behaviour varies.
- Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

Problem:  
How to handle alternatives based on type?  
How to create pluggable software components?

# Polymorphism

- In this pattern an interface that contains the services of the type is created and several adapters classes implement this interface providing their own implementation for the services. When a message is sent to one of these adapters, the adapter applies its own method implementation.
- Extensions required for new variations are easy to add, new implementations can be introduced without affecting clients.

Polymorphism has several related meanings. In this context it means: "giving the same names to services in different objects".



Application of Polymorphism Pattern to optimize the design

# Polymorphism

- When related responses (to the same event) vary by object type.
- Do not test object type and use conditional logic, apply “polymorphism.”
- The event is sent to the generalized (abstract) type, and sub-classes of the generalized type will exhibit polymorphism in having different implemented methods to respond to the event.

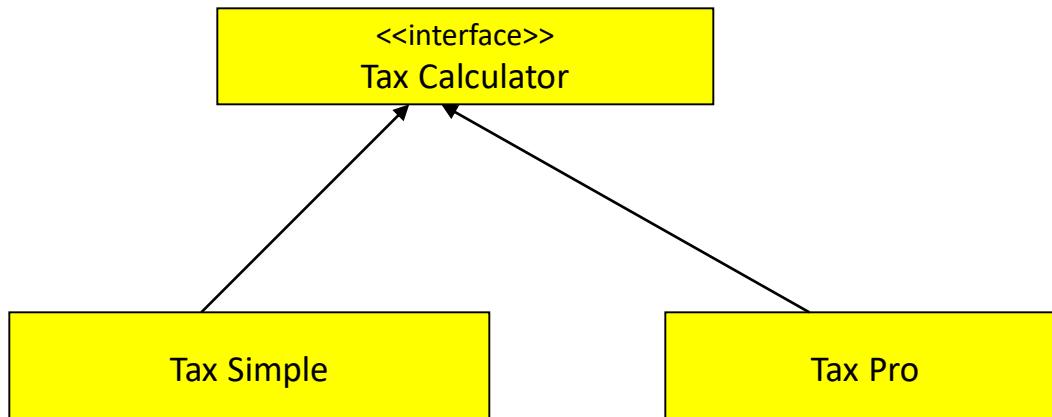
# Polymorphism

- Assign a polymorphic operation to the family of classes for which the cases vary.
  - Don't use case logic.

e.g., draw()

- Square, Circle, Triangle

e.g.





## Pure Fabrication Pattern – Problem & Solution

# Pure Fabrication

- Solution:
- Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept, something made up to support high cohesion, low coupling and reuse.
- Such class is a fabrication of the imagination. Ideally the responsibilities assigned to this fabrication support high cohesion and low coupling.

Problem:

What object should have the responsibility when you do not want to violate High Cohesion and Low Coupling.

# Pure Fabrication

- High Cohesion is supported because responsibilities are factored into a fine grained class that only focuses on a very specific set of related task.  
e.g. Instead of writing database logic in Sale object, design separate artificial class “PersistanceHandler”

Pure Fabrication is chosen by behavioural decomposition. Reuse potential may increase.



Application of Pure Fabrication Pattern to optimize the design

# Pure Fabrication

## Case

- You need to save Sale instance in a relational database.
- By Information Expert: Assign the responsibility to Sale.
- Consider Pure Fabrication of PersistentStorage Class. This term is not in the domain model but may be constructed for the convenience of the programmer.

Consider the following factory against **Expert**

- The tasks requires a relatively large number of database related operations, none of which are related to Sale (Low Cohesion)
- Sales Class will have to be coupled to a relational database which is external to the domain. (High Coupling)
- Saving objects to a database is a General task required by many classes. Placing it in sale will yield Low Reuse.



## Indirection Pattern – Problem & Solution

# Indirection

- Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an indirection between the other components.
- Lower coupling between components.

## Problem:

Where to assign a responsibility to avoid direct coupling between two or more things?

How to decouple objects so that low coupling is supported and reuse potential remains higher?

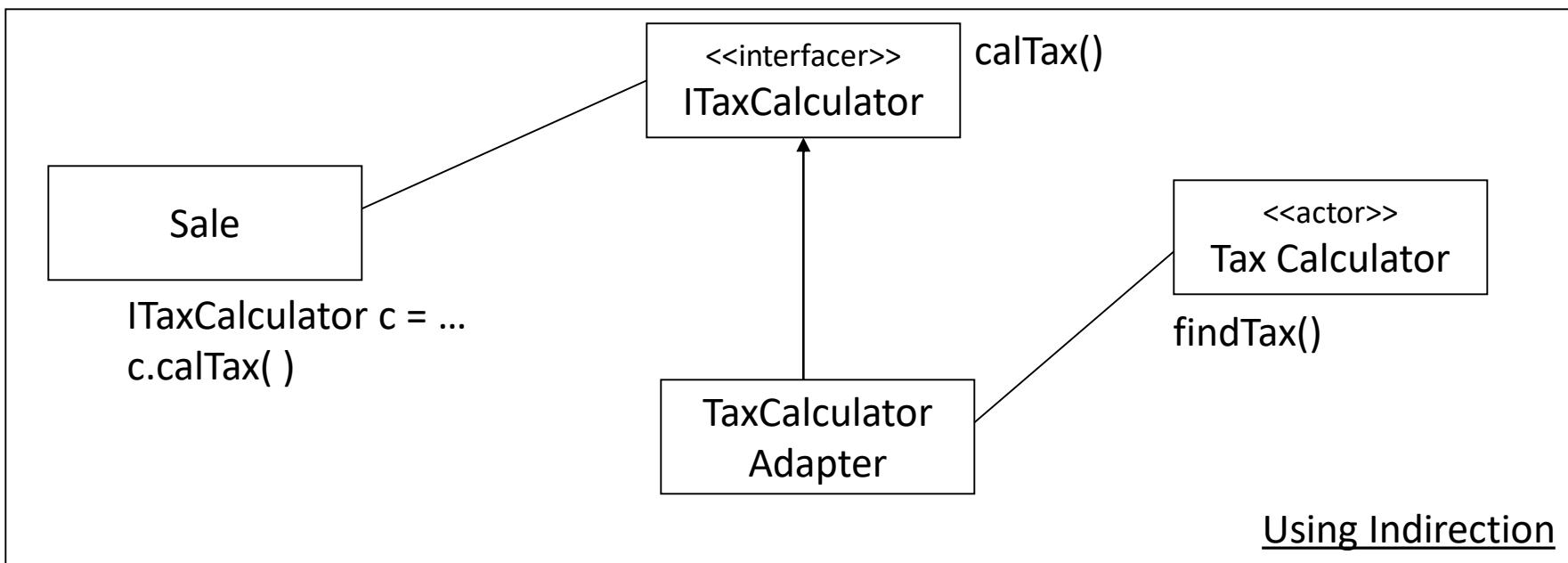
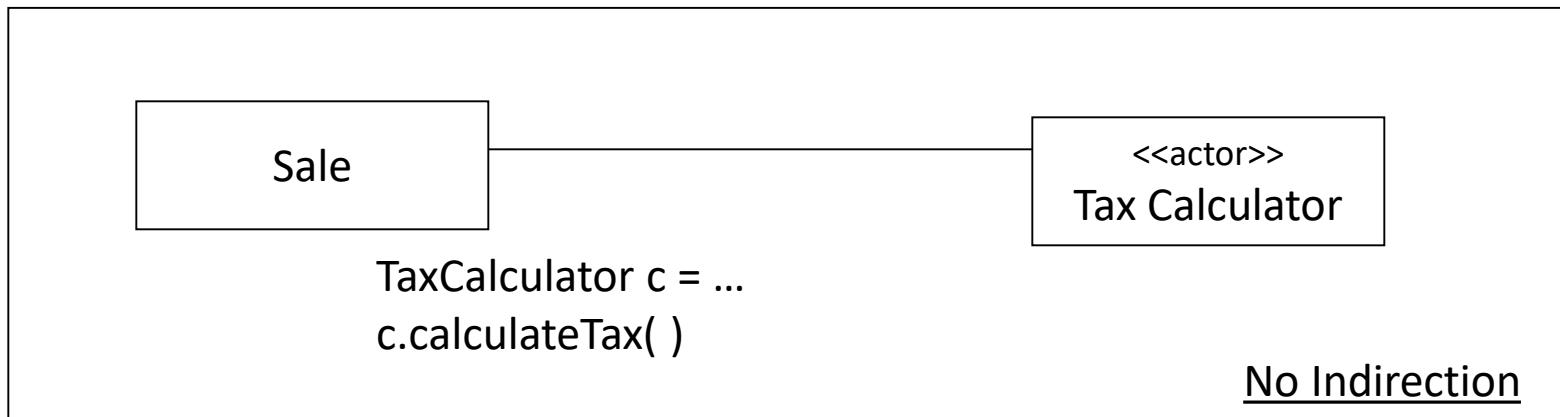


Application of Indirection Pattern to optimize the design

# Indirection

- When two objects need to interact, but they cannot get to know each other well.
- We may abstract the interaction into another object which acts as a mediator, working on behalf of one of the objects.
- Example: Listener objects in Java graphical user interface;

# Indirection





## Protected Variation Pattern – Problem & Solution

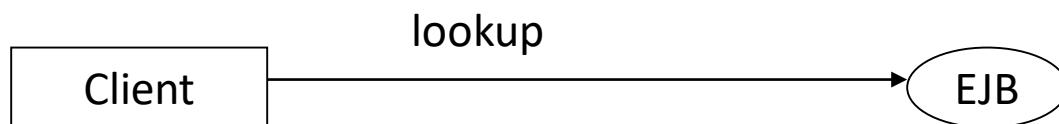
# Protected Variation

## Solution:

- Identify points of predicted variation or instability;
- assign responsibilities to create a stable interface (or protection mechanism) around them.

## Problem:

How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements.



# Protected Variation

- Technology like Service Lookup is an example of PV because clients are protected from variations in the location of services using the lookup service.
- Externalizing properties in a property file is another example of PV.

Extensions required for new variations are easy to add.

New implementations can be introduced without affecting clients.

# Protected Variation

PV through various mechanisms.

They are

Core: The “Core” mechanism for PV is as following

- Data encapsulation
- Interfaces
- Polymorphism
- Indirection

Date driven Design

- Reading codes, class file paths, class name

Service Look up

- UDDI :Universal Description, Discovery and Integration is a registry which is platform independent which is modality to register and locate web services i.e. in turn it helps to look up the services
- LDAP: It is Lightweight Directory Access Protocol which is meant for accessing (connecting), search and modify the internet directories. This is service protocol running a layer above TCP/IP stack.

# Protected Variation

## Interpreter Driven Design

- Virtual machines
- Neural networks
- Logic engines
- Rule interpreters

## Reflective of meta level design

- Java introspector
- .NET reflectors

## Uniform access

- Language supported constructs that do not change with underlying implementation
- E.g. method and attributes are invoked same way (C#), Standards (e.g. APIs like ADO.NET, ODBC, JDBC etc)

Wherever these mechanisms are encountered one has an implementation of PV in some or other form.



Application of Protected Variation Pattern to optimize the design

# Protected Variation

- This pattern is in line with the pure fabrication and Indirections in a sense that it addresses similar concerns.
- The principles are same like low coupling high cohesion, reuse but the focus is on protecting the existing objects from variations. i.e. creating a stable interface so to protect from variations in coupled objects.

**Approach:** “Find what varies and encapsulate it.”

**Step I:** Closely look for elements with direct coupling and also relatively prone to change.

**Step II:** Identify for the objects or points of predicted variation

**Step III:** Assign these responsibilities in such a way to create a stable interface around them.

# Protected Variation

- According to Larman, PV (Protected Variations) is a very important and root principle motivating most of the mechanism and patterns in programming and design to provide flexibility and protection from variation, almost every design trick in his book is a specialization of the Protected Variations pattern.

POS.

- Consider Classes: Sale, Payment, and SaleLineItem etc.
- Every sale amount has taxes applied to arrive at the total.
- There are different tax rules at different places and tax rates are different also.
- On top of this the rates and rules are bound to changes which makes the design of such systems tricky.

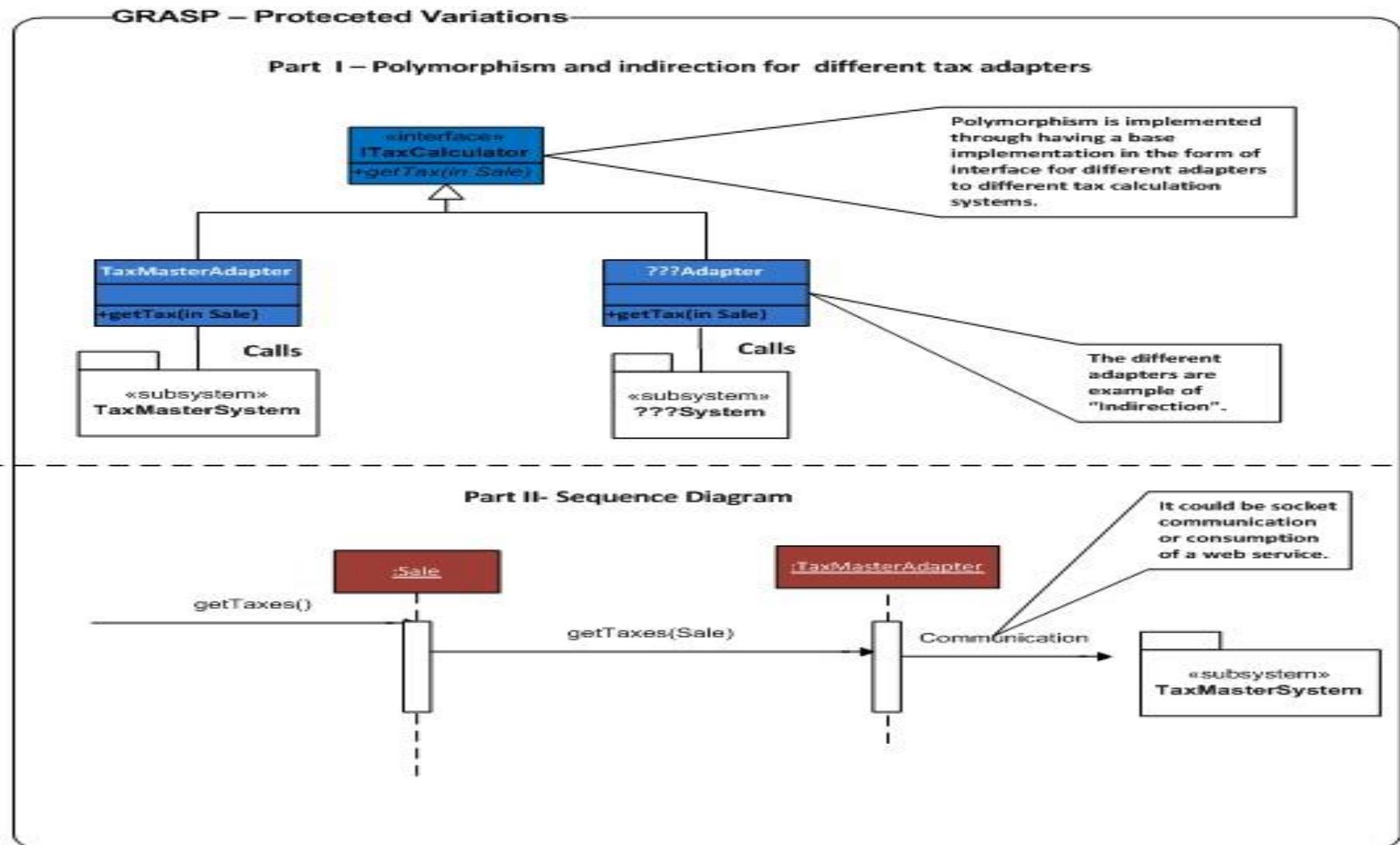
# Protected Variation

- The best approach is design the tax calculation as separate system.
- There could be different tax calculator systems (third party) available or there could be web service providing the tax calculation services.
- There might be reason for having multiple systems or move to different system.
- As mentioned above, the rules, rates are the moving target and as these changes the tax calculation systems have to follow.
- This is the point of variation or stability. In such scenario, the changes would require the consumers or users of such systems, to change also.

# Protected Variation

- These changes or variations would cause the changes in POS system or it may require revision.
- This is where the “protected Variations” pattern helps to design in such a way that the changes in the tax calculation system would not cause major problems. This is accomplished through implementation of polymorphism and indirection as depicted in the following diagram.

# Protected Variation



# Protected Variation

Class	Responsibility and method	Remarks
ITaxCalculator	Facilitator for different adapters to different systems	Provides polymorphic behaviour and handles the variations based on the type.
....Adapter	Facilitator for particular Tax System	These classes provide indirection i.e. facilitates low coupling and reuse.



## Introduction to SOLID Design Principles

# SOLID

## **first 5 principles of object-oriented design**

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

# Single Responsibility Principle (SRP)

- a class should have one, and only one, reason to change.

Just because you can, does not mean you should

# Open-Closed Principle (OCP)

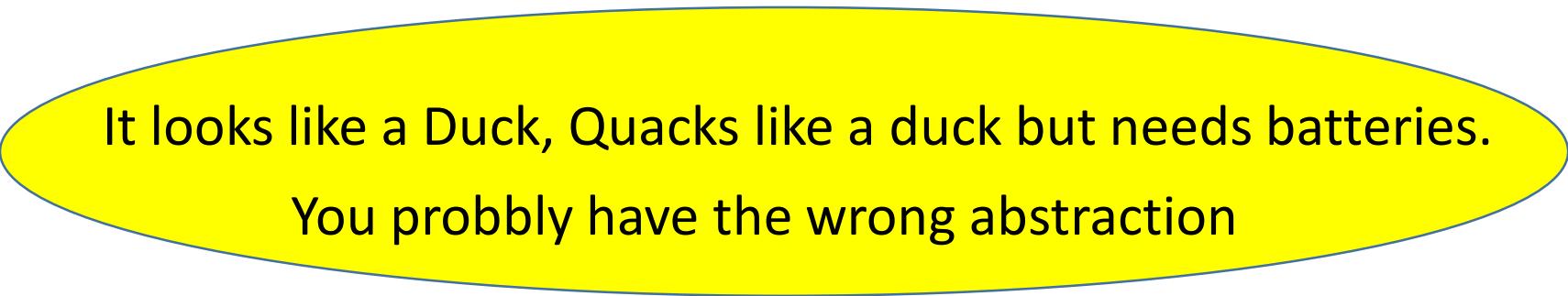
- you should be able to extend a class's behavior, without modifying it.

*Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.*

Open Heart Surgery is not necessary when putting on a coat

# Liskov Substitution Principle (LSP)

- -- derived classes must be substitutable for their base classes.



It looks like a Duck, Quacks like a duck but needs batteries.  
You probbly have the wrong abstraction

# Interface Segregation Principle (ISP)

- make fine grained interfaces that are client specific.

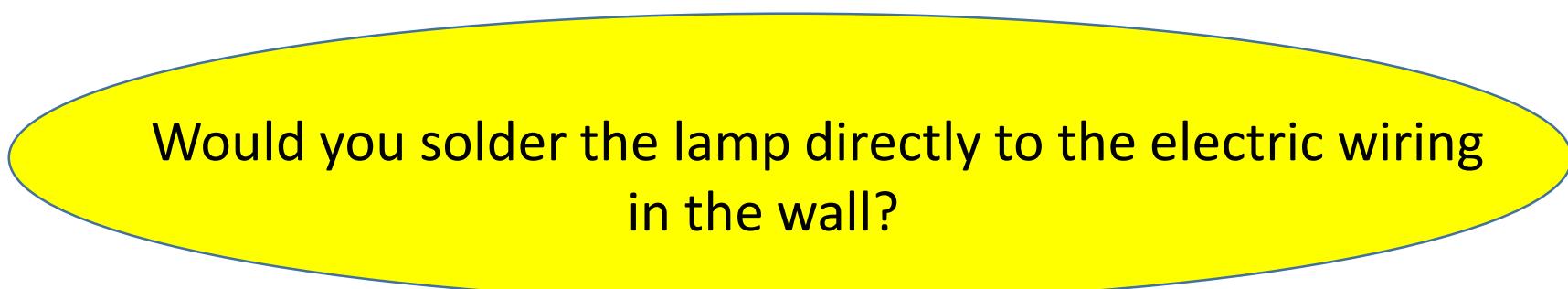
*Clients should not be forced to depend on methods  
they do not use*



*You want me to plug in one more Jack?  
Where do I do it?*

# Dependency Inversion Principle (DIP)

- depend on abstractions not on concrete implementations.



Would you solder the lamp directly to the electric wiring  
in the wall?

# Some essential Design Tips

- Keep It Simple Stupid (KISS)
- You Ain't Gonna Need It (YAGNI)
- Don't Repeat Yourself (DRY)

# Where is the stink from?

- Duplicate code
- Long method
- Large class
- Temporary field
- Switch statements
- Parallel inheritance hierarchies

# What Design needs to correct

- Rigidity
  - Software is difficult to change
- Fragility
  - Program breaks in many places when a change made in a single place
- Immobility
  - Parts could be useful in other systems, but effort and risk to separate from original system is too great

# What Design needs to correct

- Viscosity
  - Design-preserving methods are more difficult to use than the hacks
  - Development environment is slow and inefficient
- Needless complexity
  - Contains elements that aren't currently useful
- Needless repetition
  - System has lots of repeated code elements
- Opacity
  - A module is difficult to understand

# 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 Gamma | Richard Helm | Rapph Johnson | John Vlissides alias GoF.



## Single Responsibility Principle (SRP)

# Single Responsibility Principle (SRP)

- a class should have one, and only one, reason to change.

Just because you can, does not mean you should

# Single Responsibility Principle (SRP)

- Responsibility
  - What a class does
  - The more a class does, the more likely it will change
  - The more a class changes, the more likely we will introduce bugs

# Single Responsibility Principle (SRP)

- Cohesion and Coupling
  - Cohesion – How closely related are the different responsibilities of a module
  - Coupling – How much one module relies on another
  - Goal is low coupling and high cohesion



## Open-Closed Principle (OCP)

# Open-Closed Principle (OCP)

- you should be able to extend a class's behavior, without modifying it.

*Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.*

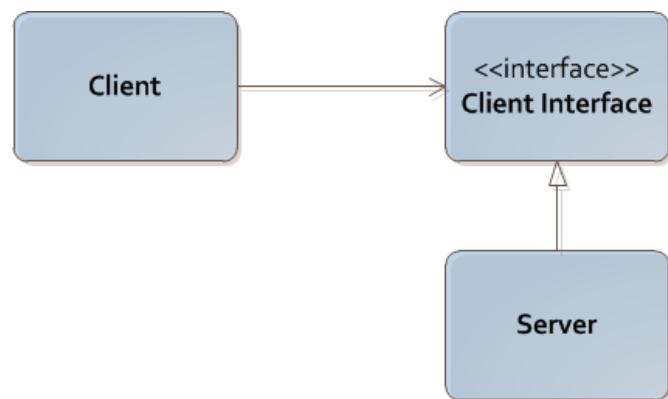
Open Heart Surgery is not necessary when putting on a coat

# OCP

- Open for extension
  - Behavior of the module can be extended
  - We are able to change what the module does
- Closed for modification
  - Extending behavior does not result in changes to source, binary, or code of the module

# Implementing OCP

- Note this situation
- Rely on abstractions
  - Interfaces
  - Abstract classes
- Strategy Pattern below

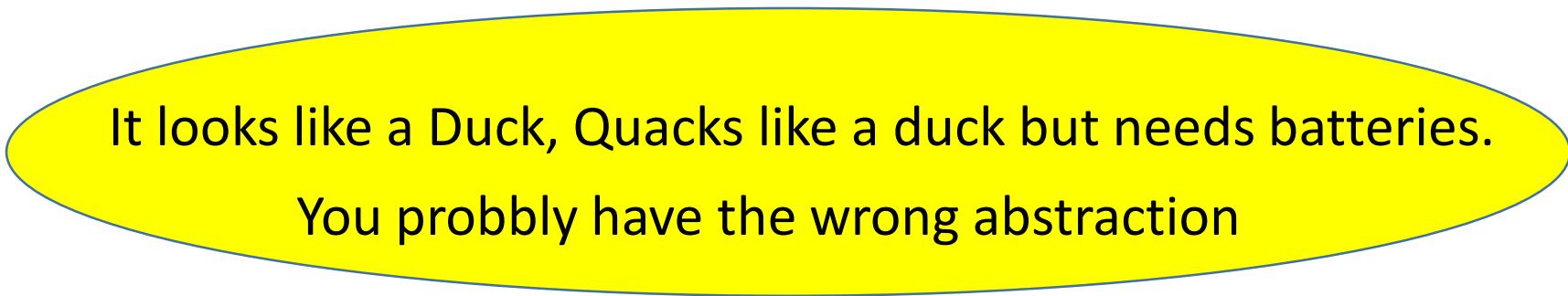




## Liskov Substitution Principle (LSP)

# Liskov Substitution Principle (LSP)

- -- derived classes must be substitutable for their base classes.



It looks like a Duck, Quacks like a duck but needs batteries.  
You probbly have the wrong abstraction

# Liskov Substitution Principle (LSP)

IS – A relation

- Basic OOP discusses inheritance with “IS-A”
- LSP says that “IS-A” refers to *behavior*
- *Behavior* is what software is really all about

Substitute

- Calling code should not know that one module is different from its substitute



## Interface Segregation Principle (ISP)

# Interface Segregation Principle (ISP)

- make fine grained interfaces that are client specific.

*Clients should not be forced to depend on methods  
they do not use*



*You want me to plug in one more Jack?  
Where do I do it?*

# Acknowledgement

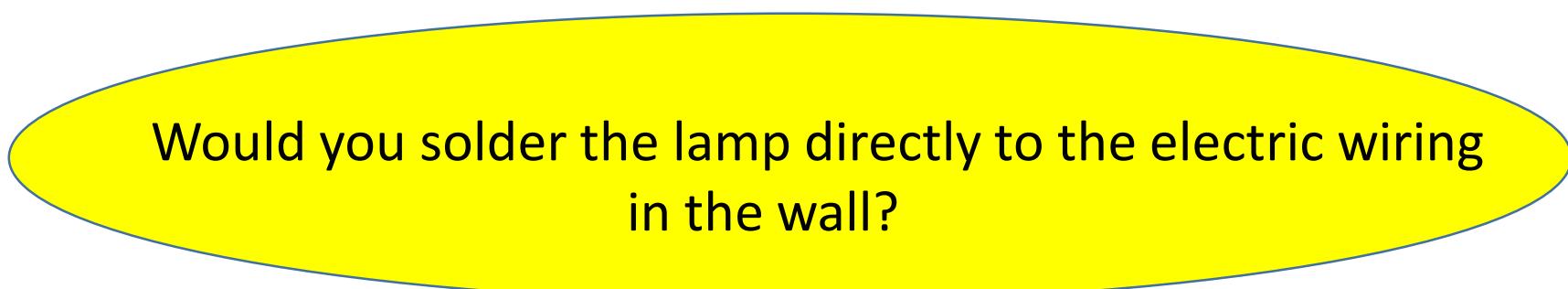
- Slides are based on Course Text Books:
  - Applying UMP and Patterns (An Introduction to Object-Oriented Analysis and Design and Iterative Development) : Craig Larman
  - UMP Distilled (A Brief Guide to the Standard Object Modeling Langauge) : Martin Fowler
  - Design Patterns (Elements of Reusable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



## Dependency Inversion Principle (DIP)

# Dependency Inversion Principle (DIP)

- depend on abstractions not on concrete implementations.



Would you solder the lamp directly to the electric wiring  
in the wall?

# Dependency Inversion Principle (DIP)

*High-level modules should not depend on low-level modules. Both should depend on abstractions*

*Abstractions should not depend on details. Details should depend upon abstractions*



# Object Oriented Analysis & Design



**BITS** Pilani  
Pilani Campus

Paramananda Barik



# Object Oriented Analysis & Design



**BITS** Pilani  
Pilani Campus

Paramananda Barik



# Object Oriented Analysis & Design

## Module-7 (RL 7.1.1)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus

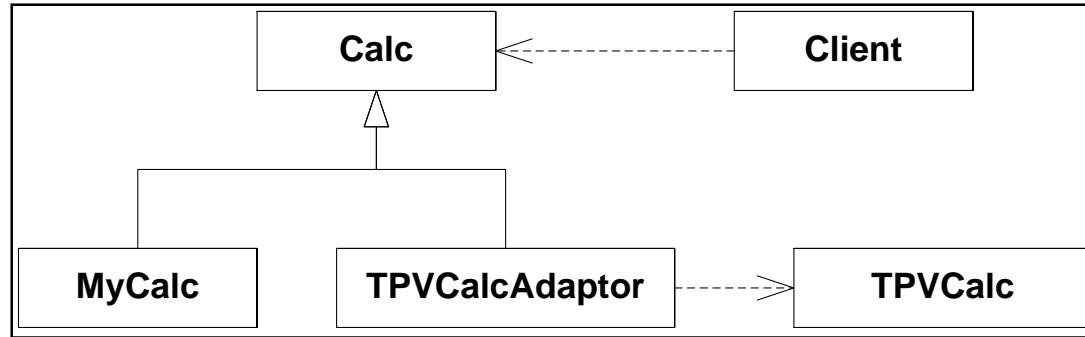


## Adapter Pattern – Problem & Solution

# Adapter (GoF) pattern.

- Problem: How to resolve incompatible interfaces
- Solution: Convert the original interface of a component into another interface through an intermediate adapter object.
- Adapters use interfaces and polymorphism to add a level of indirection to varying APIs in other components.

# Adaptor Pattern Code Example



- The calc class provides one of 2 types of Calc's
  - A self written calc MyCalc
  - A Third Party Vendor(TPV) Calc.
- The TPV Calc doesn't use the same interfaces as MyCalc:
  - Create an abstract class Calc
  - Create an Adaptor to wrap the TPVCalc

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-7 (RL 7.1.2)

Harvinder S Jabbal

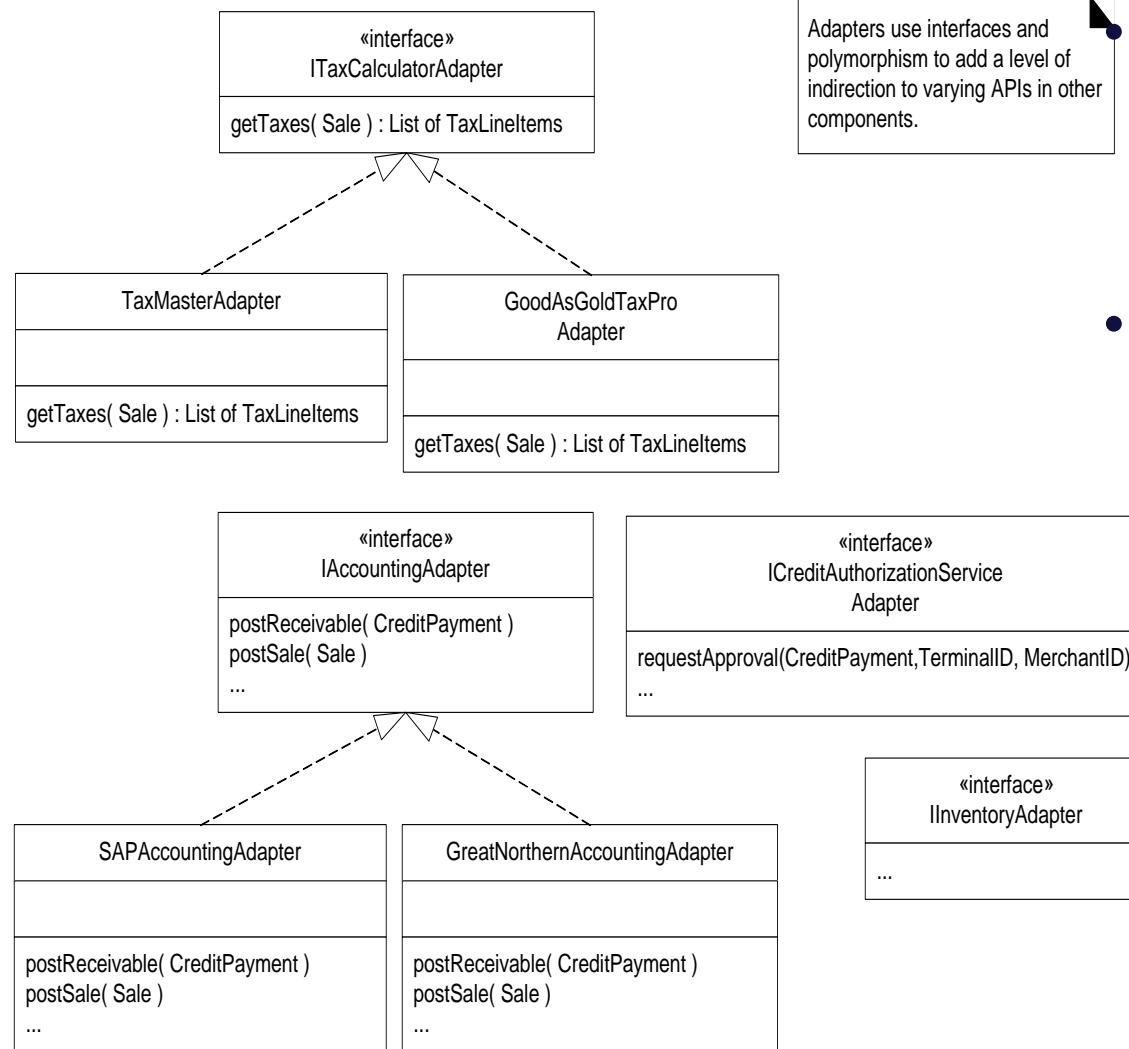


**BITS** Pilani  
Pilani Campus



## Application of Adapter Pattern to PoS

# The Adapter pattern : PoS

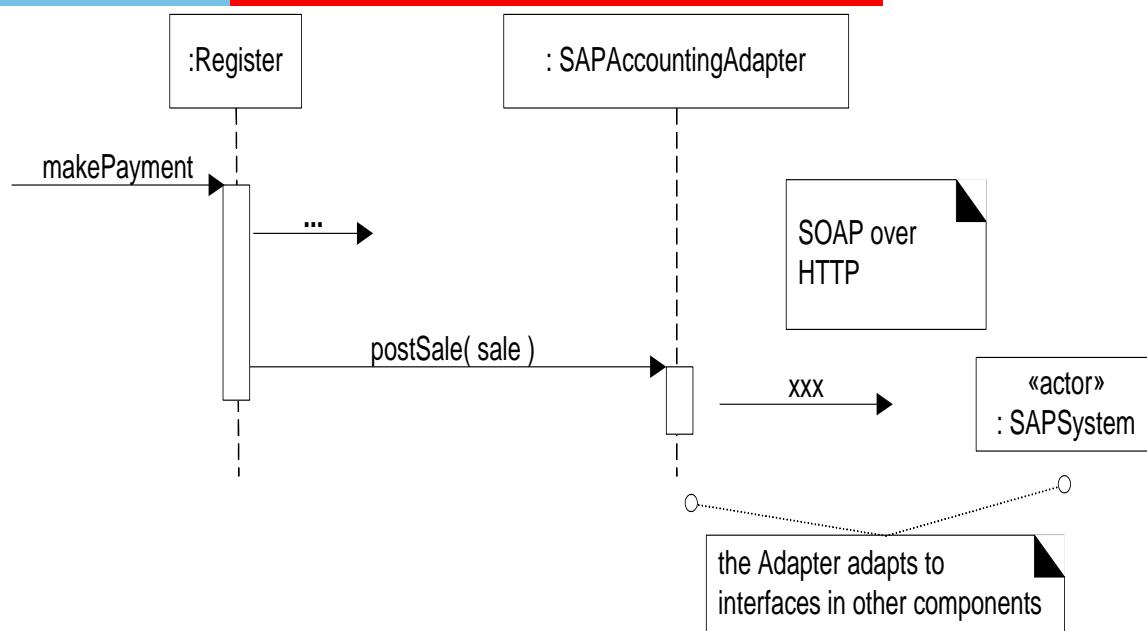


A particular adapter instance will be instantiated for the chosen external class, such as SAP.

- Adapt the `postSale` request to external service: eg SOAP XML interface over HYYPS for internet Web service offered by SAP.

# The Adapter pattern : PoS

- Note the use of pattern in Type Name.
- This helps identify the diagram to user.
- A resource adapter may be viewed as a facade to an external system



# GoF Adapter and some core GRASP principles

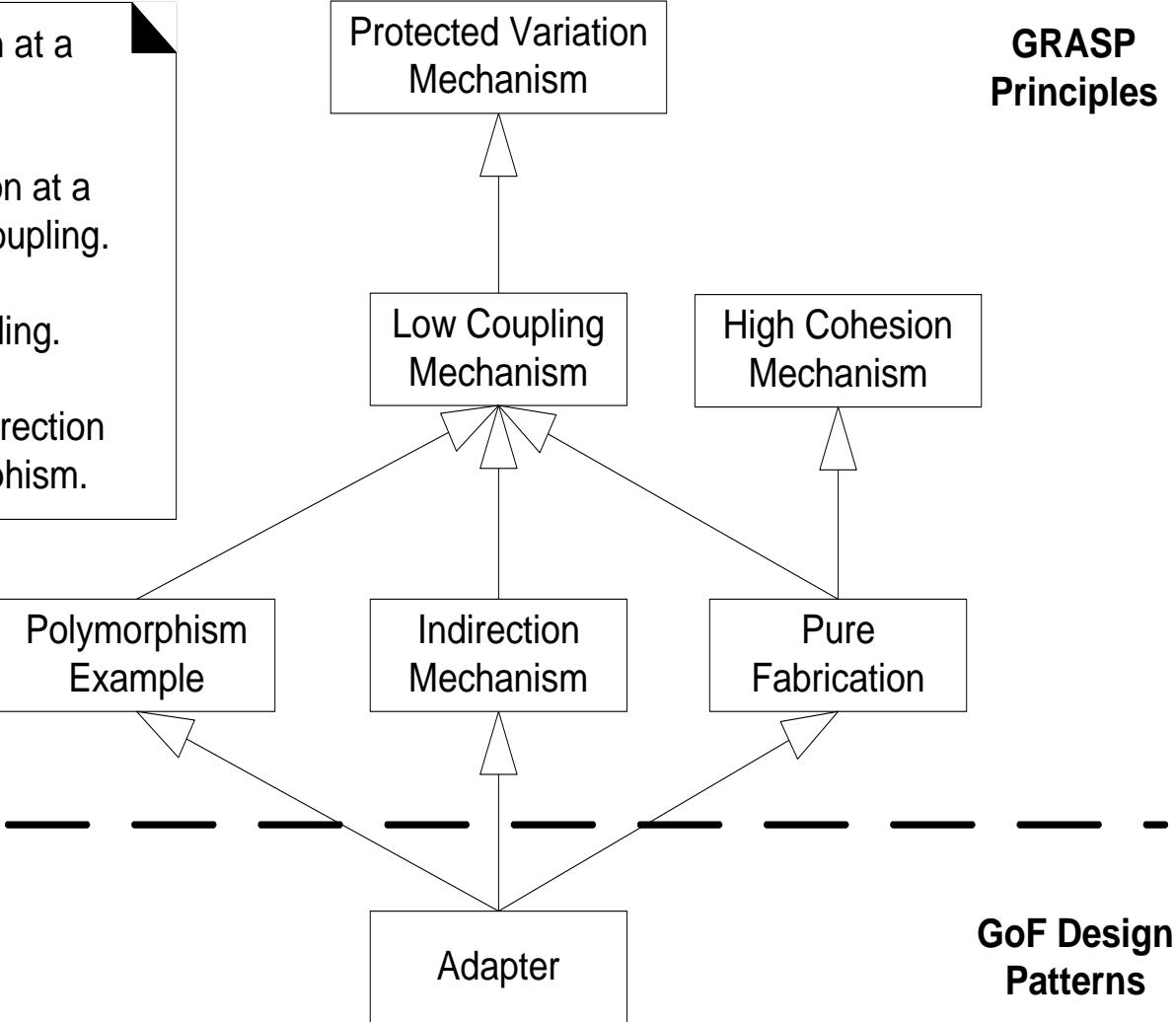


Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.



GoF Design  
Patterns

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-7 (RL 7.1.3)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus



## Factory Pattern – Problem & Solution

# Factory (GoF) pattern.

- Problem: Who should be responsible for creating objects when there are special considerations such as complex creation logic, a desire to separate the creation responsibilities for better cohesion and so forth?
- Solution: Create a Pure Fabrication object called Factory that handles the creation.
- A Factory can handle fabrication of several different types of objects. I can have different creation methods each one for the related type.

# Factory (GoF) pattern.

- We had discussed requirement of Adapter classes
- Suppose we require multiple adapter classes
- Who will create all instances of these Adapter classes
- Involving ‘Register’ domain object in this task is not proper
- Solution is Factory Class

# The Factory Pattern

## Apply Factory Pattern

- It is a pure fabrication
- Separates the responsibility of complex creation into cohesive helper objects
- Hides potential complex creation logic
- Allows introduction of performance-enhancing memory management strategies, such as object caching or recycling.

- Separation of Concern to be upheld
- Domain layer should emphasise application logic
- External connectivity should be handled by a different group of 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-7 (RL 7.1.4)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus

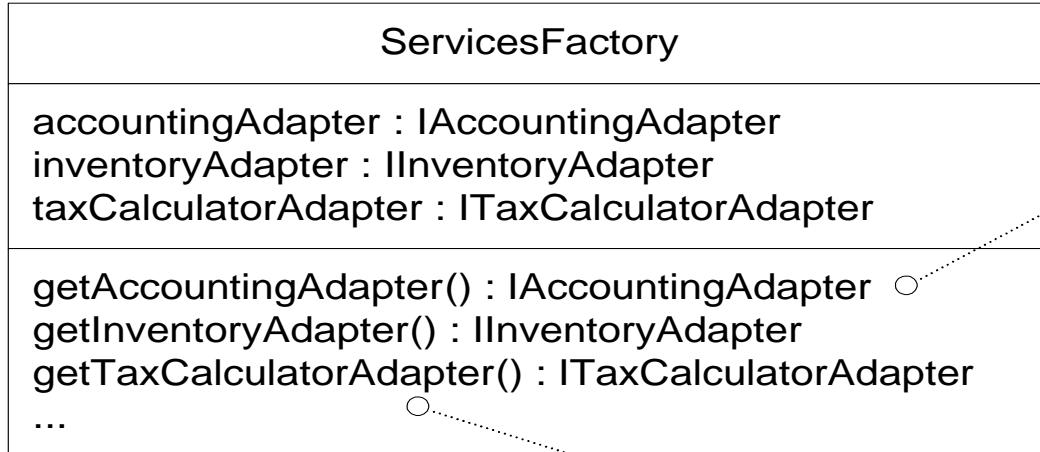


Application of Factory Pattern to PoS.

# The Factory Pattern: PoS

- Also called Simple Factory or Concrete Factory
- Simplification of GoF Abstract factory Pattern.
- Who creates the adapter?
- Who decides which class of adapter to create?

# The Factory Pattern: PoS



note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```

if ( taxCalculatorAdapter == null )
{
    // a reflective or data-driven approach to finding the right class: read it from an
    // external property

    String className = System.getProperty( "taxcalculator.class.name" );
    taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();

}

return taxCalculatorAdapter;

```

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-7 (RL 7.2.1)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus



## Singleton Pattern – Problem & Solution

# Singleton (GoF) pattern.

- Problem: Exactly one instance of a class is allowed, it is a singleton. Objects need a global and single point of access.
- Solution: Define a static method of the class that returns the singleton.
- The static method of the singleton should be named "getInstance()".
- A singleton is represented in an interaction diagram using a stereotype <<singleton>>. By doing this is not necessary to explicitly show the getInstance() message before sending a message to the singleton instance.

# Singleton (GoF) pattern.



```
public static Register getInstance()
{
    if(instance == null)
        instance = new Register();
    return(instance);
}
```

# Singleton(GoF)

---

- The Choice of Adapter was handled by ServiceFactory
  - The ServiceFactory raises another issue: Who creates the Factory itself?
  - You have to get visibility to this single ServiceFactory instance?
- 1. Only one instance of the factory is needed in the process.
  - 2. Methods of this factory may need to be called from various places in the code.

# Singleton

- The SingleFactory instance can be passed around as a parameter to every place a visibility need is discovered for it
  - These are inconvenient approaches
  - Use Singleton pattern instead.
- OR
- The objects that need visibility may be instantiated with a permanent reference

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-7 (RL 7.2.2)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus

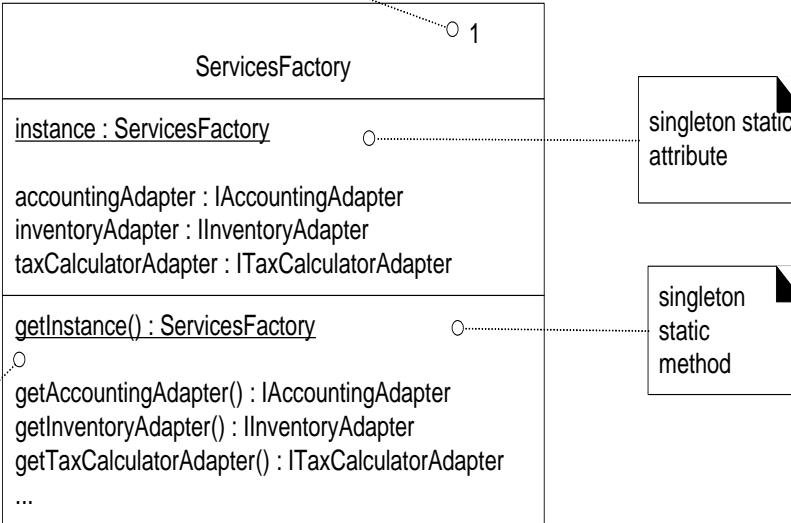


Application of Singleton Pattern to PoS.

# The Singleton pattern in the ServiceFactory Class

UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)

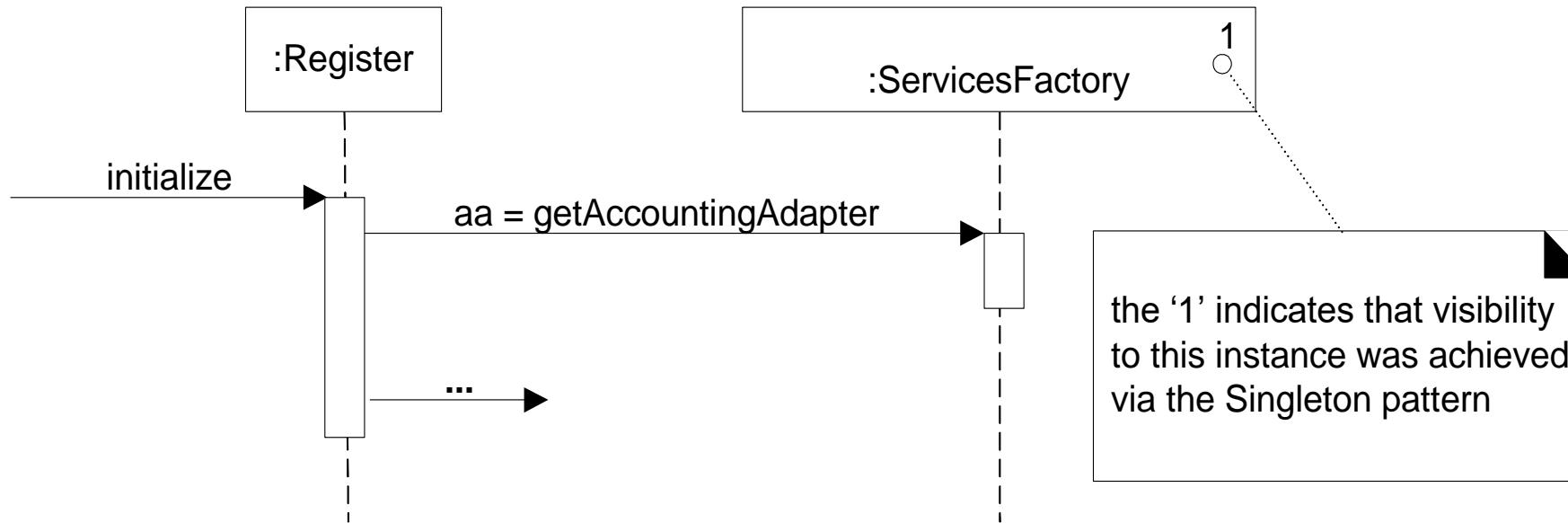
UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member



```
// static method
public static synchronized ServicesFactory getInstance()
{
if ( instance == null )
    instance = new ServicesFactory()
return instance
}
```

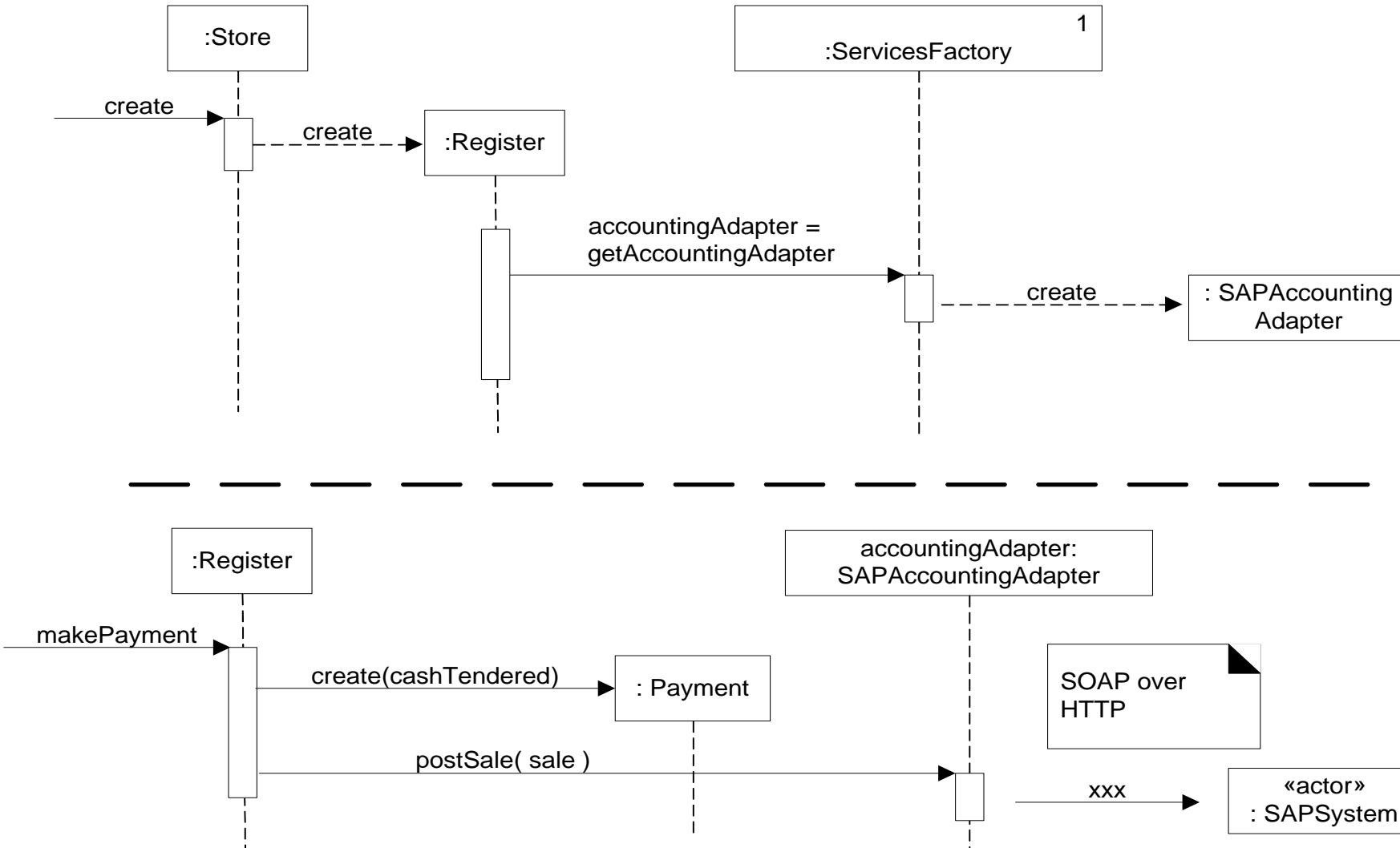
- Singleton is illustrated with a “1” on the top right
  - Key idea: class X defines a static method getInstance that itself provides a single instance of X
  - Developer has global visibility to this single instance via static getInstance method of the class.

# Implicit getInstance Singleton pattern message



- Note the “1” mark on the Service factory Class

# Adapter, Factory and Singleton pattern applied to the design



# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-7 (RL 7.2.3)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus

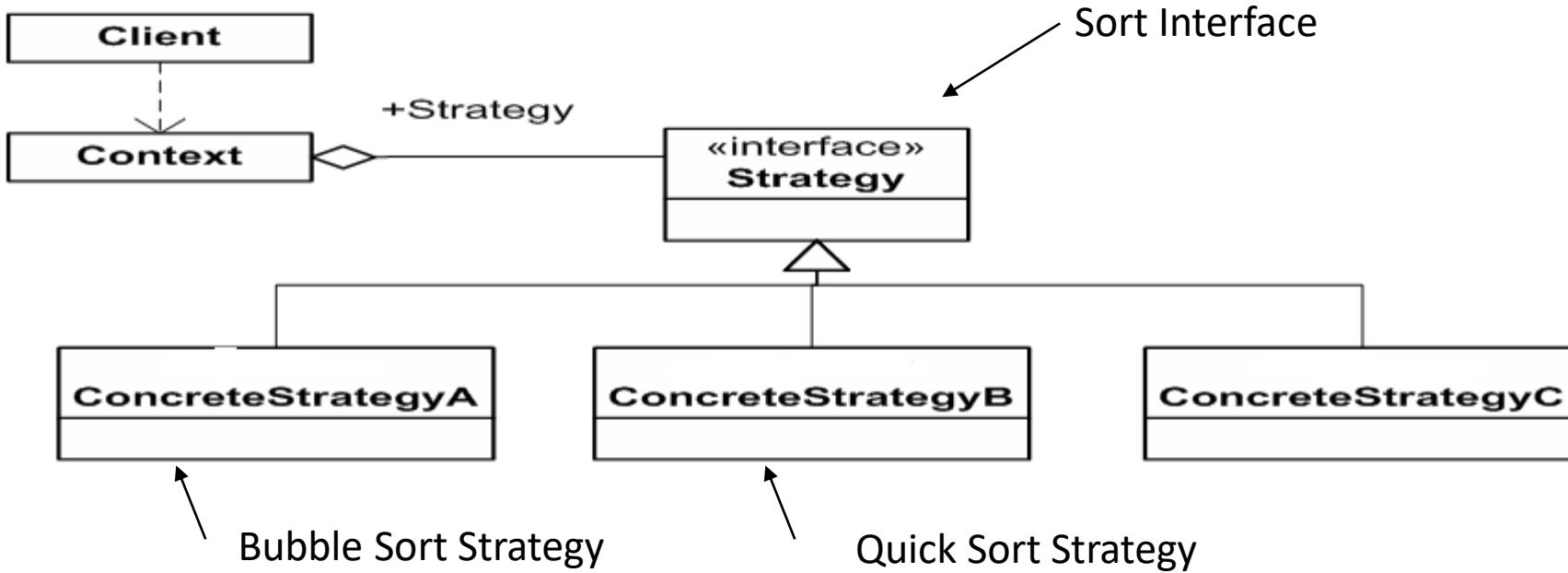


## Strategy Pattern – Problem & Solution

# Strategy (GoF) pattern.

- Problem: How to design for varying but related algorithms or policies? How to design for the ability to change these algorithms or policies?
- Solution: Define each algorithm/policy/strategy in a separate class with a common interface.
- The interface for the Strategy Object must accept as a parameter a reference of the Context Object.
- A Strategy object is used only by one Context Object.
- When a Context Object sends a message to the Strategy object a reference of the Context Object must be passed as a parameter for further collaboration.
- The Context Object must use a reference to the Strategy interface (no to the Strategy implementation) to send messages.

# Strategy (GoF) pattern.



- Client: provides algorithm to the context
- Context: uses abstract interface to algorithm
- Adding new algorithms does not impact context

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-7 (RL 7.2.4)

Harvinder S Jabbal

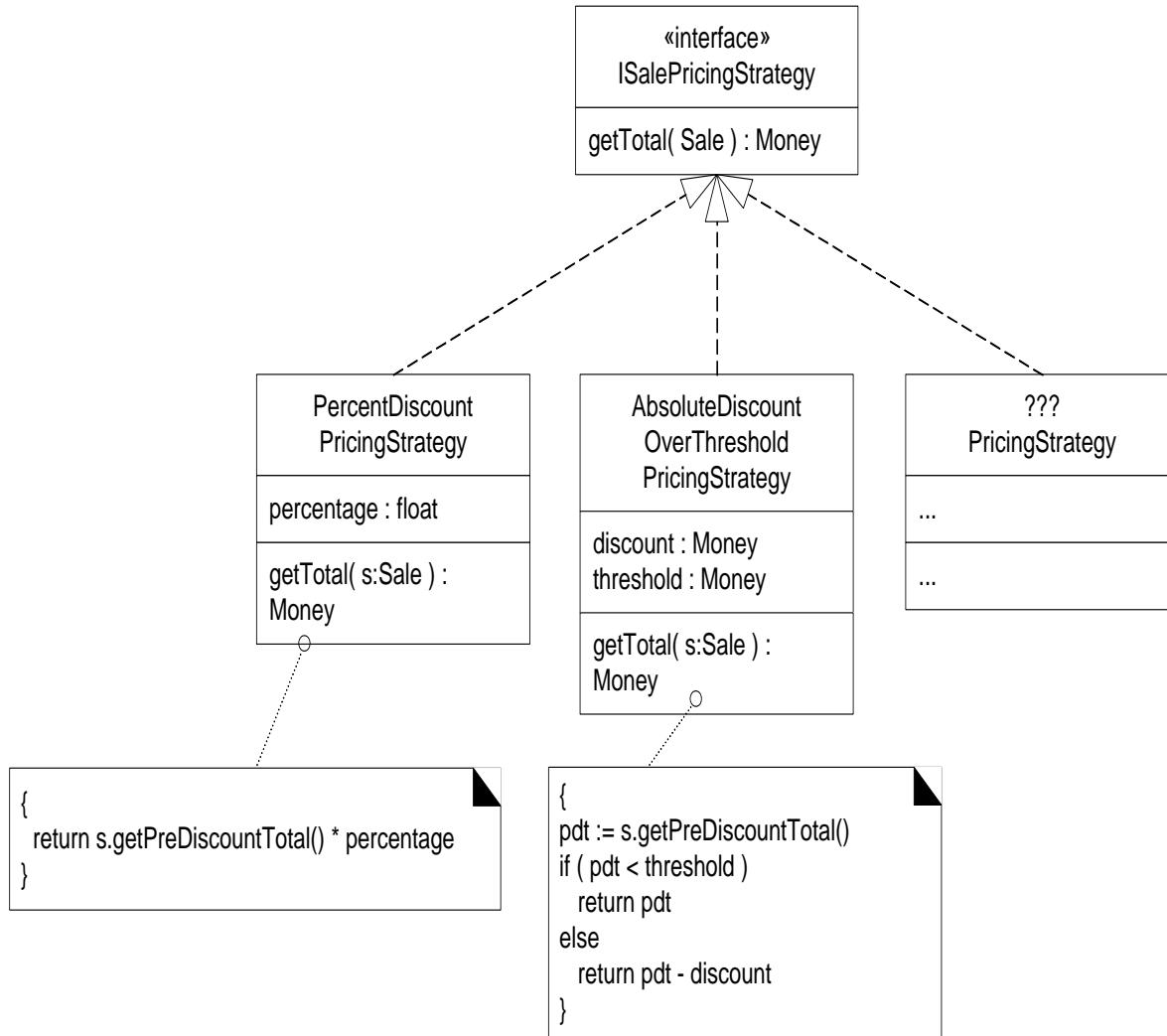


**BITS** Pilani  
Pilani Campus



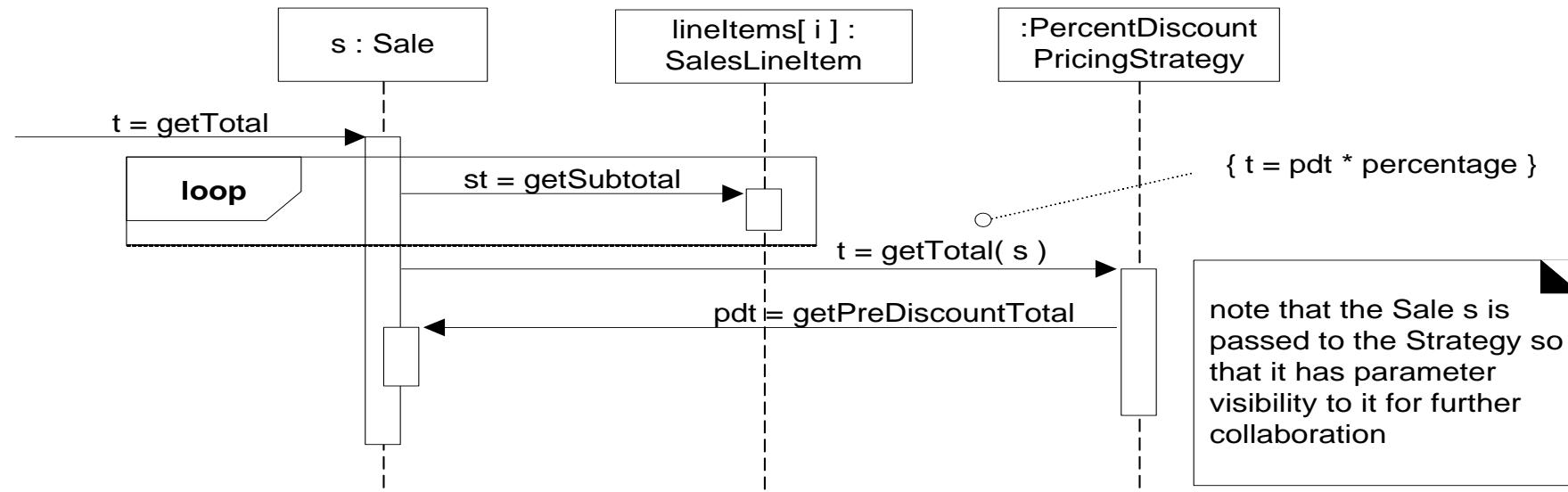
Application of Strategy Pattern to PoS.

# Pricing Strategy Classes



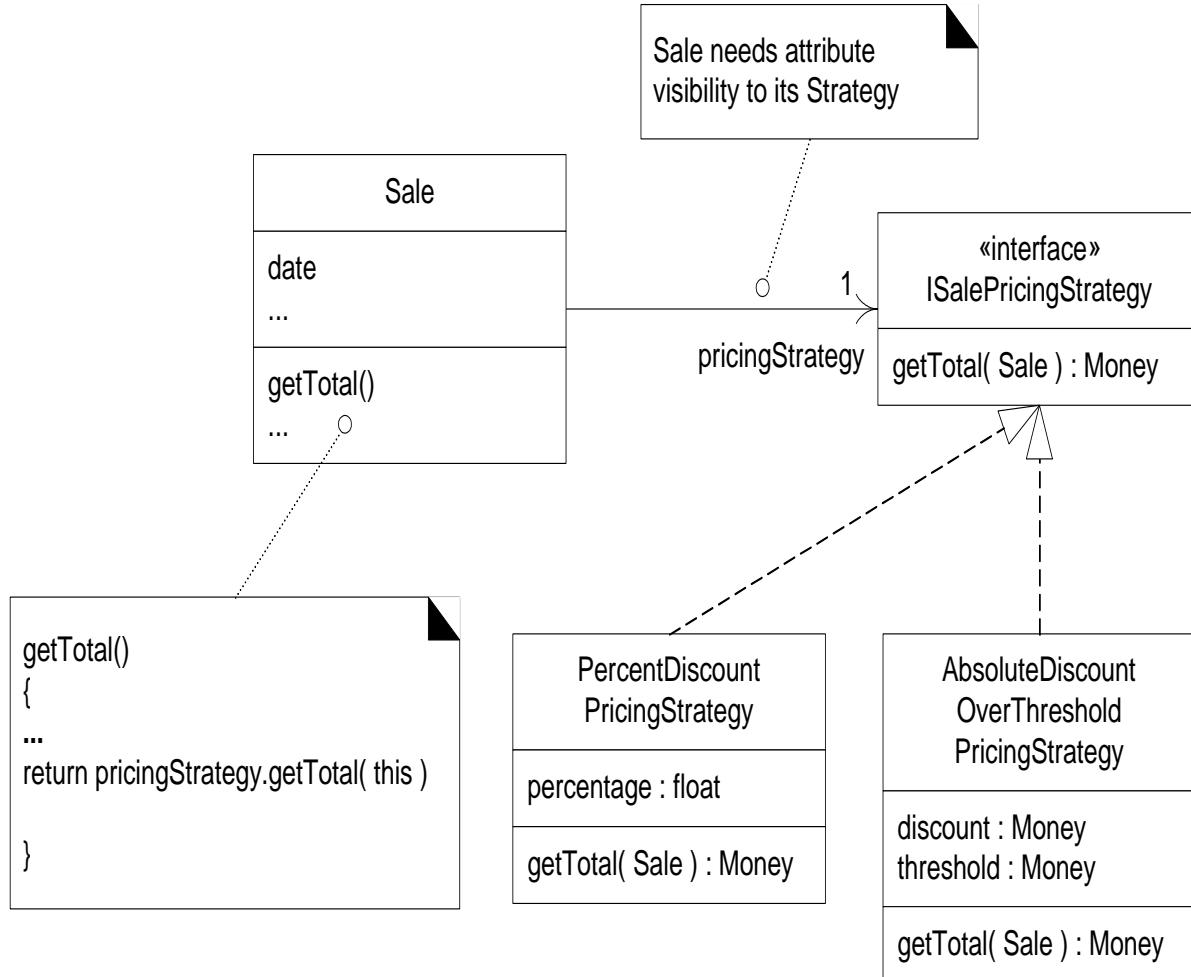
- We create multiple SalePricingStrategy classes.
- Each has a polymorphic getTotal method
- Each getTotal takes the Sale object as a parameter to find the pre-discount price.

# Strategy in collaboration



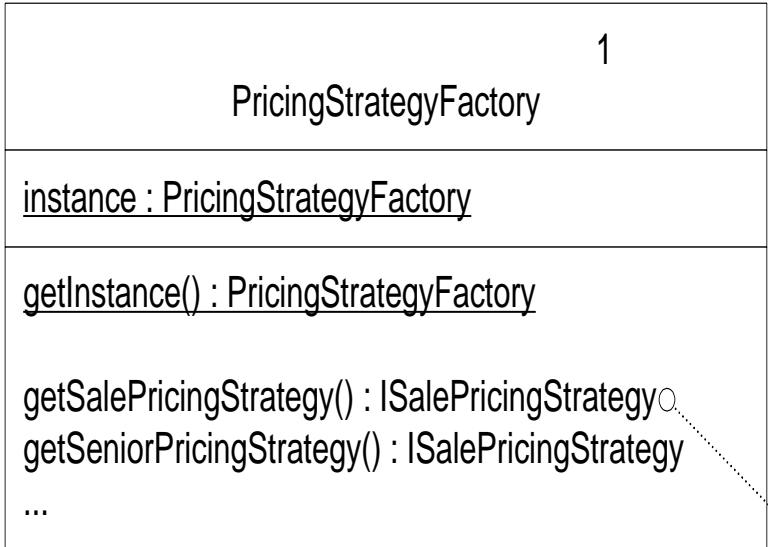
- When a `getTotal` message is sent to `Sale` it delegates some of the work to its strategy object.

# Context object needs attribute visibility to its strategy



- Note that the context object (**Sale**) needs attribute visibility to its strategy.

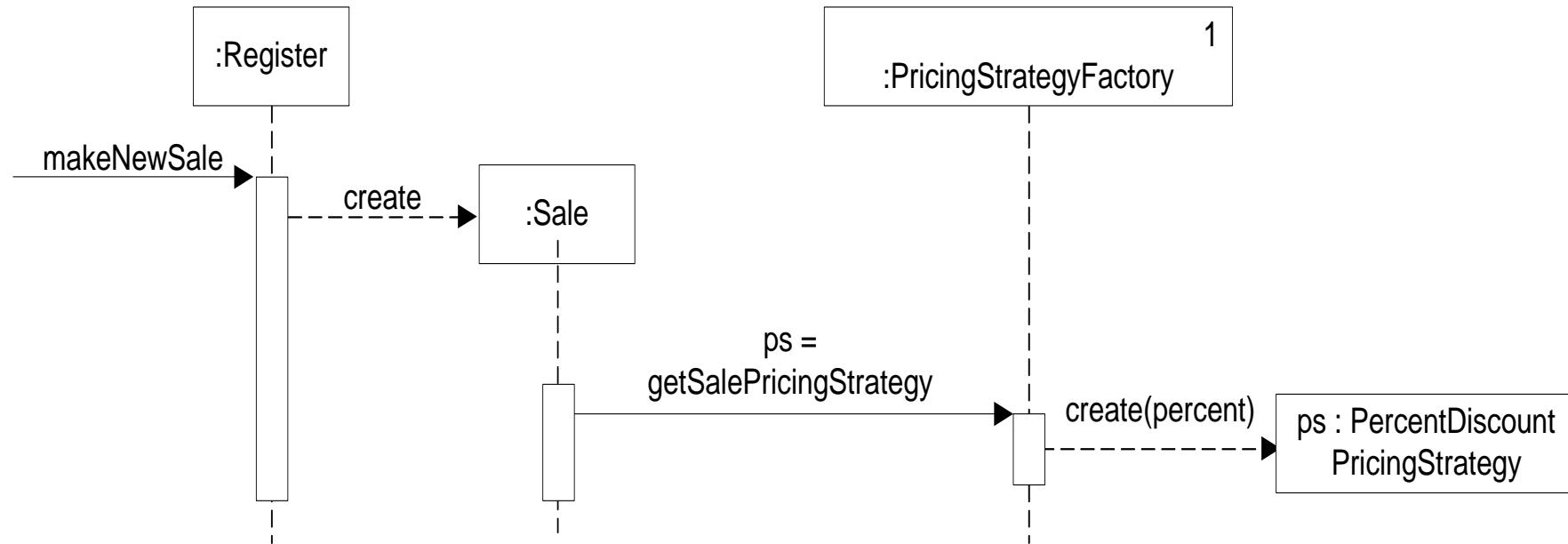
# Factory for Strategies



- As with most factories, the PricingStrategyFactory will be a singleton and access via the Singleton pattern.

```
{
    String className = System.getProperty( "salepricingstrategy.class.name" );
    strategy = (ISalePricingStrategy) Class.forName( className ).newInstance();
    return strategy;
}
```

# Creating a strategy



- When a Sales instance is created, it can ask the factory for its pricing strategy.

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-8 (RL 8.1.1)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus

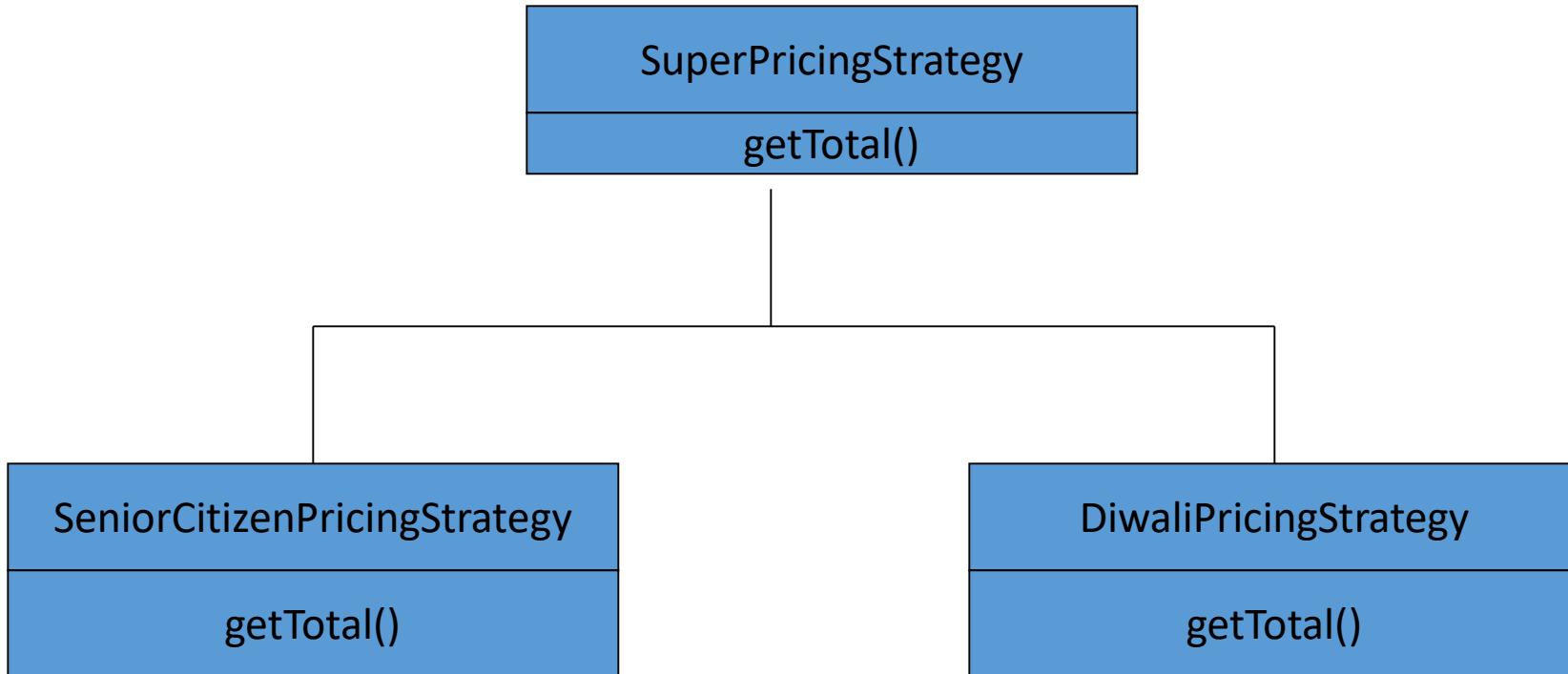


## Composite Pattern – Problem & Solution

# Composite (GoF) pattern.

- Problem: How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?
- Solution: Define classes for composite and atomic objects so that they implement the same interface.
- A Composite is an extension of the Strategy pattern. In a composite a Strategy super class contains a collection of all the strategies. All the Strategies implement the same interface and each one extends from the super class.

# Composite (GoF) pattern.



Super Class as well as derived classes should implement the same interface

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-8 (RL 8.1.2)

Harvinder S Jabbal

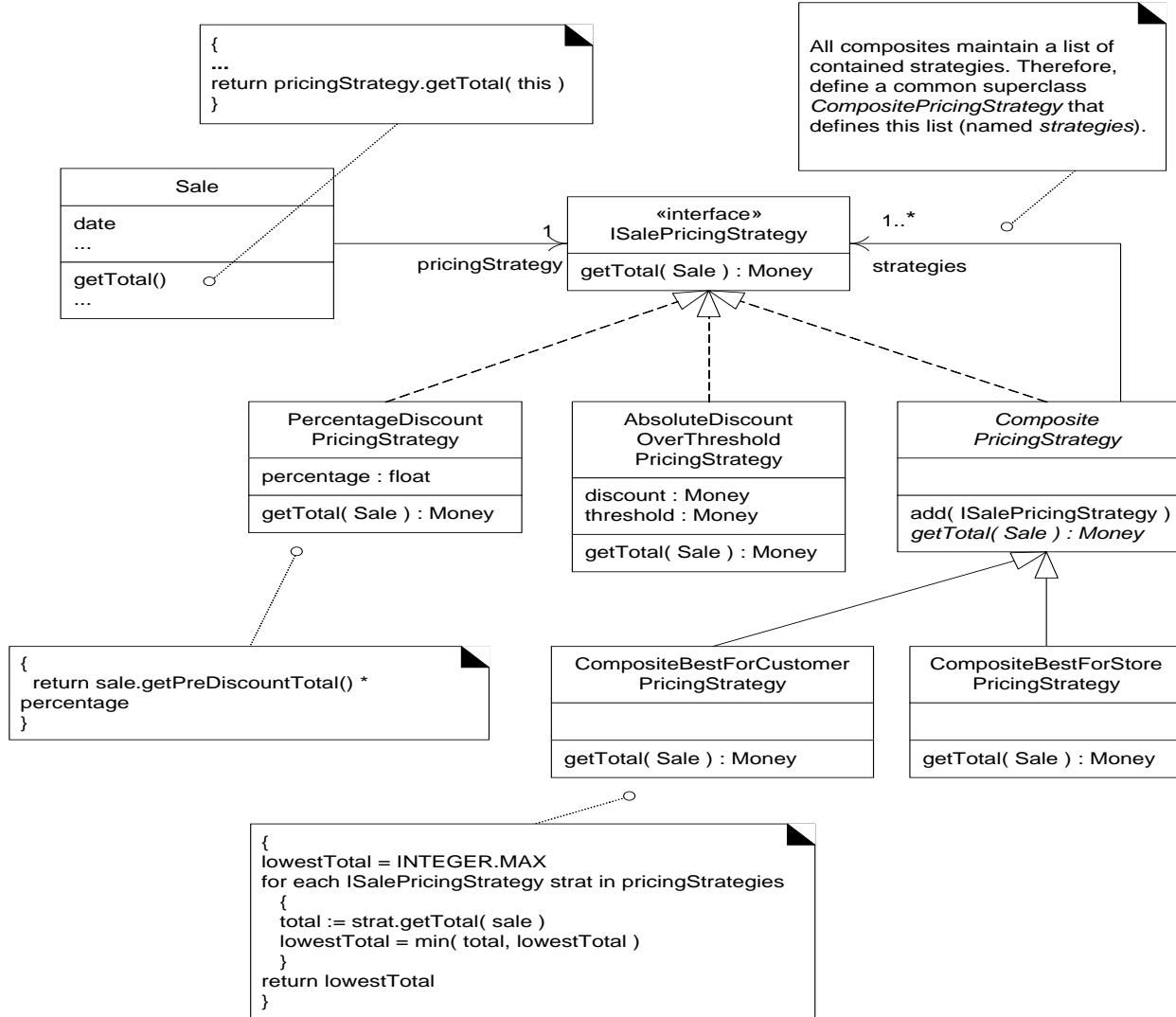


**BITS** Pilani  
Pilani Campus



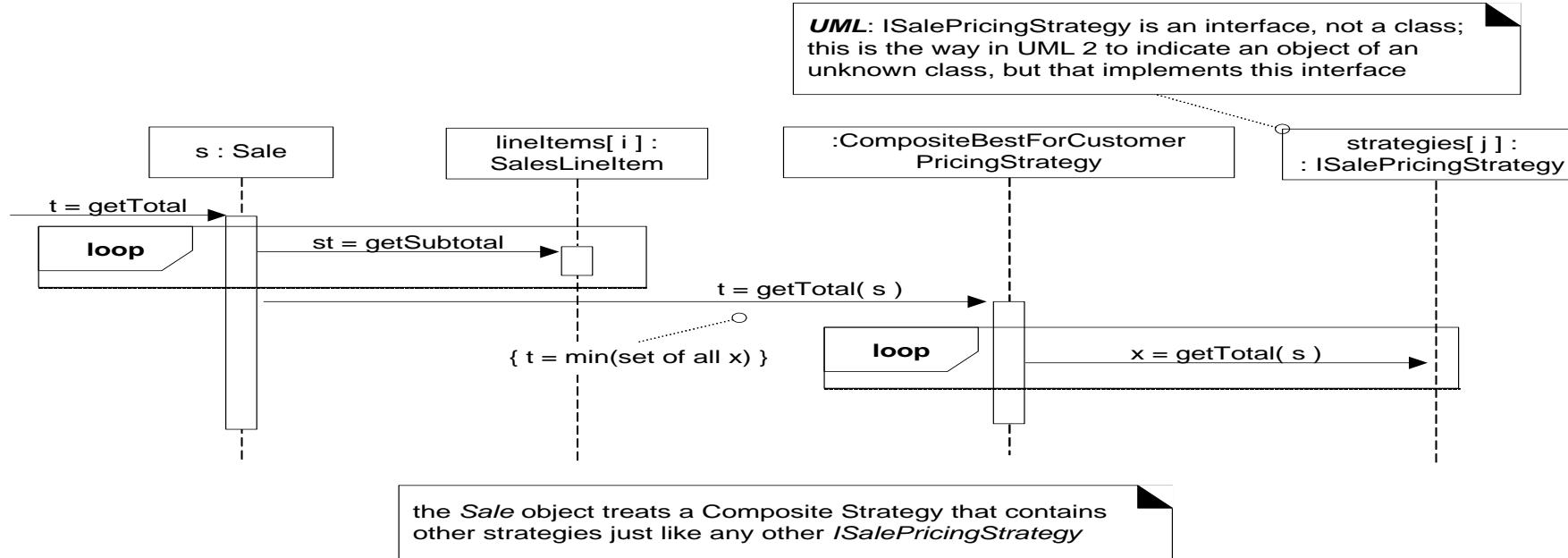
Application of Composite Pattern to PoS.

# The Composite pattern



A new class called **CompositeBestForCustomerPricingStrategy** can implement the **ISalesPricingStrategy** and itself can contain other **ISalesPricingStrategy** Objects.

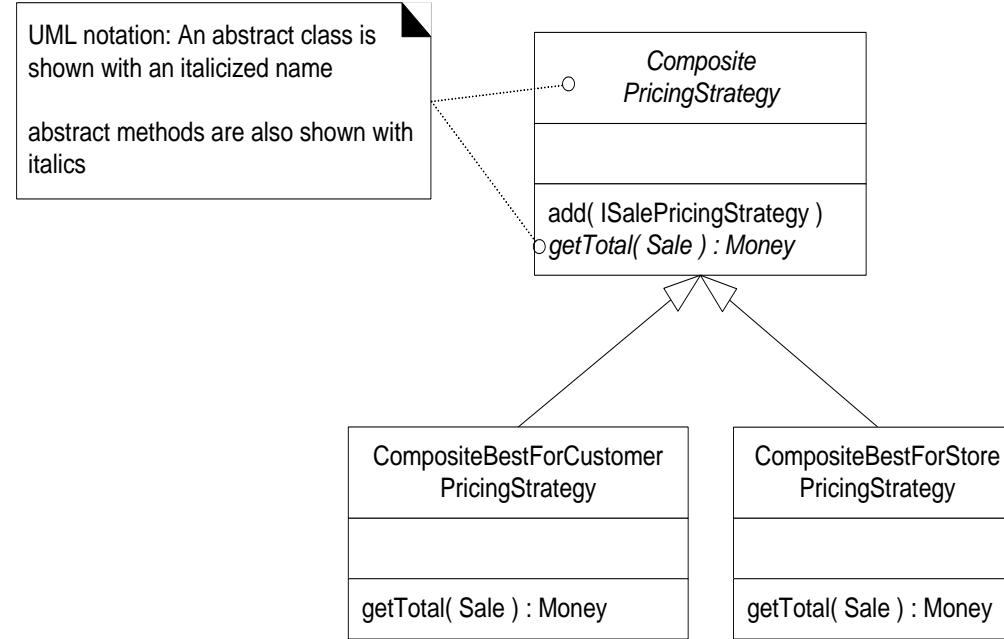
# Collaboration with a Composite



- Collaboration with a composite

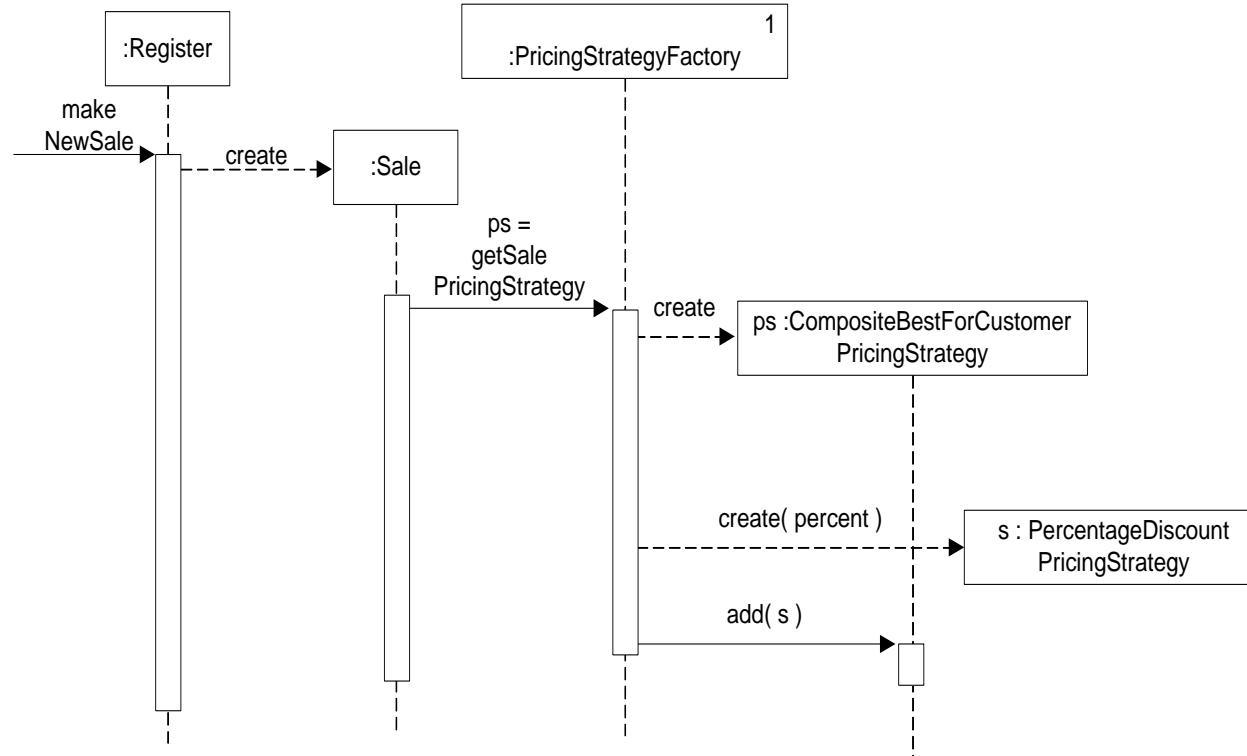
# Using Abstract Class to implement Composite

- Note the existence of add method for adding strategies in this Composite Strategy.
- Note also the abstract method getTotal to assure implementation of ISalesPricingStrategy interface.

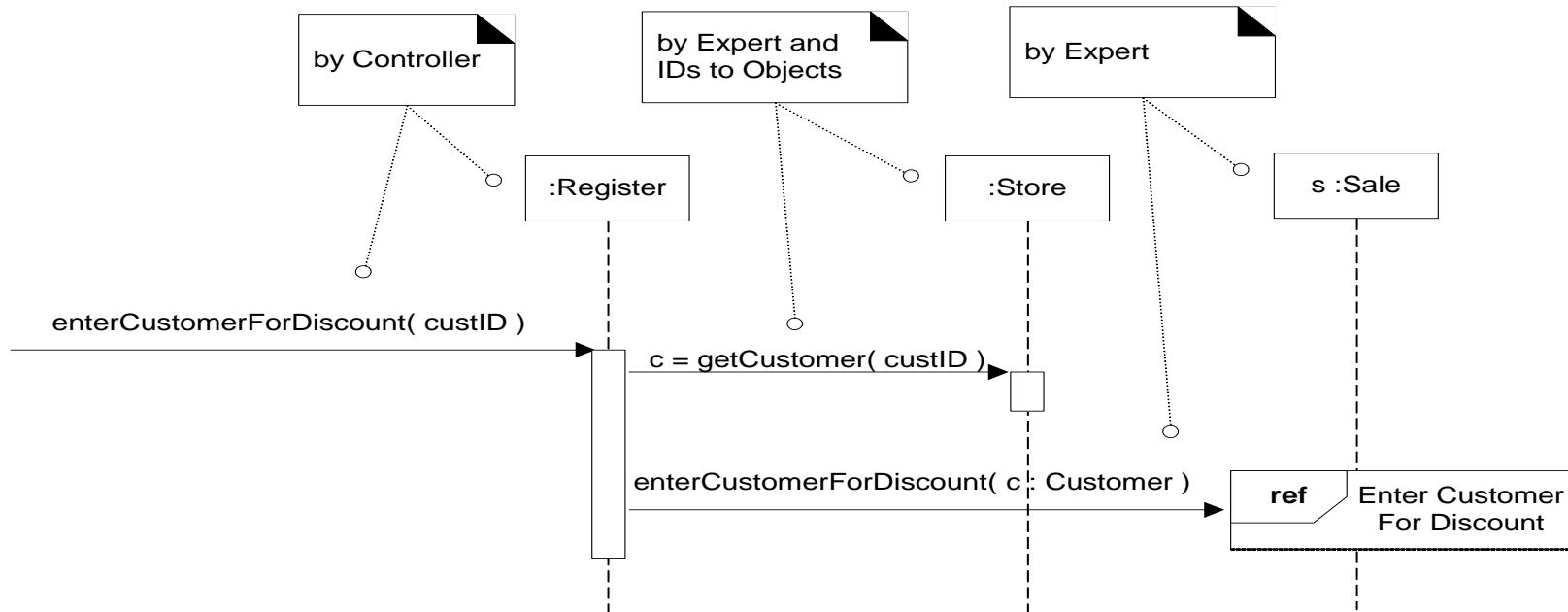


# Creating a composite strategy

- This is the strategy for creating the current store discount when the sale is created.
- May be set initially at ZERO.

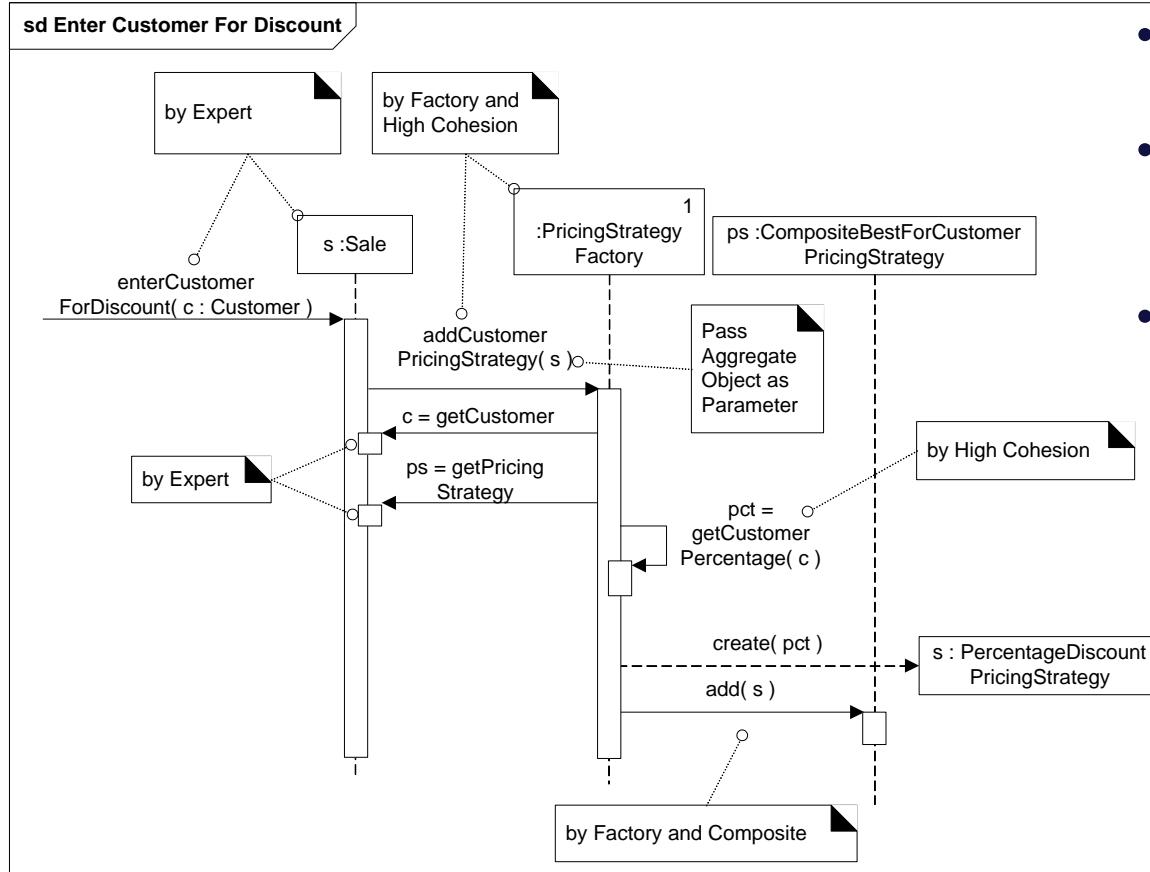


# Creating a pricing strategy for a customer discount Part 1



- Customer type discount, added when the customer type is communicated to the PoS.

# Creating a pricing strategy for a customer discount Part 2



- Elaboration of the previous diagram.
- Note the use of factory object to create additional Pricing Strategy.
- Note that even with conflicting discount strategies it looks like a single strategy to the Sale Object.

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-8 (RL 8.1.3)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus



## Facade Pattern – Problem & Solution

# Facade (GoF) pattern.

- Problem: A common unified interface to a disparate set of implementations or interfaces such as within a subsystem is required. There may be undesirable coupling to many things in the subsystem or the implementation of the subsystem may change.  
What to do?
- Solution: Define a single point of contact to the subsystem a facade object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.
- A Facade is a "front end" object that is the single point of entry for the services of a subsystem. The implementation and other components of the subsystem are private and can't be seen by external components.
- Facades are often accessed via Singleton.
- The Facade hides a subsystem behind an object.

# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-8 (RL 8.1.4)

Harvinder S Jabbal



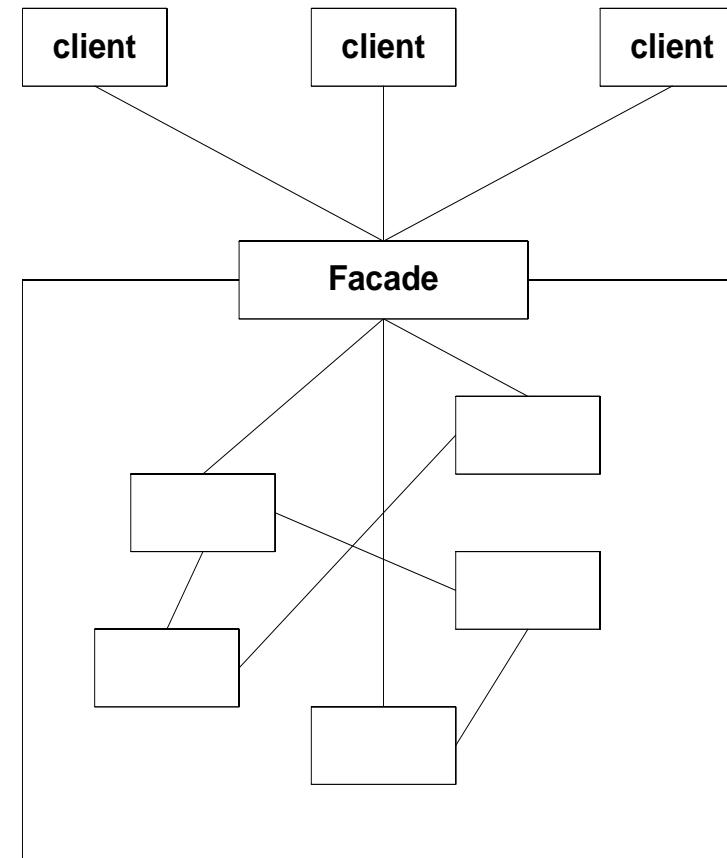
**BITS** Pilani  
Pilani Campus



Application of Facade Pattern to PoS.

# Façade Pattern

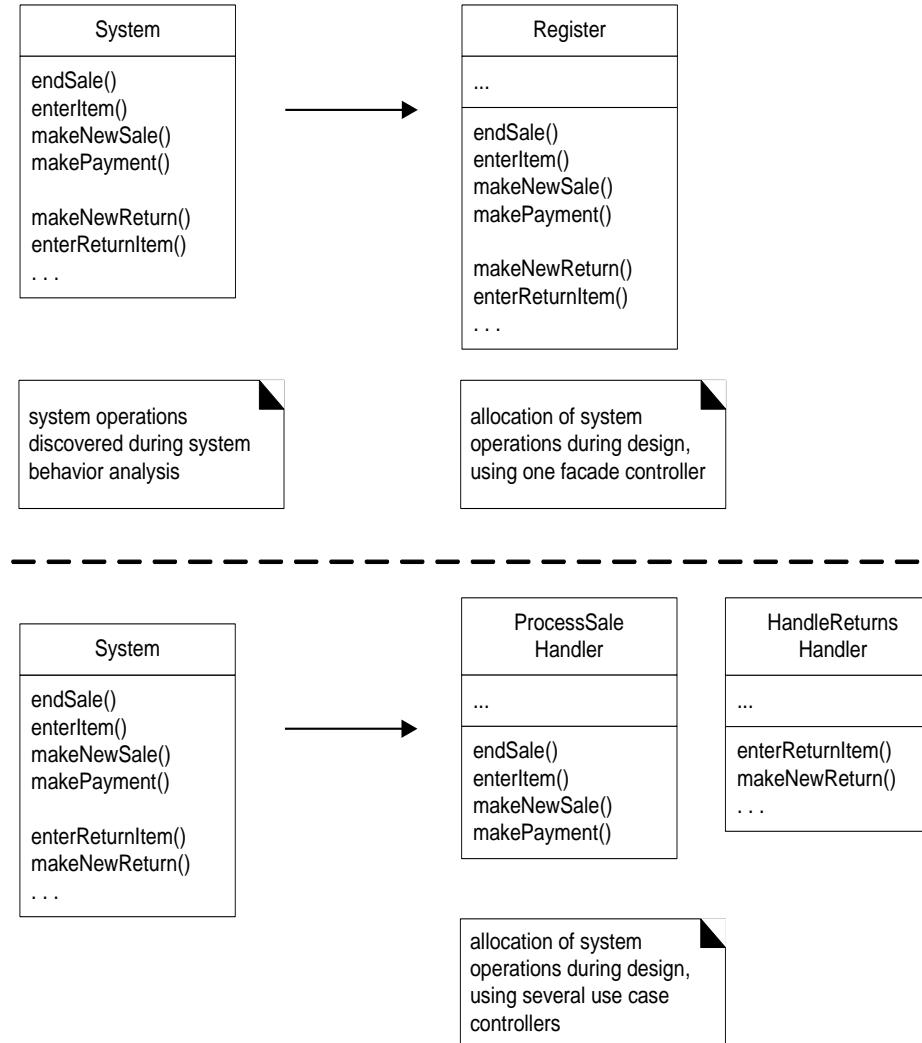
- Provide a simple interface onto a complex interface.
  - Encapsulate generalities
    - ProductDB instead of SQL.
    - EmployeeDB instead of SQL
- Decouple clients from a package.
  - The implementation a façade hides can change without impacting the clients
- Creating layers in the system.
  - a single point of entry allows the system to enforce new rules in a central location
    - ProductDB could check enforce rebate/recall notices.



# Facade

- When disparate sets of objects need a single point of contact...
- Create a “façade” object to represent the entire sub-system.
- Same as the controller pattern.
- E.g. Register Object. Change in implementation of domain objects (Sale, Payment does not affect on client)

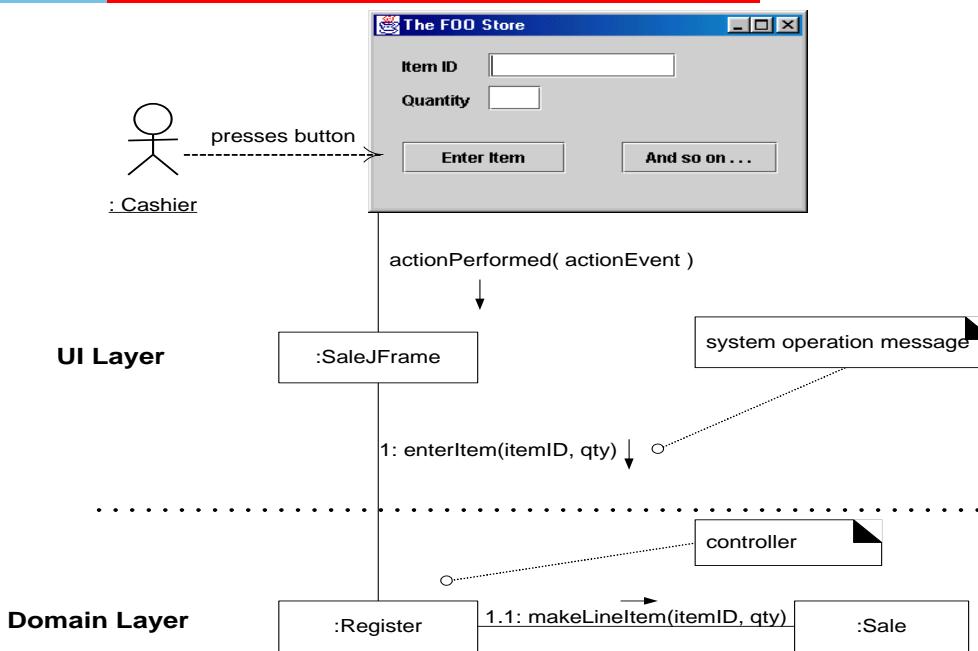
# System Behavioral Analysis



The assignment of responsibility for systems operation may be assigned to one or more classes. See examples on left.

# Assignment by UI Layer

- Controller is basically a delegation pattern.
- It does not itself do work.
- UI Layer should not get involved with business logic.
- Controller pattern common choices of developer with respect to domain object delegate that receive the work request.
- Controller is a façade for the domain layer from UI layer.



# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-8 (RL 8.2.1)

Harvinder S Jabbal



**BITS** Pilani  
Pilani Campus



# Observer/Delegation Event/Publish Subscribe Pattern – Problem & Solution

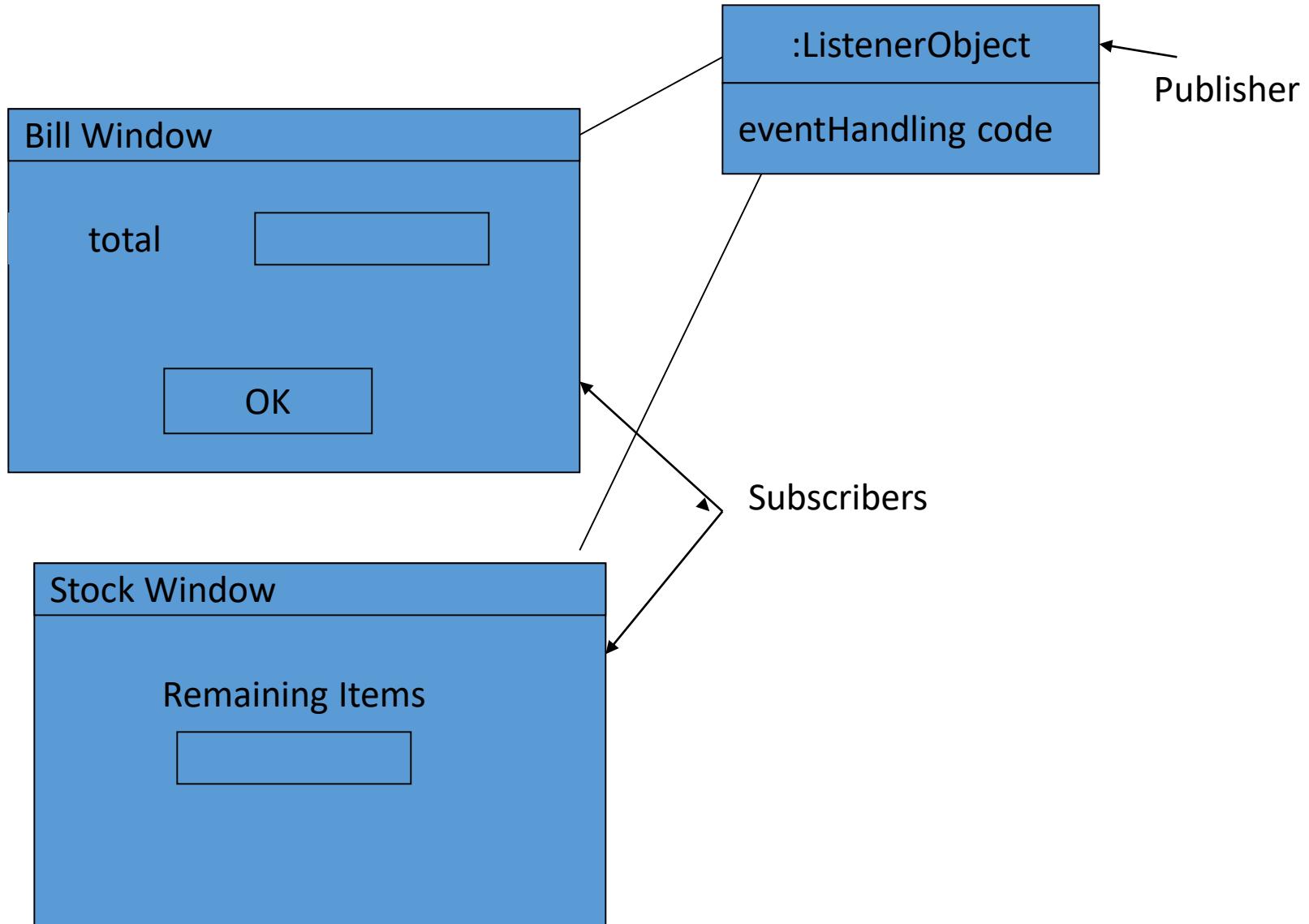
# Observer/Publish-Subscribe/Delegation Event Model (GoF) pattern.

- Problem: Different kinds of subscriber objects are interested in the state changes or events of a publisher object and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?
- Solution: Define a "subscriber" or "listener" interface.
- Subscribers implement this interface.
- The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.
- One publisher can have many subscribers for an event.

# Publish-Subscribe

- When different objects may be interested in the events of changes in an object, which does not want to couple too tightly with them.
- Act as the publisher and allow other objects to subscribe to events, and notify them if there is a subscription.
- Also called: the Delegation Event Model, or the Observer Model.

# Publish-Subscribe



# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.



# Object Oriented Analysis & Design

## Module-8 (RL 8.2.2)

Harvinder S Jabbal

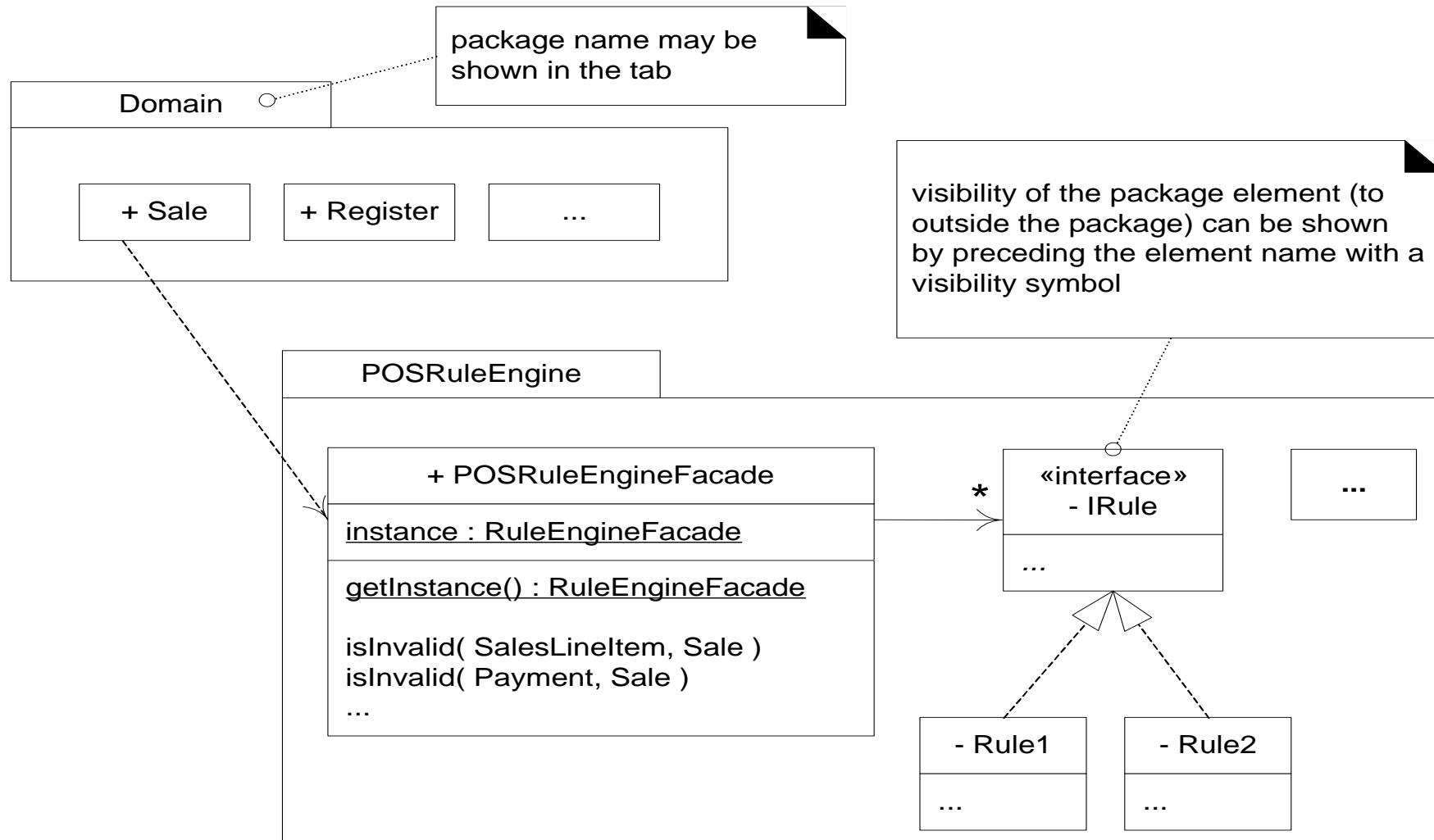


**BITS** Pilani  
Pilani Campus



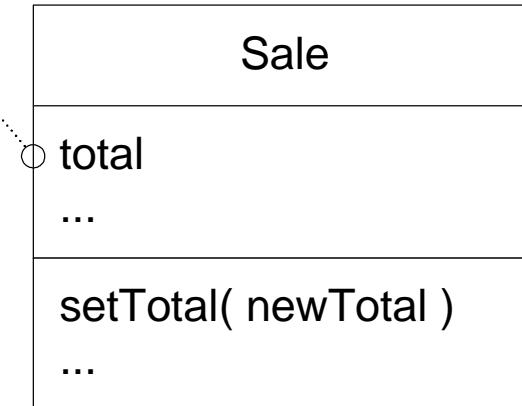
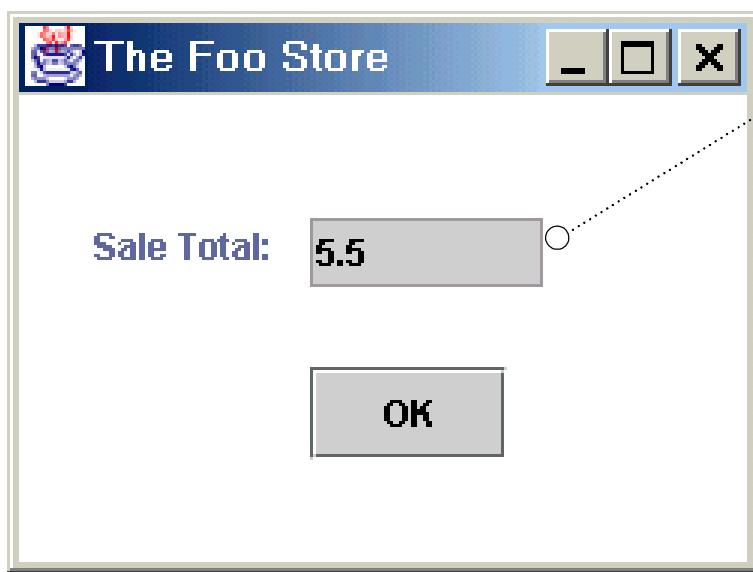
Application of Publish Subscribe Pattern to PoS.

# UML Package Diagram with a Facade

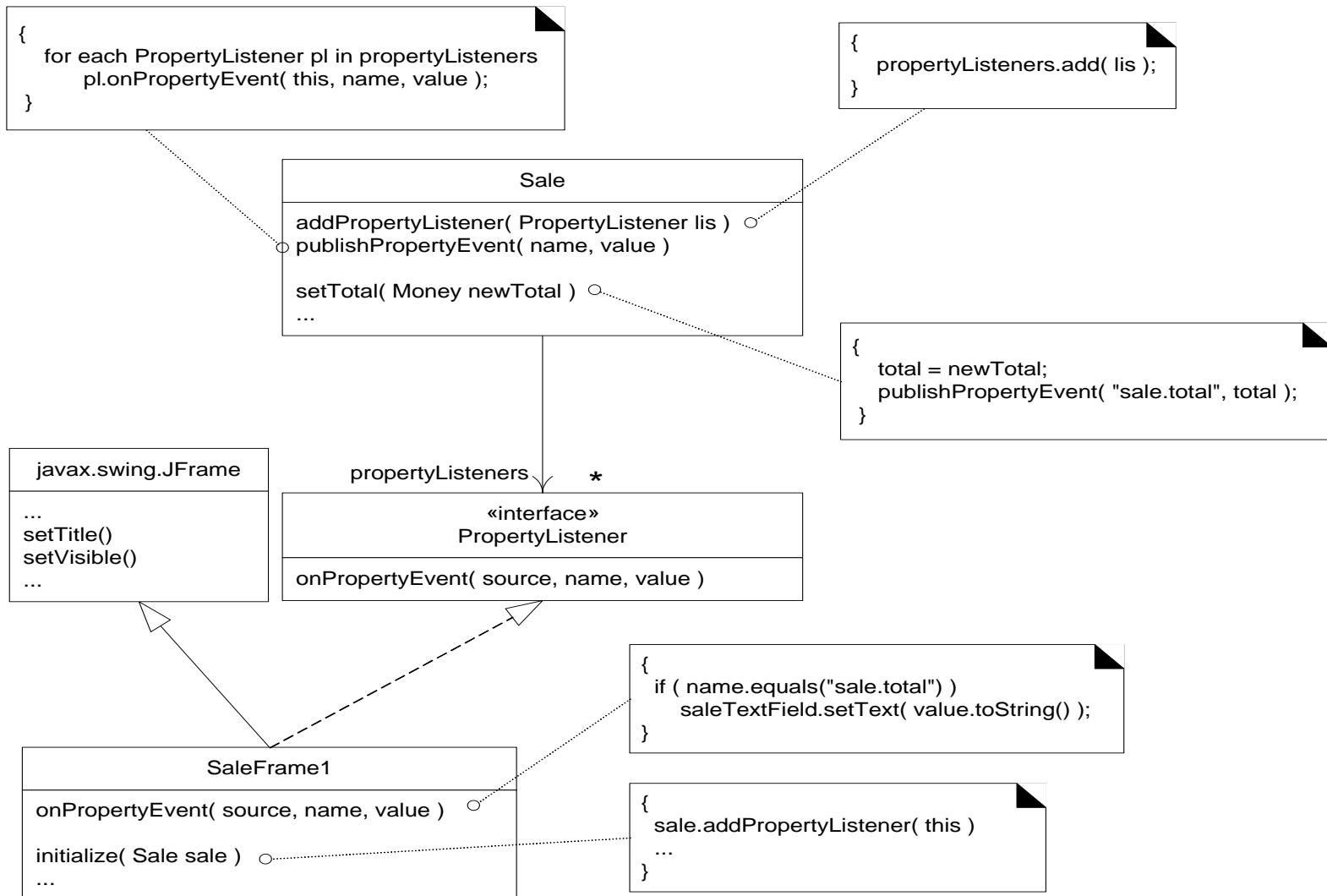


# Update the interface when the sale total changes

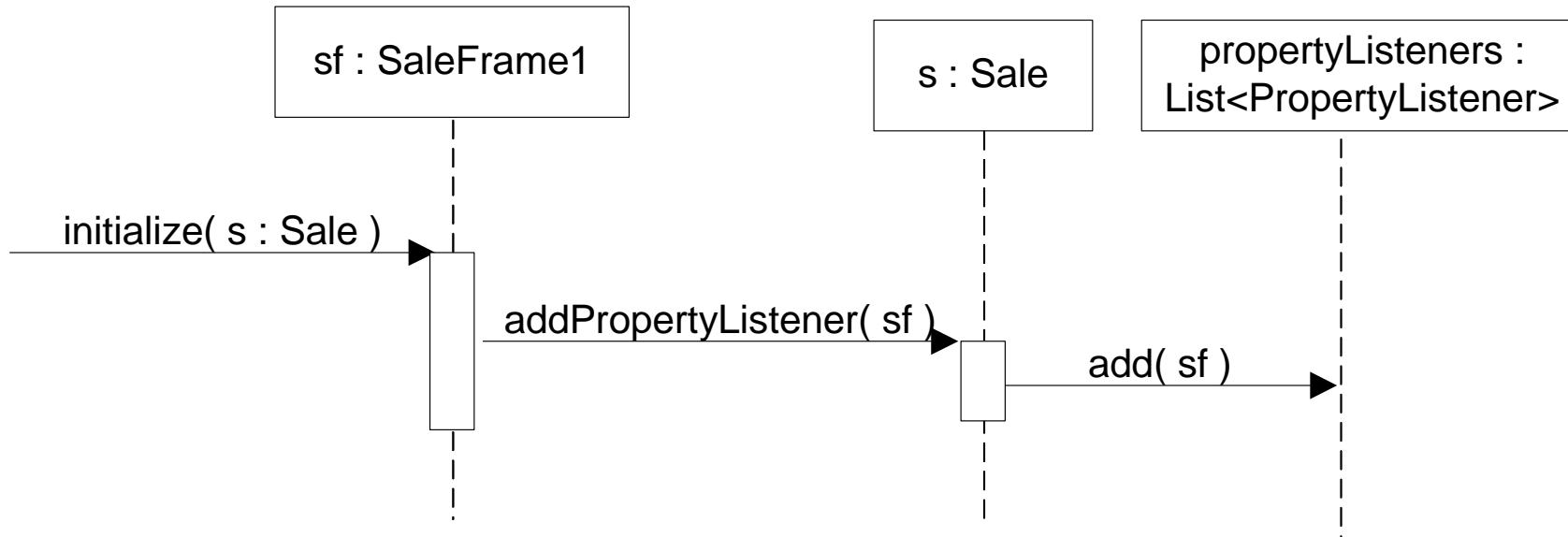
Goal: When the total of the sale changes, refresh the display with the new value



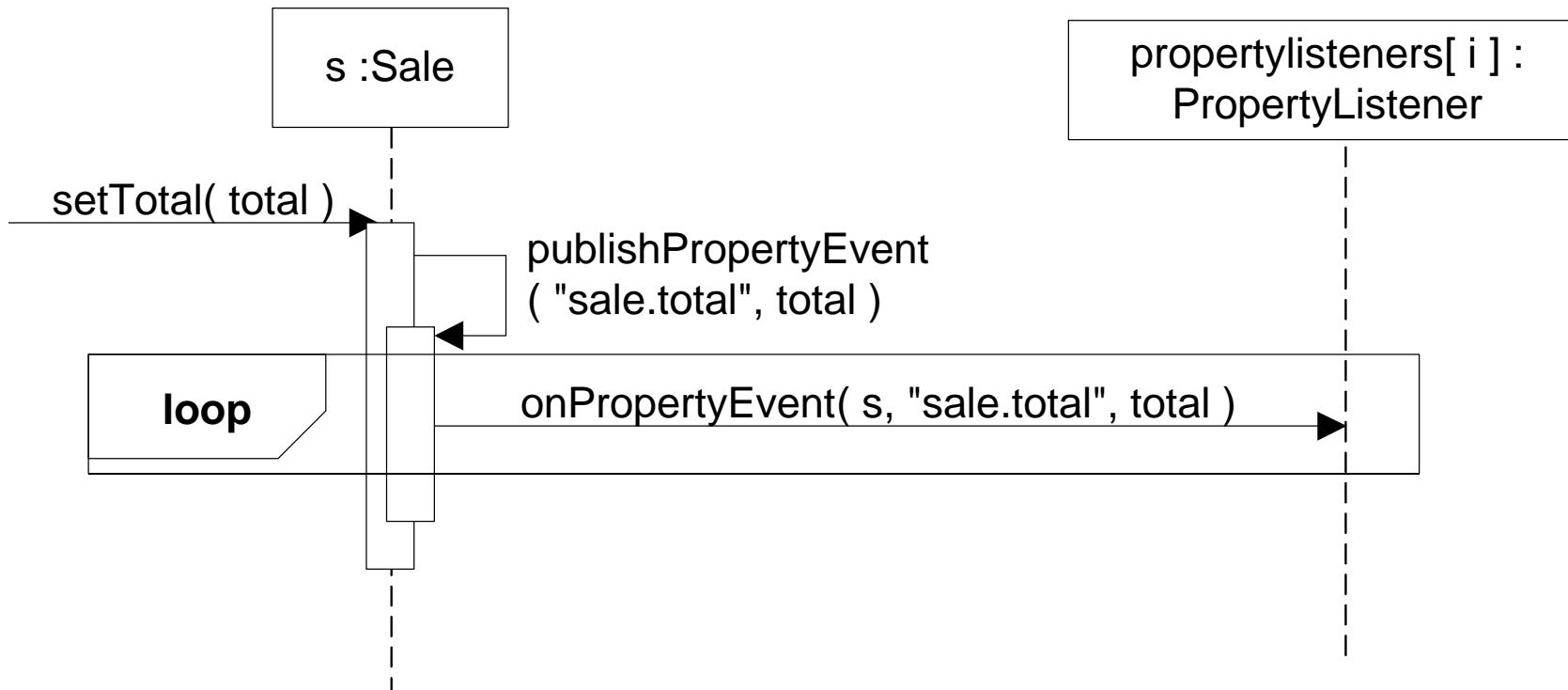
# The Observer Pattern



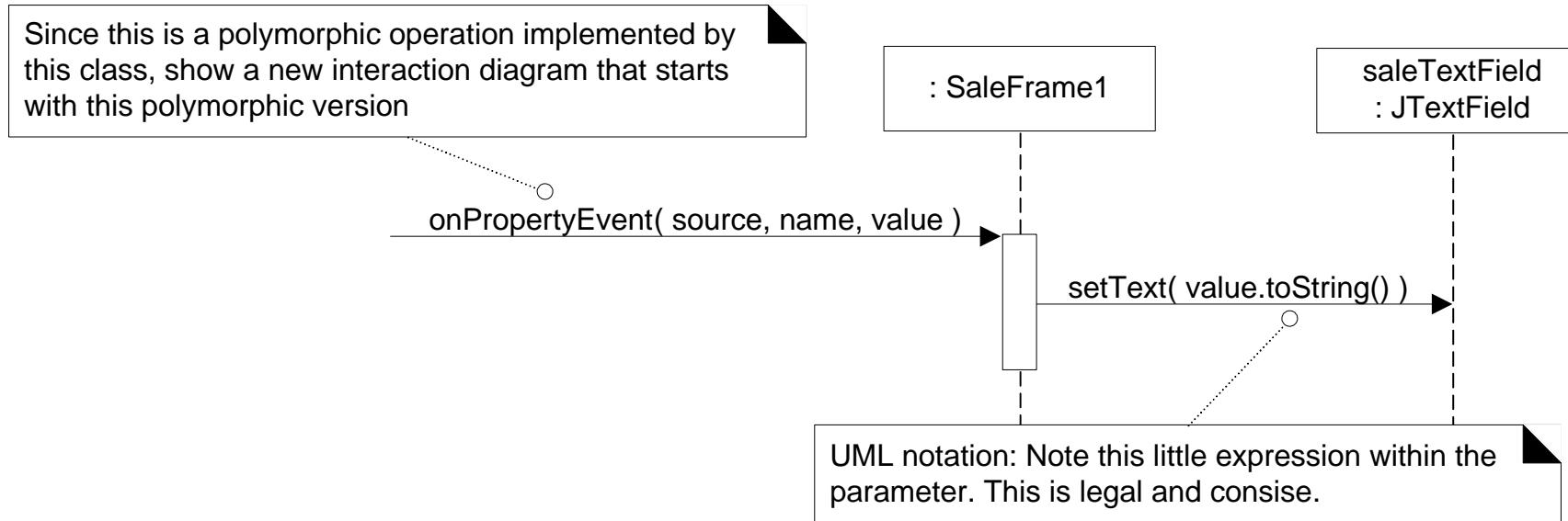
# The Observer SaleFrame1 subscribes to the publisher Sale



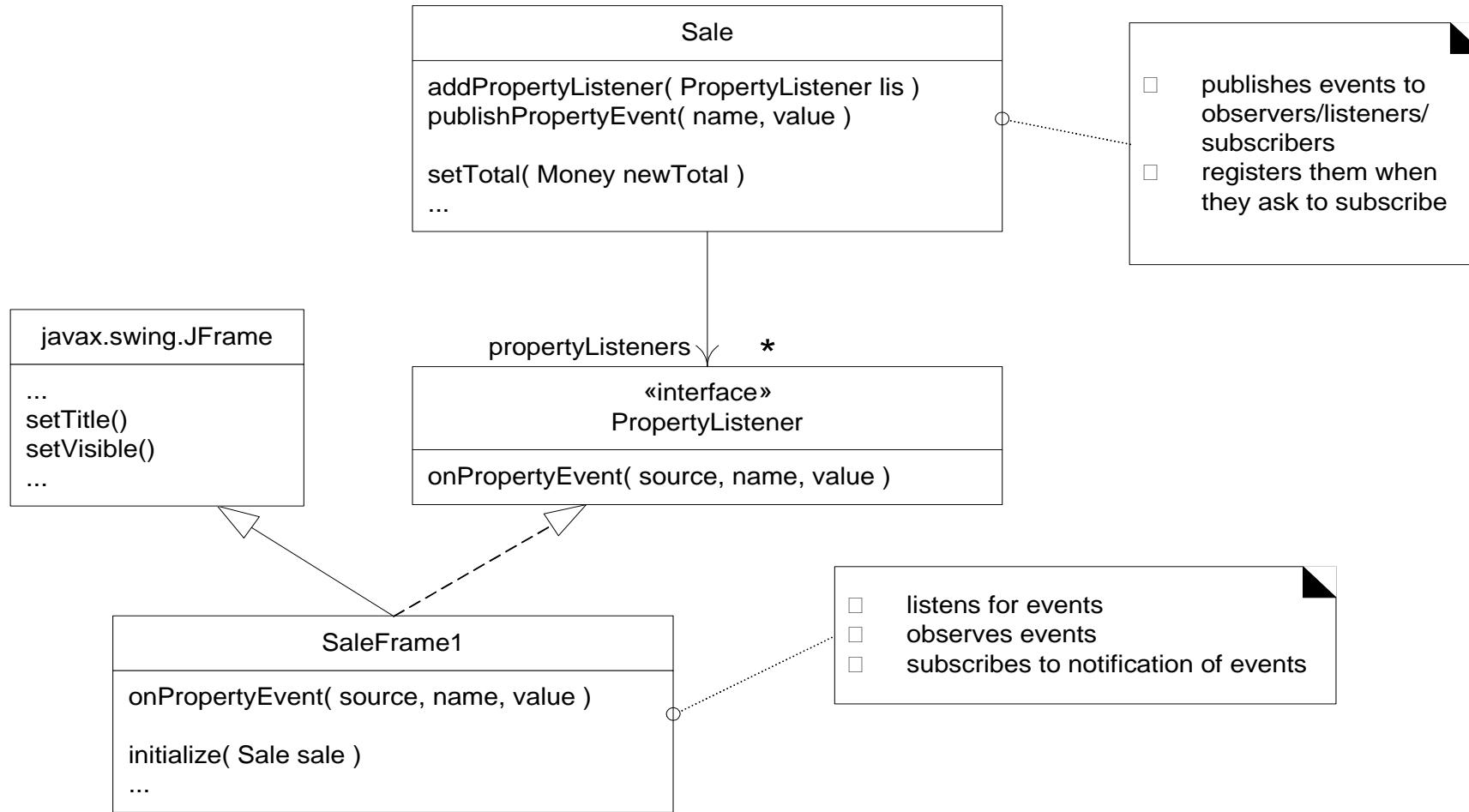
# The Sale publishes a property event to all its subscribers



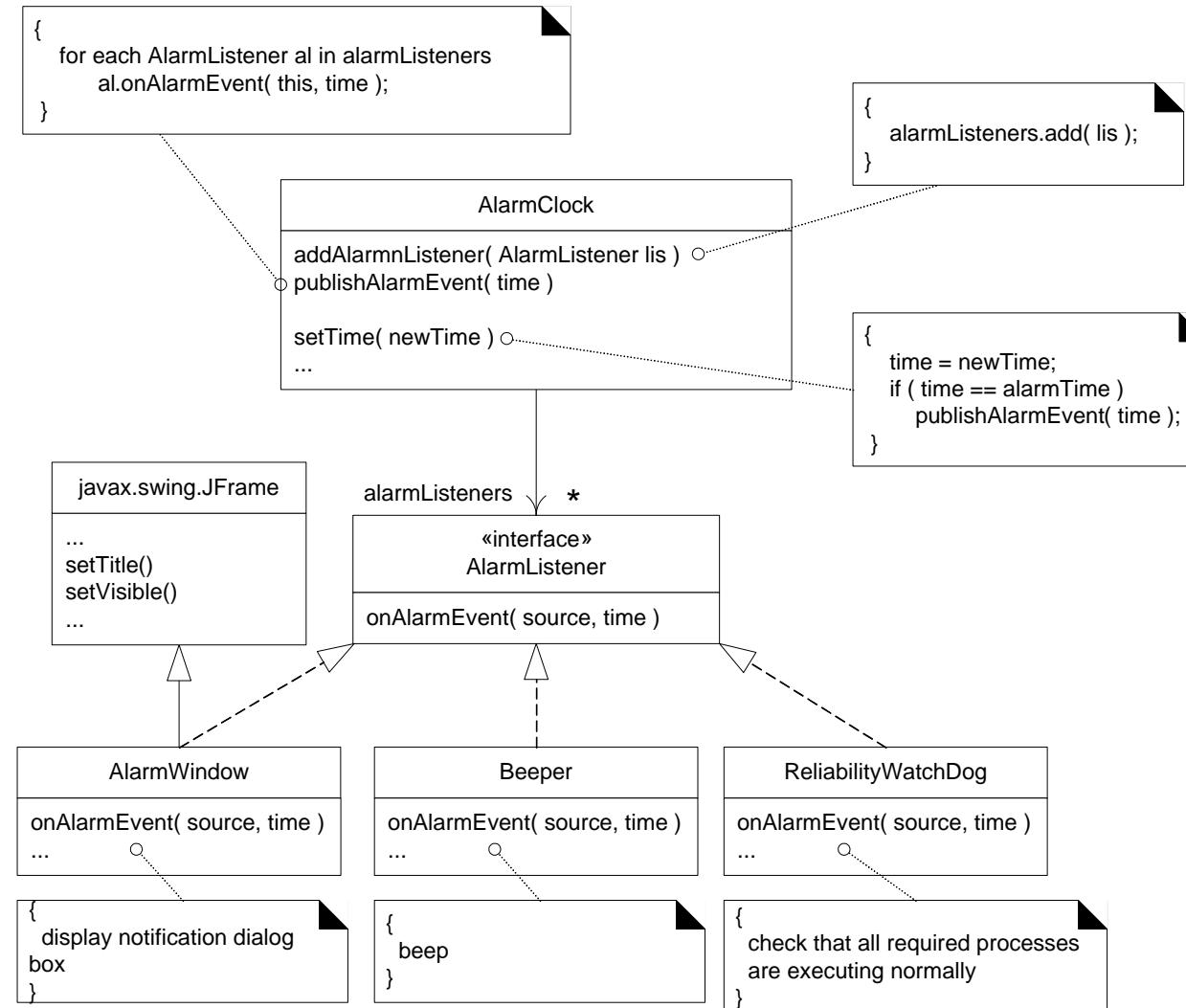
# The subscriber SaleFrame1 receives notification of a published event



# Who is the observer, listener, subscriber and publisher



# Observer applied to alarm events, with different subscribers



# 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) : GoF  
Erich Gama | Richard Helm |  
Ralph Johnson | John Vlissides.