



# IT 314: Software Engineering

*Object-Oriented System Design*

# Recap (Analysis Modeling)

---

1. Analyze the problem statement
  - Identify functional requirements
  - Identify nonfunctional requirements
  - Identify constraints (pseudo requirements)
2. Build the functional model:
  - Develop use cases to illustrate functional requirements
3. Use case realizations by building **dynamic models**:
  - Develop **sequence diagrams** to illustrate the interaction between **domain objects**
  - Develop **state diagrams** for **domain objects** with interesting behavior
4. Build the domain object model:
  - Develop **class diagrams** (at domain level) showing the structure of the system



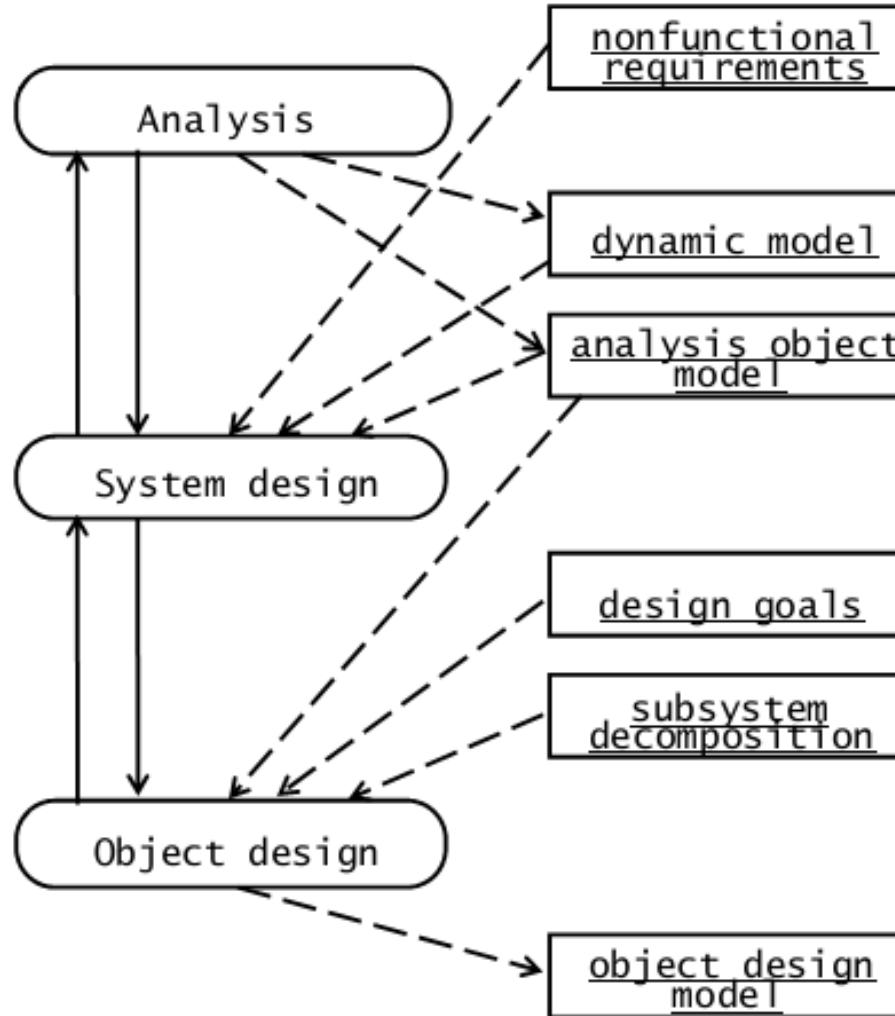
---

# **Designing the System**

---

# The activities of System Design

---



# Analysis Model

---

- Analysis model = **first cut** at design model
- In theory: refine analysis model into design model in a structure-preserving way
  - But may not be possible due to **design/implementation** constraints (Platforms, NFRs, Reuse, etc.)
  - Design is **not algorithmic** activity
- What is in design but not analysis:
  - Decisions on **performance** and **distribution requirements**; optimizations etc.
  - Decisions on **choosing architecture**
  - Decisions on **data structures** and **persistent (database)** storage
  - Decision on **Reuse**
  - Analysis should prepare for design by giving a **thorough understanding of the requirements**

# Analysis vs. Design model

---

Conceptual model (avoids implementation issues)	Physical model (blueprint of implementation)
Applicable to several designs	Specific for one implementation
Three conceptual stereotypes (entity, boundary, and control)	Any number of physical stereotypes depending on programming language
Less formal	More formal
Less expensive to develop	More expensive to develop
Few layers	Many layers
May not be maintained	Should be maintained

Jacobson/Booch/Rumbaugh

---

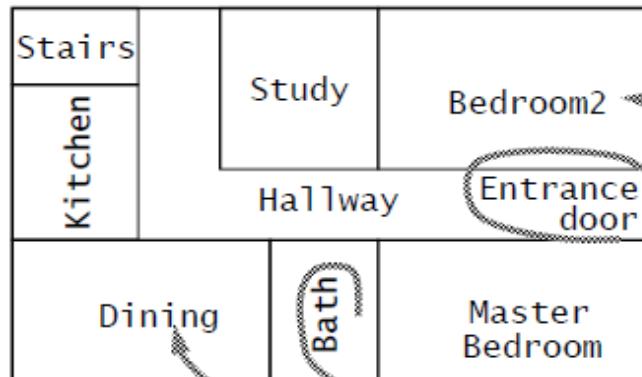


# System Design

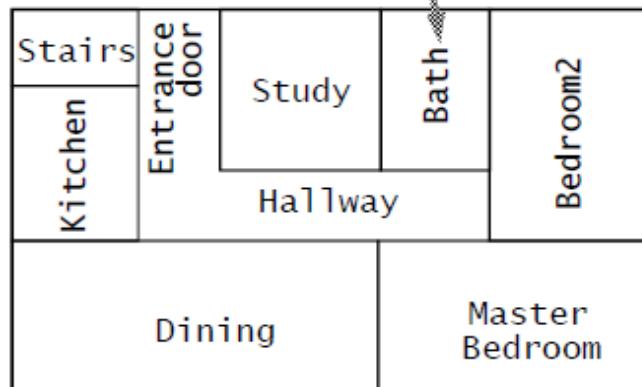
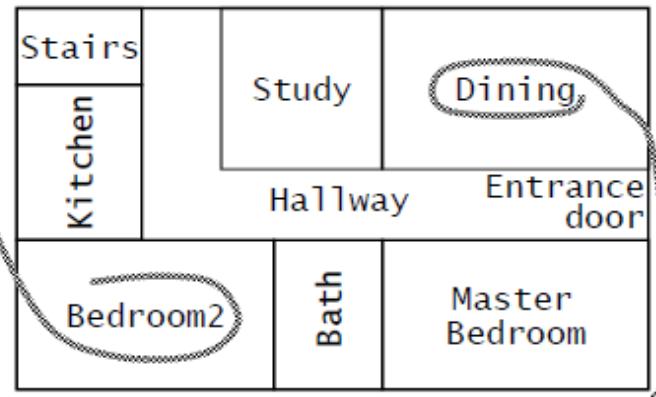
---

- Identify **design** goals
- Design the **initial subsystem decomposition**
  - Analysis model and use cases
  - Use standard **Architectural Styles**
- Refine the subsystem decomposition

**Version 2**



**Version 1**



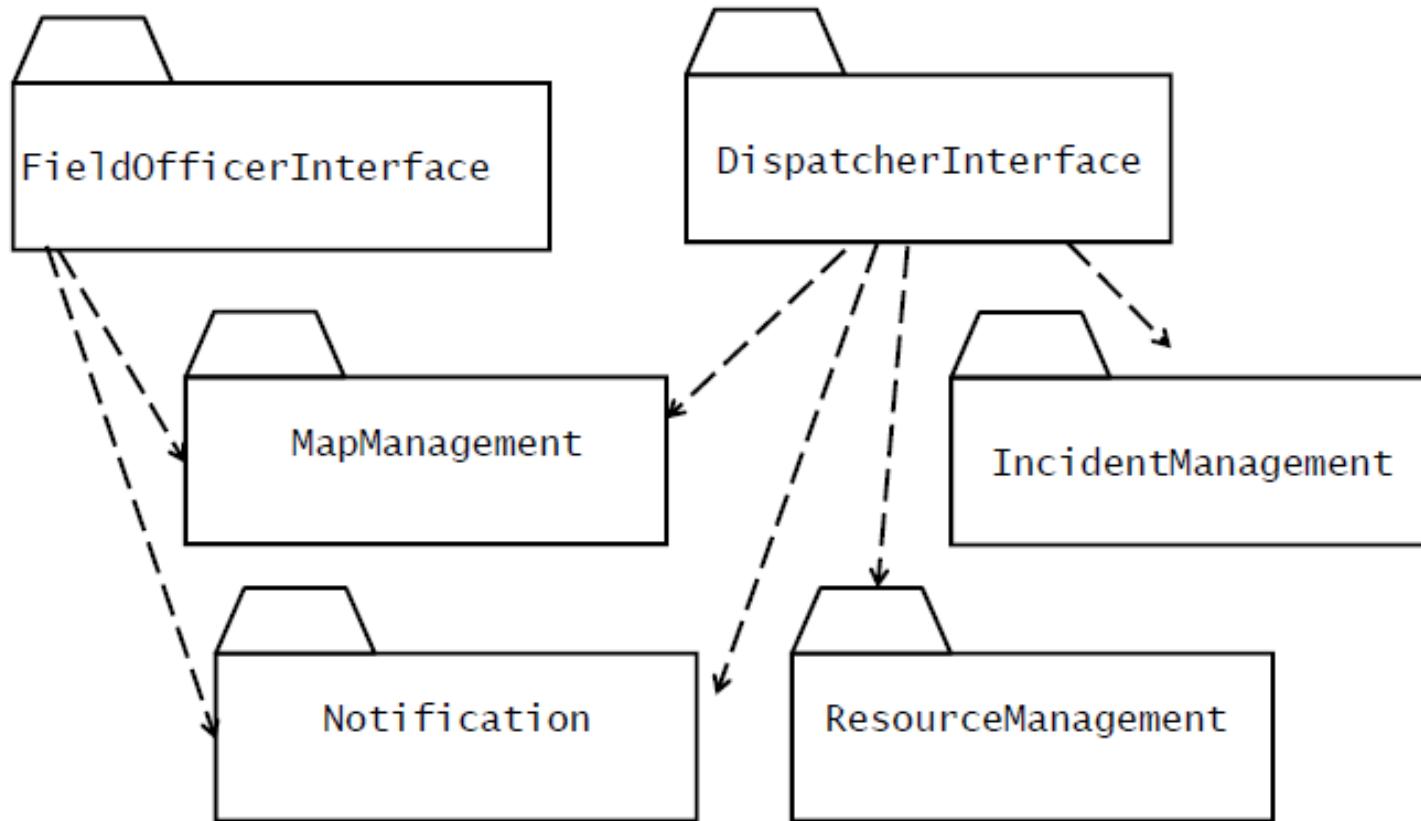
**Version 3**

Example of iterative floor plan design.  
Three successive versions show how we minimize walking distance and take advantage of sunlight.

N

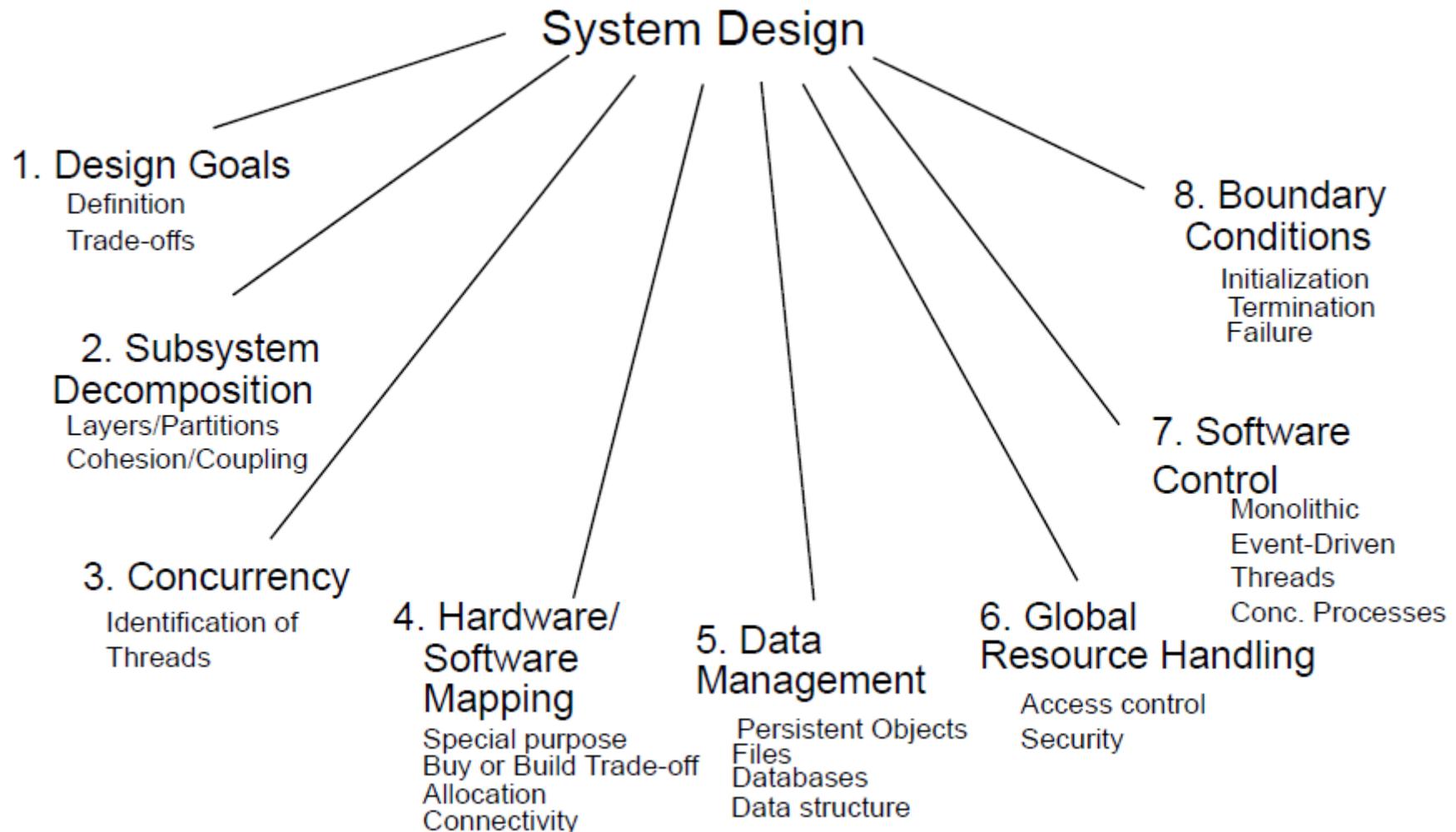
# Example: FRIEND

---



# Subsystem Design

---



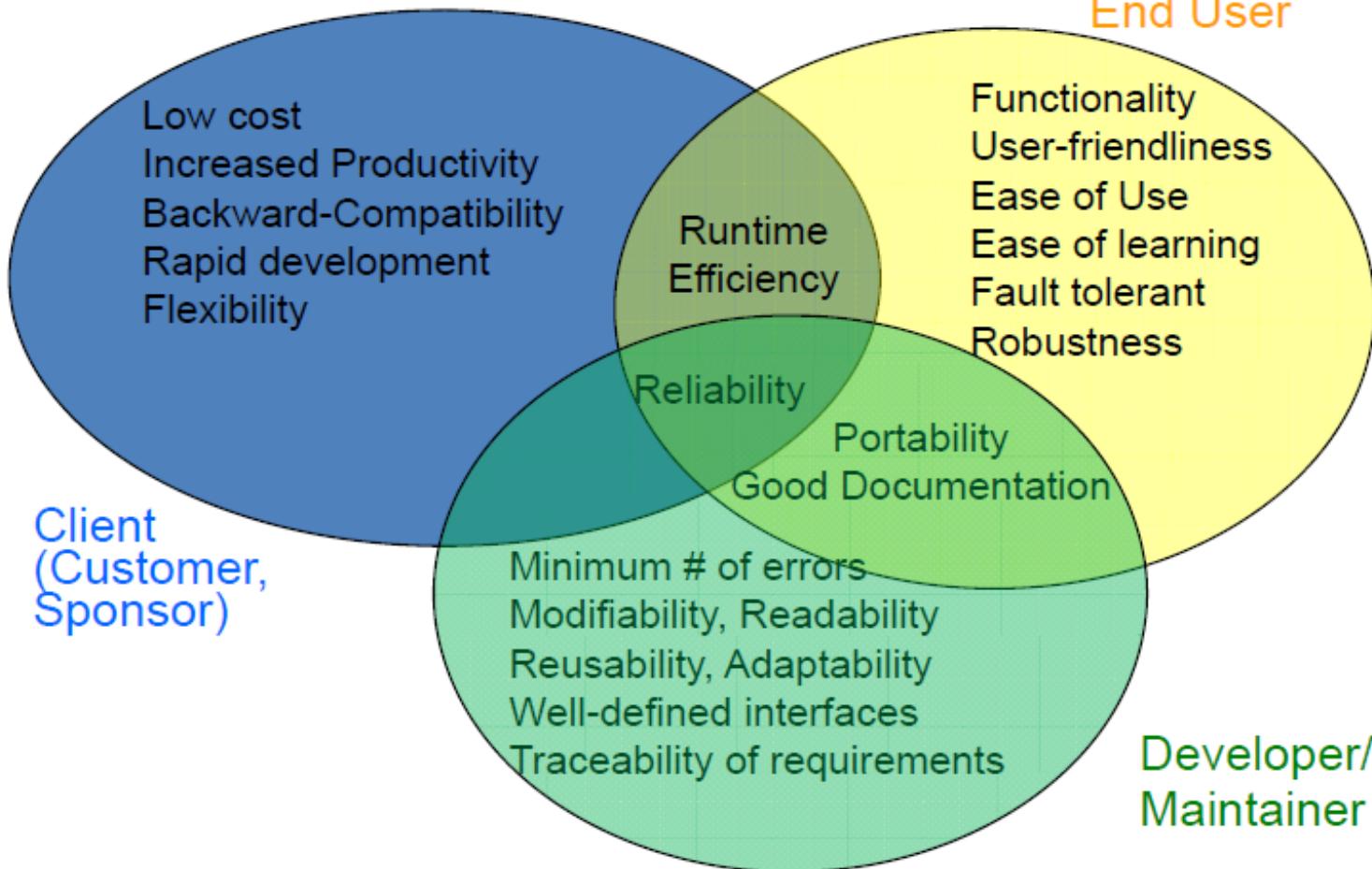
# List of Design Goals

---

- Reliability
- Modifiability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- Backward-compatibility
- Cost-effectiveness
- Robustness
- High-performance
- Good documentation
- Well-defined interfaces
- User-friendliness
- Reuse of components
- Rapid development
- Minimum # of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility

# Relationship between Design Goals

---





# Typical design Trade-Offs

---

- Functionality vs. Usability
  - Cost vs. Robustness
  - Efficiency vs. Portability (or Modifiability)
  - Rapid development vs. Functionality
  - Cost vs. Reusability
  - Backward Compatibility vs. Readability
-



# Design the System

---

## Subsystem decomposition

- Subsystems
- Subsystem Structure
- Subsystem Interfaces

# Subsystem Decomposition

---

## Subsystem (UML: Package)

- Collection of classes, associations, operations, events and constraints that are interrelated
- Seed for subsystems: Objects and Classes.

## (Subsystem) Services:

- Group of operations provided by the subsystem
- Seed for services: Subsystem use cases

## (Services are specified by) Subsystem interface:

- Specifies interaction and information flow from/to subsystem boundaries, but not inside the subsystem.
- Should be clear, well-defined and small.
- Often called API: Application programmer's interface (but this term should be used during implementation, not during Subsystem Design)

# Choosing Subsystem

---

Criteria for subsystem selection:

Most of the interaction should be within subsystems, rather than across subsystem boundaries

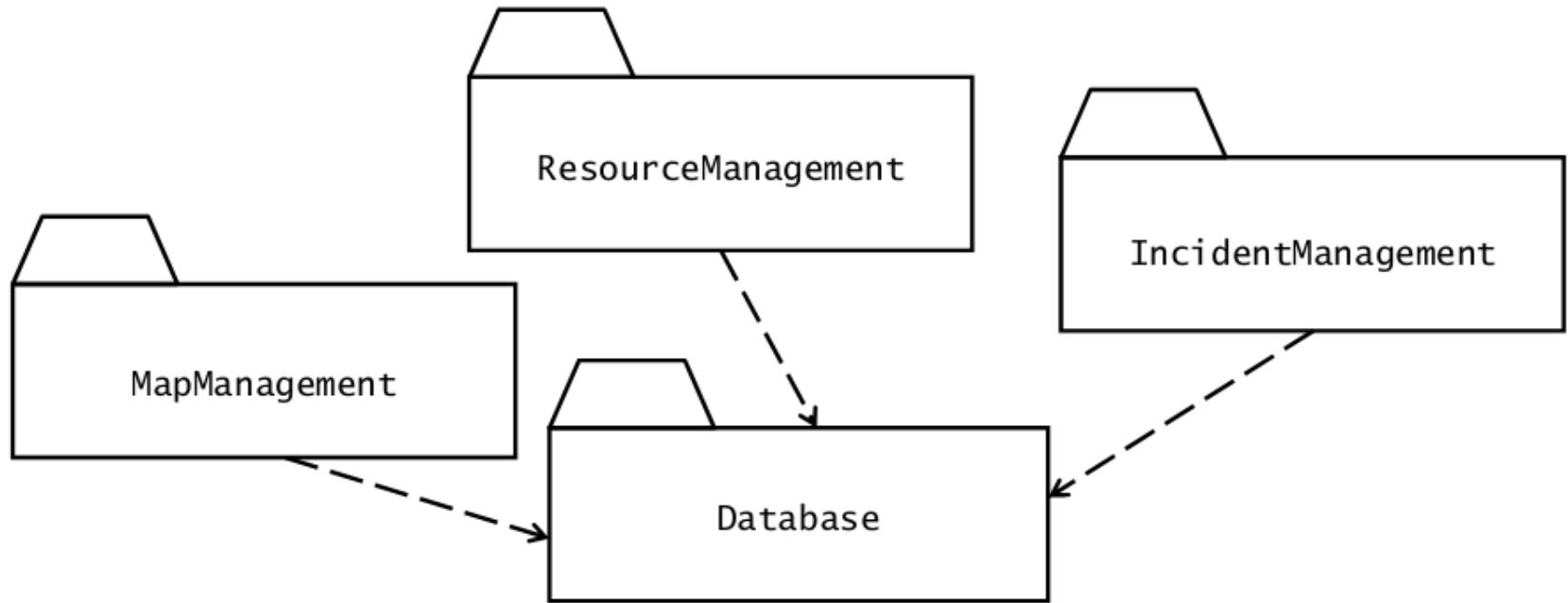
(High cohesion and Low Coupling).

- Does one subsystem always call the other for the service?
- Which of the subsystems call each other for the service?

# Example design alternatives: FRIEND

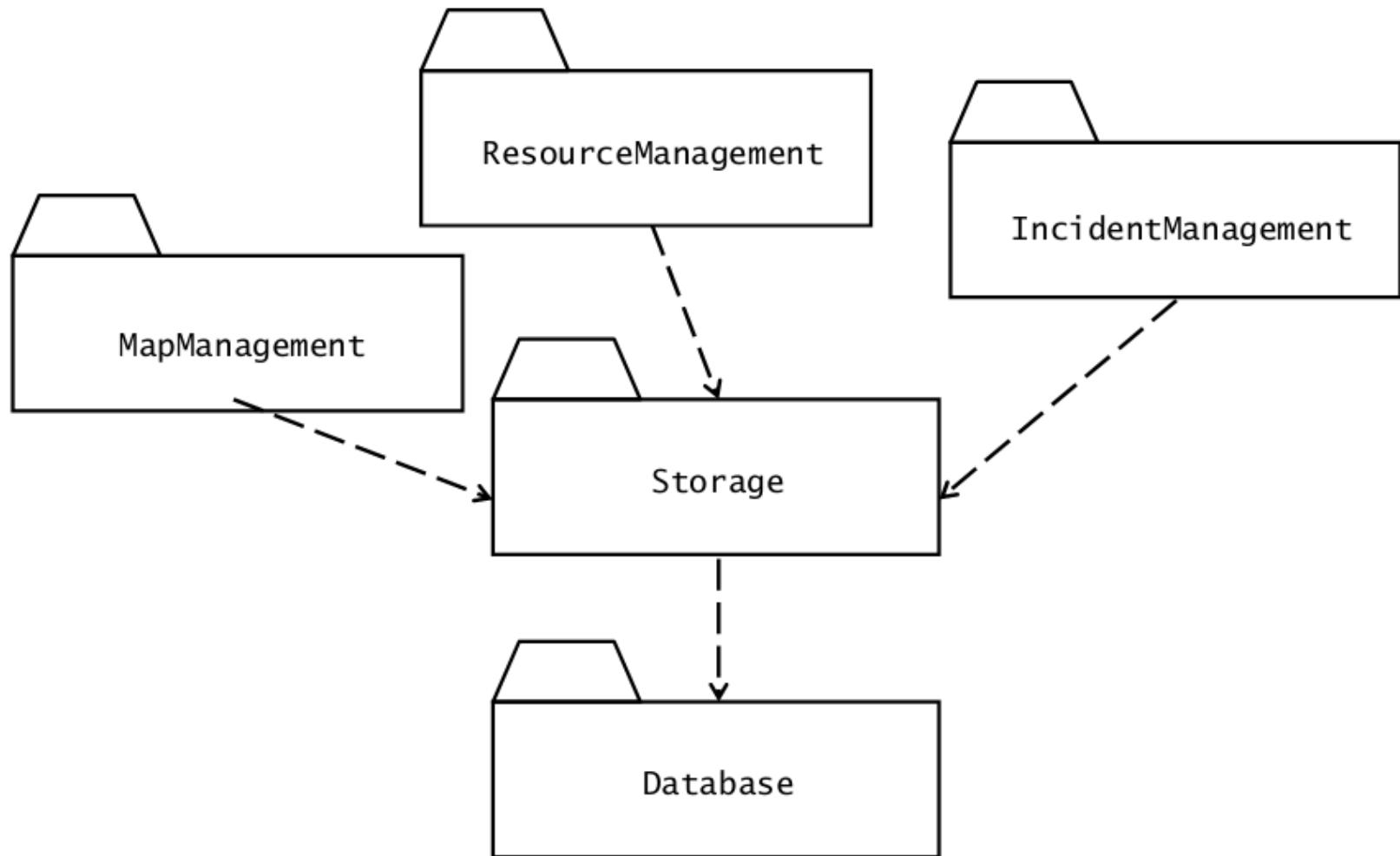
---

## Alternative 1: Direct access to the Database subsystem

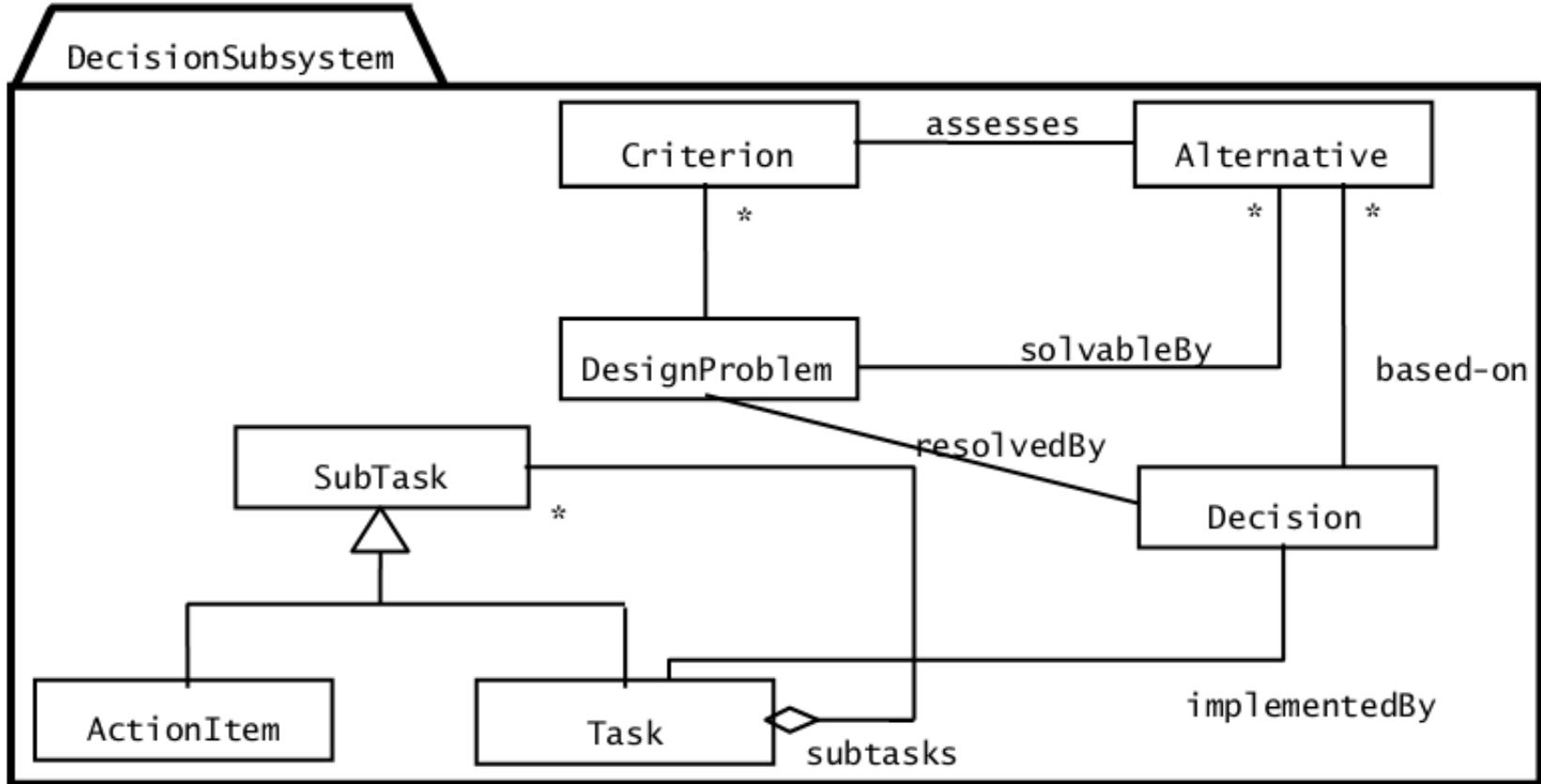


# Example design alternatives: FRIEND

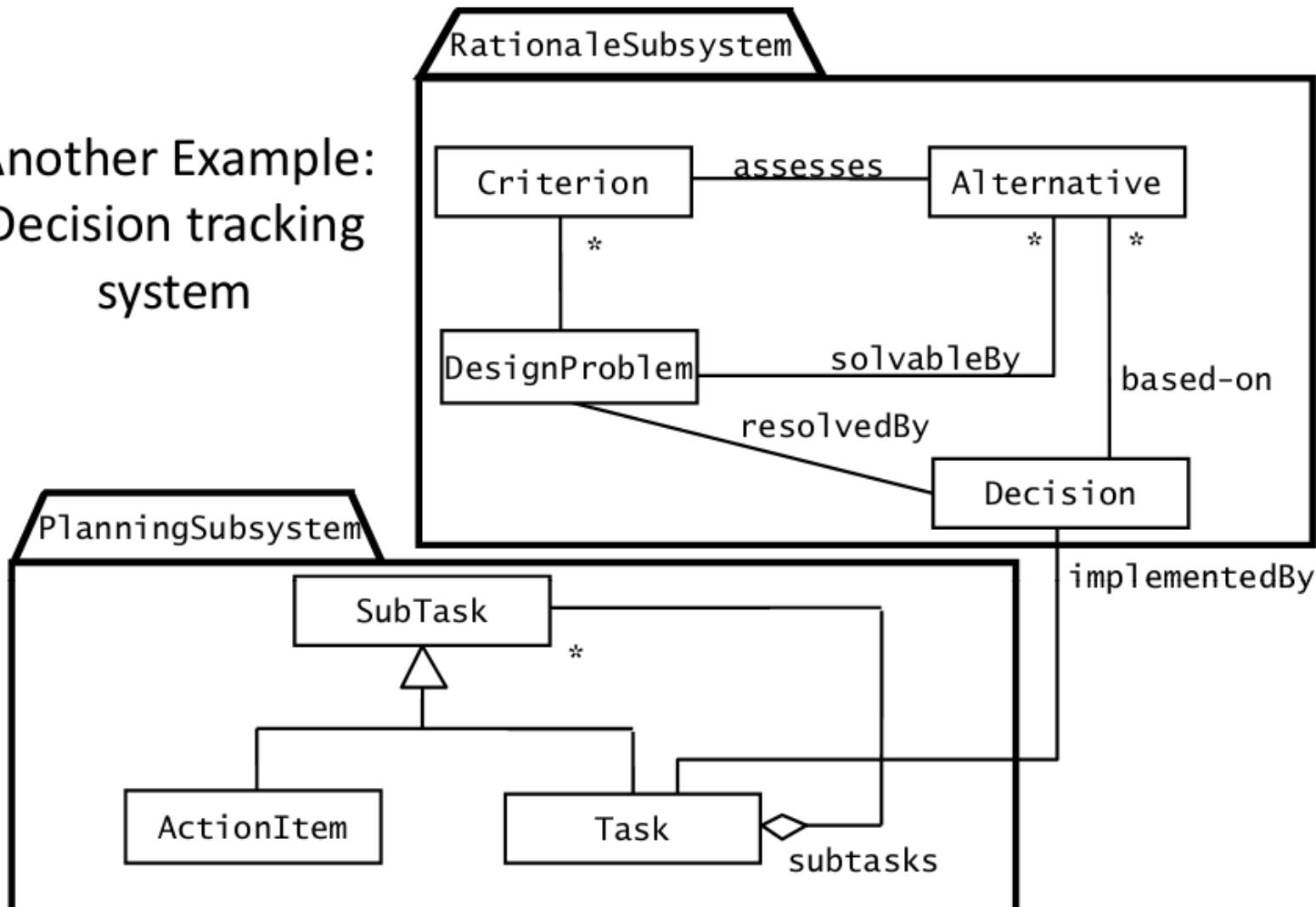
---



# Another Example: Decision tracking system



## Another Example: Decision tracking system



# Choosing Subsystems

---

**Primary Question:**

What kind of service is provided by the subsystems (subsystem interface) ?

**Secondary Question:**

Organization: Can the subsystems be hierarchically ordered ([layers](#)), and /or partitioned?

Deployment: How the subsystem is going to be deployed?

Q. What kind of model is good for describing [layers](#) and [partitions](#)?

# Subsystem Decomposition: Partitions and Layers

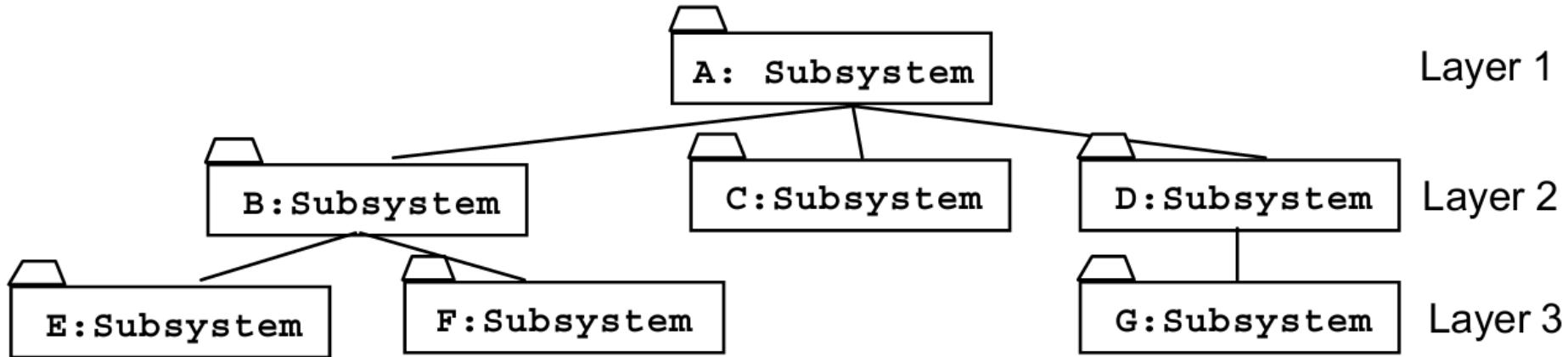
---

A large system is usually decomposed into subsystems using both, layers and partitions

- A Partition divide a system into several independent (or weakly-coupled) subsystems that provide services on the same level of abstraction.
- A layer is a subsystem that provides subsystem services to a higher layers (level of abstraction)
  - A layer can only depend on lower layers
  - A layer has no knowledge of higher layers

# Subsystem Decomposition: Partitions and Layers

---



## Subsystem Decomposition Heuristics:

- No more than **7+/-2** subsystems

More subsystems increase cohesion but also complexity (more services)

- No more than **4+/-2** layers, use 3 layers (good)

# Relationship between Subsystems

---

## Partition relationship

- The subsystem have mutual but not deep knowledge about each other
- Partition A “Calls” partition B and partition B “Calls” partition A

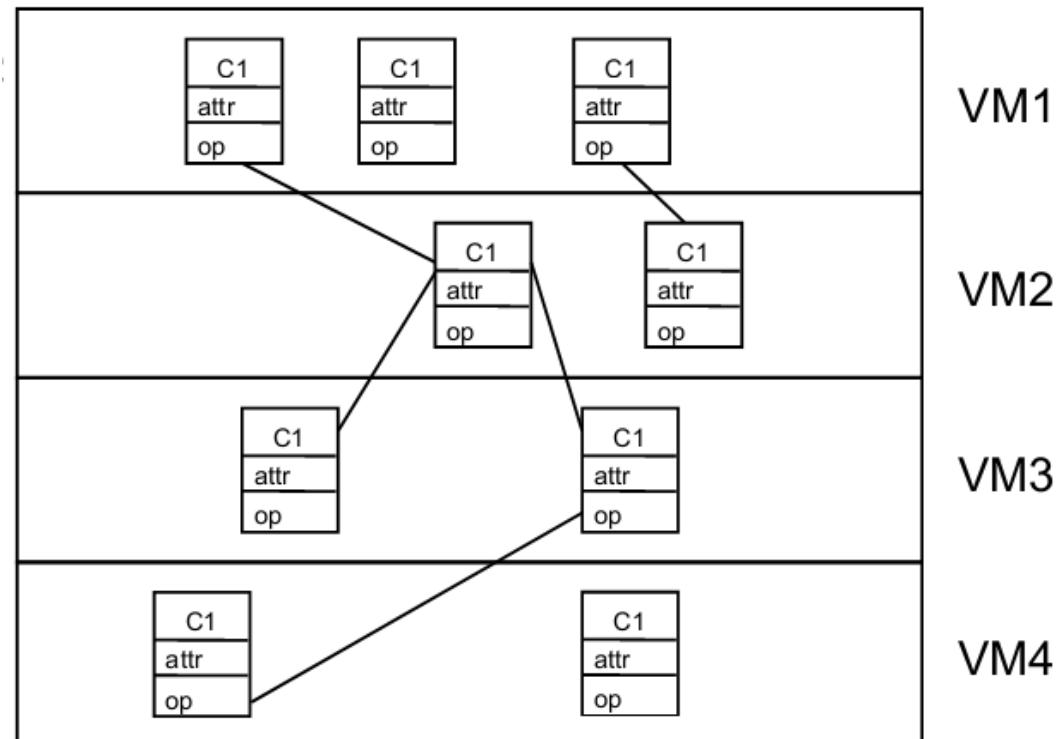
## Layer relationship

- Layer A “Calls” Layer B (runtime)
- Layer A “Depends on” Layer B (“make” dependency, compile time)

# Closed Architecture (Opaque Layering)

Any layer can only invoke operations from the immediate layer below

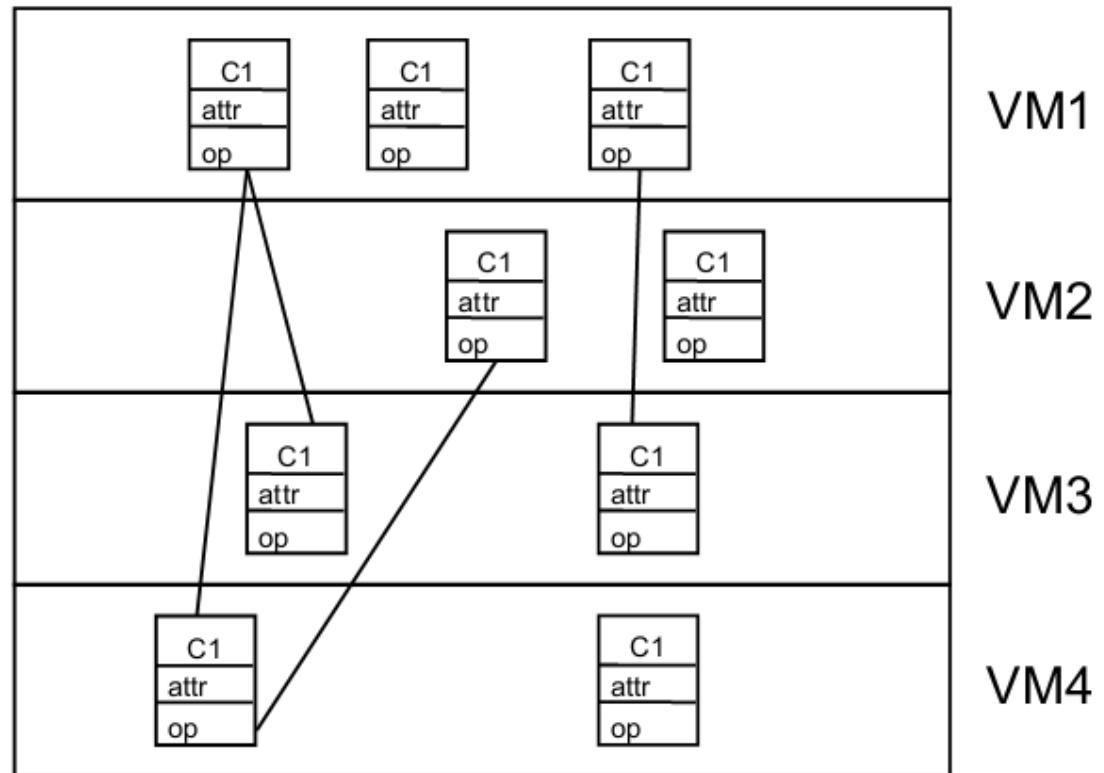
Design Goal: High maintainability, flexibility



# Open Architecture (Transparent Layering)

Any layer can invoke operations from any layers below

Design Goal: Runtime, efficiency



# Software Architectural Styles

---

## Subsystem decomposition

- Identification of subsystems, services, and their relationship to each other.

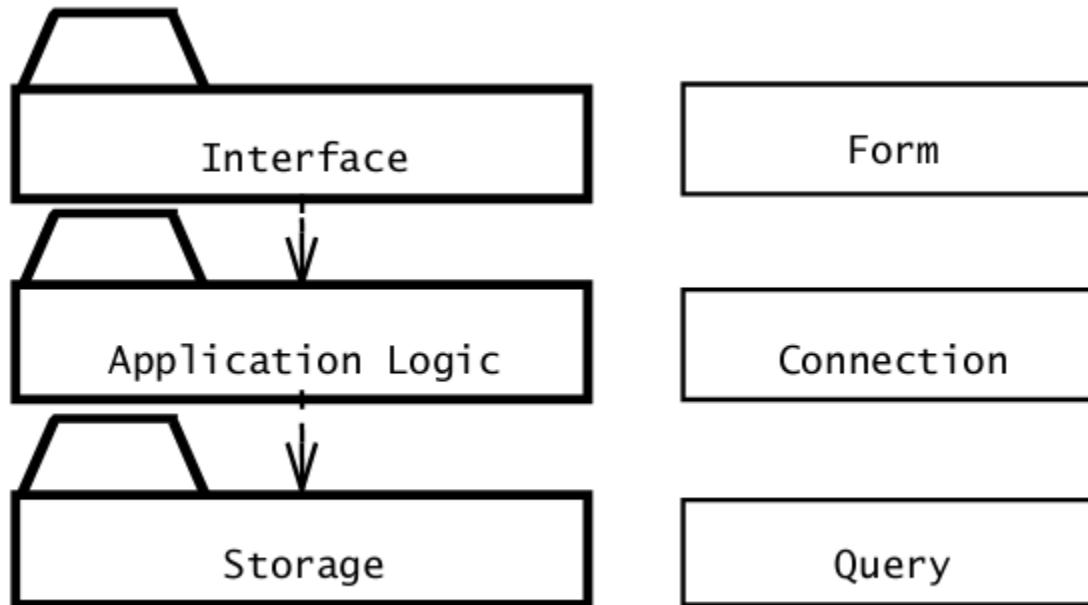
Specification of the system decomposition is critical.

## Organization of services (Patterns for software architecture)

- Client/Server
- Peer-To-Peer
- Repository
- Model/View/Controller
- Pipes and Filters
- ...and others

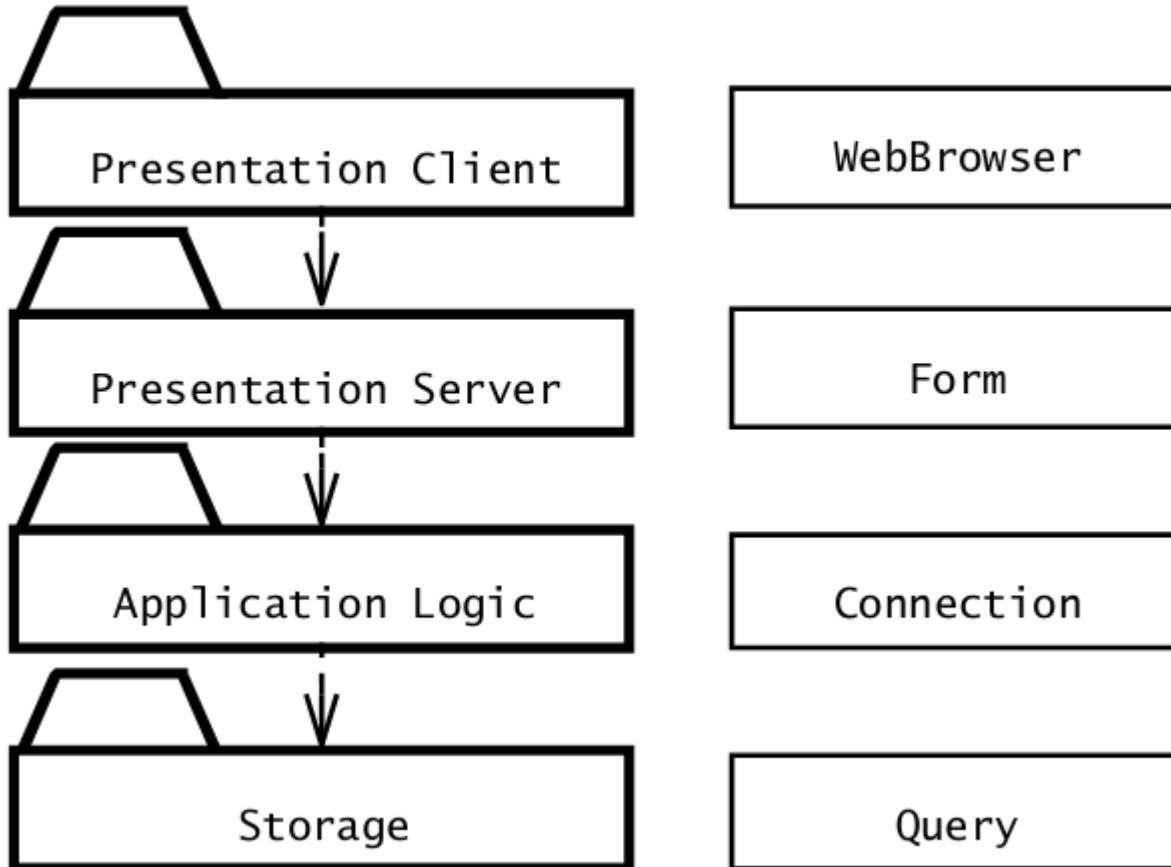
# C/S: Three-tier architectural style

---



# Four-tier architectural style

---



## Issues with C/S

---

Layered systems (in dedicated client-server mode) do not provide peer-to-peer communication

Example: Database receives queries from application but also sends notifications to application when data have changed

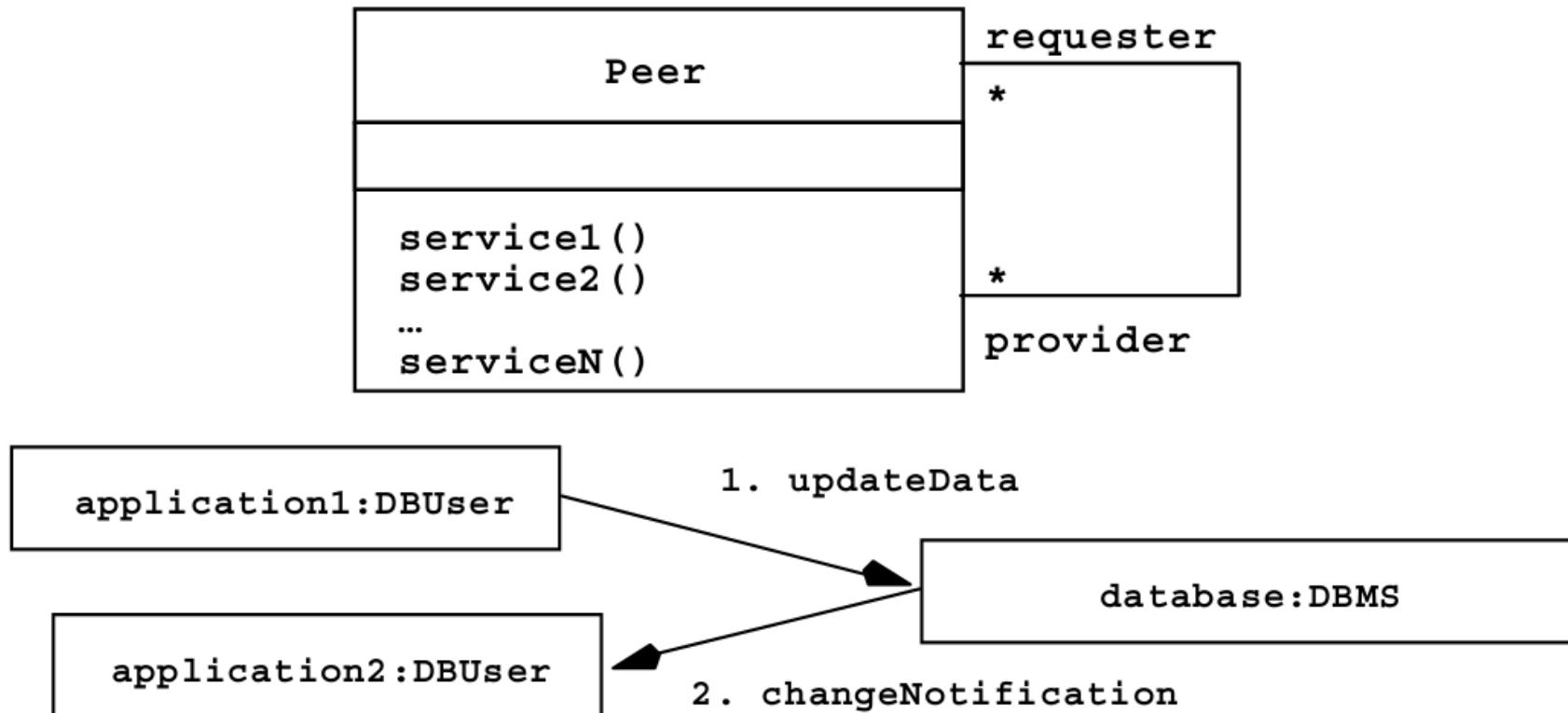
# Peer-to-Peer Architectural Style

---

Generalization of Client/Server Architecture

Clients can be servers and servers can be clients

More difficult because of possibility of deadlocks



# Model/View/Controller

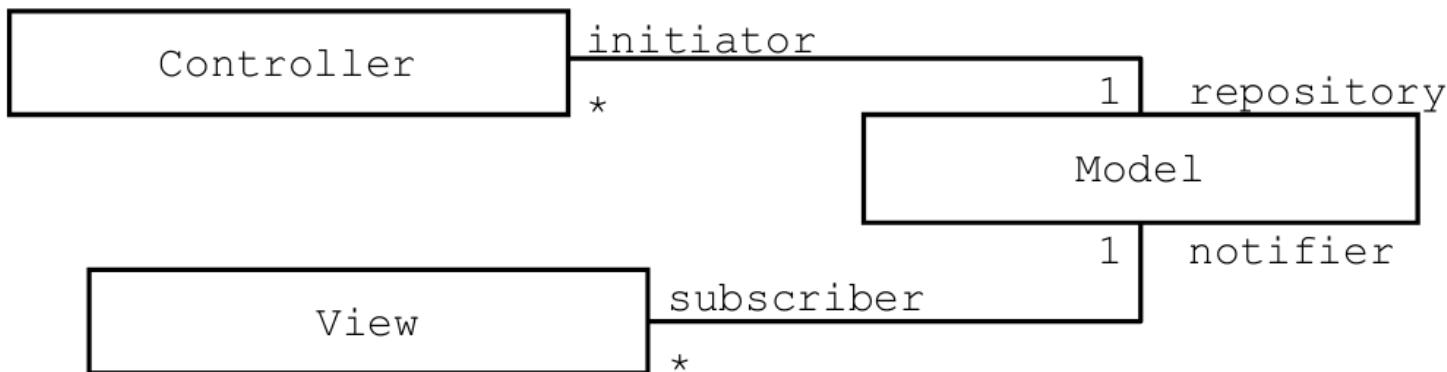
---

Subsystems are classified into 3 different types

- Model subsystem: Responsible for application domain knowledge
- View subsystem: Responsible for displaying application domain objects to the user
- Controller subsystem: Responsible for sequence of interactions with the user and notifying views of changes in the model.

MVC is a special case of a repository architecture:

- Model subsystem implements the central data structure, the Controller subsystem explicitly dictate the control flow



# Example: MVC Architectural Style

The image shows a Mac OS X desktop environment. On the left, a file browser window titled "CBSE" is open, displaying a list of files in the "Computer" folder. The list includes "9DesignPatterns.ppt", "9DesignPatterns.ppt2", "10DesignPatterns2.ppt", "11Testing.ppt", "11Testing2.ppt", and "12Object Design.ppt". The "View" menu is selected. On the right, a contextual information dialog for "9DesignPatterns.ppt2" is displayed, showing details like kind, size, and location.

Name	Date
9DesignPatterns.ppt	Fri, Jan 25, 2002, 14:17
9DesignPatterns.ppt2	Wed, Jan 23, 2002, 14:17
10DesignPatterns2.ppt	Thu, Jan 24, 2002, 14:17
11Testing.ppt	Fri, Jan 25, 2002, 14:17
11Testing2.ppt	Fri, Jan 25, 2002, 14:17
12Object Design.ppt	Fri, Jan 25, 2002, 14:17

**9DesignPatterns.ppt2 Info**

**General:**

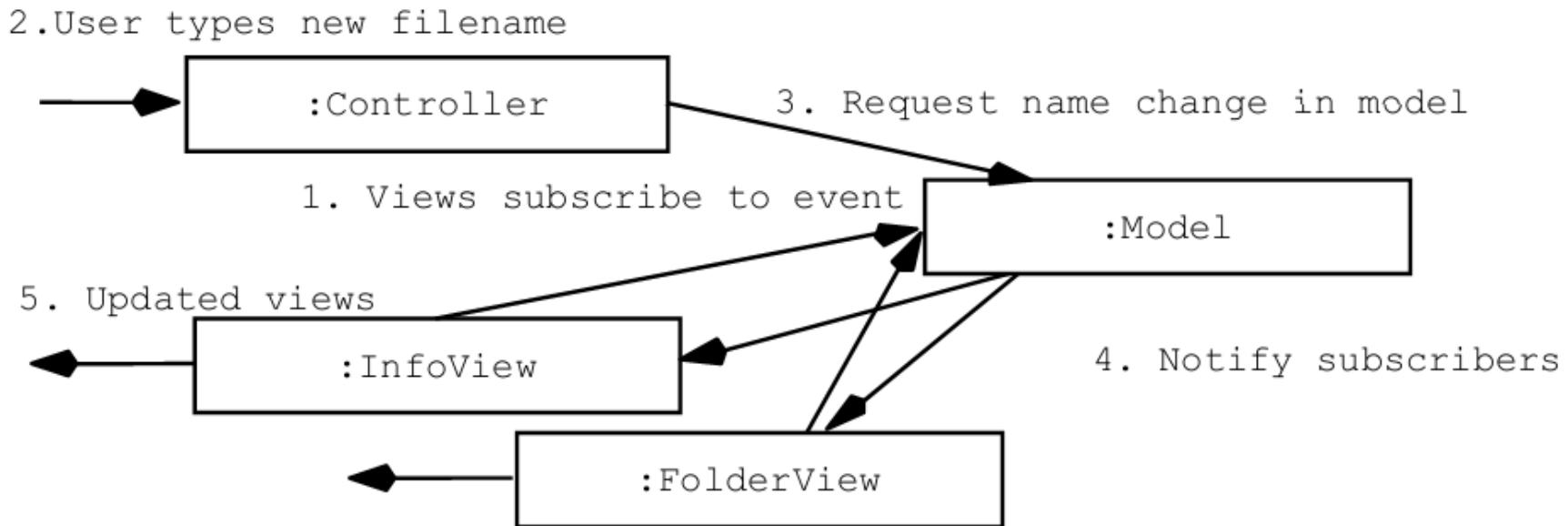
**9DesignPatterns.ppt2**

Kind: Microsoft PowerPoint document  
Size: 268 KB on disk (269,598 bytes)  
Where: Macintosh  
HD:Users:bob:Documents:teaching:  
CBSE:CBSE:  
Created: Wed, Jan 23, 2002, 14:17  
Modified: Wed, Jan 23, 2002, 14:17

Stationery Pad  
 Locked

► Name & Extension:  
► Open with:  
► Preview:  
► Ownership & Permissions:  
► Comments:

# Sequence of Events (Collaborations)



# Summary

---

Design is the process of adding details to the requirements analysis and making implementation decisions

- An evolutionary activity
- Consists of
  - Sub-system Design (Choosing an Architecture)
  - Interface Design
  - Object Design (Solution domain)

Next Lectures...  
Object Design  
Specifying Interfaces  
Package diagram  
Design Patterns

---



# IT314: Software Engineering

*Software Effort Estimation*  
Use Case Points



# Estimation?

---

- A large proportion of industrial systems development projects significantly
    - overrun budget or
    - are delivered after schedule (or not delivered at all), or
    - are not delivered with the specified functionality.
- Need for early and precise effort estimates.
- Can use case models be used to improve estimation?



# Approach...

---

- A use case model defines the functional scope of the system to be developed. The functional scope is the basis for top-down estimation.
- Estimation parameters can be derived from a use case model.
- Following a use case driven development process, a high-level use case model is available in the inception phase, and a detailed use case model is available at the start of the elaboration phase.
- Many companies use a system's use case model in the estimation process.
- How can a use case model best be applied in estimating software development effort ?

# Use Case Modeling

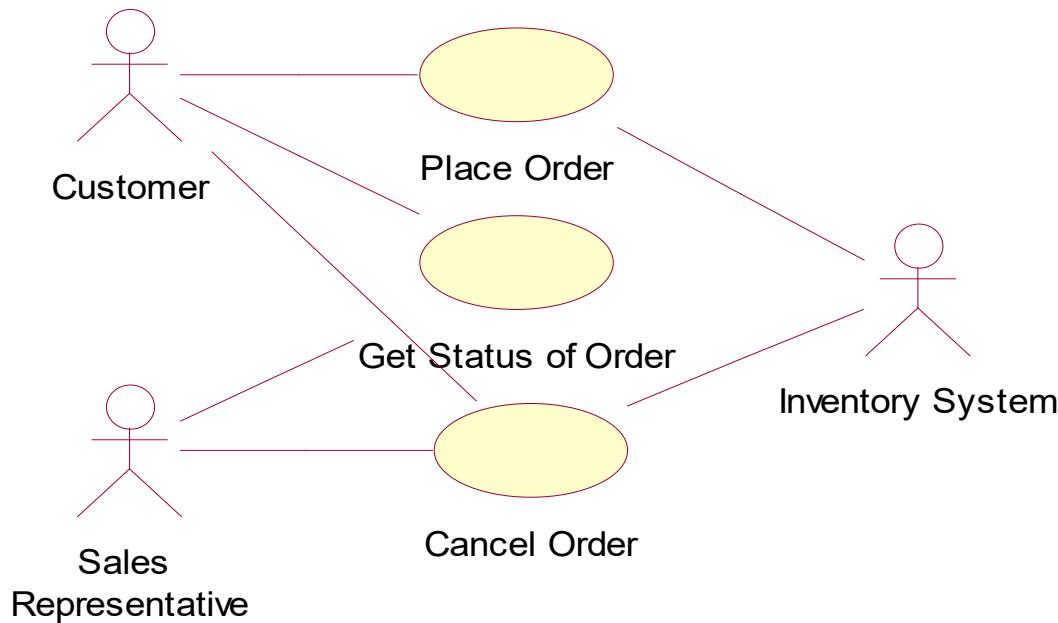
---

A use case model describes a system's intended functions and its environment. It has two parts:

1. A diagram that provides an overview of actors and use cases, and their interactions.
    - An actor represents a role that the user can play with regard to the system.
    - A use case represents an interaction between an actor and the system.
  2. The use case descriptions detail the requirements by documenting the flow of events between the actors and the system.
-

# Example of Use Case Diagram

---



# Example of Use Case Description

---

Use Case Name: Place Order

Short description:

The customer provides address information and a list of product codes.  
The system confirms the order.

Basic flow of events:

1. Customer enters name and address
2. Customer enters product codes for items he wishes to order
3. The system will supply a product description and price for each item
4. The system will keep a running total of items ordered as they are entered
5. The customer enters credit card information
6. The system validates the credit card information
7. The system issues a receipt to the customer

Alternative flow of events:

3.1 The product is out of stock:

- 3.1.1 The system informs the customer that the product can not be ordered.

6.1 The credit card is invalid

- 6.1.1 The system informs the customer that his credit card is invalid
- 6.1.2 The customer can enter credit card information again or cancel the order.

Pre-Conditions: The customer is logged on to the system

Post-Conditions: The order has been submitted

Extension Points: None

---



# Use Case Point (UCP)

---

- The UCP considers the Use Case (UC) model to estimate the size and effort of an object-oriented system.
- The UCP is an adaptation from FP and MKII FP (United Kingdom Software Metrics Association (UKSMA))..

# Estimation Method

---

- Each actor and use case is categorized according to complexity and assigned a weight.
  - The complexity of a use case is measured in number of transactions.
- The unadjusted use case points are calculated by adding the weights for each actor and use case.
- The unadjusted use case points are adjusted based on the values of 13 technical factors and 8 environmental factors.
- Finally the adjusted use case points are multiplied with a productivity factor.

# Estimation Method

---

Use Case Classification	No. of Transactions	Weight
Simple	1 to 3 transactions	5
Average	4 to 7 transactions	10
Complex	8 or more transactions	15

$$\text{UUCW} = (\text{Total No. of Simple Use Cases} \times 5) + (\text{Total No. Average Use Case} \times 10) + (\text{Total No. Complex Use Cases} \times 15)$$

Actor Classification	Type of Actor	Weight
Simple	External system that must interact with the system using a well-defined API	1
Average	External system that must interact with the system using standard communication protocols (e.g. TCP/IP, FTP, HTTP, database)	2
Complex	Human actor using a GUI application interface	3

$$\text{UAW} = (\text{Total No. of Simple actors} \times 1) + (\text{Total No. Average actors} \times 2) + (\text{Total No. Complex actors} \times 3)$$

---

# Adjust Based on Technical Factors

---

Factor number	Factor description	Weight
T1	Distributed system	2
T2	Response or throughput performance objective	1
T3	End-user efficiency	1
T4	Complex internal processing	1
T5	Code must be reusable	1
T6	Easy to install	0.5
T7	Easy to use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent	1
T11	Includes special security features	1
T12	Provides direct access for third parties	1
T13	Special user training facilities are required	1

# Adjust Based on Environmental Factors

---

Factor number	Factor description	Weight
F1	Familiar with RUP	1.5
F2	Application experience	0.5
F3	Object-oriented experience	1
F4	Lead analyst capability	0.5
F5	Motivation	1
F6	Stable requirements	2
F7	Part-time workers	-1
F8	Difficult programming language	-1

# Producing an Estimate

---

- The unadjusted actor weight, UAW, is calculated adding the weights for each actor.
- The unadjusted use case weights, UUCW, is calculated correspondingly.
- The unadjusted use case points, UUCP, = UAW + UUCW.
- The technical factor, TCF, =  $.6 + (.01 * \sum_{1..13} T_n * Weight_n)$ .
- The environmental factor, EF, =  $1.4 + (-.03 * \sum_{1..8} F_n * Weight_n)$ .
- $UCP = UUCP * TCF * EF$
- $Estimate = UCP * Productivity\ factor$

# Results from Case Studies

---

Company	Project	Use Case Estimate	Expert Estimate	Actual Effort	Deviation use case est.	Deviation exp. est.
Mogul	A	2550	2730	3670	-31%	-26%
Mogul	B	2730	2340	2860	-5%	-18%
Mogul	C	2080	2100	2740	-24%	-23%
CGE&Y	A	10831	7000	10043	+8%	-30%
CGE&Y	B	14965	12600	12000	+25%	+5%
IBM	A	4086	2772*	3360	+22%	-18%
Students project	A	666		742	-10%	
Students project	B	487		396	+23%	
Students project	C	488		673	-25%	

# Characteristics of Projects

---

Project	Estimate produced	Use case model	Characteristics
Mogul – A	Before	Detailed	<b>Duration = 7 months, Team = 6, Development tools = Java and Web-logic Application domain = Banking</b>
Mogul – B	After	Detailed	<b>Duration = 3 months, Team = 6, Development tools = MSVisual studio Application domain = CRM within banking</b>
Mogul – C	After	Sequence diagrams	<b>Duration = 4 months, Team = 5, Development tools = Java and Web-logic Application domain = Banking</b>
CGE&Y – A	After	No details	<b>Duration = 7 months, Team = 6, Development tools = Java2Enterprise and Websphere, Application domain = Internet application for banking</b>
CGE&Y – B	Before	Detailed	<b>Duration = 9 months, Team = 3 - 4, Development tools = C++, Application domain = Real-time system, part of larger commercial system</b>

# Characteristics cont.

---

Project	Estimate produced	Use case model	Characteristics
IBM	After	Detailed	<b>Duration = 3 months, Team = 8,</b> <b>Development tools = Smalltalk, Java and C++,</b> <b>Application domain = Internet solution for home shopping.</b>
Students project	Before	Detailed	<b>Duration = 2 months, Team = 6,</b> <b>Development tools = Java and C++,</b> <b>Application domain = News service on the internet.</b>
Students project	Before	Detailed	<b>Duration = 2 months, Team = 5,</b> <b>Development tools = Java,</b> <b>Application domain = Travelling information service for wap.</b>
Students project	Before	Detailed	<b>Duration = 2 months, Team = 5,</b> <b>Development tools = Java and IDEA 2.0,</b> <b>Application domain = Generating source code from UML .</b>



# Prerequisites for Applying the UCP Method

---

## 1. Correctness of the use case model:

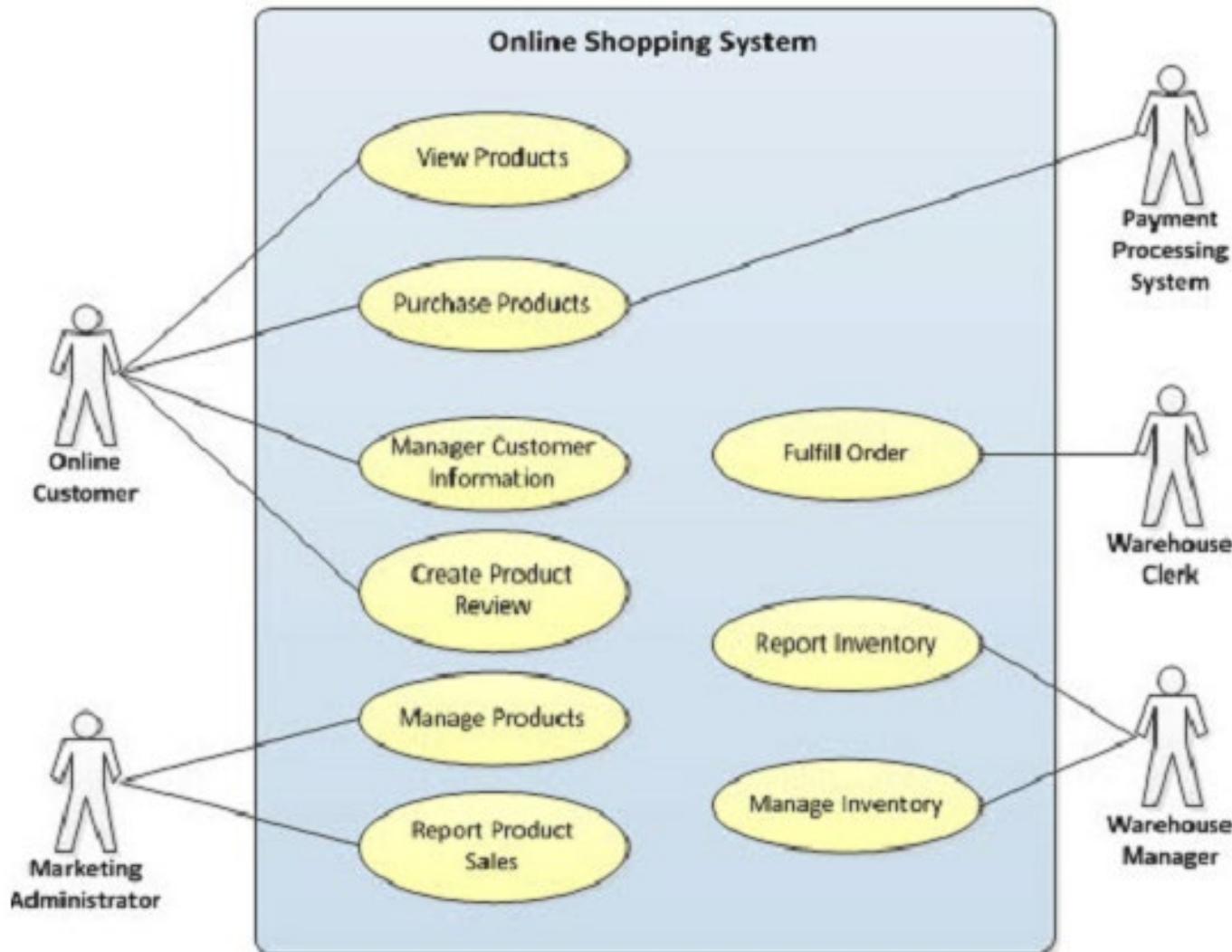
The use case model should include the functional requirements of all the user groups. The main challenge is sufficient access to skilled and motivated domain experts.

## 2. Level of detail:

The use case model should be described at an appropriate level of detail. The main challenges are to obtain balanced use cases and avoid "infinite" expansion. Possible solutions are guidelines and good examples of use case models.

# Example !!

---



# UUCW & UAW

---

$\text{UUCW} = (\text{Total No. of Simple Use Cases} \times 5) + (\text{Total No. Average Use Cases} \times 10) + (\text{Total No. Complex Use Cases} \times 15)$

For the Online Shopping System, the  $\text{UUCW} = (2 \times 5) + (3 \times 10) + (4 \times 15) = 100$

$\text{UAW} = (\text{Total No. of Simple Actors} \times 1) + (\text{Total No. Average Actors} \times 2) + (\text{Total No. Complex Actors} \times 3)$

For the Online Shopping System,  $\text{UAW} = (1 \times 1) + (0 \times 2) + (4 \times 3) = 13$

---

<b>Factor</b>	<b>Description</b>	<b>Weight</b>	<b>Assigned Value</b>	<b>Weight x Assigned Value</b>
T1	Distributed system	2.0	5	10
T2	Response time/performance objectives	1.0	5	5
T3	End-user efficiency	1.0	3	3
T4	Internal processing complexity	1.0	2	2
T5	Code reusability	1.0	3	3
T6	Easy to install	0.5	1	0.5
T7	Easy to use	0.5	5	2.5
T8	Portability to other platforms	2.0	2	4
T9	System maintenance	1.0	2	2
T10	Concurrent/parallel processing	1.0	3	3
T11	Security features	1.0	5	5
T12	Access for third parties	1.0	1	1
T13	End user training	1.0	1	1
<b>Total (TF):</b>				<b>42</b>

$$TCF = 0.6 + (TF/100)$$

For the Online Shopping System,  $TCF = 0.6 + (42/100) = 1.02$

$$TCF = 1.02$$

<b>Factor</b>	<b>Description</b>	<b>Weight</b>	<b>Assigned Value</b>	<b>Weight x Assigned Value</b>
E1	Familiarity with development process used	1.5	3	4.5
E2	Application experience	0.5	3	1.5
E3	Object-oriented experience of team	1.0	2	2
E4	Lead analyst capability	0.5	5	2.5
E5	Motivation of the team	1.0	2	2
E6	Stability of requirements	2.0	1	2
E7	Part-time staff	-1.0	0	0
E8	Difficult programming language	-1.0	4	-4
<b>Total (EF):</b>				<b>10.5</b>

Next, the ECF is calculated:

$$\text{ECF} = 1.4 + (-0.03 \times \text{EF})$$

For the Online Shopping System,  $\text{ECF} = 1.4 + (-0.03 * 10.5) = 1.085$

$$\text{ECF} = 1.085$$

$$\text{UCP} = (\text{UUCW} + \text{UAW}) \times \text{TCF} \times \text{ECF}$$

For the Online Shopping System,  $\text{UCP} = (100 + 13) \times 1.02 \times 1.085 = 125.06$

$$\text{UCP} = 125.06$$

Now that the size of the project is known, the total effort for the project can be estimated. For the Online Shopping System example, 28 man hours per use case point will be used.

$$\text{Estimated Effort} = \text{UCP} \times \text{Hours/UCP}$$

For the Online Shopping System,  $\text{Estimated Effort} = 125.06 \times 28$

$$\text{Estimated Effort} = 3501 \text{ Hours}$$

# Limitations of UCP

---

UCs in UCP could only be classified into one of the three categories

- Simple
- Average
- Complex

*(no matter how many sections it has or how big it is)*

For example, consider three UCs: UC1, UC2 and UC3 with respectively 70, 2 and 8 transactions.

According to the UCP classification tables,

- UC2 is considered simple
- UC1 and UC3 are complex.

However, UC1 is much more complex than UC3.

---

# Use Case Size Points (USP)

---

- Applied to each UC separately
- It is possible to estimate time and cost for the development of a particular UC and not only for the whole system
- USP measures the functionality by considering the structures and sections of a UC
  - counting the number and weight of scenarios, actors, precondition and postconditions.
- USP components
  - Actors
  - Pre-conditions
  - Post-conditions
  - Flow of events (Basic, Alternative)
  - Exception

# Actors Classification

---

- Each actor has its complexity (CA) determined according to the data provided to or received from the UC being classified.
- The total complexity of actors in the UC (TPA) is calculated by

$$TPA = \sum_{i=1}^n CA_i$$

n - Number of actors in the UC

Complexity	Data	UUSP
Simple	$\leq 5$	2
Average	6 to 10	4
Complex	$> 10$	6

# Precondition Classification

---

- Each precondition of the UC has the complexity (CPrC) determined according to the number of logical expressions tested by the condition
- The total complexity of the preconditions (TPPrC) is given by

$$TPPrC = \sum_{i=1}^n CPrC_i$$

n - Number of preconditions in the UC

Complexity	Tested Expression	UUSP
Simple	1 logical expression	1
Average	2 or 3 logical expressions	2
Complex	3 logical expressions	3

# Postcondition Classification

---

- The complexity of the postconditions (CPoC) is determined according to the number of related entities
- The total complexity of postconditions (TPPoC) is given by

$$TPPoC = \sum_{i=1}^n CPoC_i$$

n - Number of postconditions in the UC

Complexity	Entities	UUSP
Simple	$\leq 3$	1
Average	4 to 6	2
Complex	$> 6$	3

# Main Scenario Classification

---

- The main scenario must be classified according to its number of entities and to the number of elementary steps needed to the scenario conclusion.
- The complexity of the scenario (PCP) is given by the sum of both values (number of entities + number of steps)

Complexity	Entities + Steps	UUSP
Very Simple	$\leq 5$	4
Simple	6 to 10	6
Average	11 to 15	8
Complex	16 to 20	12
Very Complex	$> 20$	16

# Alternative Scenario Classification

---

- All the alternative scenarios are classified similarly to the main scenario.
- Each alternative scenario receives a number of points (PCA) according to

Complexity	Entities + Steps	UUSP
Very Simple	$\leq 5$	4
Simple	6 to 10	6
Average	11 to 15	8
Complex	16 to 20	12
Very Complex	$> 20$	16

- The total complexity of the alternative scenarios (TPCA) is given by

$$TPCA = \sum_{i=1}^n PCA_i$$

n - Number of alternative scenario in the UC

---

# Exception Classification

---

- Each exception present in the UC must also be analyzed according to its complexity (CE), determined by the number of logical expressions tested to detect the exception occurrence.
- The total points added by exceptions (TPE) are determined by

$$TPE = \sum_{i=1}^n CE_i$$

n - Number exceptions in the UC

Complexity	Tested Expressions	UUSP
Simple	1 logical expression	1
Average	2 or 3 logical expressions	2
Complex	> 3 logical expressions	3

# Unadjusted Use-case Size Point (UUSP)

---

- The Unadjusted Use-case Size Point (UUSP) is given by the sum of the complexity values of all sections of the UC

$$\begin{aligned} UUSP = & TPA + TPPrC + PCP + \\ & TPCA + TPE + TPPoc \end{aligned}$$

# Application of the adjustment factor (value between 0 to 5)

Factor	Requirement	Influence
F1	Data communication	I1
F2	Distributed processing	I2
F3	Performance	I3
F4	Equipment utilization	I4
F5	Transaction Capacity	I5
F6	On-line input of data	I6
F7	User efficiency	I7
F8	On-line update	I8
F9	Code reuse	I9
F10	Complex processing	I10
F11	Easiness of deploy	I11
F12	Easiness operation	I12
F13	Many places	I13
F14	Facility of change	I14

Technical Factors  
(adapted from FP)

$$FTA = 0.65 + (0.01 * \sum_{i=1}^{14} I_i)$$

Factor	Description	Influence
E1	Formal development process existence	I1
E2	Experience with the application being developed	I2
E3	Experience of the team with the used technologies	I3
E4	Presence on an experienced analyst	I4
E5	Stable requirements	I5

Environmental Factors

$$FAA = (0.01 * \sum_{i=1}^5 I_i)$$

# Adjusted USP

---

$$USP = UUSP * (FTA - FAA)$$

Next lecture....

PERT Chart

Project Planning Document

Testing Plans

Testing Strategy

*Example:*

*Source- [https://en.wikipedia.org/wiki/Use\\_Case\\_Points](https://en.wikipedia.org/wiki/Use_Case_Points)*

---



# IT 314: Software Engineering

*Object Relationships: Association (Aggregation, Composition)*



# Object Relationships

---

- Association
  - Aggregation
  - Composition
- Inheritance
- Dependency

# Aggregation

---

- Aggregation is a strong form of association in which an aggregate object is made of constituent parts.
- The aggregate is semantically an extended object that is treated as a UNIT in many operations, although physically it is made of several lesser objects.
- Aggregation is a transitive relation:
  - if A is a part of B and B is a part of C then A is also a part of C
- Aggregation is an antisymmetric relation:
  - If A is a part of B then B is not a part of A.

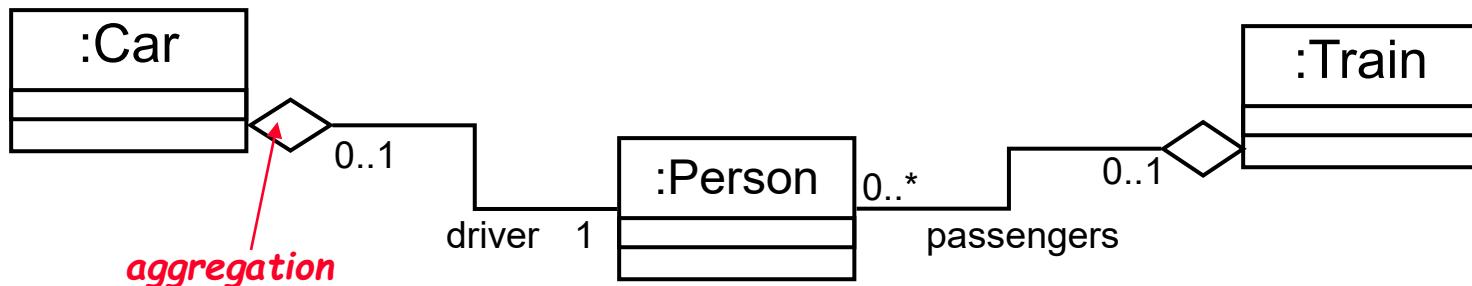
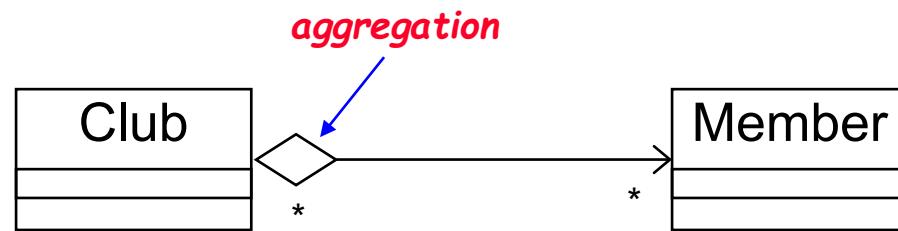
# Aggregation versus Association

---

- Aggregation is a special form of association, not an independent concept.
- Aggregation adds semantic connotations:
  - If two objects are tightly bound by a part-whole relation it is an aggregation.
  - If the two objects are usually considered as independent, even though they may often be linked, it is an association.
- Discovering aggregation
  - Would you use the phrase **part of**?
  - Do some operations on the whole automatically apply to its parts?
  - Do some attributes values propagates from the whole to all or some parts?
  - Is there an asymmetry to the association, where one class is subordinate to the other?

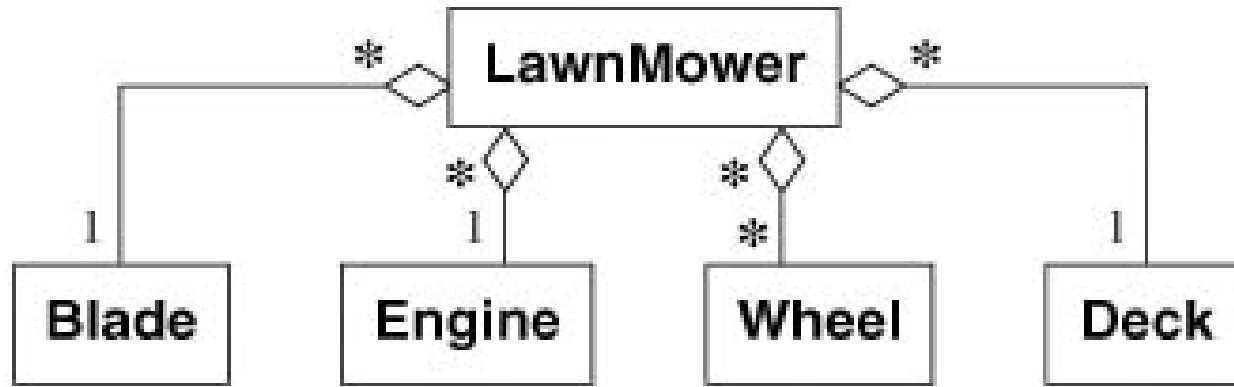
# Aggregation

↳ This is the “Has-a” or “Whole/part” relationship



# Example

---



# Aggregation versus Composition

---

- **Composition** is a form of aggregation with additional constraints:
  - A constituent part can belong to at most one assembly (whole).
    - it has a coincident lifetime with the assembly.
    - Deletion of an assembly object triggers automatically a deletion of all constituent objects via composition.
  - Composition implies ownership of the parts by the whole.
    - Parts cannot be shared by different wholes.

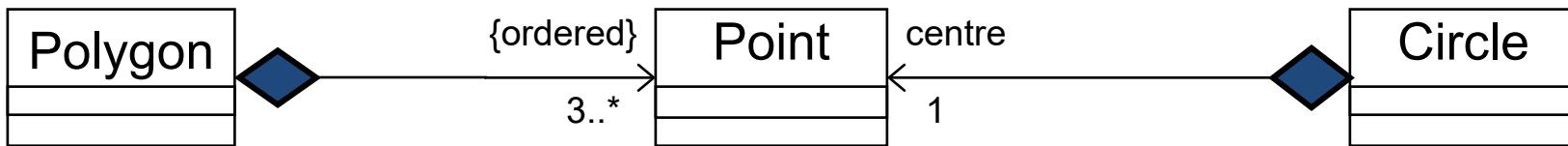
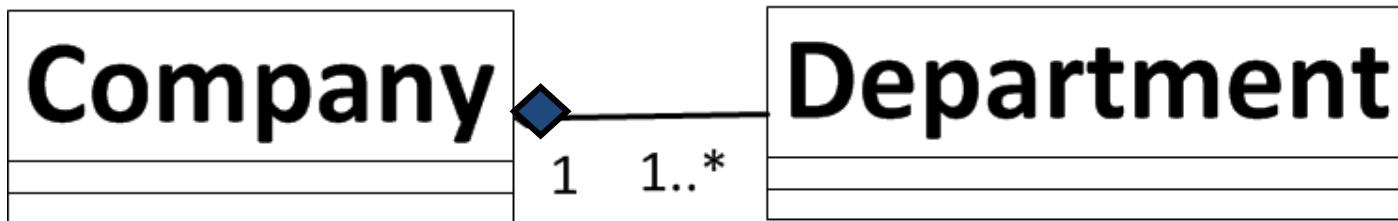
# Aggregation vs. Composition

## Ü Aggregation

- ↳ This is the “Has-a” or “Whole/part” relationship

## Ü Composition

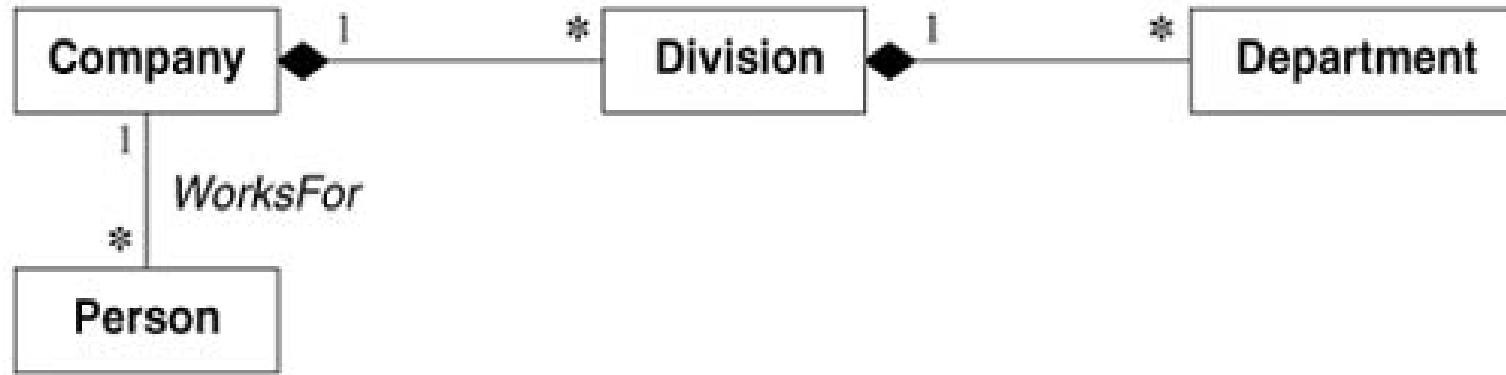
- ↳ Strong form of aggregation that implies ownership:
  - if the whole is removed from the model, so is the part.
  - the whole is responsible for the disposition of its parts
  - Note: Parts can be removed from the composite (where allowed) before the composite is deleted



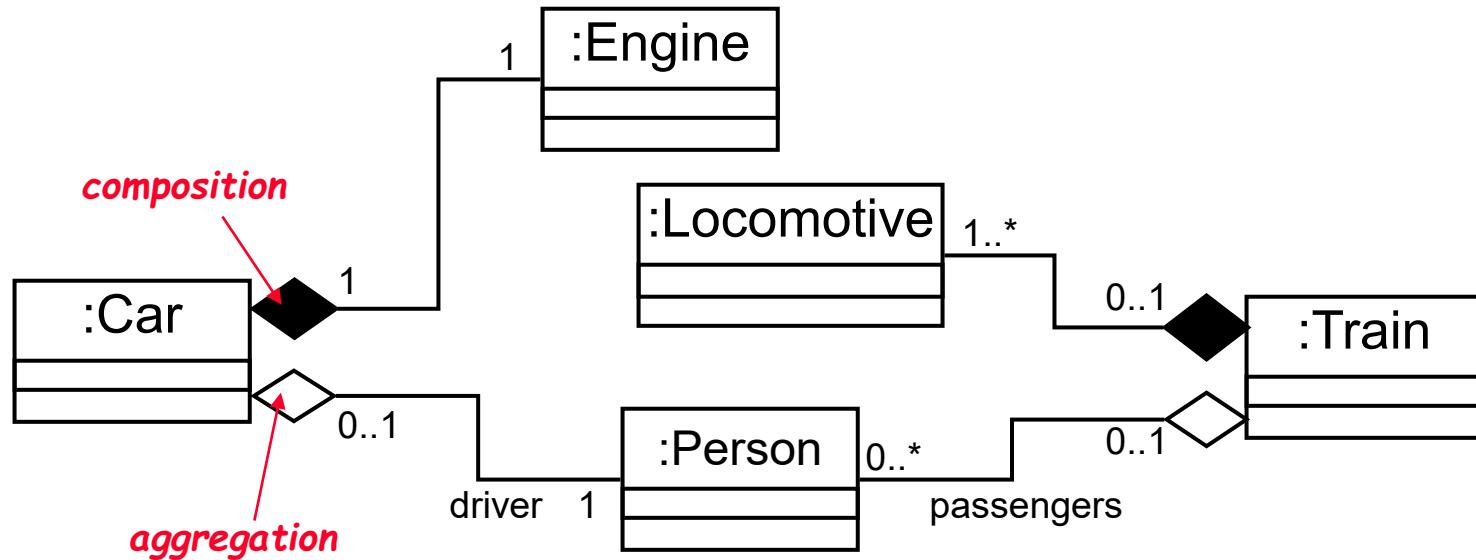
*Note: No sharing - any instance of point can be part of a polygon or a circle, but not both (Why?)*

# Example

---



# Aggregation and Composition



# Inheritance

---

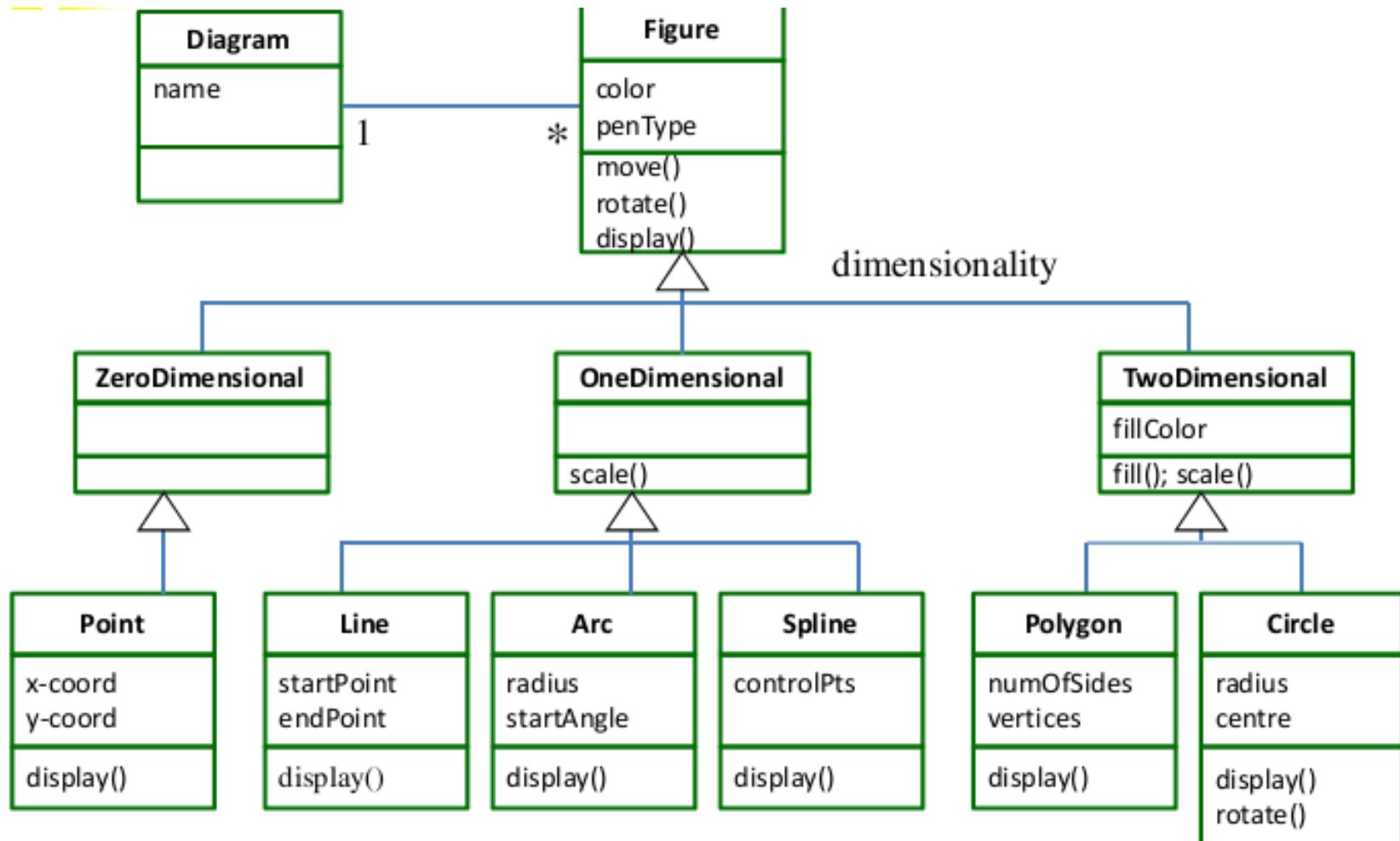
- A relationship between a set of closely related classes having similarities and differences
- Superclass - subclass relationship
- A subclass inherits all the members of its parent and may add attributes and methods of its own
- Superclass holds common attributes, operations, and associations; subclasses add specific attributes, operations, and associations
- *is-a or a-kind-of or a-type-of* relationship
- Depicts a hierarchy of relationship

# Inheritance

---

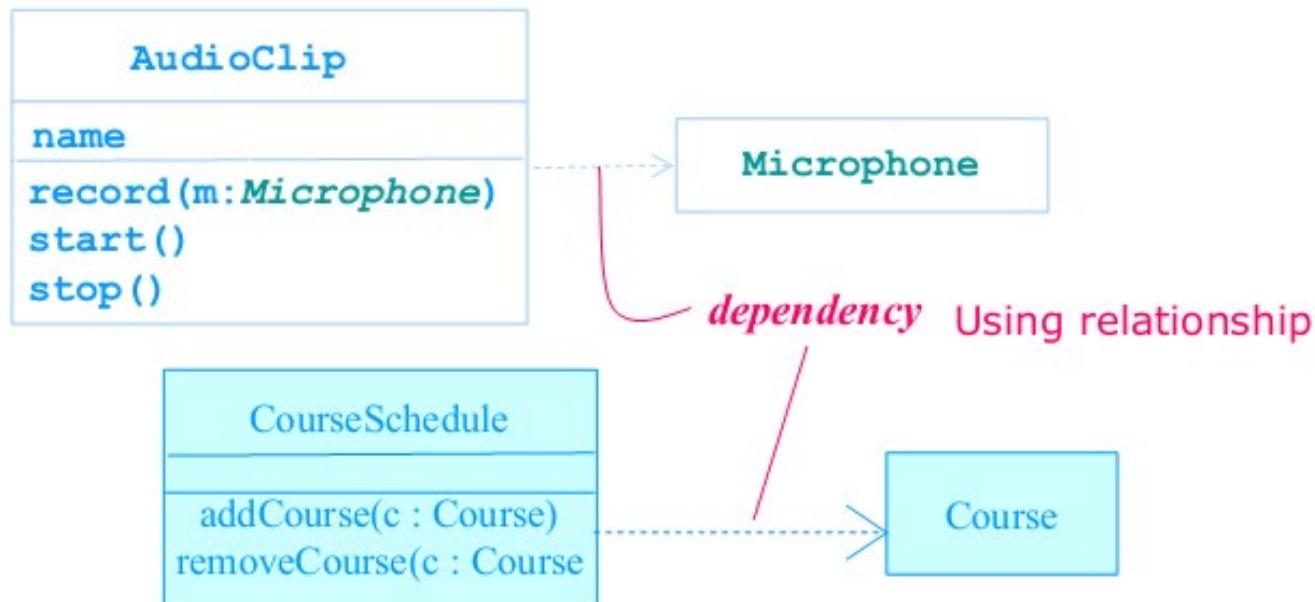
- A subclass inherits attributes and methods from its superclass
- Inheritance is also referred as Generalization-Specialization relationship - opposite perspectives
- Inheritance is also referred as Classification

# Example of Inheritance



# Dependency

- Dependency: A using relationship between two classes
  - A change in the specification of one class may affect the other
  - But not necessarily the reverse



# Dependency

---

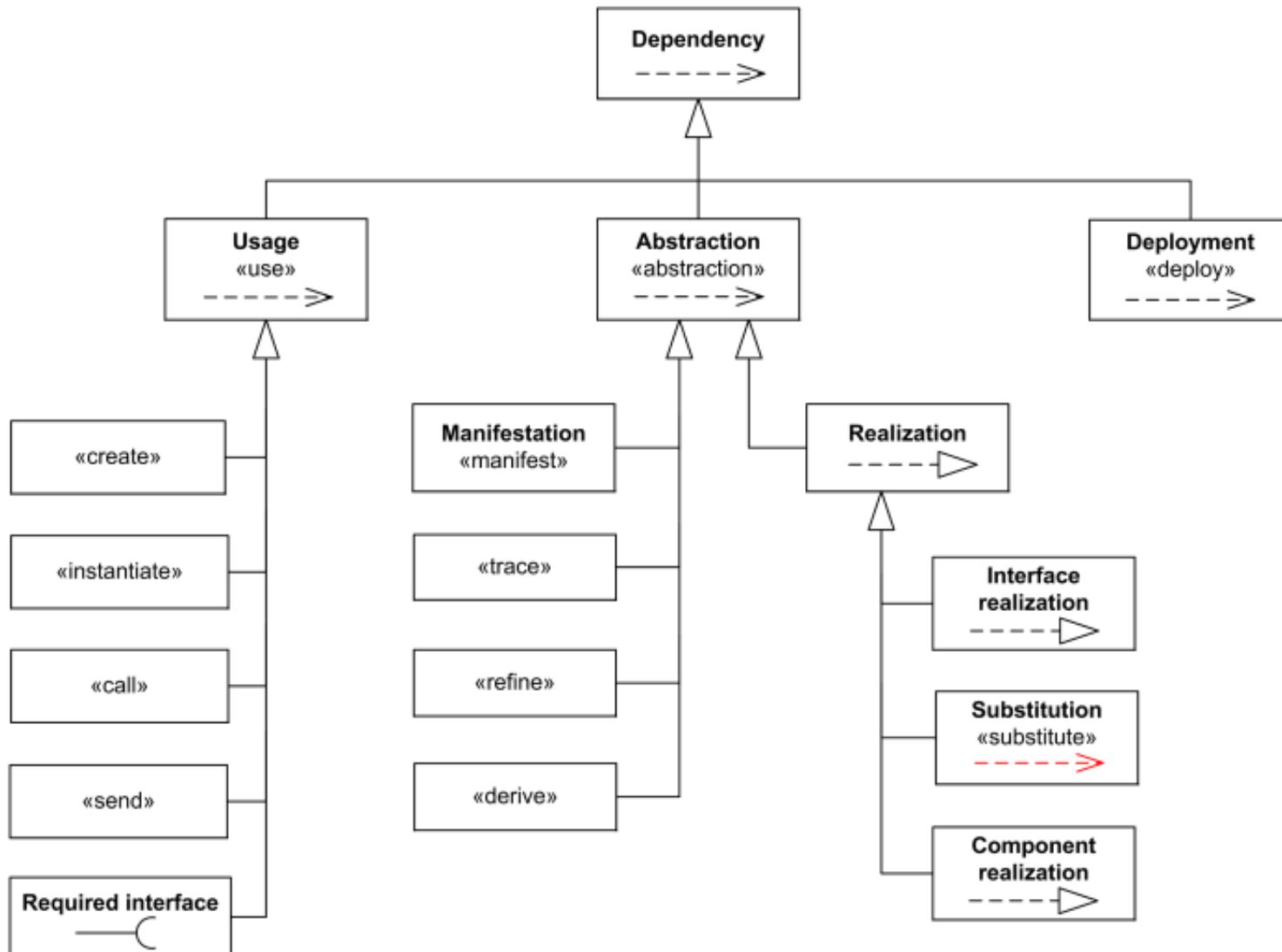
- A dependency is generally shown as a dashed arrow pointing from the **client** (dependent) at the tail to the **supplier** (provider) at the arrowhead.
- A change in one thing mat affect another.



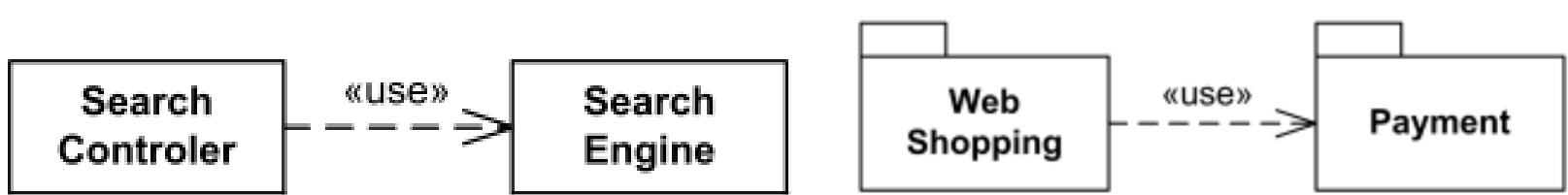
Wrong: *Car class has a dependency on the CarFactory class.*

Right: *CarFactory class depends on the Car class.*

# Dependency relationship overview diagram



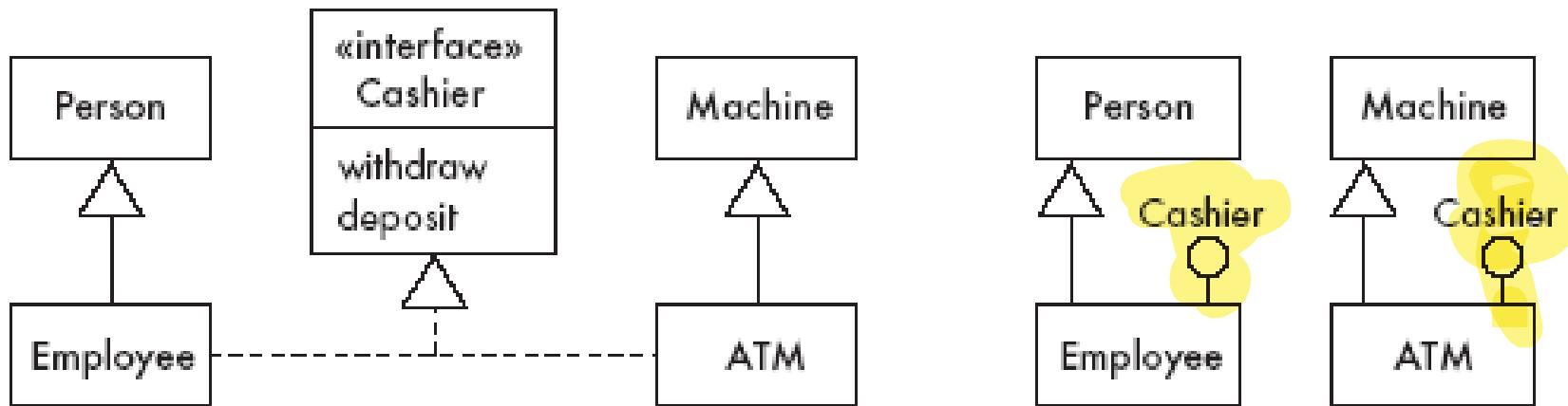
- A usage is a dependency relationship in which one element (client) requires another element (or set of elements) (supplier) for its full implementation or operation.
- The abstraction relates two elements representing the same concept but at different levels of abstraction.
- The deployment is a dependency which shows allocation (deployment) of an artifact to a deployment target.



# Interfaces

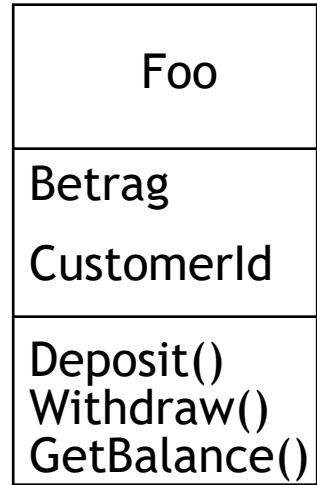
An interface describes a *portion of the visible behaviour* of a set of objects.

- An *interface* is similar to a class, except it lacks instance variables and implemented methods



# Object Modeling in Practice: Class Identification

---

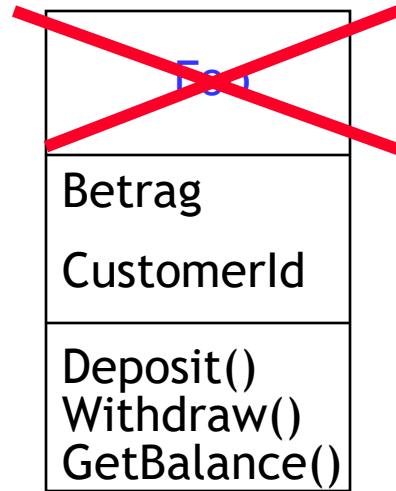
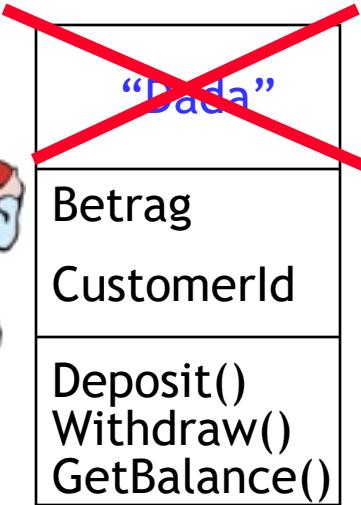


Class Identification: Name of Class, Attributes and Methods

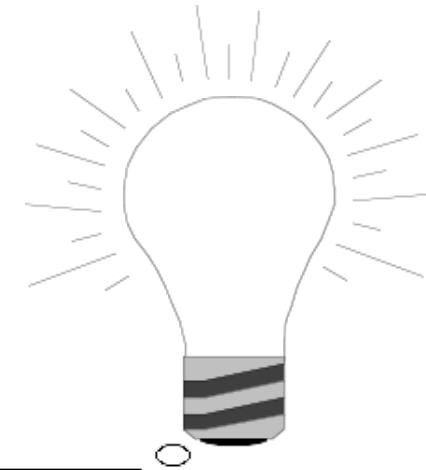
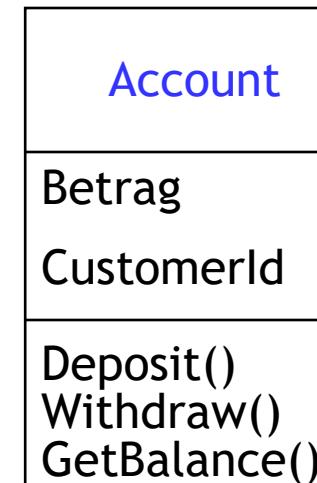
---

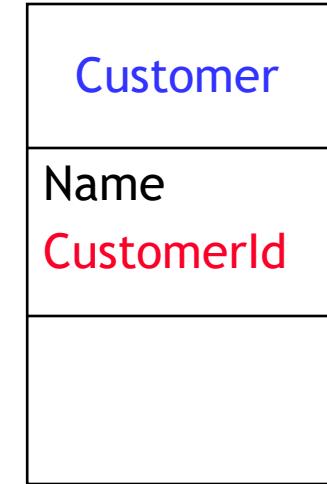
# Object Modeling in Practice: Encourage Brainstorming

---



Naming is important!  
Is **Foo** the right name?



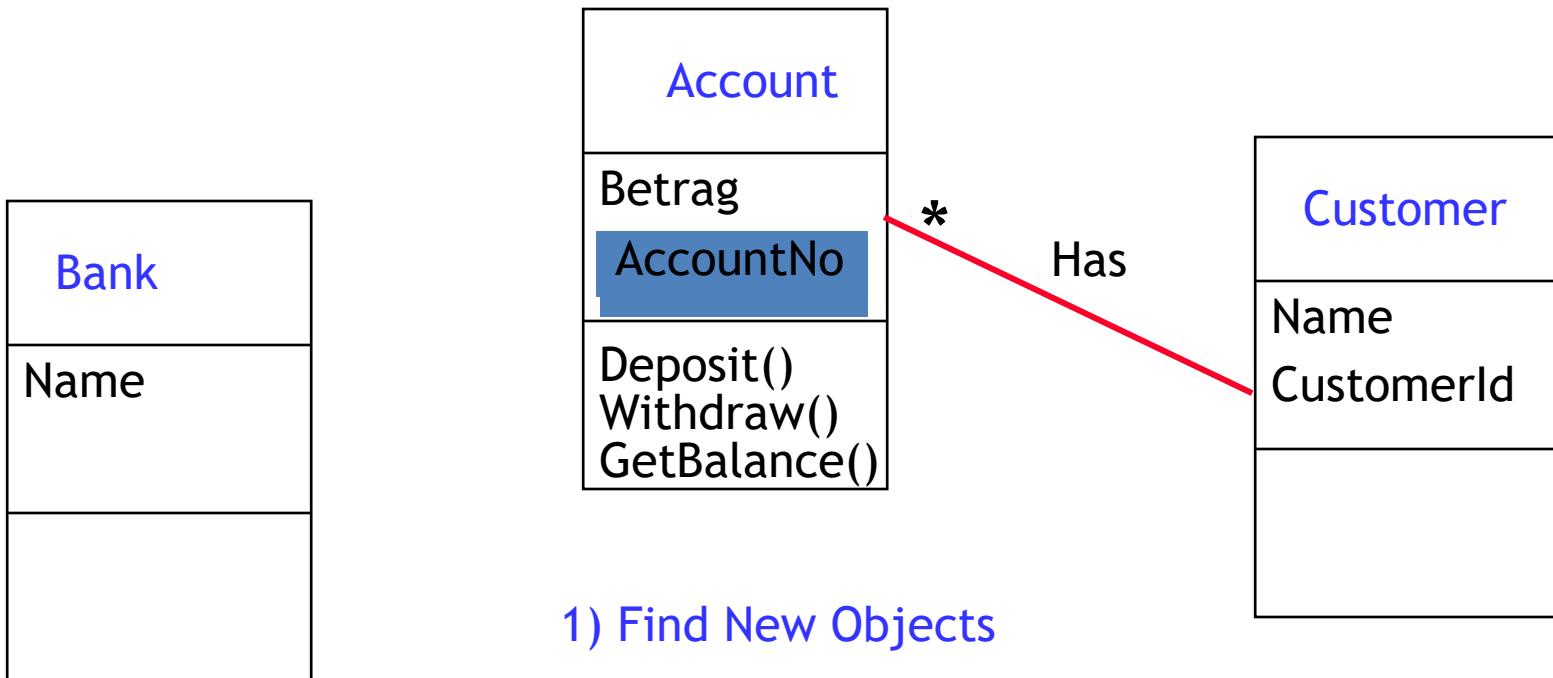


1) Find New Classes

2) Iterate on Names, Attributes and Methods

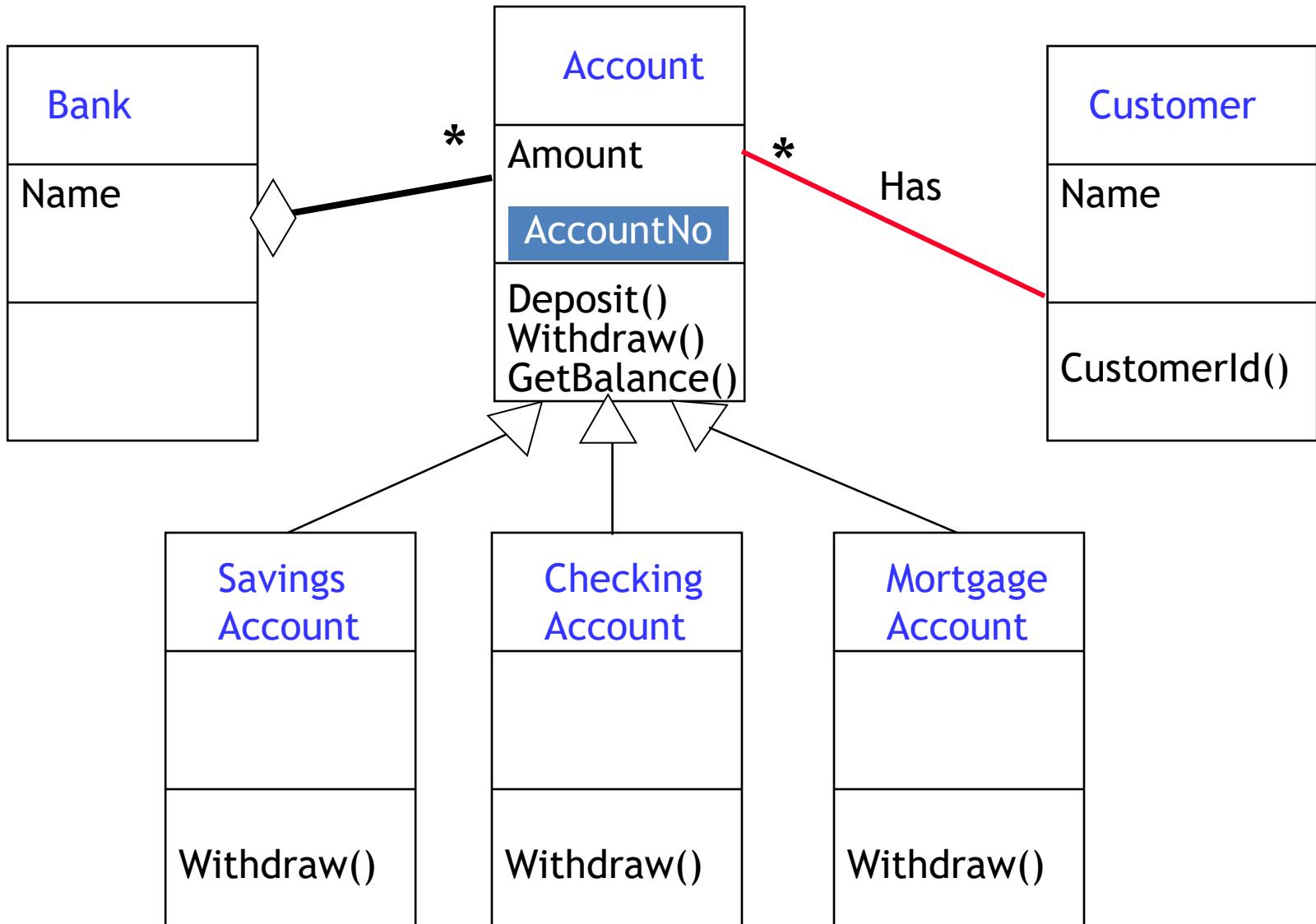
# Object Modeling in Practice: A Banking System

---



- 1) Find New Objects
  - 2) Iterate on Names, Attributes and Methods
  - 3) Find Associations between Objects
  - 4) Label the associations
  - 5) Determine the multiplicity of the associations
-

# Practice Object Modeling: Iterate, Categorize!





# Identifying associations and attributes

---

- Start with classes you think are most **central** and important
  - Decide on the clear and obvious data it must contain and its relationships to other classes.
  - Work outwards towards the classes that are less important.
  - Avoid adding many associations and attributes to a class
    - A system is simpler if it manipulates less information
-



# Tips about identifying and specifying valid associations

---

- An association should exist if a class
  - *possesses*
  - *controls*
  - *is connected to*
  - *is related to*
  - *is a part of*
  - *has as parts*
  - *is a member of*, or
  - *has as members*

some other class in your model

- Specify the multiplicity at both ends
  - Label it clearly.
-



# Identifying attributes

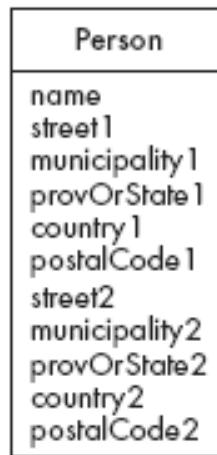
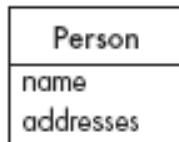
---

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number

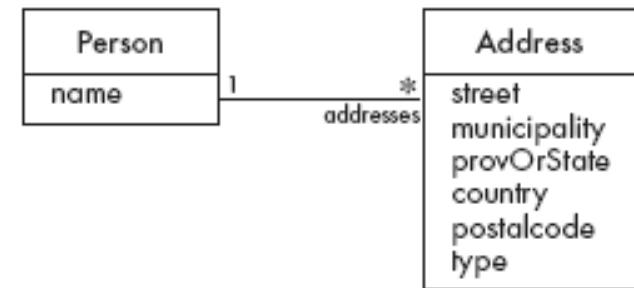
# Tips about identifying and specifying valid attributes

---

- It is not good to have many duplicate attributes
- If a subset of a class's attributes form a coherent group, then create a distinct class containing these attributes



Bad, due to  
a plural attribute

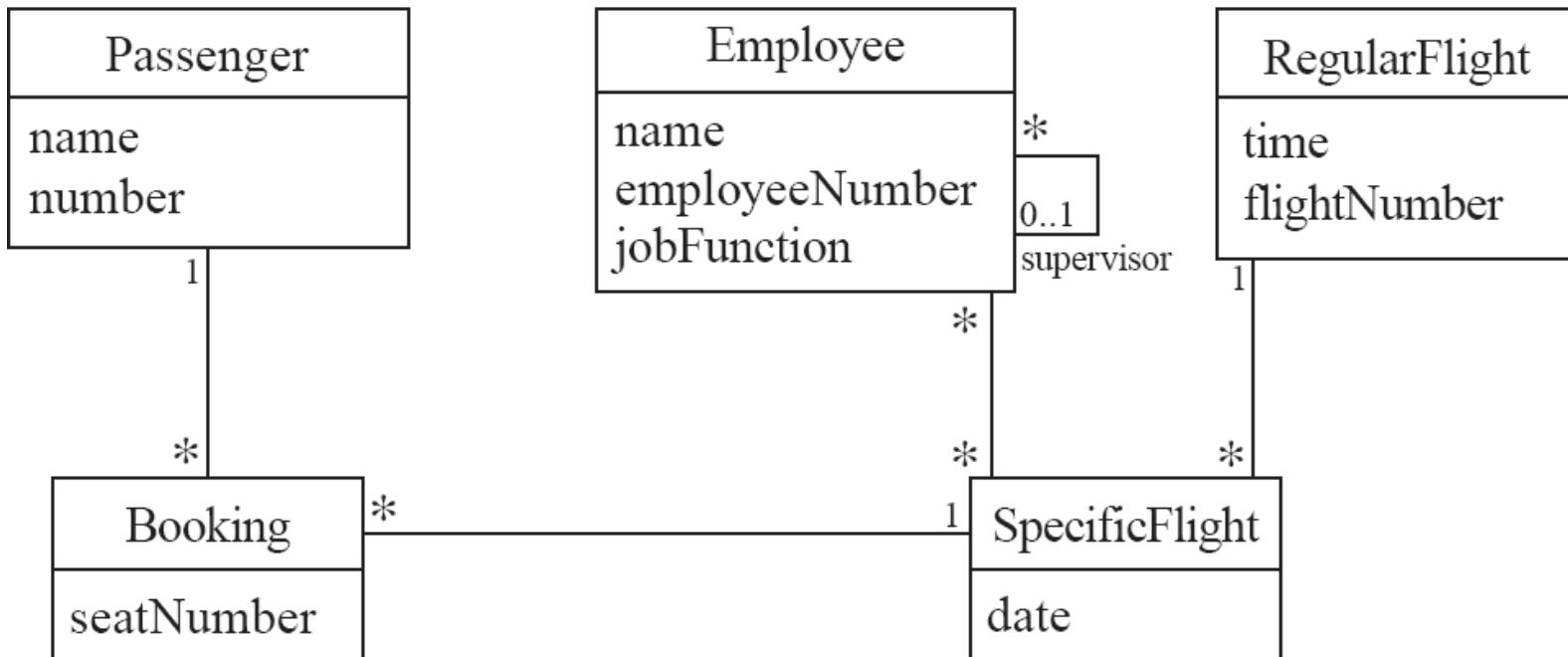


Bad, due to too many  
attributes, and the  
inability to add more  
addresses

Good solution. The type indicates whether it  
is a home address, business address etc.

# An example (attributes and associations)

---

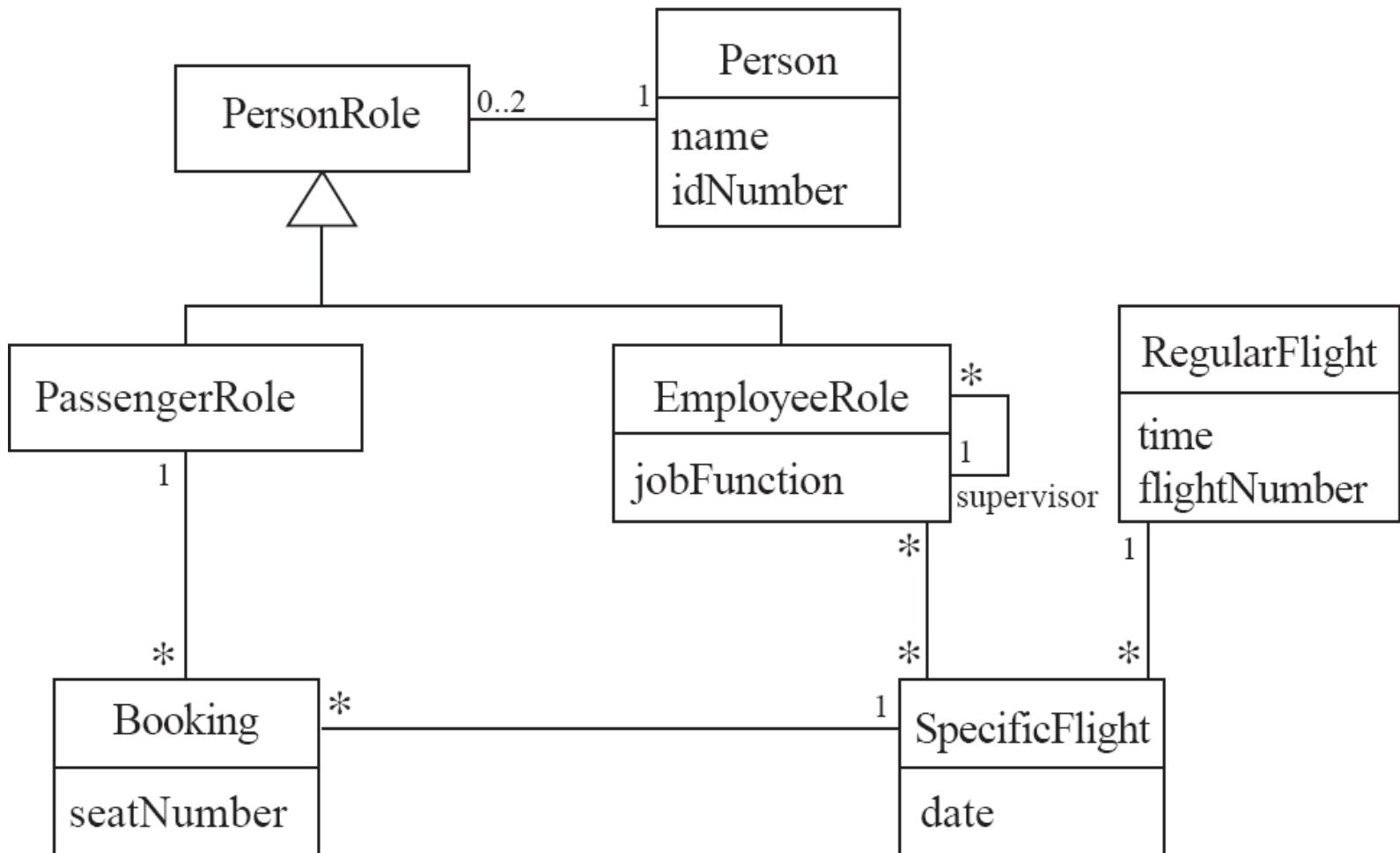


# Identifying generalizations and interfaces

---

- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
- Create an *interface*, instead of a superclass if
  - The classes are very dissimilar except for having a few operations in common
  - One or more of the classes already have their own superclasses
  - Different implementations of the same class might be available

# An example (generalization)

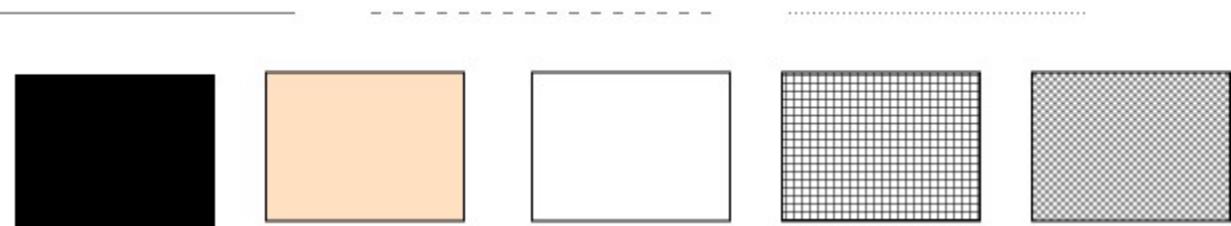


# Enumerations

---

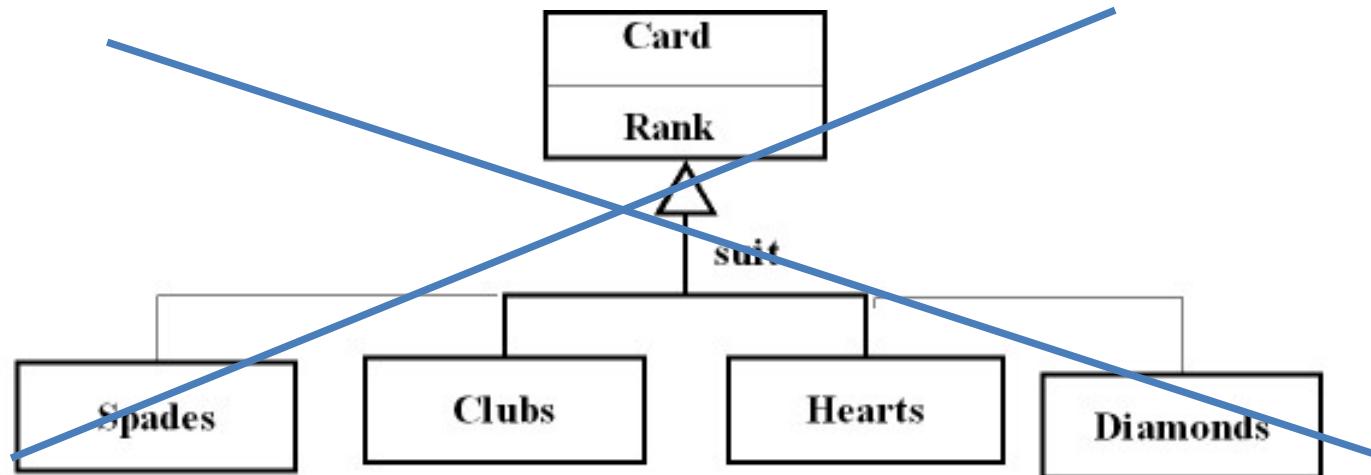
- A data type is a description of values.
- An enumeration is a data type that has a finite set of values.
  - For example, accessPermission (read, write, read-write), colors

**Figure.penType**



**TwoDimensional.fillStyle**

# Modeling Enumerations



Card  
suit: suit  
rank: rank

<<enumeration>>  
Suit  
Spades  
clubs  
hearts  
diamonds

<<enumeration>>  
Rank  
ace  
king  
queen  
.....

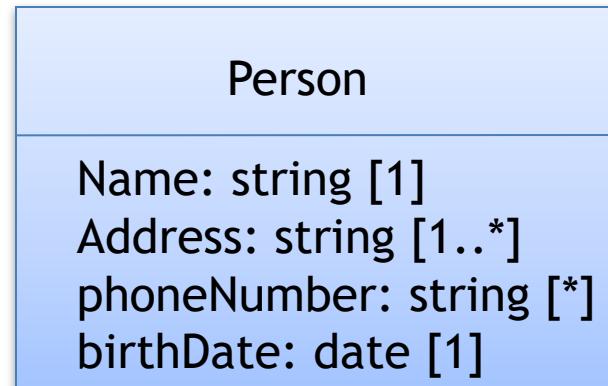
Do not use a generalization to capture the values of an enumerated attribute

# Multiplicity for an attribute

---

Specifies the number of possible values for each instantiation of an attribute.

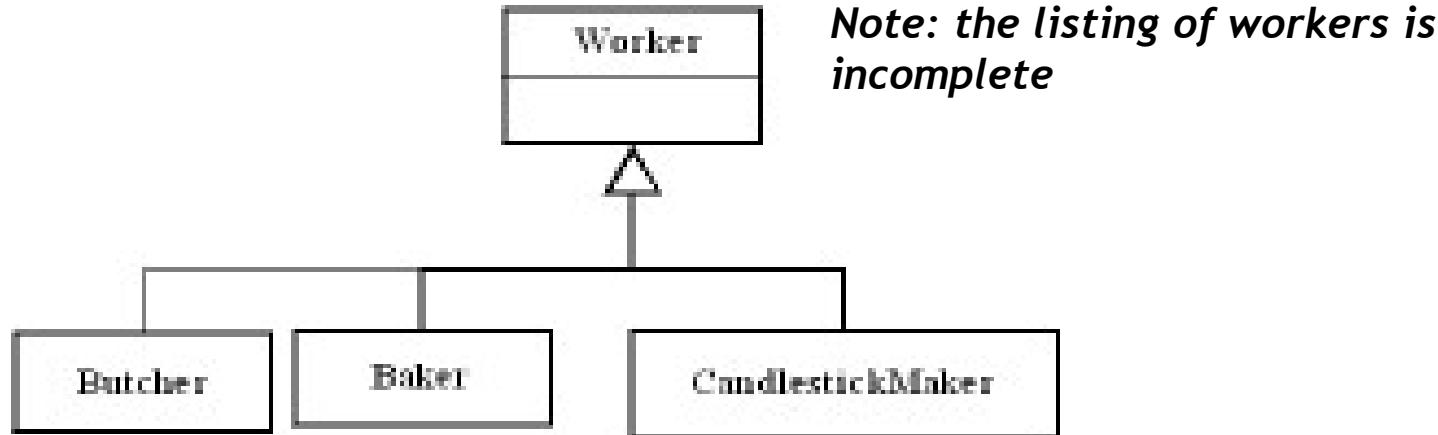
- single value [1]
- optional single value [0..1]
- many [\*]



# Abstract Class

---

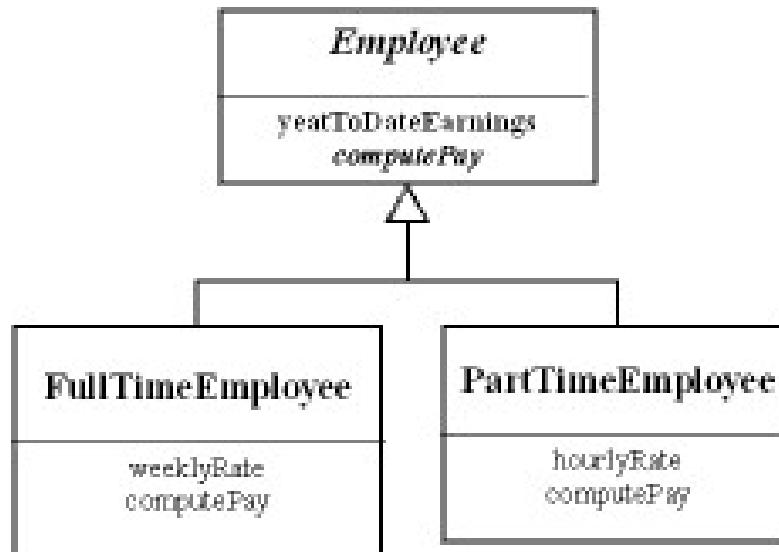
An abstract class is a class that has no direct instances but whose descendant classes have direct instances.



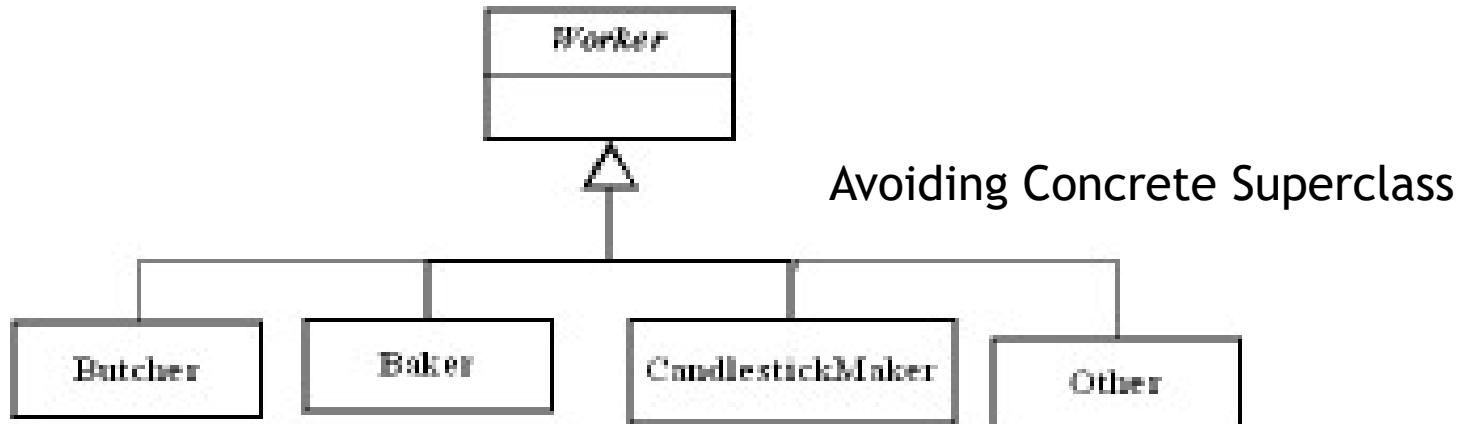
A concrete class is a class that is instantiable; it can have direct instances.

---

# Abstract Class



Abstract Class  
&  
Abstract Operation

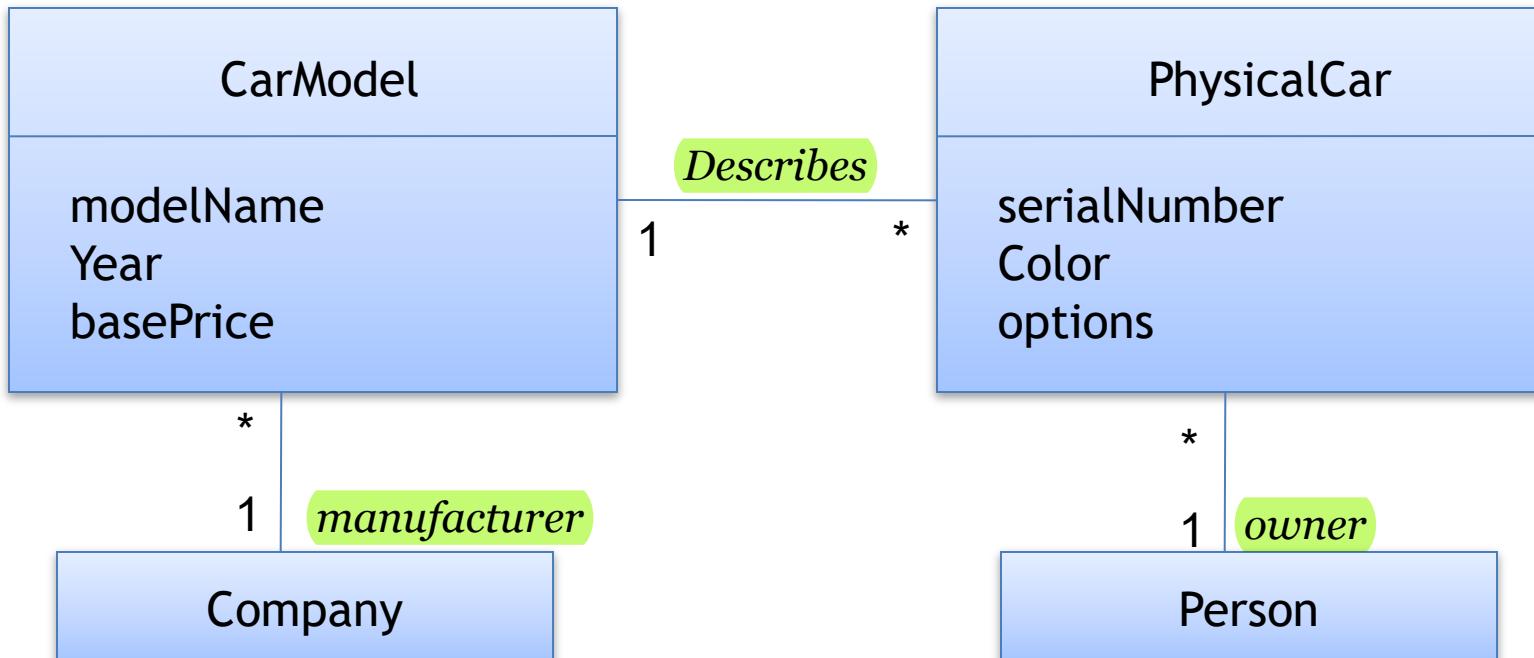


Avoiding Concrete Superclass

# Metadata

Metadata is data that describes other data

- A Class Definition is Metadata.
- Models are inherently metadata, describe the things being modeled
- Programming Language implementations also use metadata.



# Reification

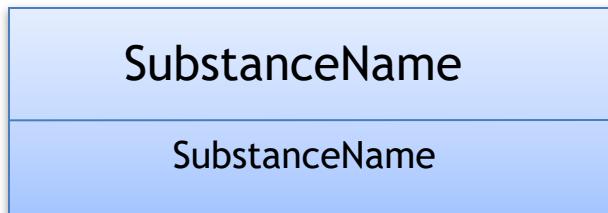
---

Reification is the promotion of something that is not an object into an object.

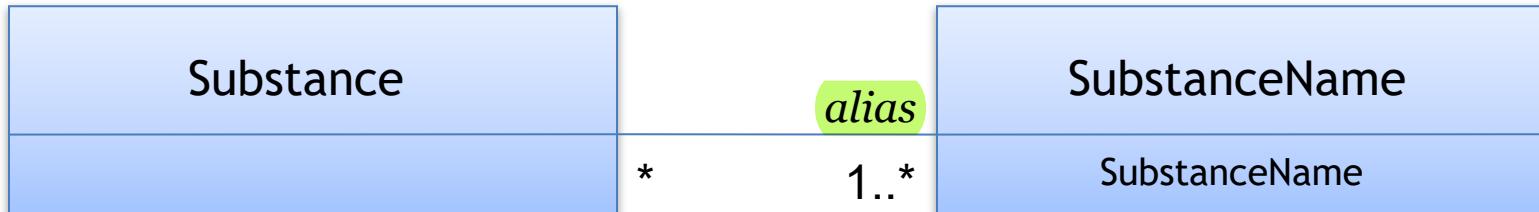
- It lets you to shift the level of abstraction
- For example, Consider a Database Manager. A developer could write code for each application to read and write files.

*Writing for all other applications????*

*Reify: Data services and use DATA Manager.*



**Reification: Promote attribute to a Class**



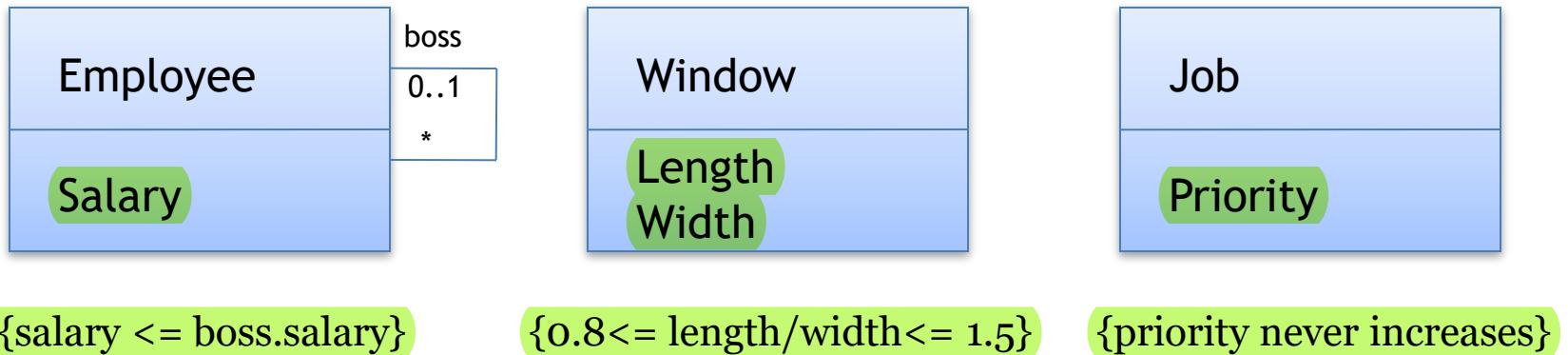
# Constraints

---

A **Constraint** is a Boolean condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets.

- It restricts the values that elements can assume.
- Natural Language or a formal language (**OCL; Object Constraint Language**)

## Constraints on Objects



# Constraints

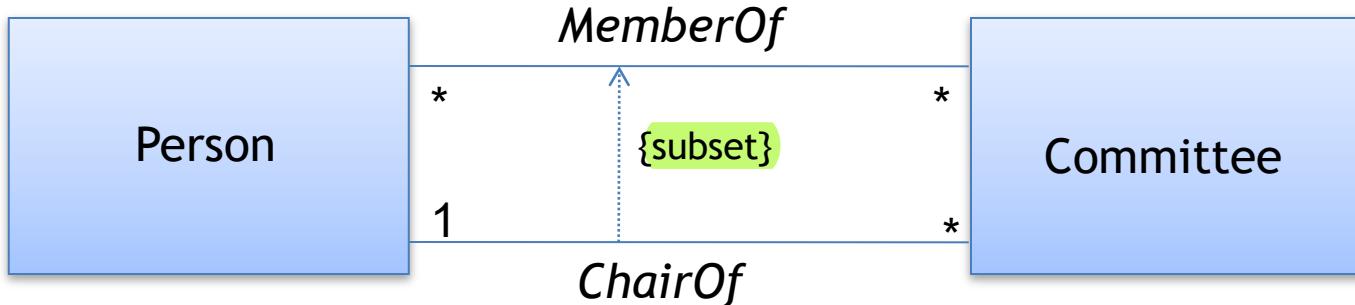
---

## Constraints on Links

Multiplicities, association names, association end names, qualification, {ordered}, {sequence} and many more....

Explicit constraint:

*The chair of a committee is a member of the committee; the ChairOf association is a subset of the MemberOf association.*



Subset constraints between associations

---



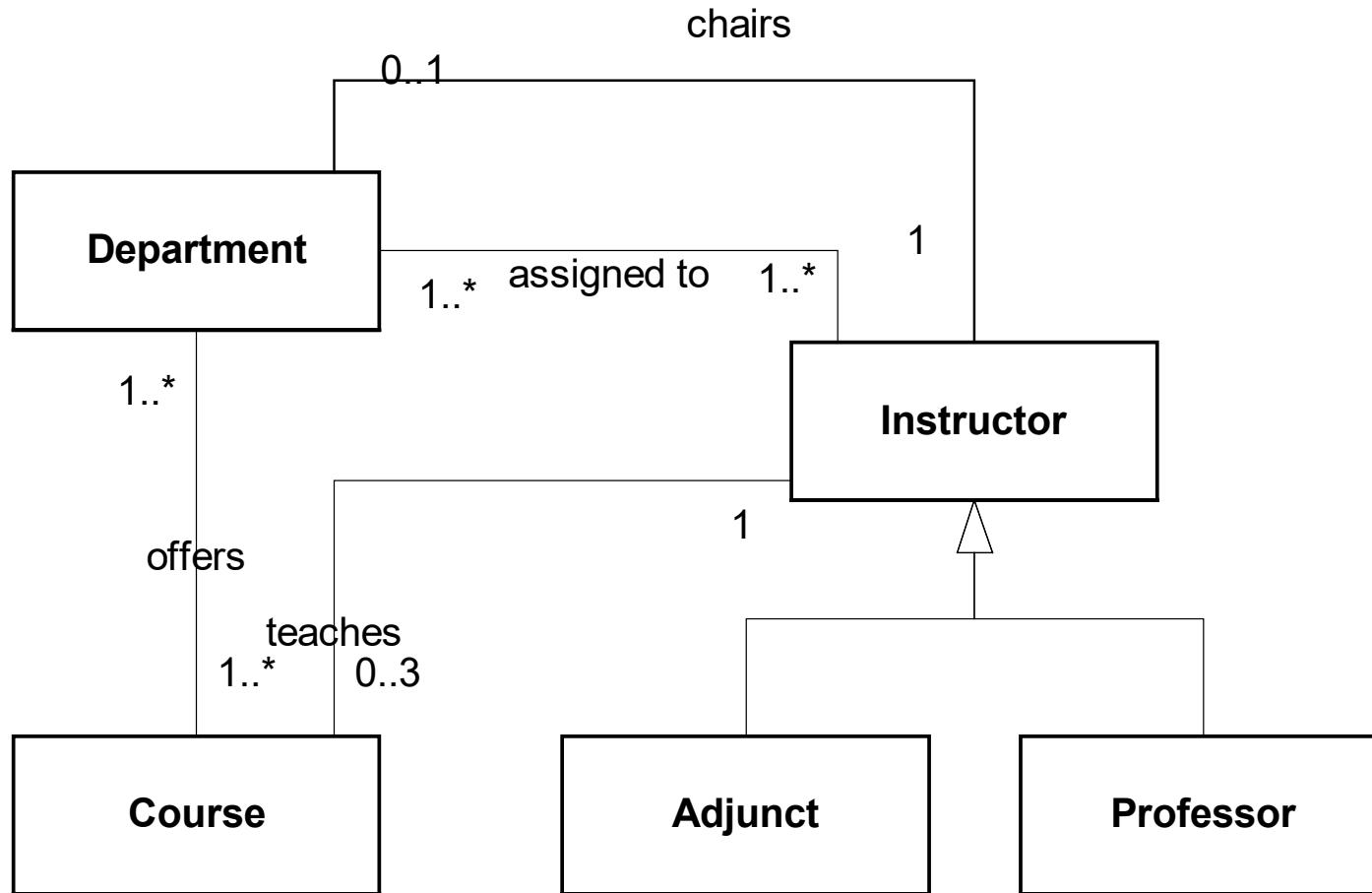
## Example 1: University Courses

---

- Some instructors are professors, while others have job title adjunct
  - Departments offer many courses, but a course may be offered by >1 department
  - Courses are taught by instructors, who may teach up to three courses
  - Instructors are assigned to one (or more) departments
  - One instructor also serves a department chair
-

# Class Diagram for Univ. Courses

---



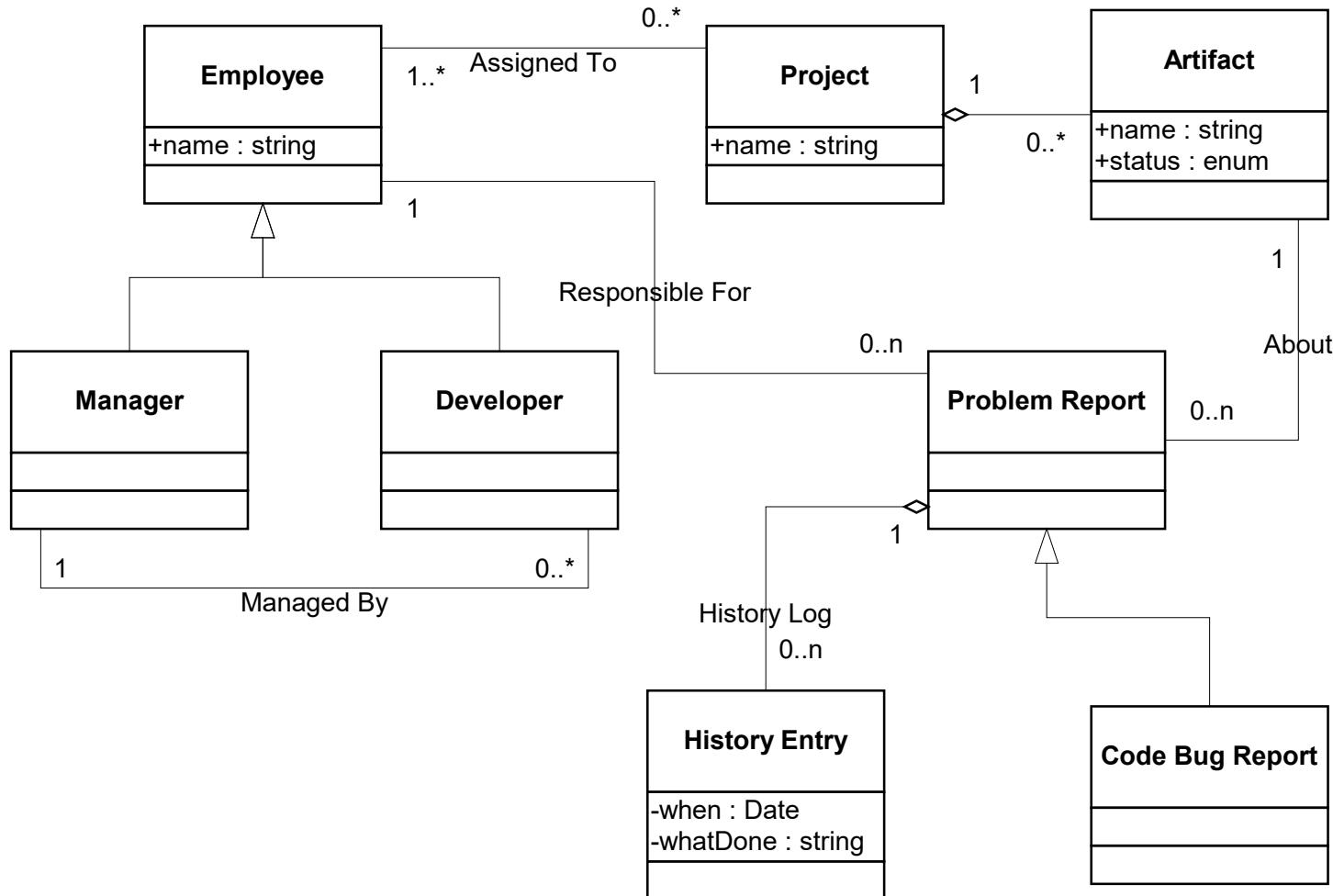


## Example 2: Problem Report Tool

---

- A CASE tool for storing and tracking problem reports
    - Each report contains a problem description and a status
    - Each problem can be assigned to someone
    - Problem reports are made on one of the “artifacts” of a project
    - Employees are assigned to a project
    - A manager may add new artifacts and assign problem reports to team members
-

# Class Diagram for Prob. Rep. Tool





---

**Questions??**

---



# IT 314: Software Engineering

*Object Relationships: Association*



# Object Relationships

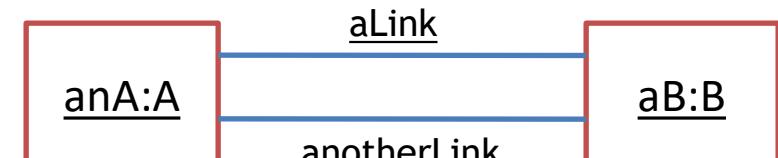
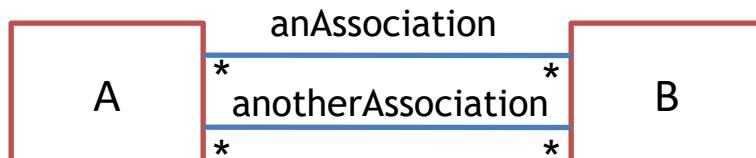
---

- Association
  - Aggregation
  - Composition
- Inheritance
- Dependency

# Link & Association

---

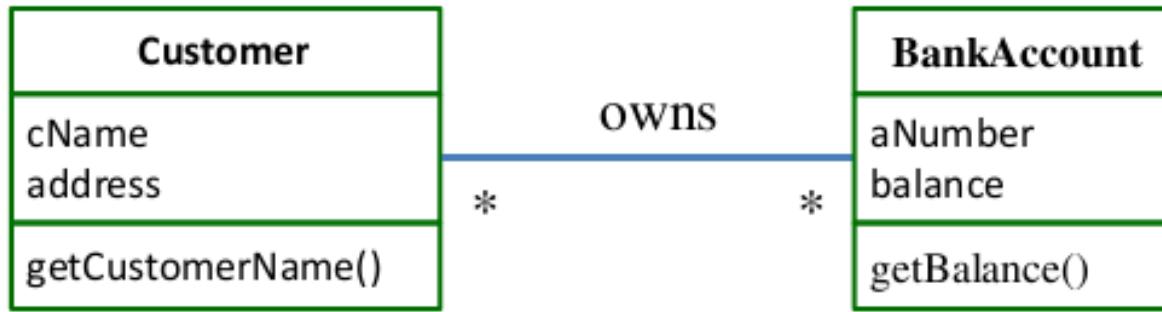
- Link is a physical or conceptual connection among objects
- Association is a description of a group of links with common structure and common semantics



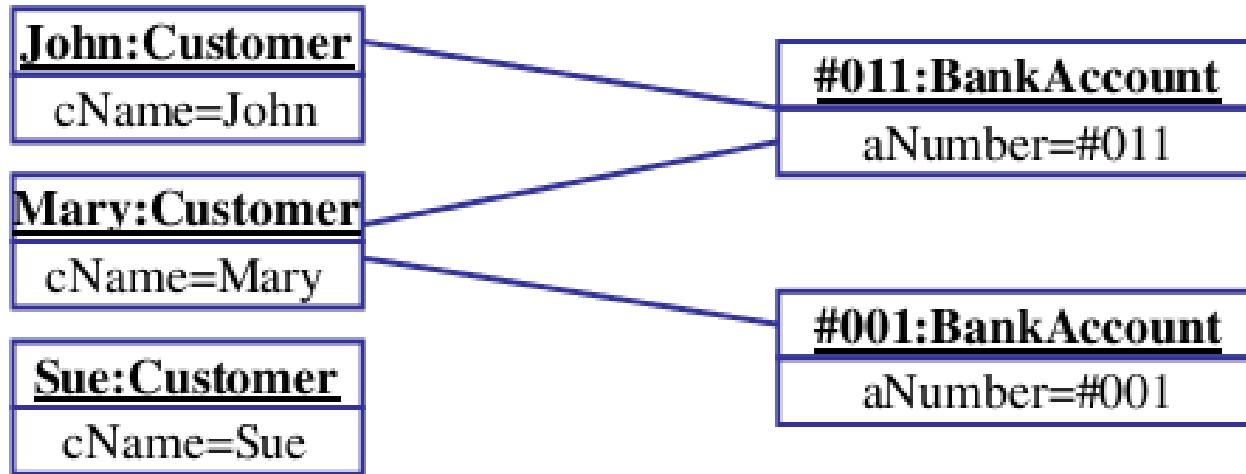
*Class Diagram*

*Object Diagram*

# Association



Class Diagram



Object Diagram

# Multiplicity

---

Multiplicity specifies “the number of instances of one class that may relate to a single instance of an associated class”.

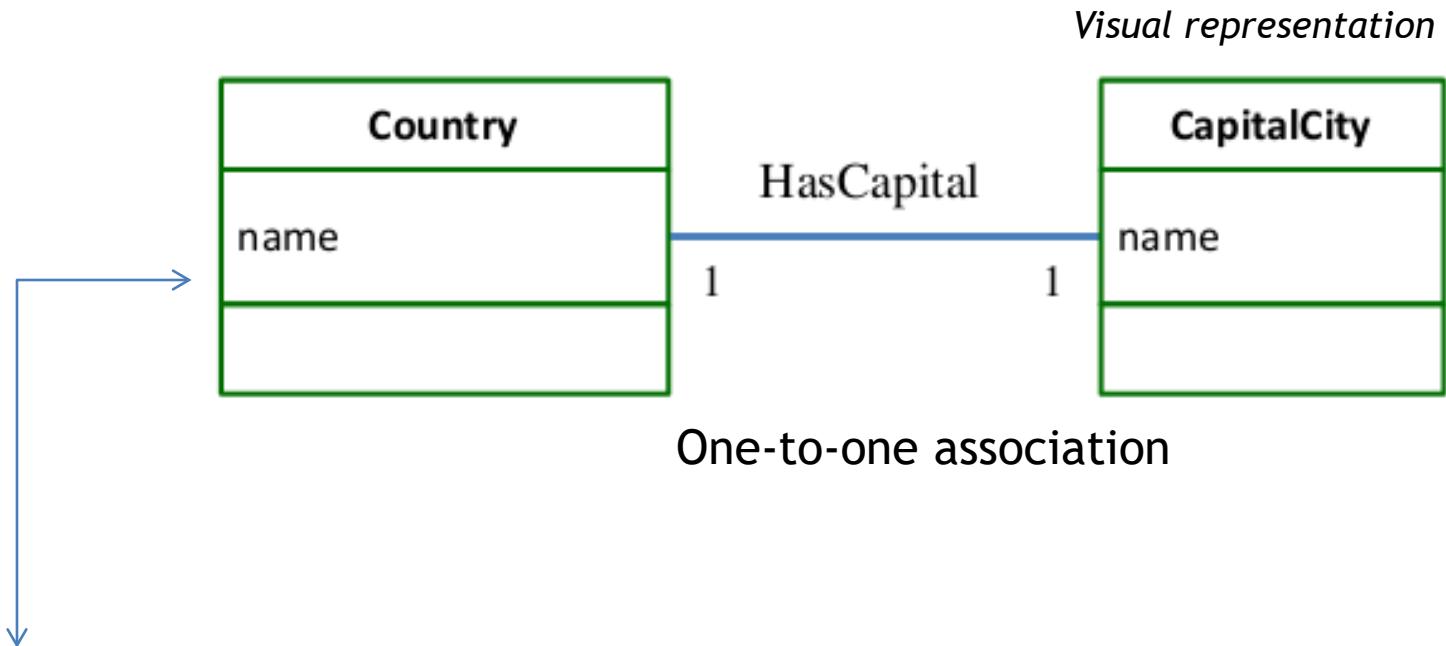
- “One” or “Many”
- *Infinite (subset of the nonnegative integers)*

UML Specifies multiplicity with an interval

- “1” (*exactly one*)
- “1...\*” (*one or more*)
- “3..5” (*three to five, inclusive*)
- “\*” (*zero or more*)

# Association

---

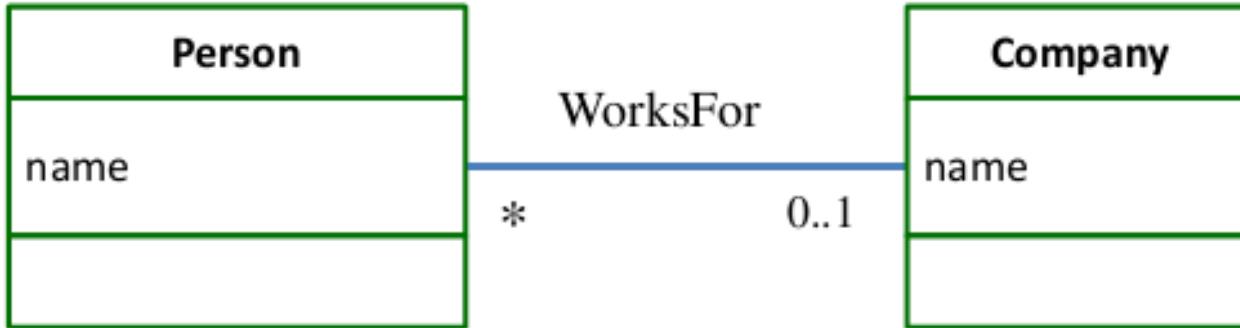


**Each country has one capital city**

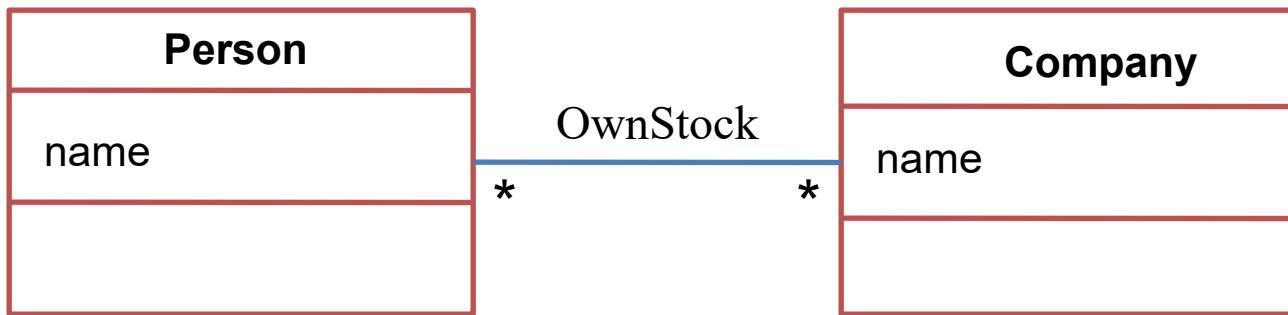
*Textual representation*

# Association

---

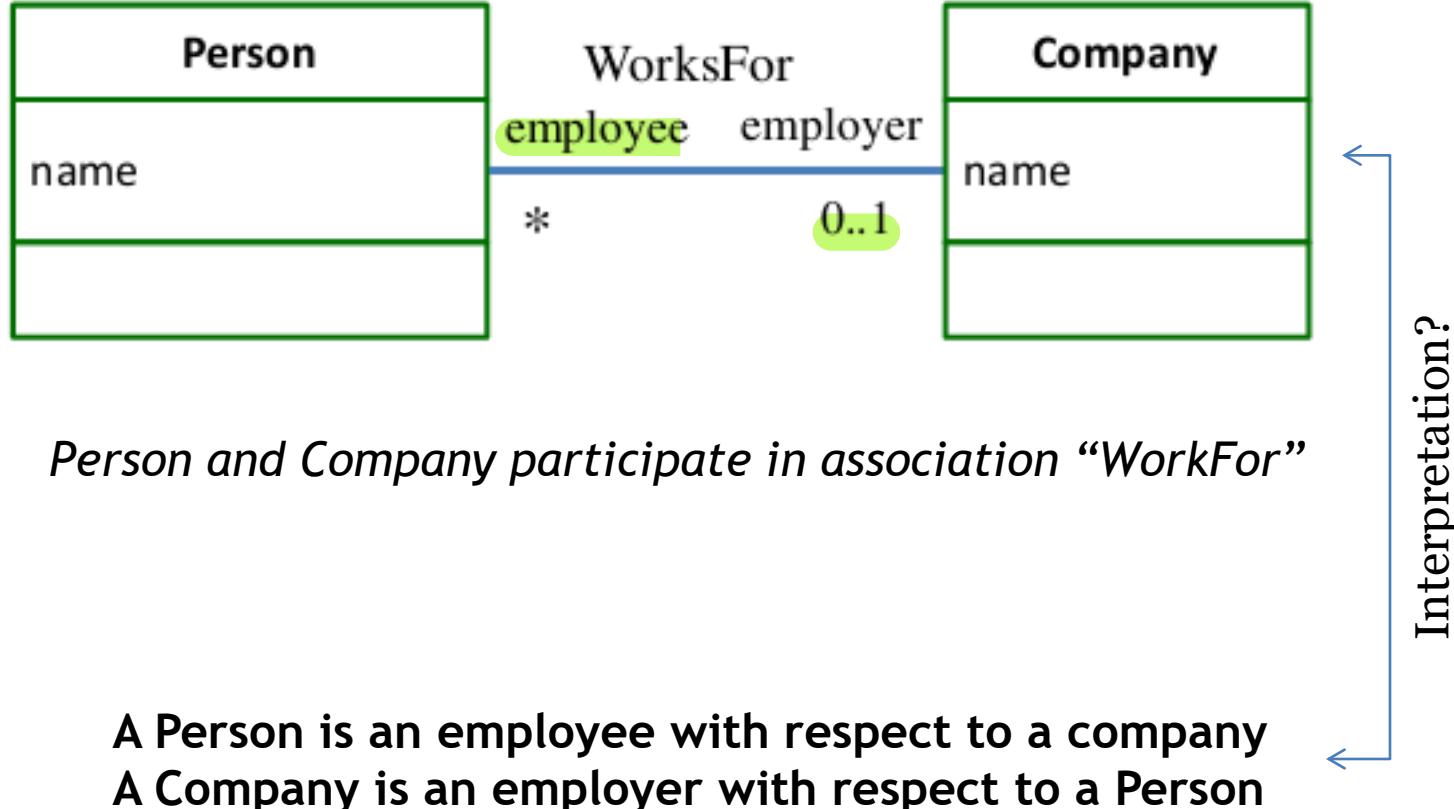


Many-to-one association



Many-to-many association

# Association End Names



# Association End Names

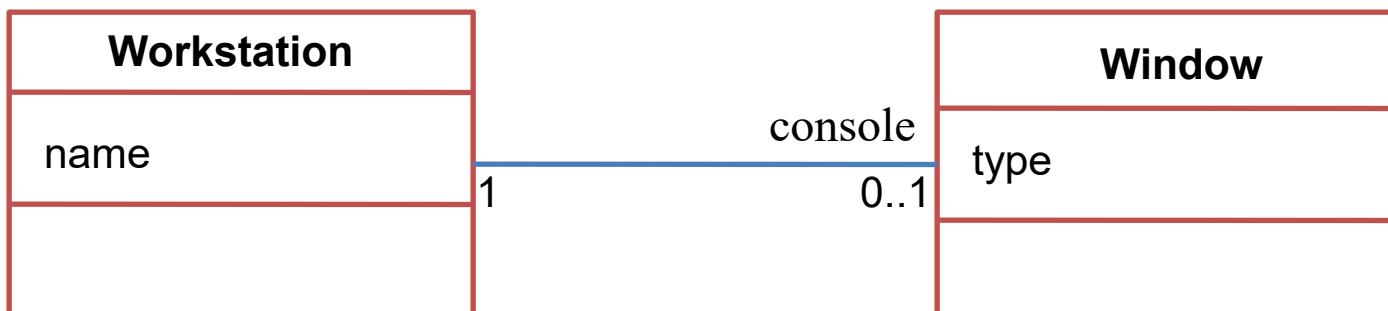
---

A Workstation may have one of its window designated as the console to receive general error messages.

*Interpretation??*



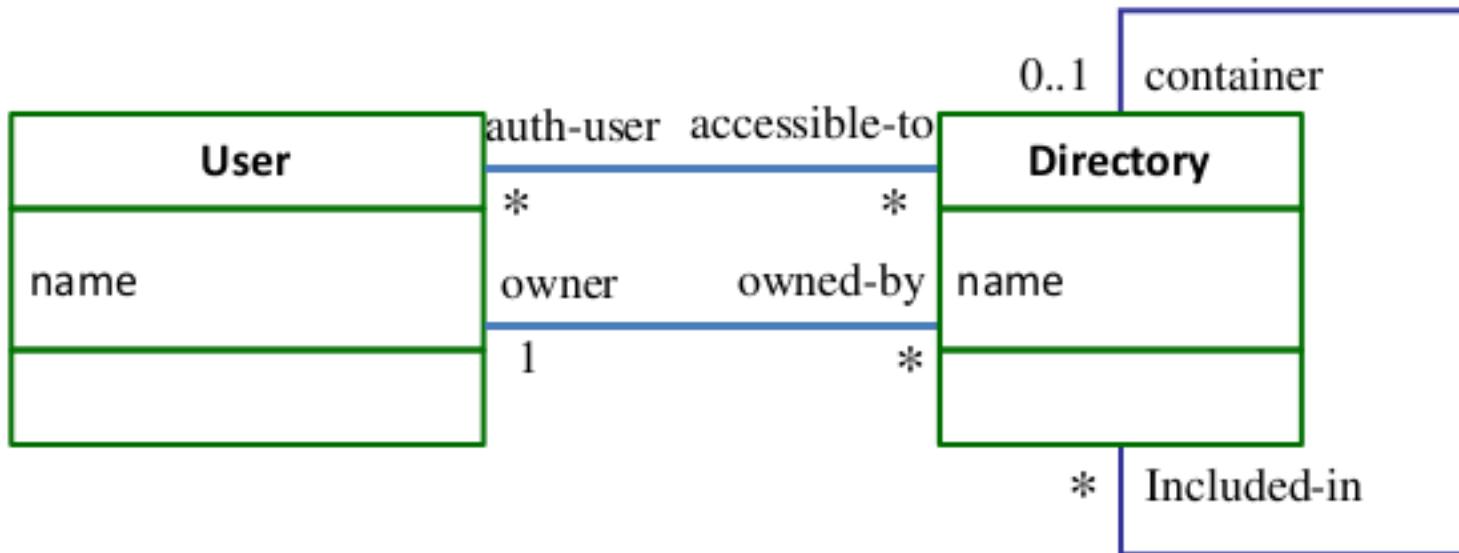
*Is it possible that no console window exists?*



# Association End Names

---

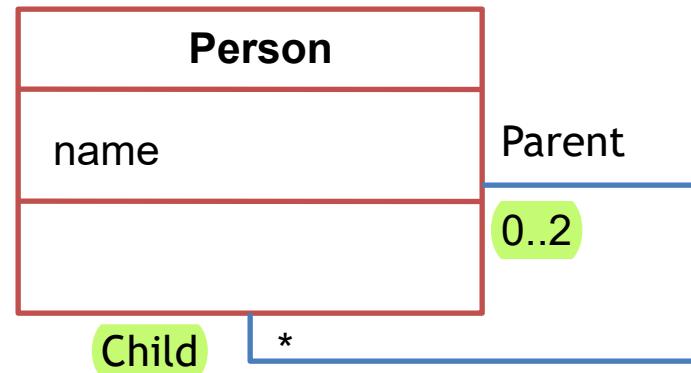
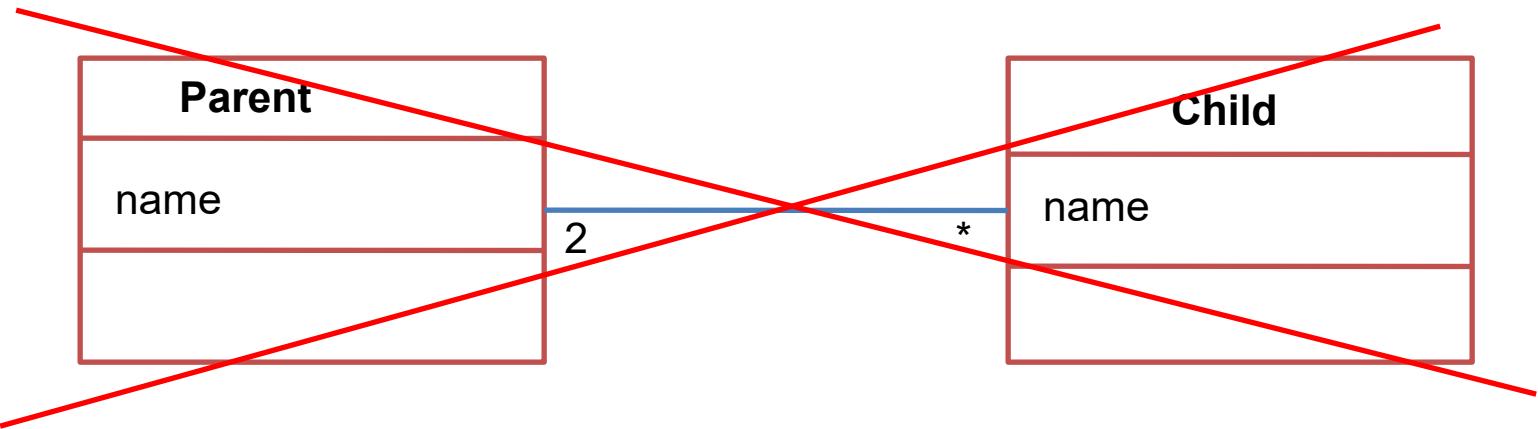
Association End Names are necessary for associations between two objects of a same class



# Question?

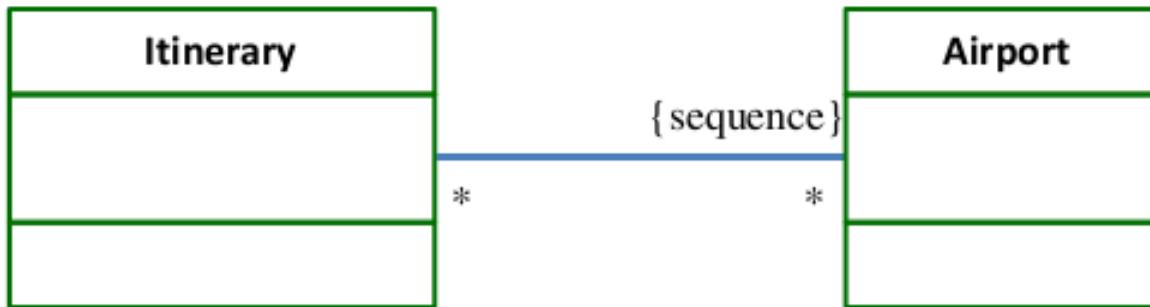
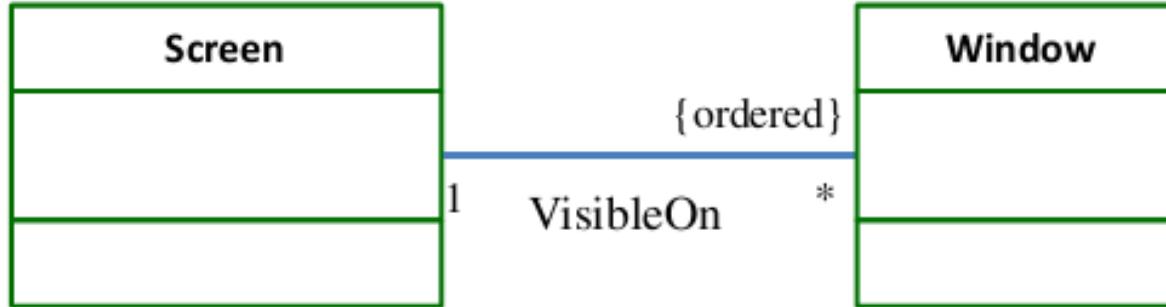
---

How to model parent-child relationship?



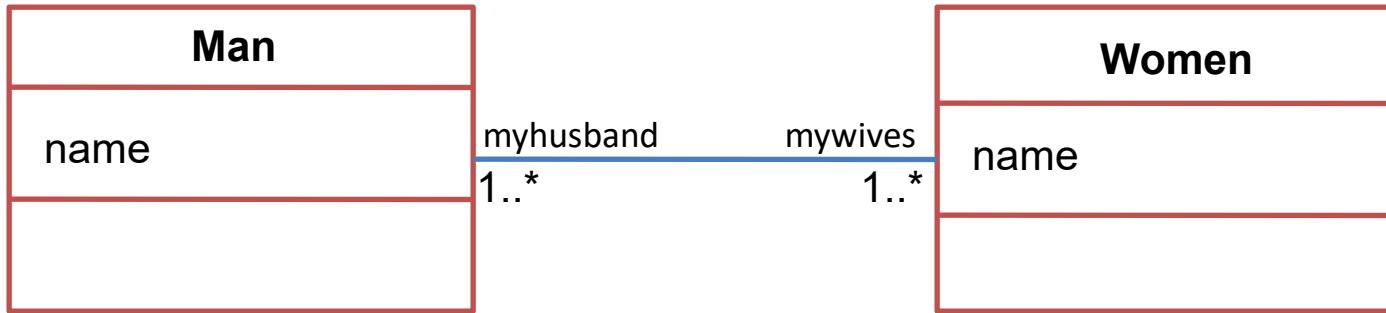
# Association: Ordered, Sequence, Bag

---



# Question?

---



{.....}

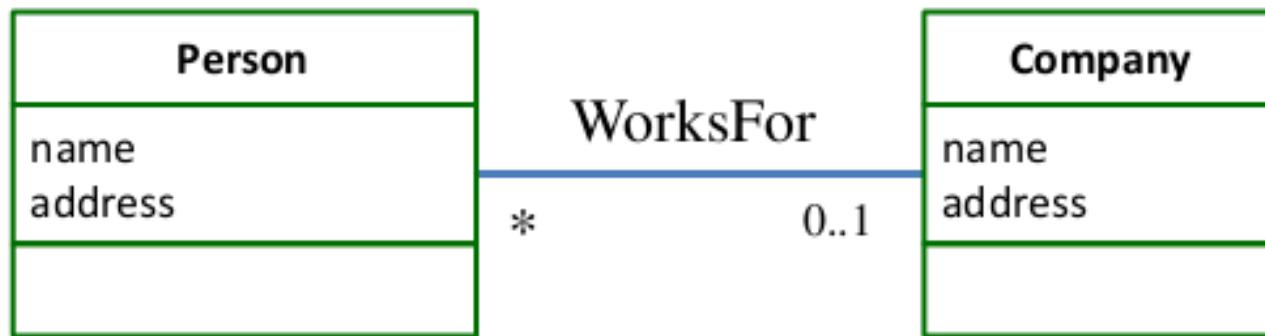
*Ordered  
Sequence*

# Association Class

---

UML offers the ability to describe links of association with attributes like any class.

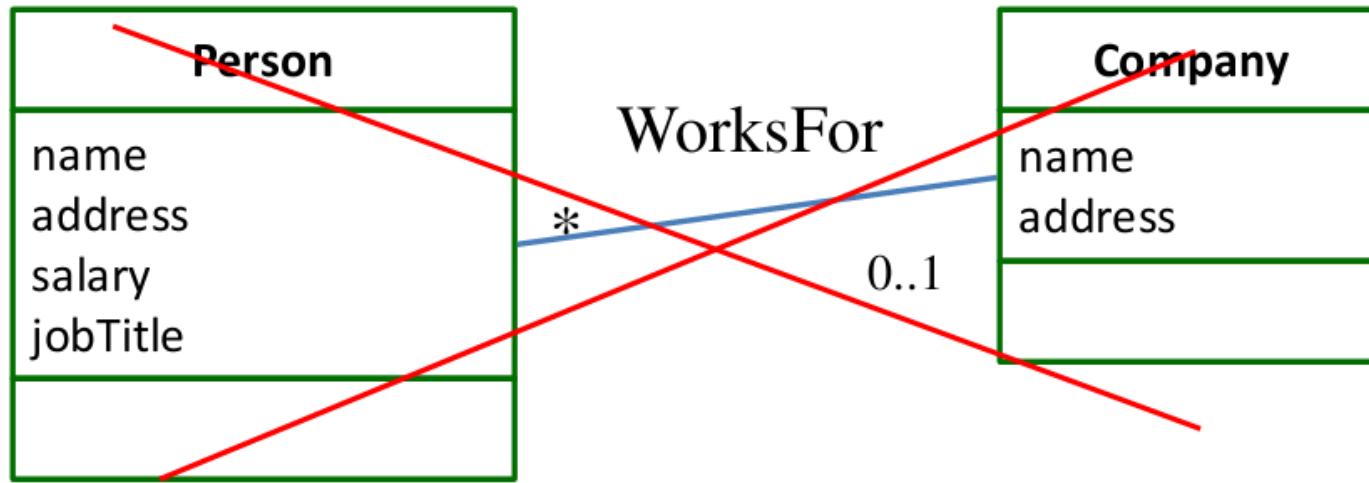
An association class is an association that is also a class.



Salary ?  
Job Title ?

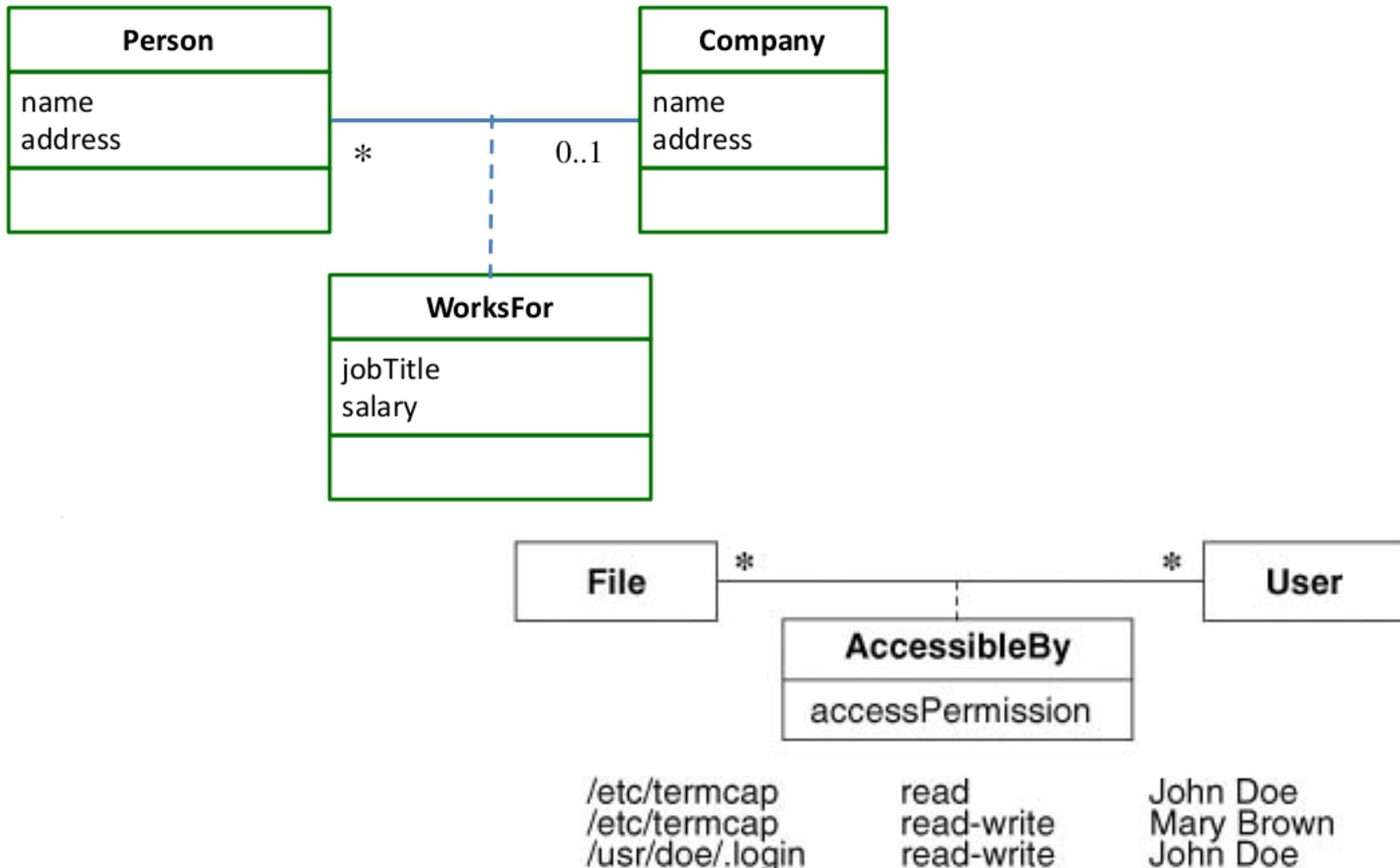
# Association Class

---



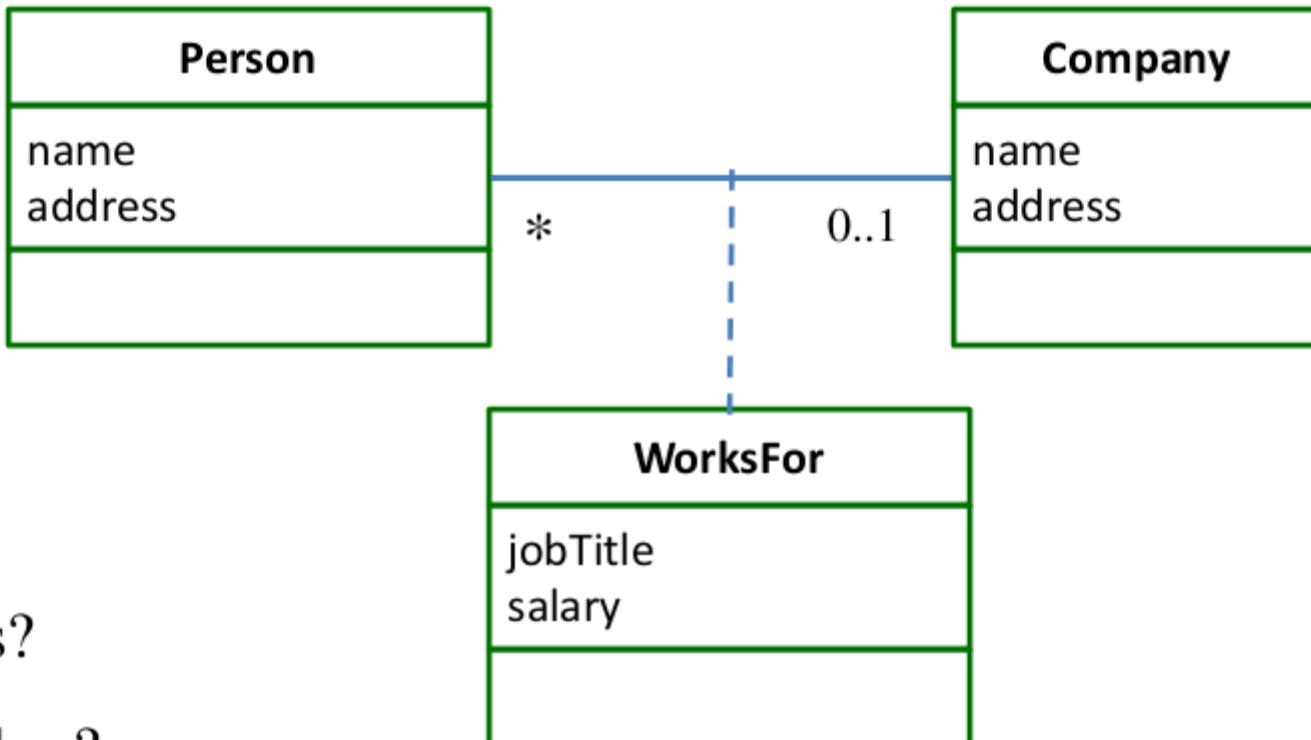
Salary ?  
Job Title ?

# Association Class



# Association Class

---

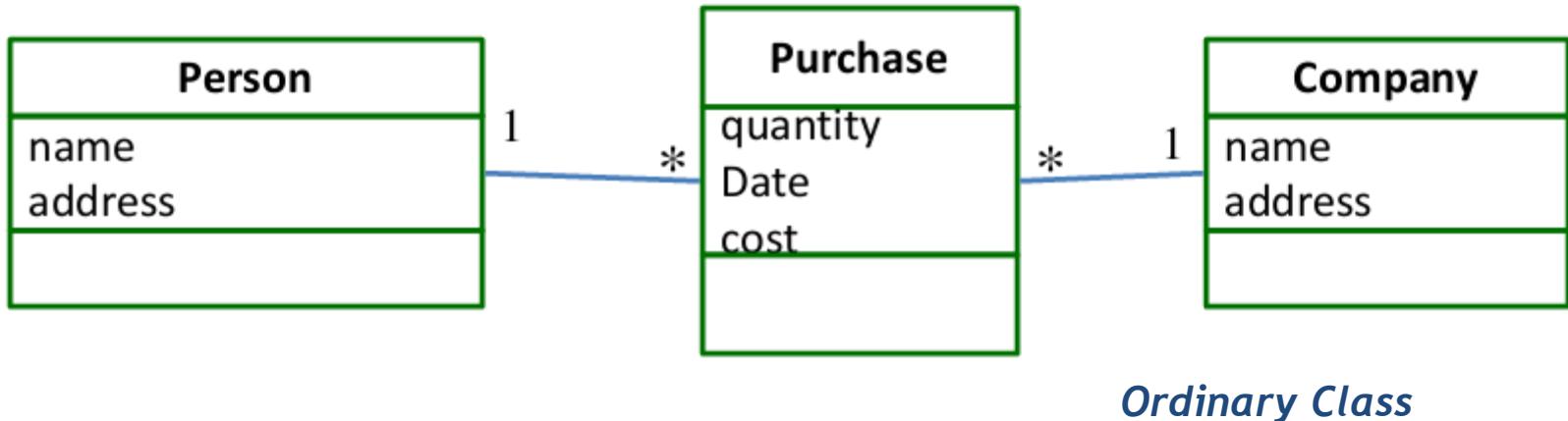
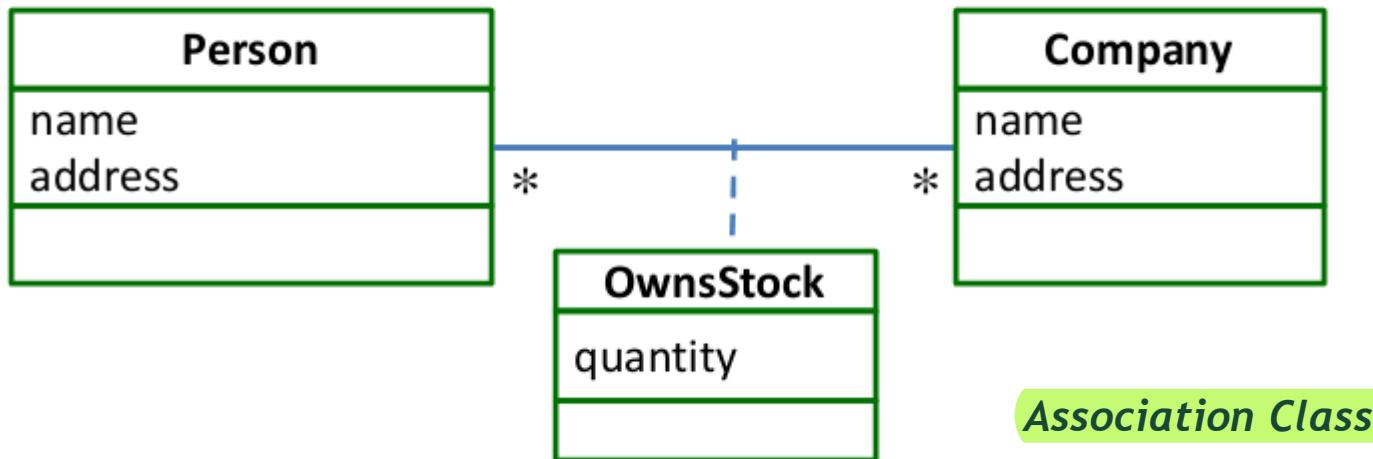


Boss?

Worker?

performanceRating?

# Association Class vs Ordinary Class



*Ordinary Class*



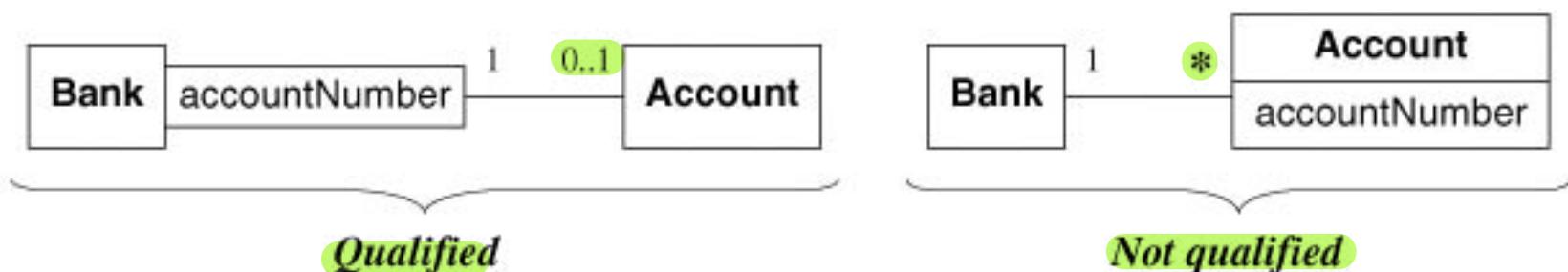
# Question?

---

Users may be authorized on many **workstations**. Each **authorization** carries a *priority and access privileges*. A user has a *home directory* for each authorized workstation, but several workstations and users can share the same home directory.

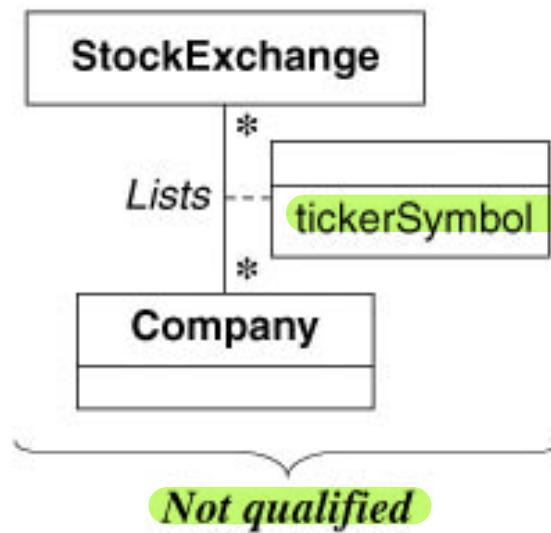
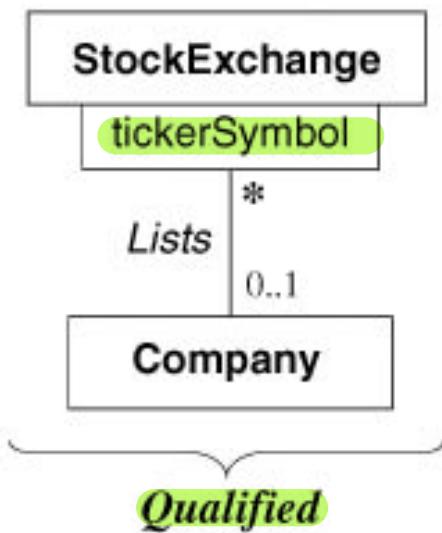
# Qualified Association

- A qualified association is an association in which an attribute called Qualifier disambiguates the objects for a ‘many’ association end.
- A qualifier selects among the target objects, reducing the effective multiplicity for ‘many’ to ‘one’.
- Both below models are acceptable but the qualified model adds information.



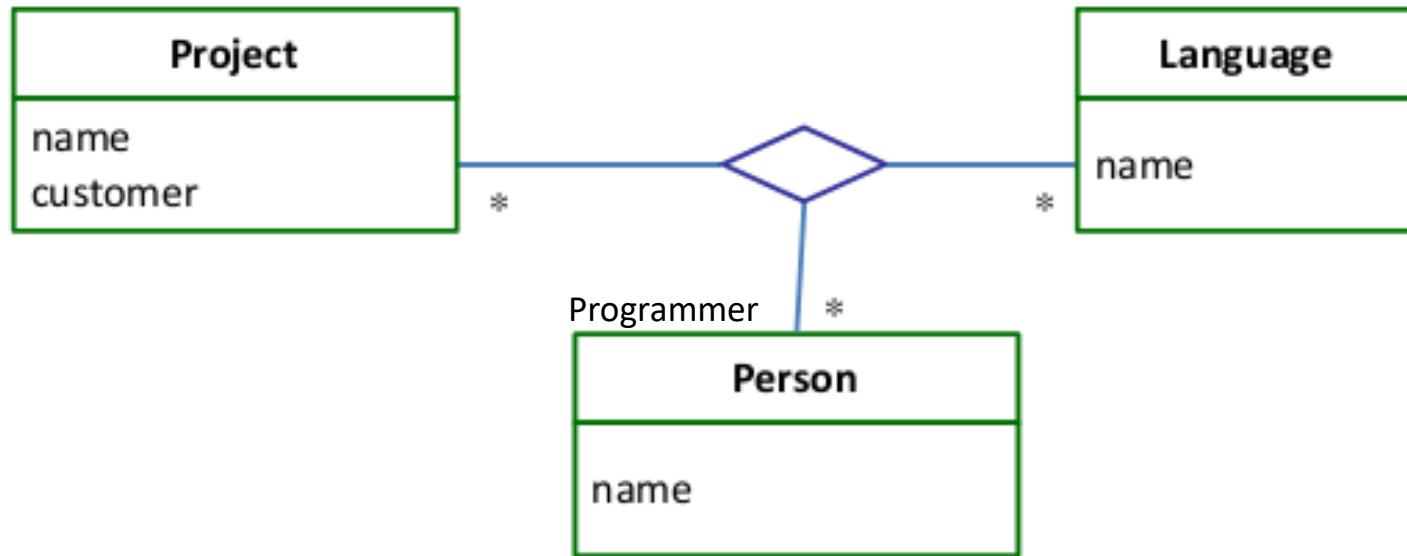
# Qualified Association

---



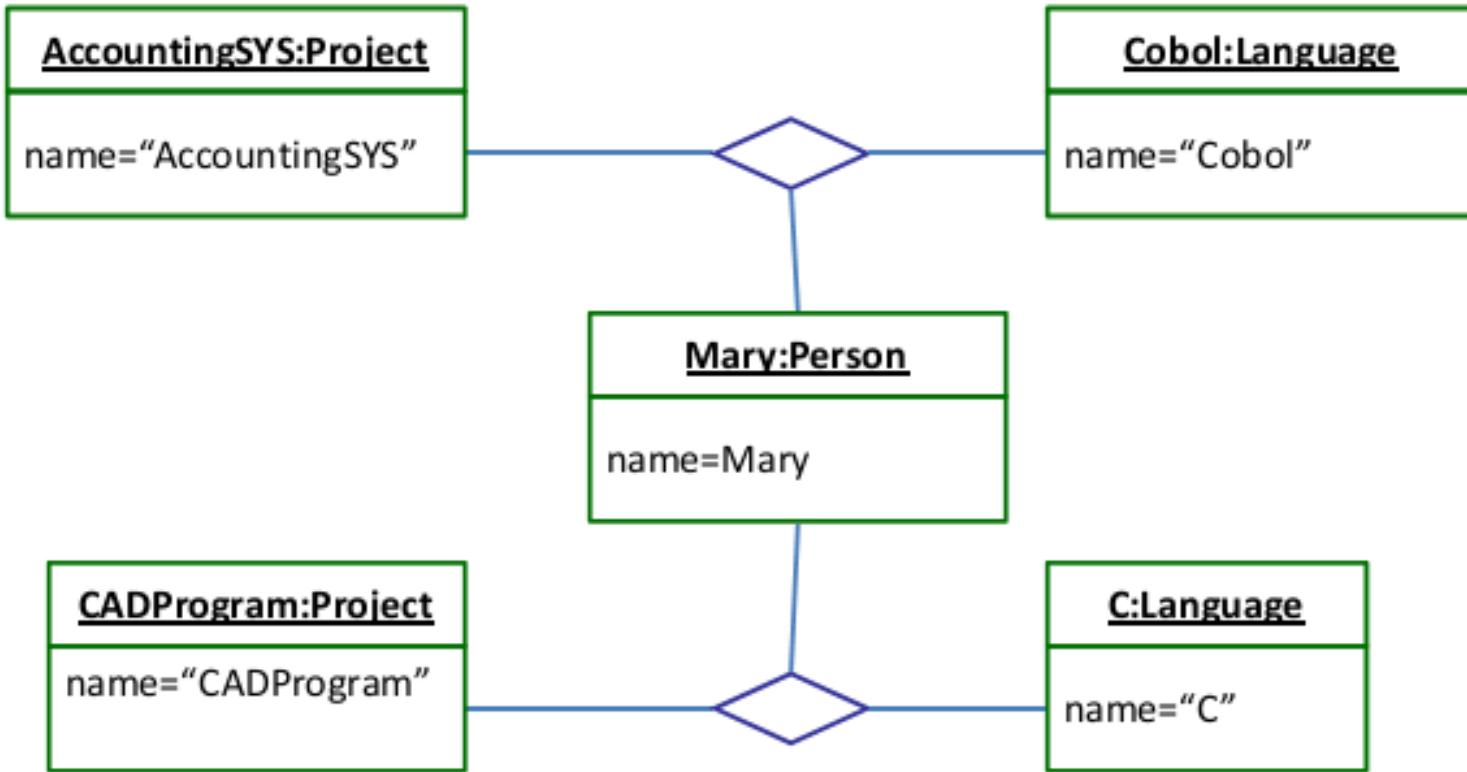
# N-ary Association

---



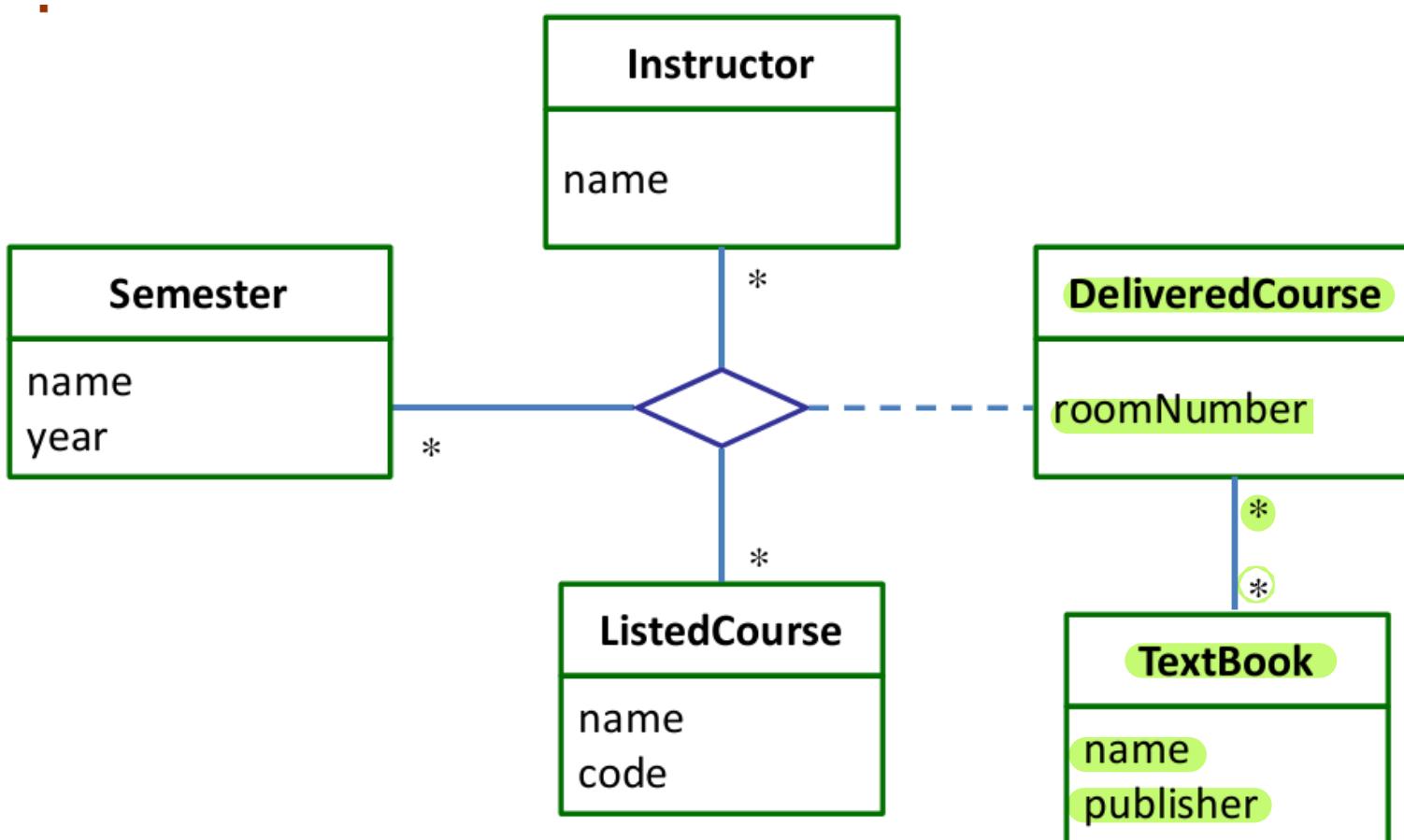
*Class Diagram*

# N-ary Association



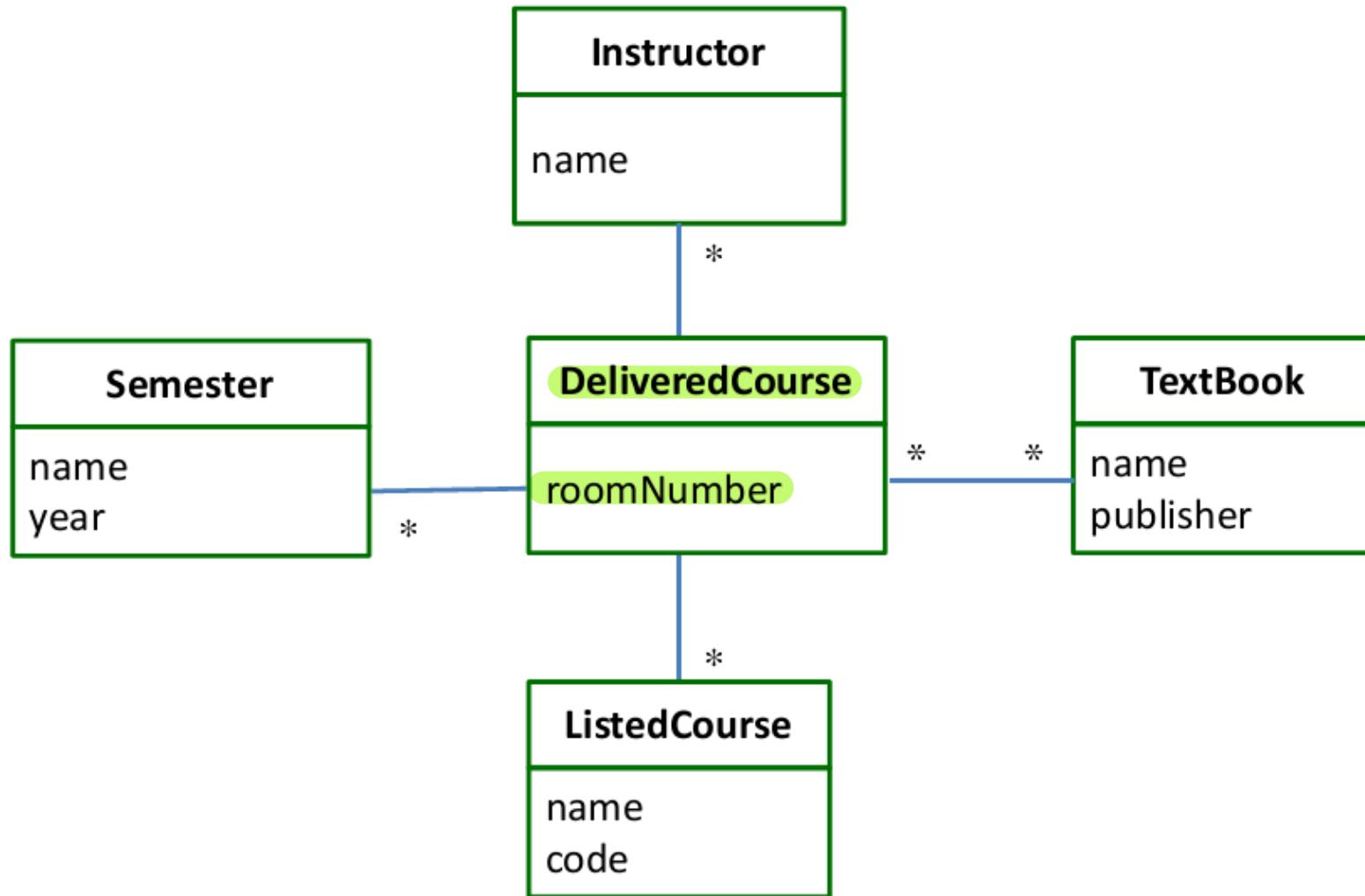
*Instance Diagram*

# N-ary Association Class



# N-ary Association Class

---



# Summary

---

- Objects may have relationships (among the same type or different types)
- Loosely coupled (relatively independent) objects are related with Association
- Association is shown by (optional) association name and association end names
- Multiplicity represents the cardinality of the relationship stating one object of a class is associated with how many number of objects of the other class



---

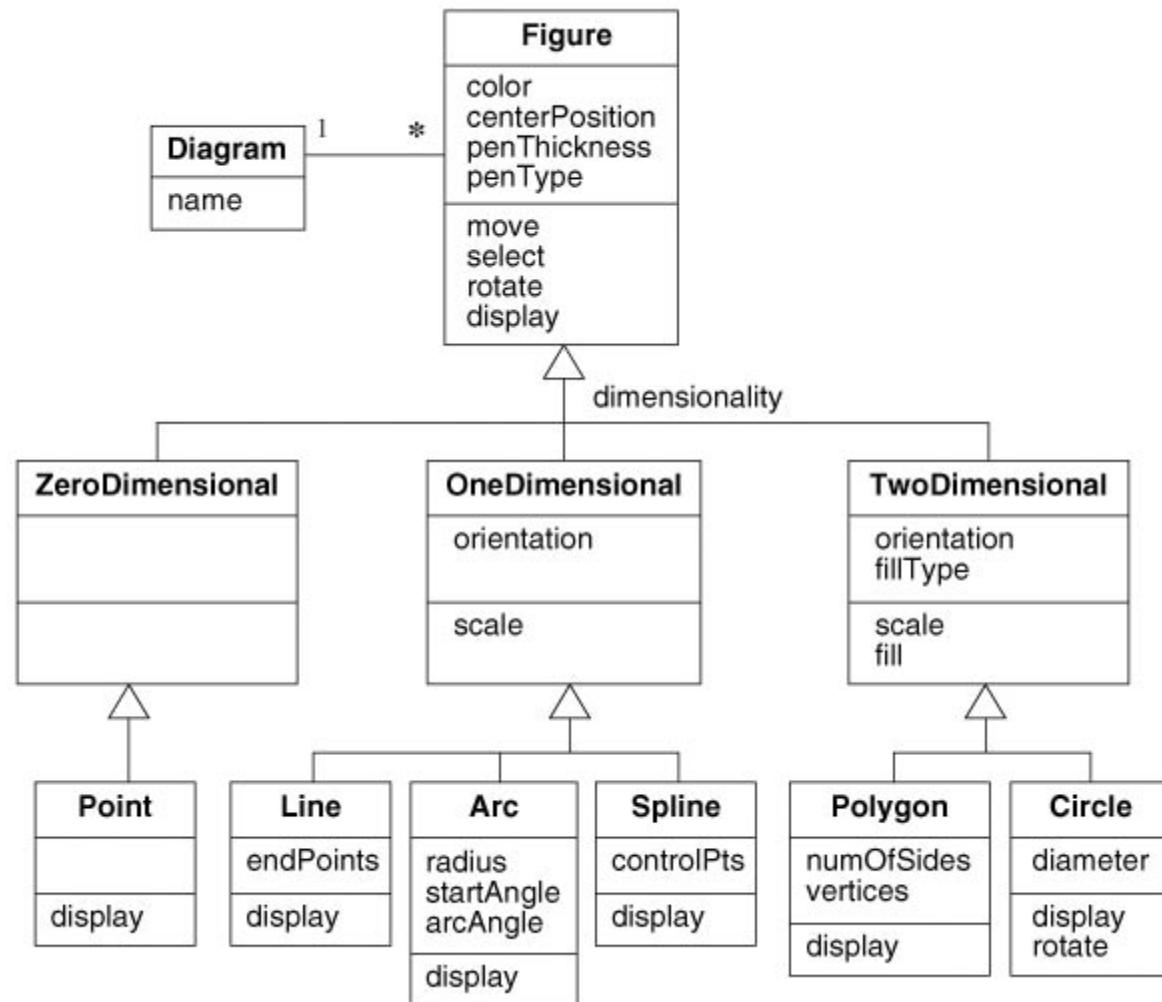
Next Lecture...

## **Object Relationships: Association (Aggregation, Composition) & Inheritance, Dependency**

# Generalization/Inheritance

---

- Generalization is the relationship between a class (**superclass**) and one or more **variations** of the class (**subclasses**).
  - Generalization organizes classes by their **similarities** and their **differences**, structuring the descriptions of objects.
  - A superclass holds **common** attributes, attributes and associations.
  - The subclasses **adds specific** attributes, operations, and associations. They **inherit** the features of their superclass.
  - Often **Generalization** is called a “**IS A**” relationship
  - **Simple generalization** organizes classes into a **hierarchy**.
  - A subclass may **override** a superclass **feature** (attribute default values, operation) by **redefining a feature with the same name**.
  - Never override the signature of methods.
-



# Use of generalization

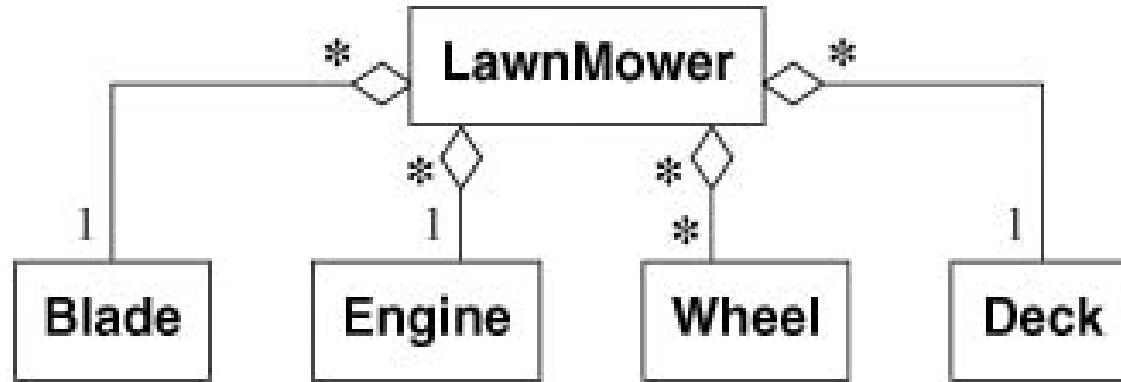
---

- Used for three purposes:
  - Support of polymorphism:
    - polymorphism increases the flexibility of software.
    - Adding a new subclass and automatically inheriting superclass behavior.
  - Structuring the description of objects:
    - Forming a taxonomy (classification), organizing objects according to their similarities. It is much more profound than modeling each class individually and in isolation of other similar classes.
  - Enabling code reuse:
    - Reuse is more productive than repeatedly writing code from scratch.

# Aggregation

---

- Aggregation is a **strong form of association** in which an **aggregate object** is made of **constituent parts**.
- The **aggregate** is semantically an extended object that is treated as a **UNIT** in many operations, although **physically it is made of several lesser objects**.
- Aggregation is a **transitive relation**:
  - if A is a part od B and B is a part of C then A is also a part of C
- Aggregation is an **antisymmetric relation**:
  - If A is a part of B then B is not a part of A.



# Aggregation versus Association

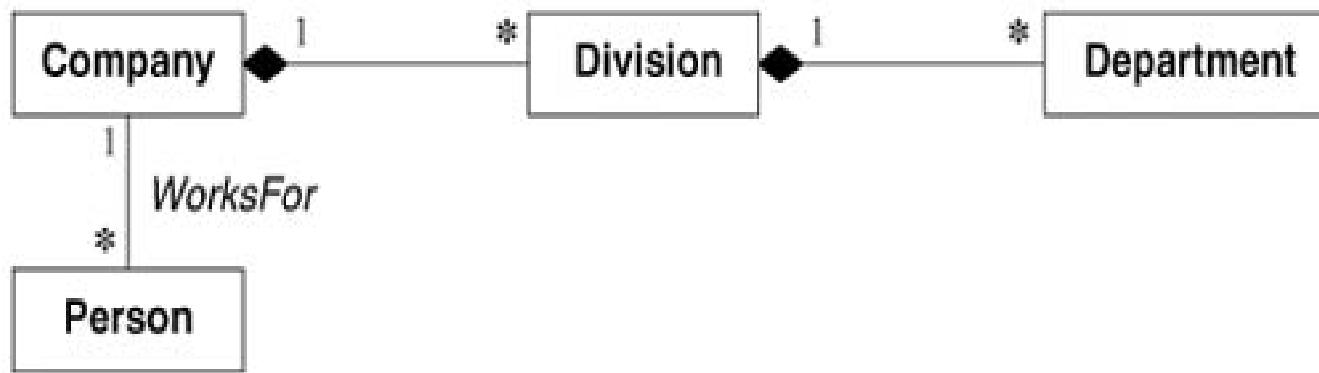
---

- Aggregation is a special form of association, not an independent concept.
- Aggregation adds semantic connotations:
  - If two objects are tightly bound by a part-whole relation it is an aggregation.
  - If the two objects are usually considered as independent, even though they may often be linked, it is an association.
- Discovering aggregation
  - Would you use the phrase **part of**?
  - Do some operations on the whole automatically apply to its parts?
  - Do some attributes values propagates from the whole to all or some parts?
  - Is there an asymmetry to the association, where one class is subordinate to the other?

# Aggregation versus Composition

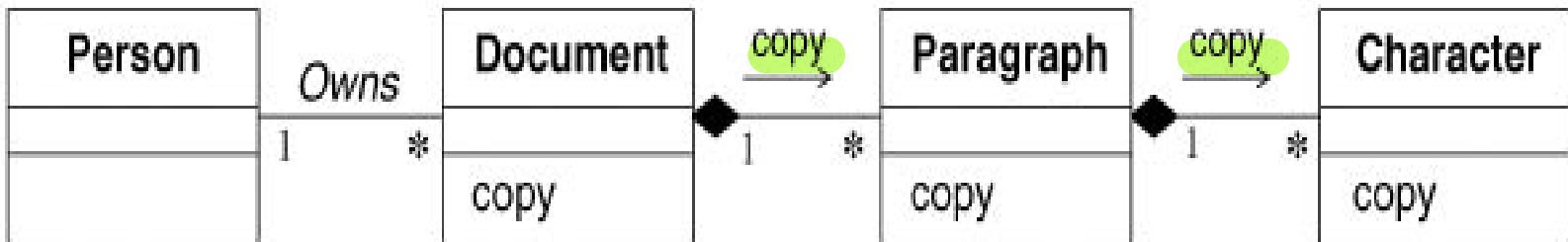
---

- **Composition** is a form of aggregation with additional constraints:
  - A constituent part can belong to **at most one** assembly (whole).
    - it has a **coincident lifetime** with the assembly.
    - Deletion of an assembly object triggers automatically a deletion of all constituent objects via composition.
  - Composition implies ownership of the parts by the whole.
    - Parts cannot be shared by different wholes.



# Propagation of operations

- Propagation is the automatic application of an operation to a network of objects when the operation is applied to some starting object.
- Propagation of operations to parts is often a good indicator of propagation.





---

# Questions??

---



# IT 314: Software Engineering

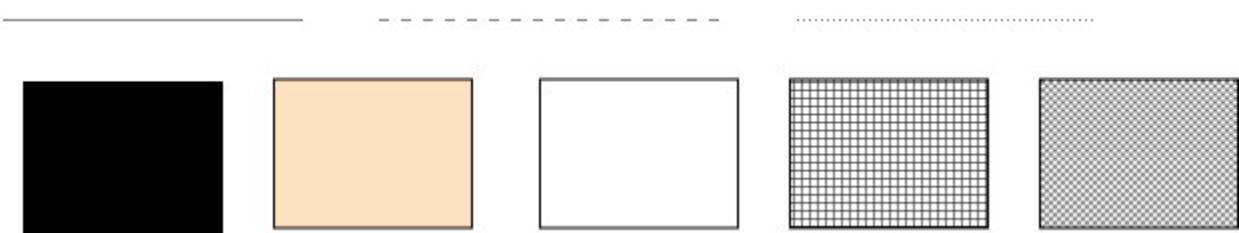
*Advanced Class Modeling  
Identification of Objects, Classes and their Relationships:  
Exercises*

# Enumerations

---

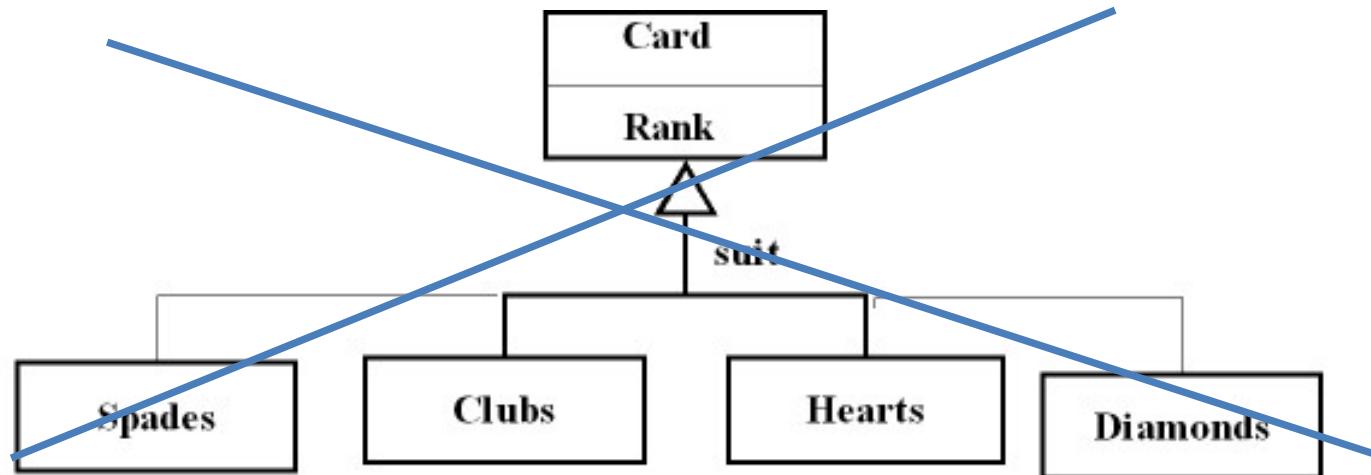
- A data type is a description of values.
- An enumeration is a data type that has a finite set of values.
  - For example, accessPermission (read, write, read-write), colors

**Figure.penType**



**TwoDimensional.fillStyle**

# Modeling Enumerations



Card  
suit: suit  
rank: rank

<<enumeration>>  
Suit  
Spades  
clubs  
hearts  
diamonds

<<enumeration>>  
Rank  
ace  
king  
queen  
....

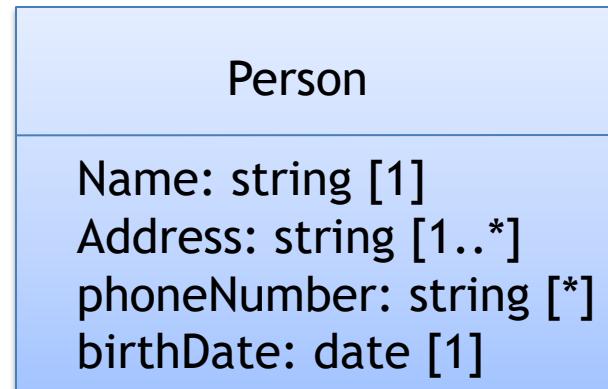
Do not use a generalization to capture the values of an enumerated attribute

# Multiplicity for an attribute

---

Specifies the number of possible values for each instantiation of an attribute.

- single value [1]
- optional single value [0..1]
- many [\*]

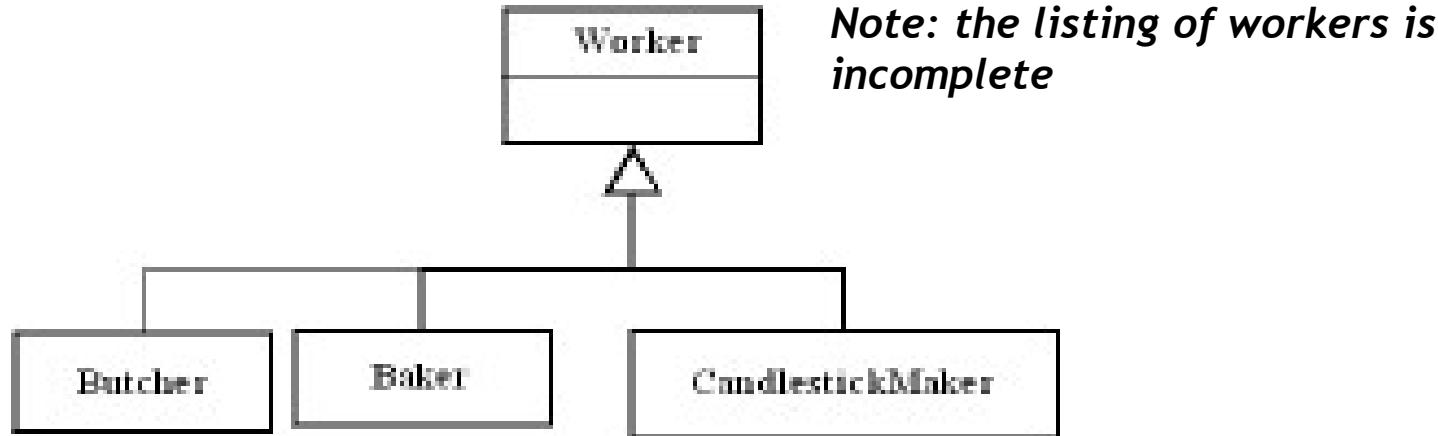




# Abstract Class

---

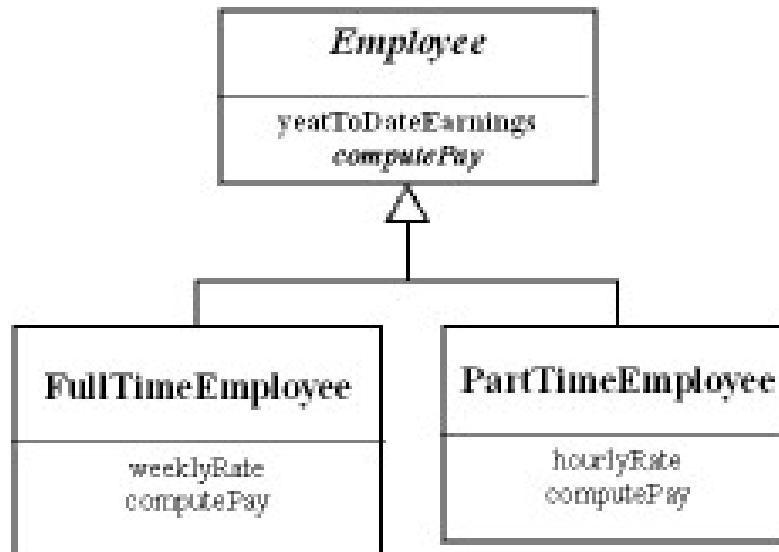
An **abstract class** is a class that has no direct instances but whose descendant classes have direct instances.



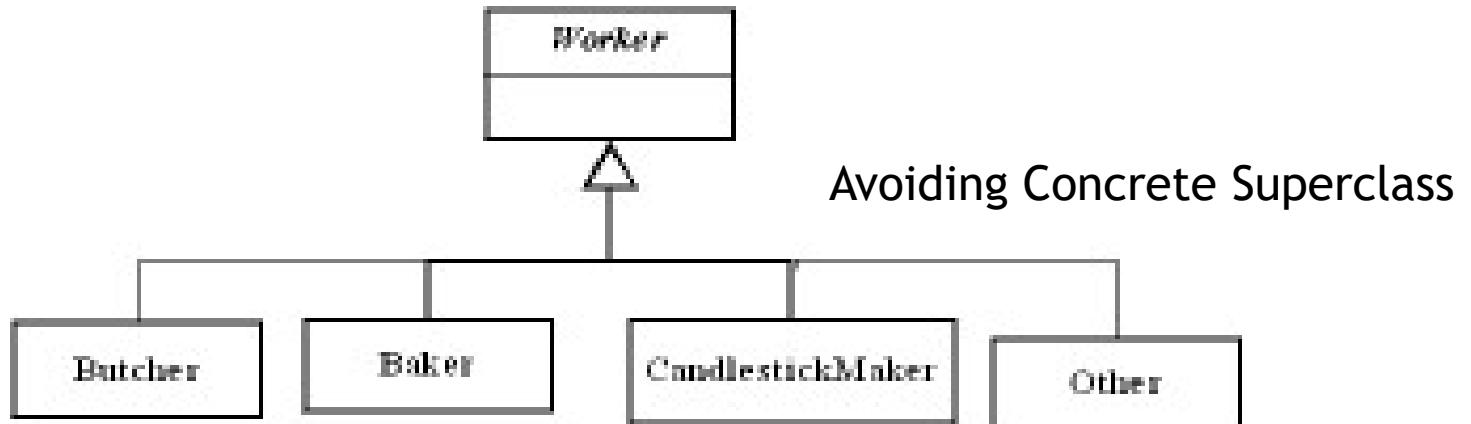
A **concrete class** is a class that is instantiable; it can have direct instances.

---

# Abstract Class



Abstract Class  
&  
Abstract Operation

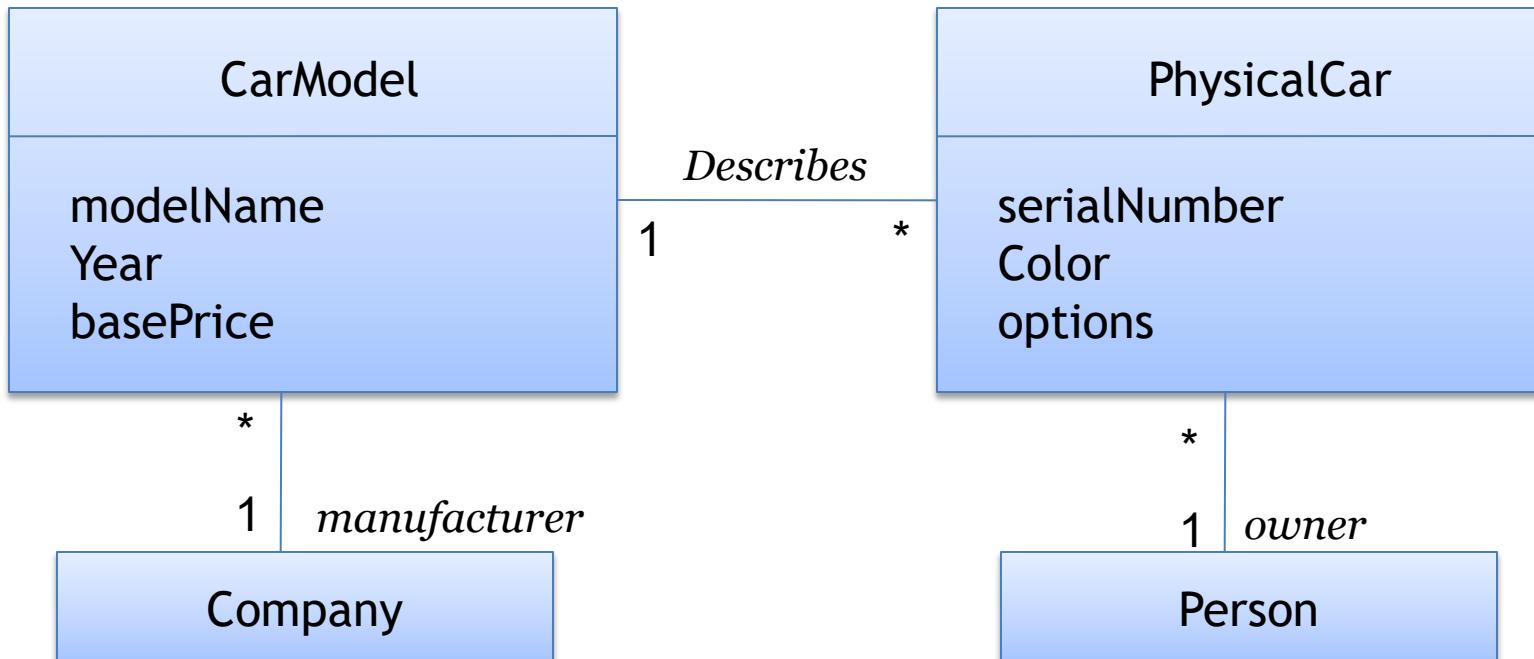


Avoiding Concrete Superclass

# Metadata

Metadata is data that describes other data

- A Class Definition is Metadata.
- Models are inherently metadata, describe the things being modeled
- Programming Language implementations also use metadata.



# Reification

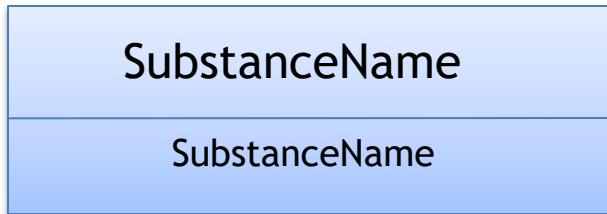
---

Reification is the promotion of something that is not an object into an object.

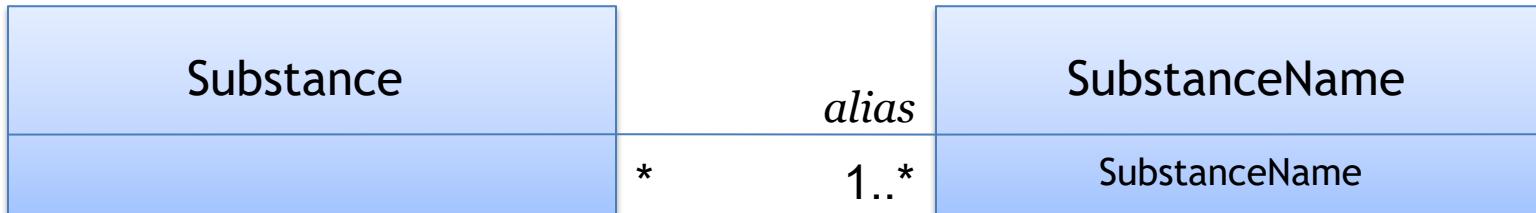
- It lets you to shift the level of abstraction
- For example, Consider a Database Manager. A developer could write code for each application to read and write files.

*Writing for all other applications????*

*Reify: Data services and use DATA Manager.*



***Reification: Promote attribute to a Class***



# Constraints

---

A **Constraint** is a Boolean condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets.

- It restricts the values that elements can assume.
- Natural Language or a formal language (**OCL; Object Constraint Language**)

## Constraints on Objects



{salary <= boss.salary}



{0.8<= length/width<= 1.5}



{priority never increases}

# Constraints

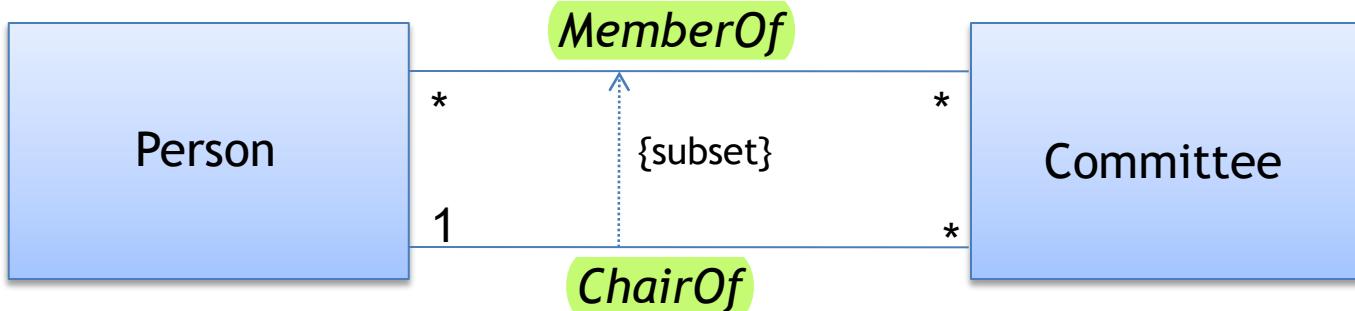
---

## Constraints on Links

Multiplicities, association names, association end names, qualification, {ordered}, {sequence} and many more....

Explicit constraint:

*The chair of a committee is a member of the committee; the ChairOf association is a subset of the MemberOf association.*



Subset constraints between associations

---



---

# Exercises

---



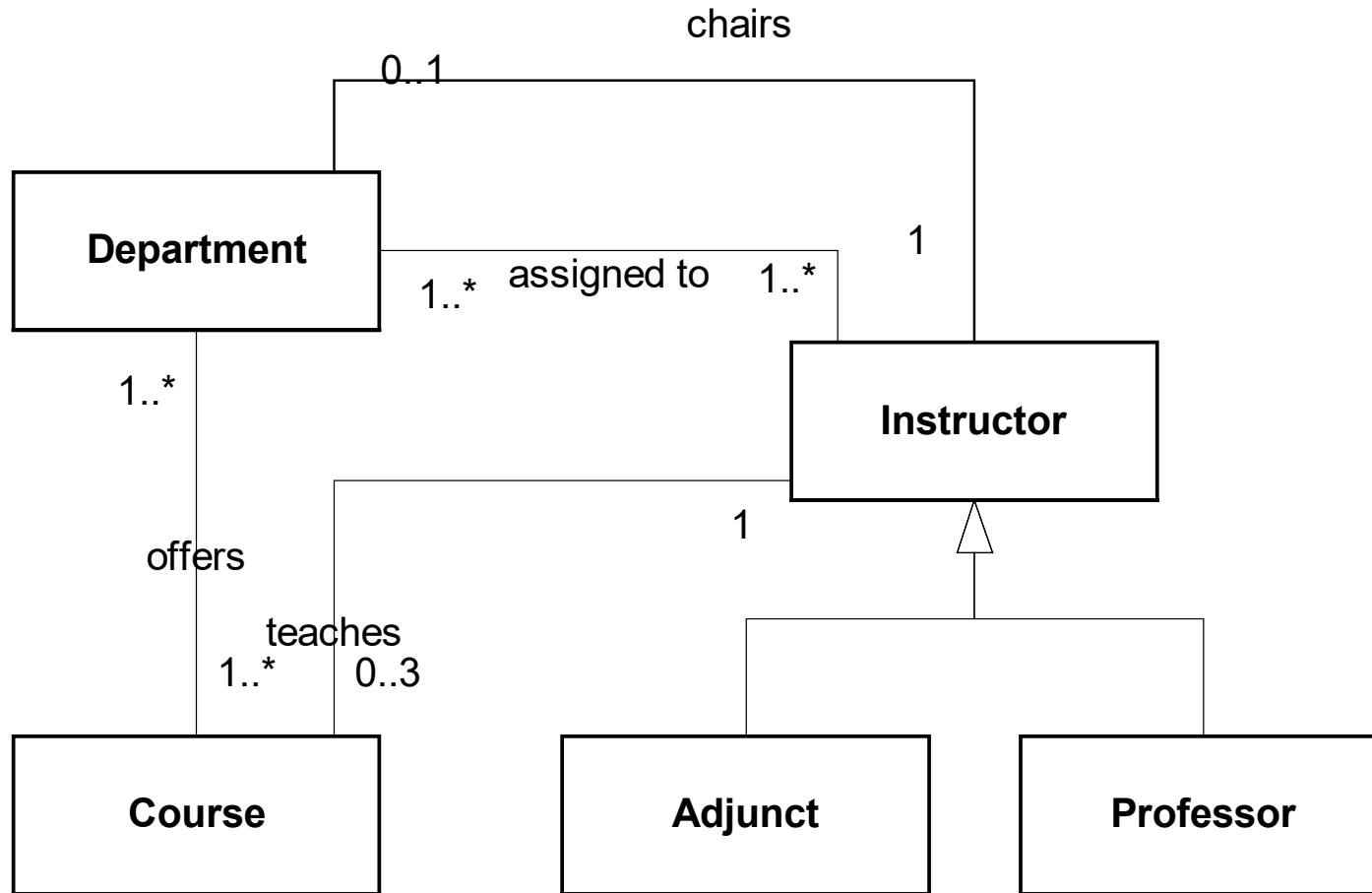
## Example 1: University Courses

---

- Some instructors are professors, while others have job title adjunct
  - Departments offer many courses, but a course may be offered by >1 department
  - Courses are taught by instructors, who may teach up to three courses
  - Instructors are assigned to one (or more) departments
  - One instructor also serves a department chair
-

# Class Diagram for Univ. Courses

---



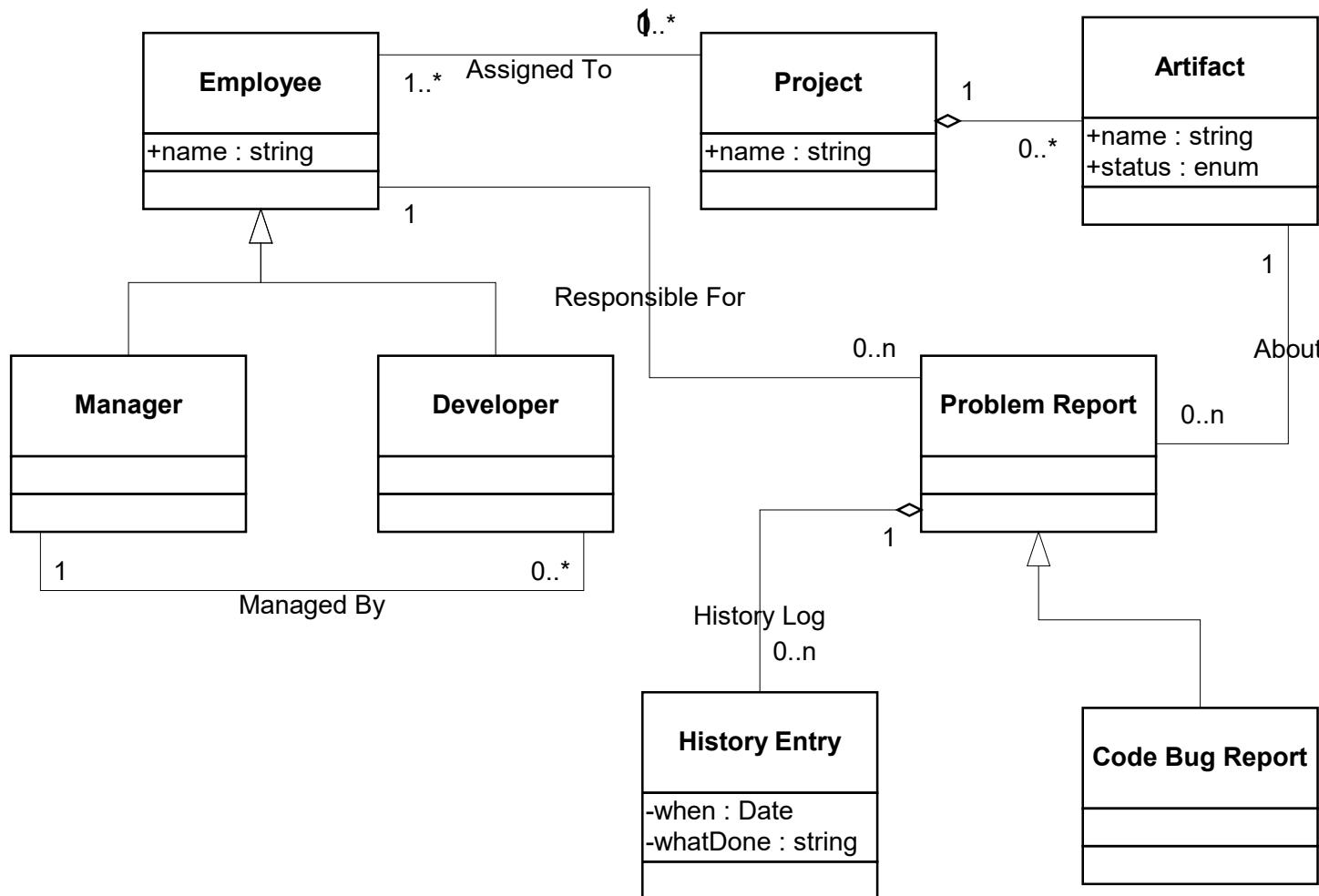


## Example 2: Problem Report Tool

---

- A CASE tool for storing and tracking problem reports
    - Each report contains a problem description and a status
    - Each problem can be assigned to someone
    - Problem reports are made on one of the “artifacts” of a project
    - Employees are assigned to one or more project
    - A manager may add new artifacts and assign problem reports to team members
-

# Class Diagram for Prob. Rep. Tool





---

**Questions??**

---

# Outline

- Overview of sequence diagrams
- Syntax and semantics
- Examples





# **an overview of sequence diagrams**

# **What is a UML sequence diagram?**

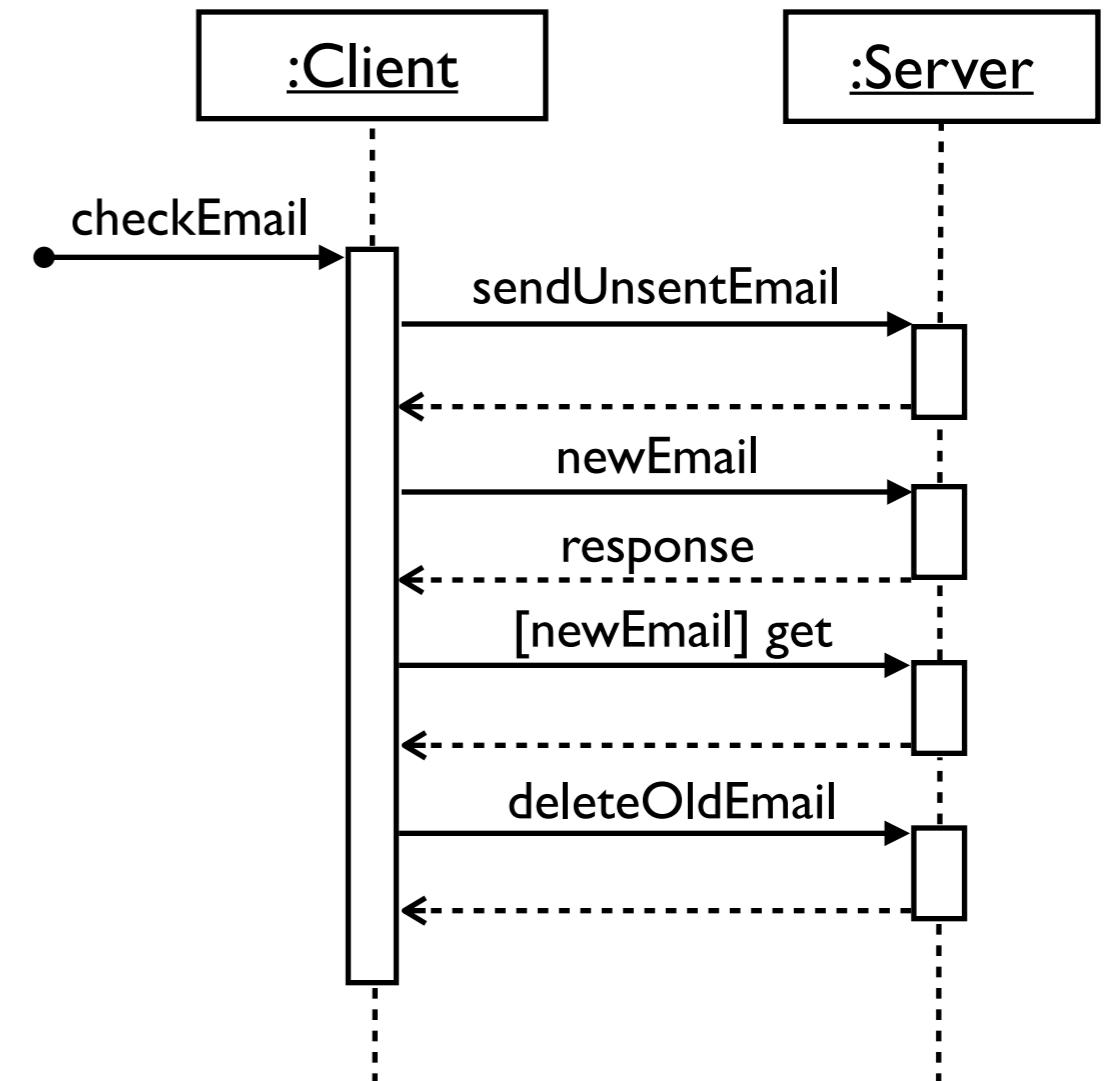
# **What is a UML sequence diagram?**

- **Sequence diagram:** an “interaction diagram” that models a single scenario executing in a system
  - 2nd most used UML diagram (behind class diagram)
  - Shows what messages are sent and when

# What is a UML sequence diagram?

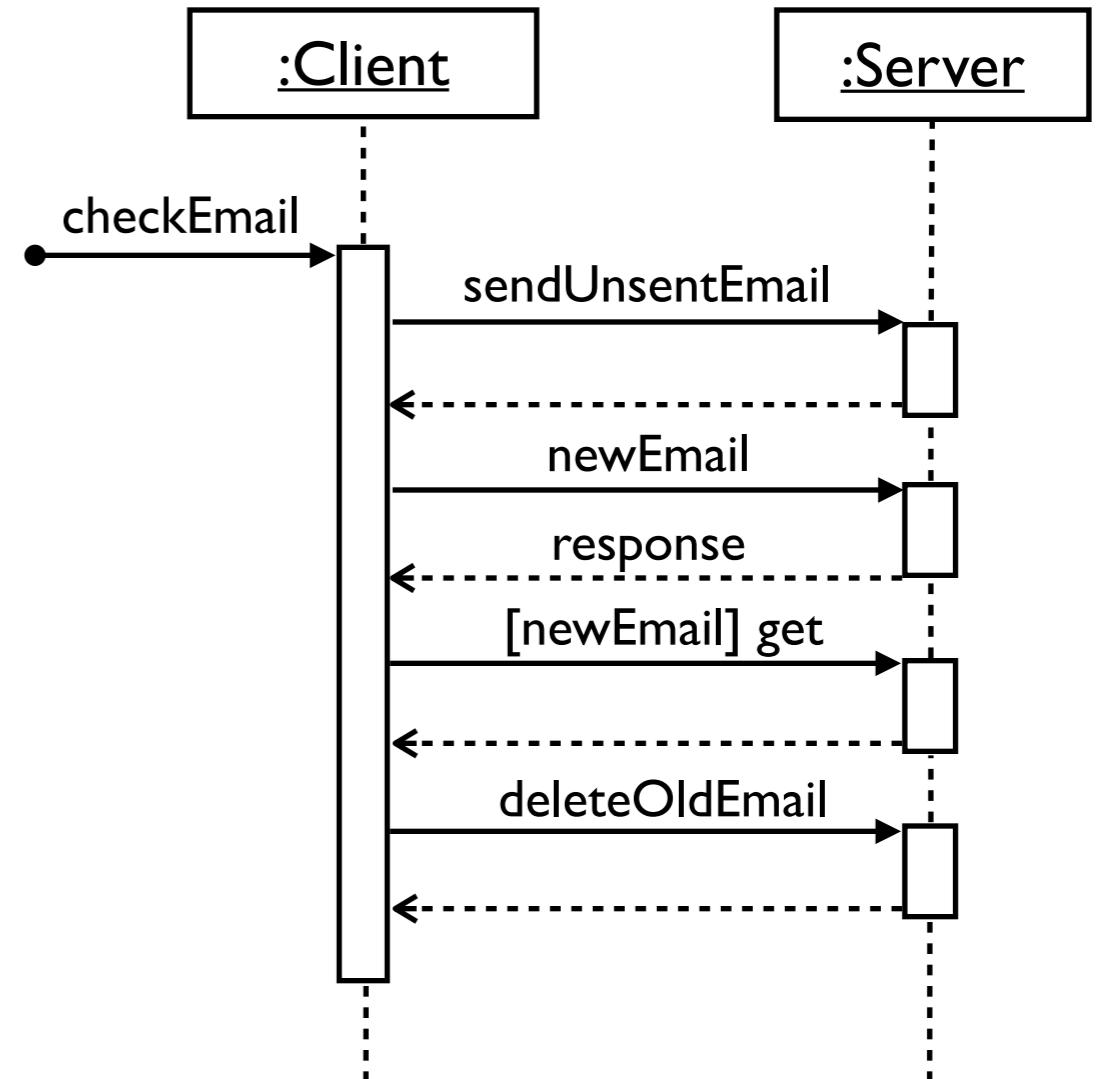
- **Sequence diagram:** an “interaction diagram” that models a single scenario executing in a system
  - 2nd most used UML diagram (behind class diagram)
  - Shows what messages are sent and when
- Relating UML diagrams to other design artifacts:
  - CRC cards → class diagrams
  - Use cases → sequence diagrams

# Key parts of a sequence diagram



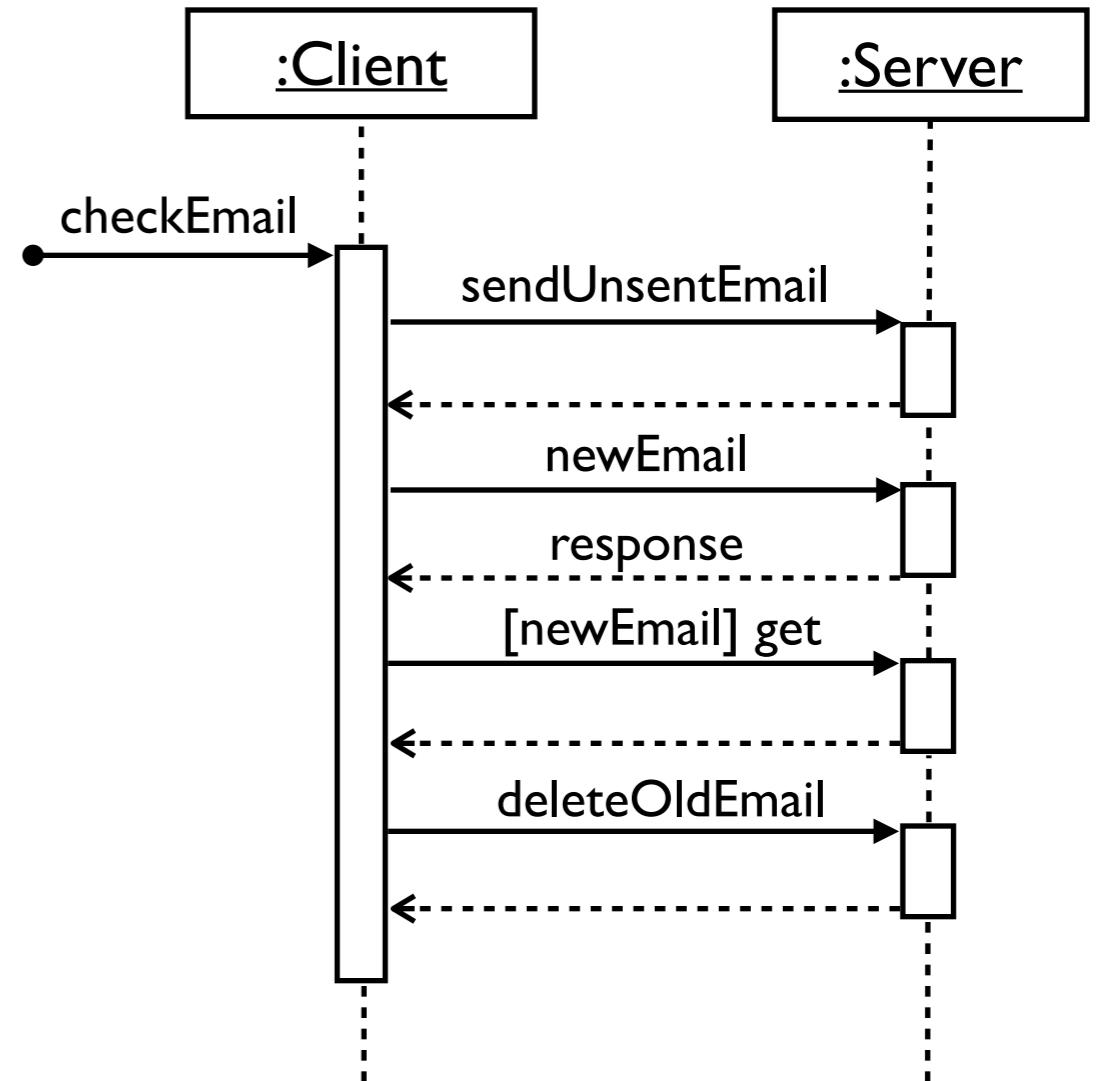
# Key parts of a sequence diagram

- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow



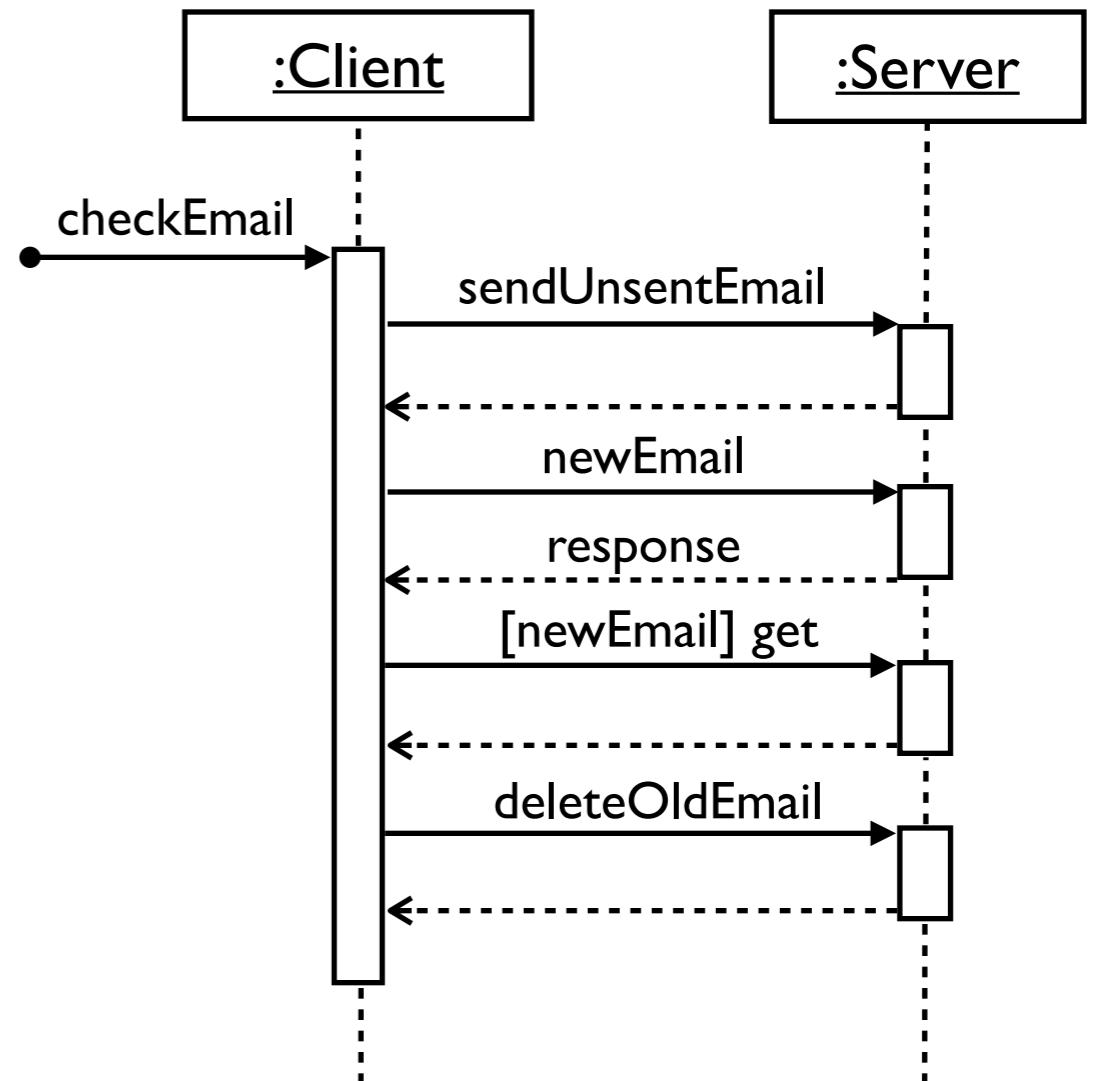
# Key parts of a sequence diagram

- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow
- **Message:** communication between objects



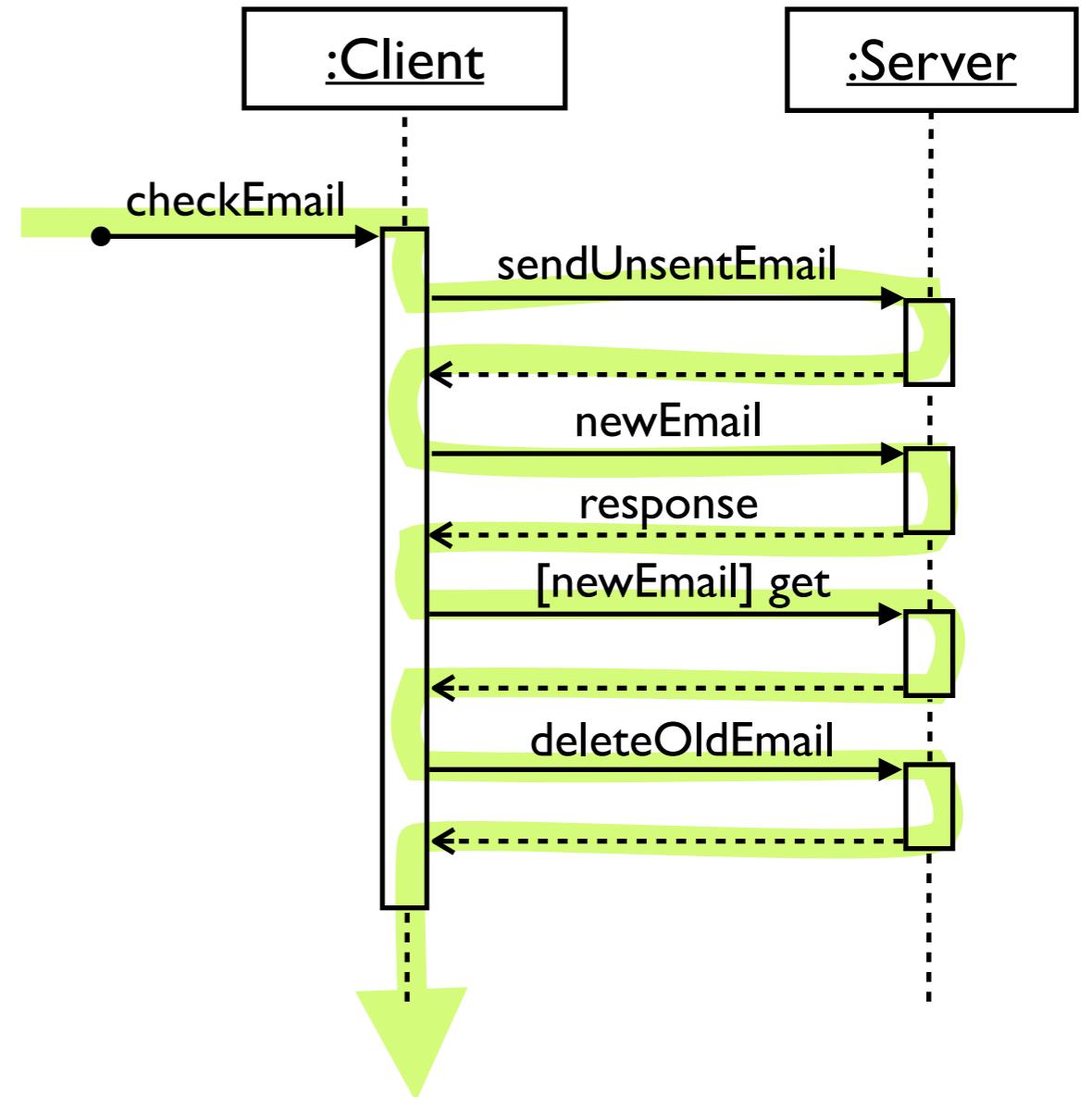
# Key parts of a sequence diagram

- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow
- **Message:** communication between objects
- **Axes in a sequence diagram:**
  - **horizontal:** which participant is acting
  - **vertical:** time (↓ forward in time)



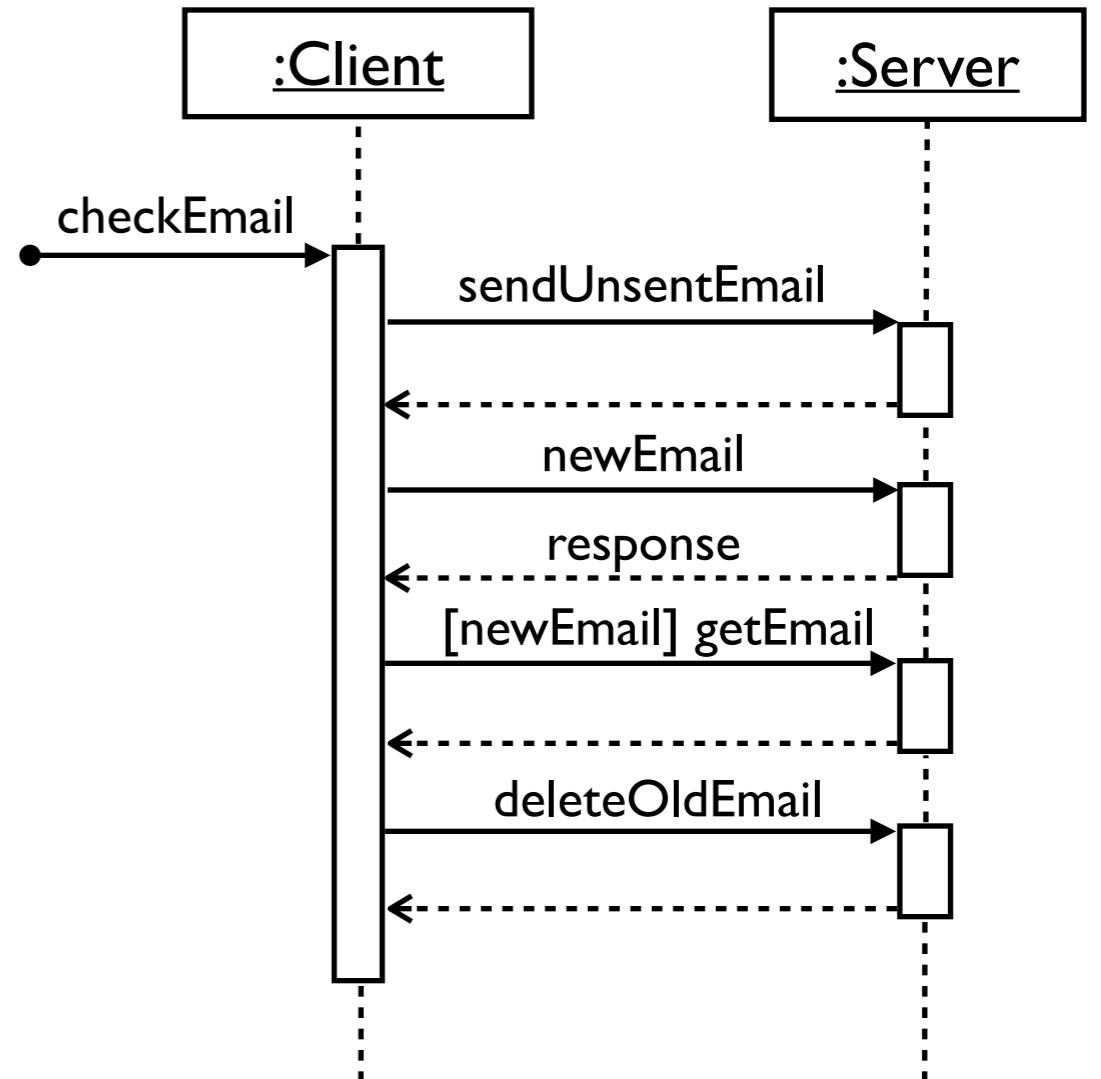
# Key parts of a sequence diagram

- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow
- **Message:** communication between objects
- Axes in a sequence diagram:
  - **horizontal:** which participant is acting
  - **vertical:** time (↓ forward in time)



# Sequence diagram from a use case

1. The user presses the “check email” button.
2. The client first sends all unsent email to the server.
3. After receiving an acknowledgement, the client asks the server if there is any new email.
4. If so, it downloads the new email.
5. Next, it deletes old thrashed email from the server.





**sequence diagrams: syntax and semantics**

# Representing objects

objectname:classname

Named object

Smith:Patient

Anonymous object

:Patient

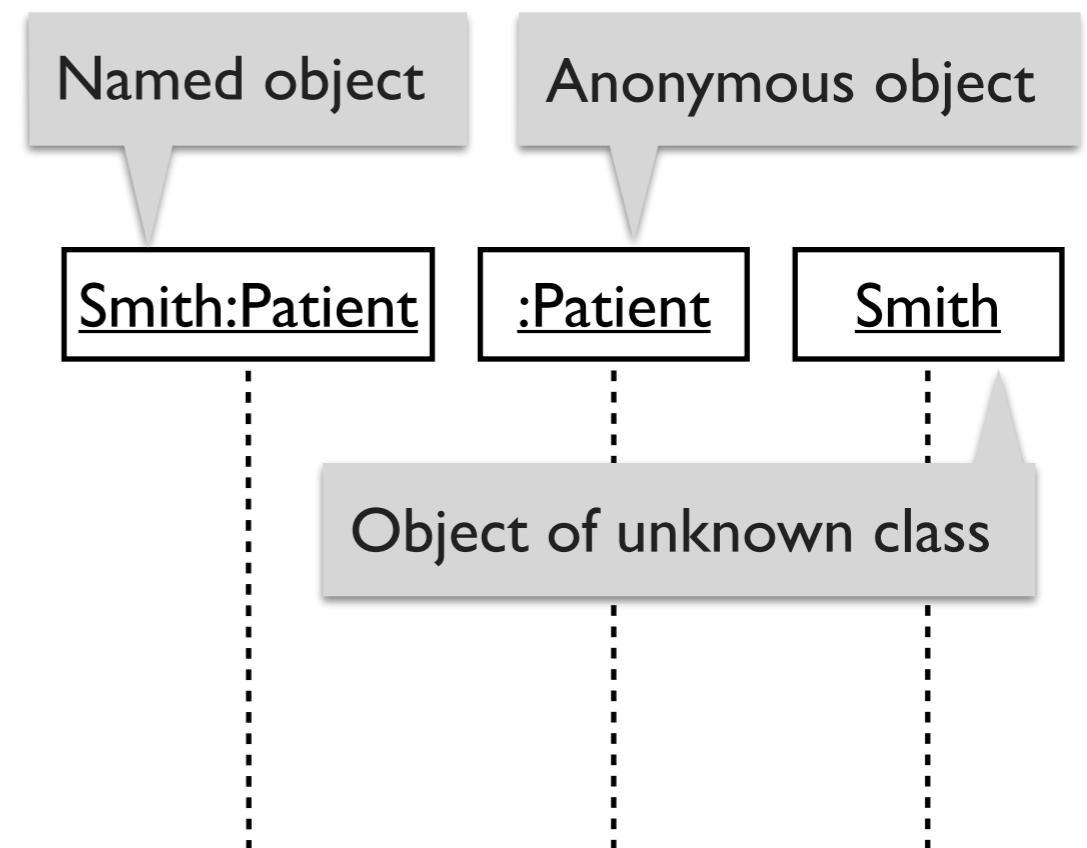
Smith

Object of unknown class

# Representing objects

objectname:classname

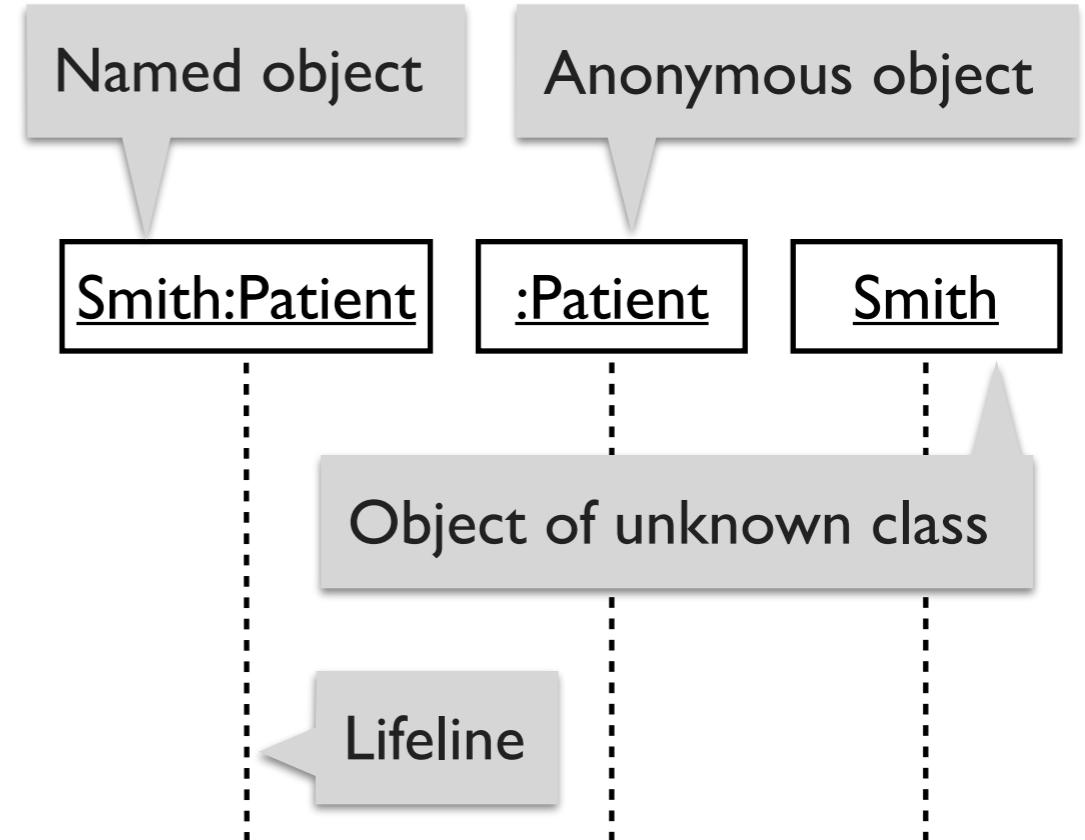
- An **object**: a **box** with an underlined label that specifies the object type, and optionally the object name.
  - Write the object's name if it clarifies the diagram.



# Representing objects

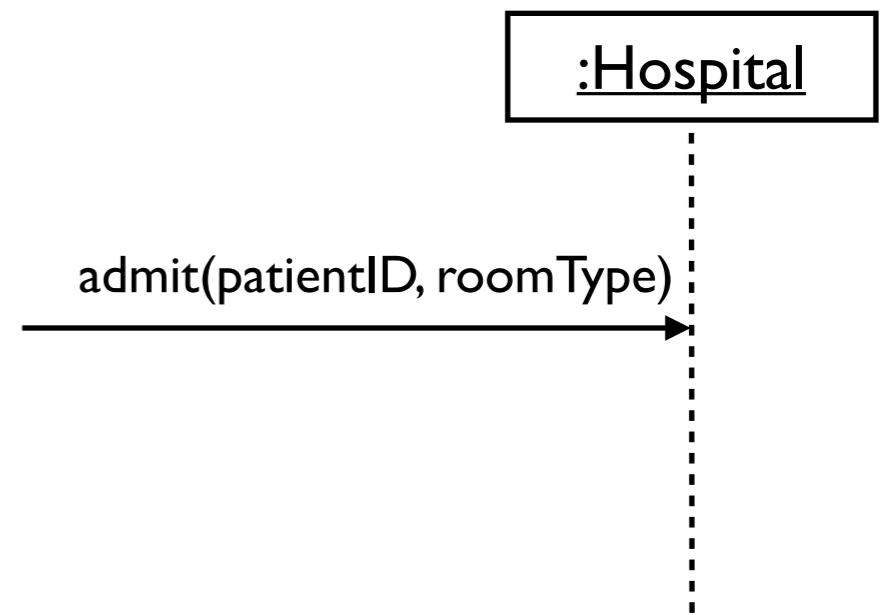
objectname:classname

- An **object**: a **box** with an underlined label that specifies the object type, and optionally the object name.
  - Write the object's name if it clarifies the diagram.
- An object's "life line" is represented by a dashed vertical line.
  - Represents the life span of the object during the scenario being modeled.



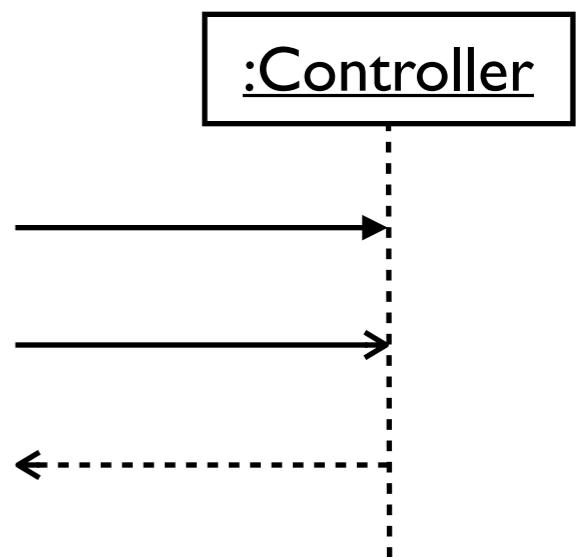
# Representing messages between objects

- A **message** (method call): **horizontal arrow** to the receiving object.
  - Write message name and arguments above the arrow.



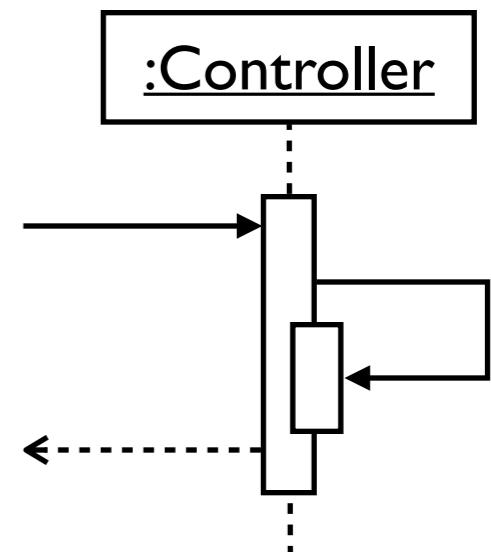
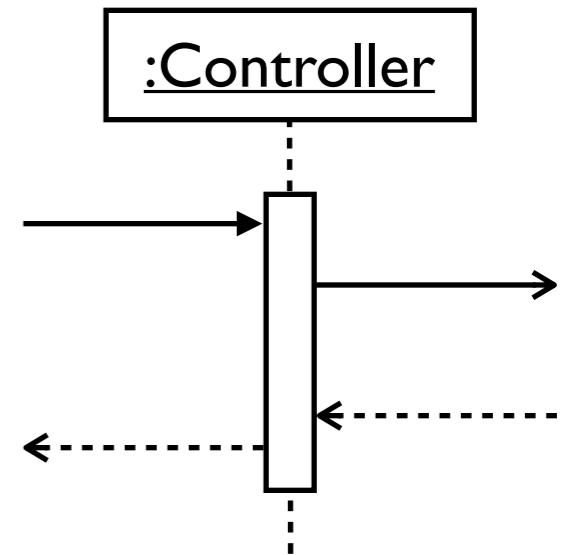
# Different types of messages

- Type of arrow indicates types of messages:
  - Synchronous message: solid arrow with a solid head.
  - Asynchronous message: solid arrow with a stick head.
  - Return message: dashed arrow with stick head.



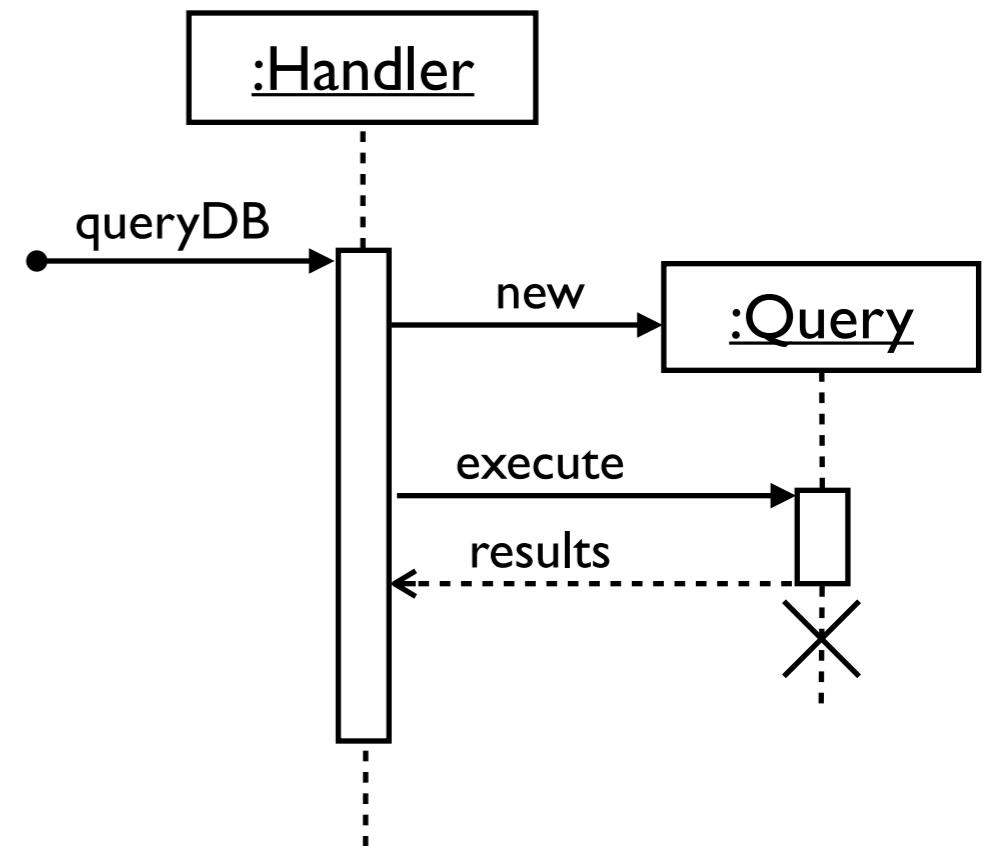
# Indicating method execution

- **Activation:** thick box over object's life line, drawn when an object's method is on the stack
  - Either that object is running its code, or it is on the stack waiting for another object's method to finish
- Nest activations to indicate an object calling itself.



# Lifetime of objects

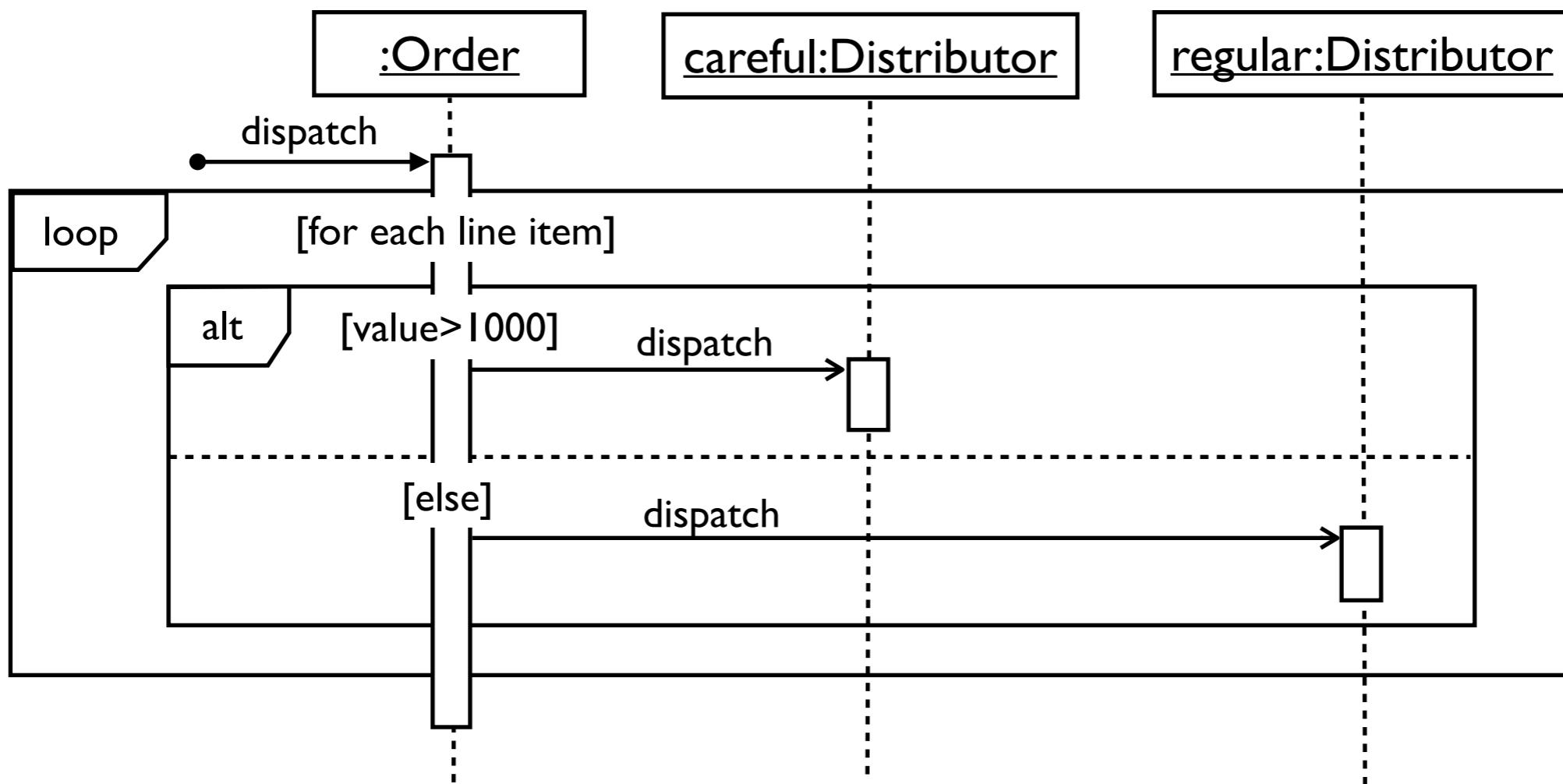
- **Object creation:** an arrow with **new** written above it
  - An object created after the start of the scenario appears lower than the others.
- **Object deletion:** X at the bottom of object's lifeline
  - Java doesn't explicitly delete objects; they fall out of scope and are garbage collected.



# Alternatives, options, and loops

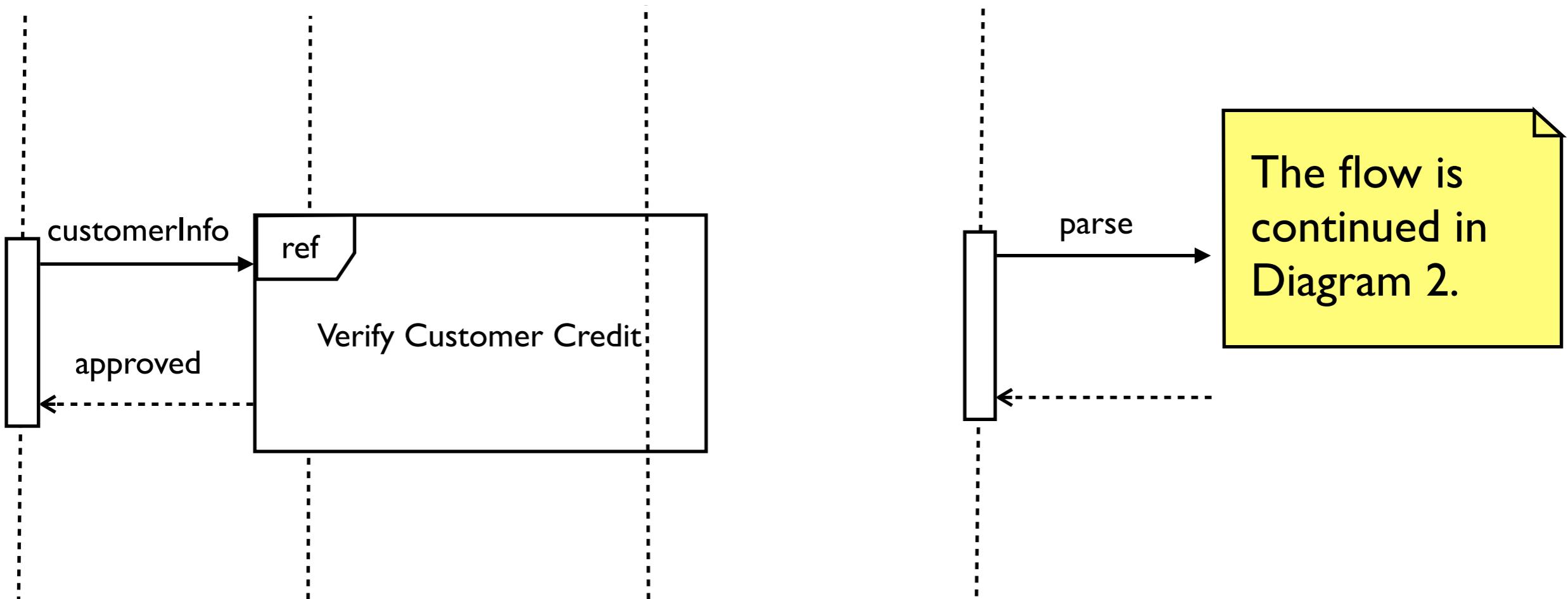
- Frame: a box around part of a sequence diagram

- if → (opt) [condition]
- if/else → (alt) [condition], separated by horizontal dashed line
- loop → (loop) [condition or items to loop over]

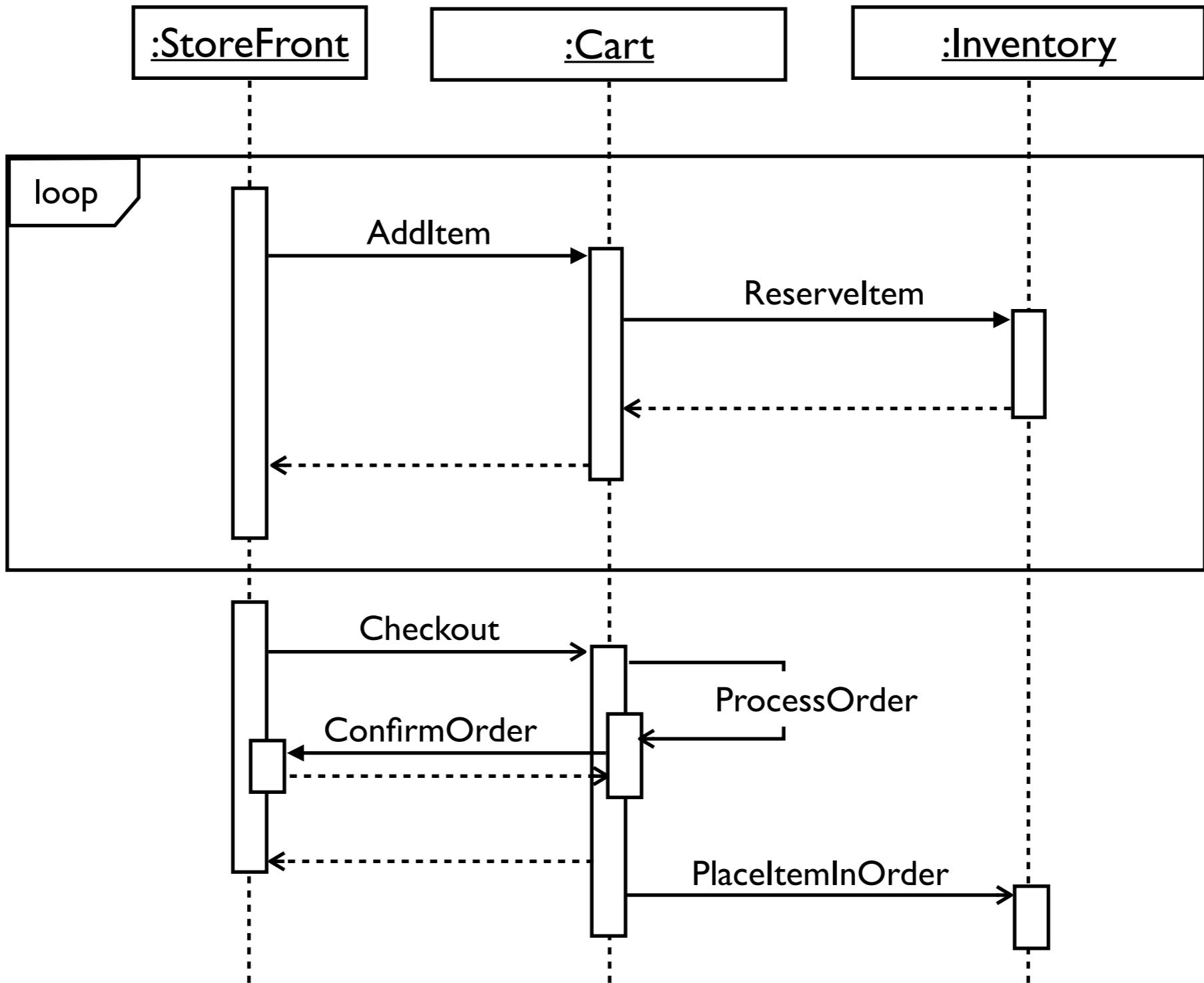


# Linking sequence diagrams

- If one sequence diagram is too large or refers to another diagram:
  - An unfinished arrow and comment.
  - A **ref** frame that names the other diagram.

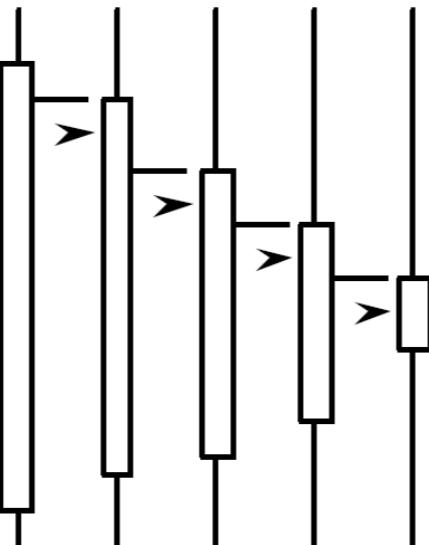
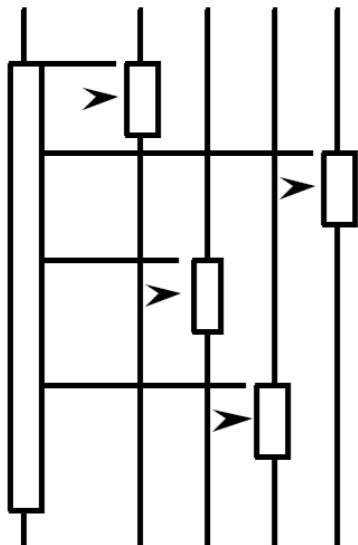


# Example sequence diagram



# Forms of system control

Centralized as the work depends on the last object called and no other object is killed or ends process before that



What can you say about the control flow of each of these systems?

- Is it centralized?
- Is it distributed?

Distributed as the work is done by all the objects in equal amount

# **Why use sequence diagrams? Why not code it?**

# **Why use sequence diagrams? Why not code it?**

- A good sequence diagram is still above the level of the real code (not all code is drawn on diagram)

# **Why use sequence diagrams? Why not code it?**

- A good sequence diagram is still above the level of the real code (not all code is drawn on diagram)
- Sequence diagrams are language-agnostic (can be implemented in many different languages)

# **Why use sequence diagrams? Why not code it?**

- A good sequence diagram is still above the level of the real code (not all code is drawn on diagram)
- Sequence diagrams are language-agnostic (can be implemented in many different languages)
- Non-coders can read and write sequence diagrams.

# **Why use sequence diagrams? Why not code it?**

- A good sequence diagram is still above the level of the real code (not all code is drawn on diagram)
- Sequence diagrams are language-agnostic (can be implemented in many different languages)
- Non-coders can read and write sequence diagrams.
- Easier to do sequence diagrams as a team.

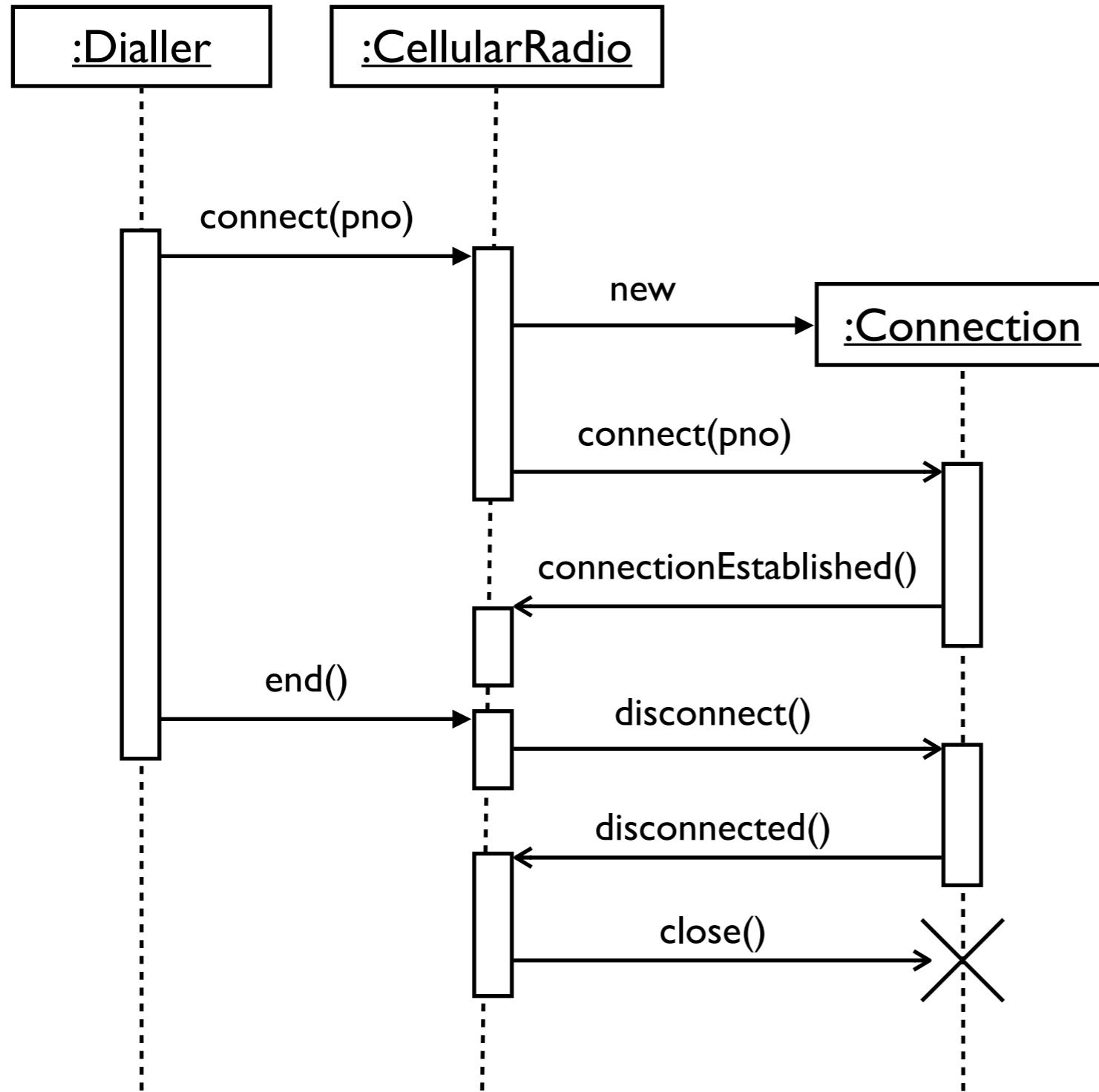
# **Why use sequence diagrams? Why not code it?**

- A good sequence diagram is still above the level of the real code (not all code is drawn on diagram)
- Sequence diagrams are language-agnostic (can be implemented in many different languages)
- Non-coders can read and write sequence diagrams.
- Easier to do sequence diagrams as a team.
- Can see many objects/classes at a time on same page (visual bandwidth).

# Study

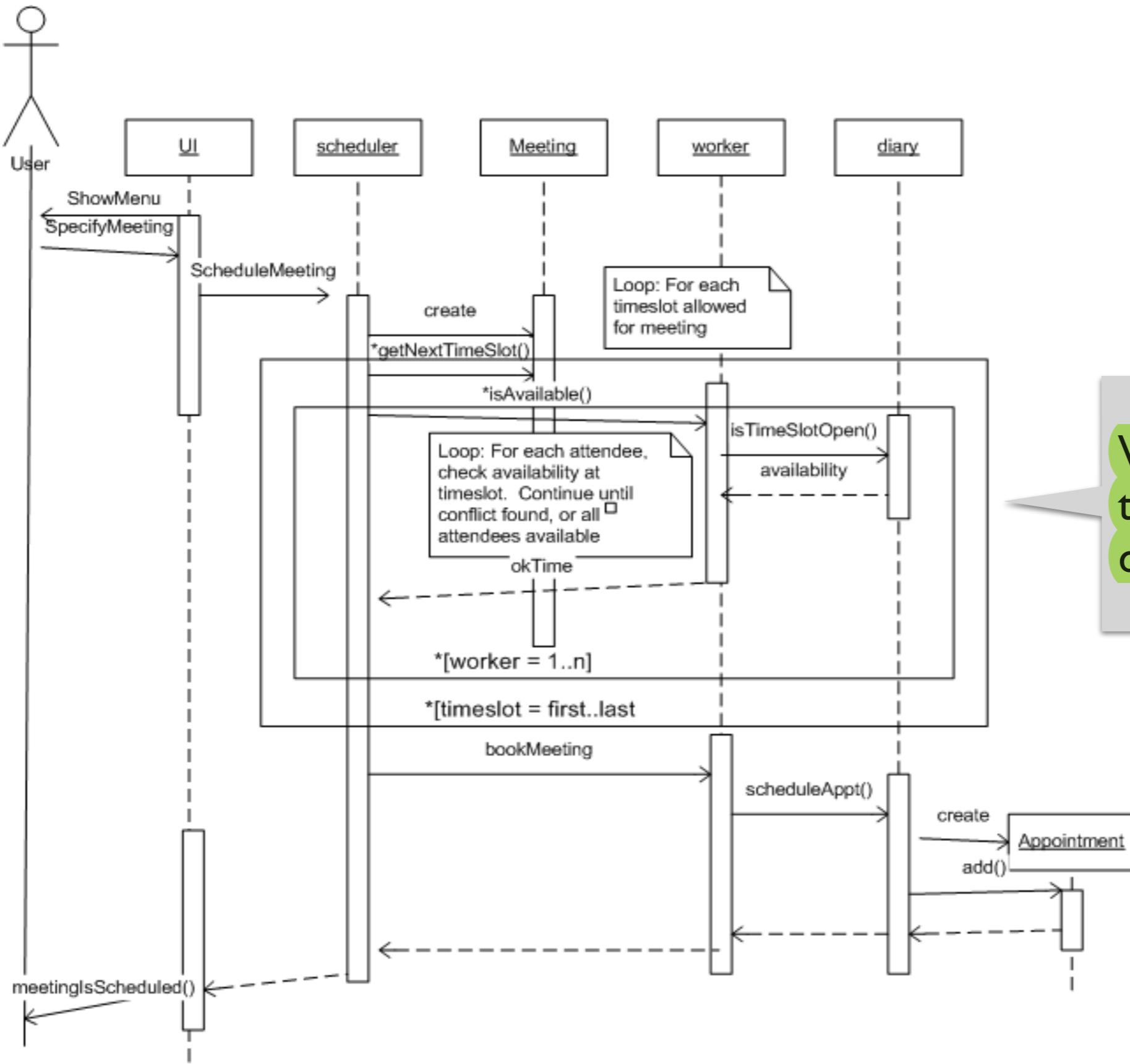
## sequence diagrams: examples

# Flawed sequence diagram I



What's wrong with  
this sequence  
diagram?

# Flawed sequence diagram 2

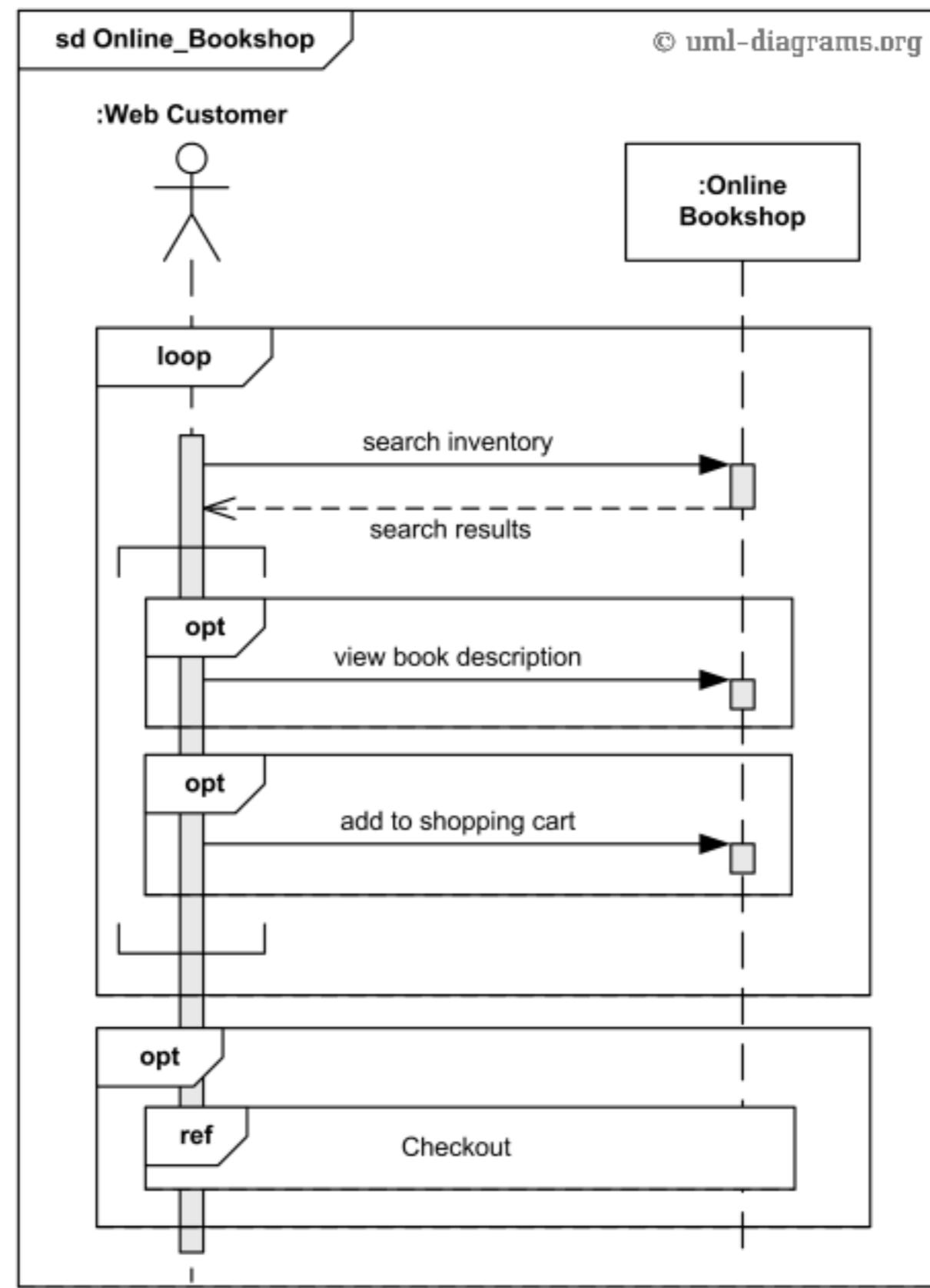


What's wrong with  
this sequence  
diagram?

# Online bookstore example

1. The customer begins the interaction by searching for a book by title.
2. The system will return all books with that title.
3. The customer can look at the book description.
4. The customer can place a book in the shopping cart.
5. The customer can repeat the interaction as many times as desired.
6. The customer can purchase the items in the cart by checking out.

# Online bookstore sequence diagram



# Summary

- A sequence diagram models a single scenario executing in the system.
- Key components include participants and messages.
- Sequence diagrams provide a high-level view of control flow patterns through the system.





# IT 314: Software Engineering

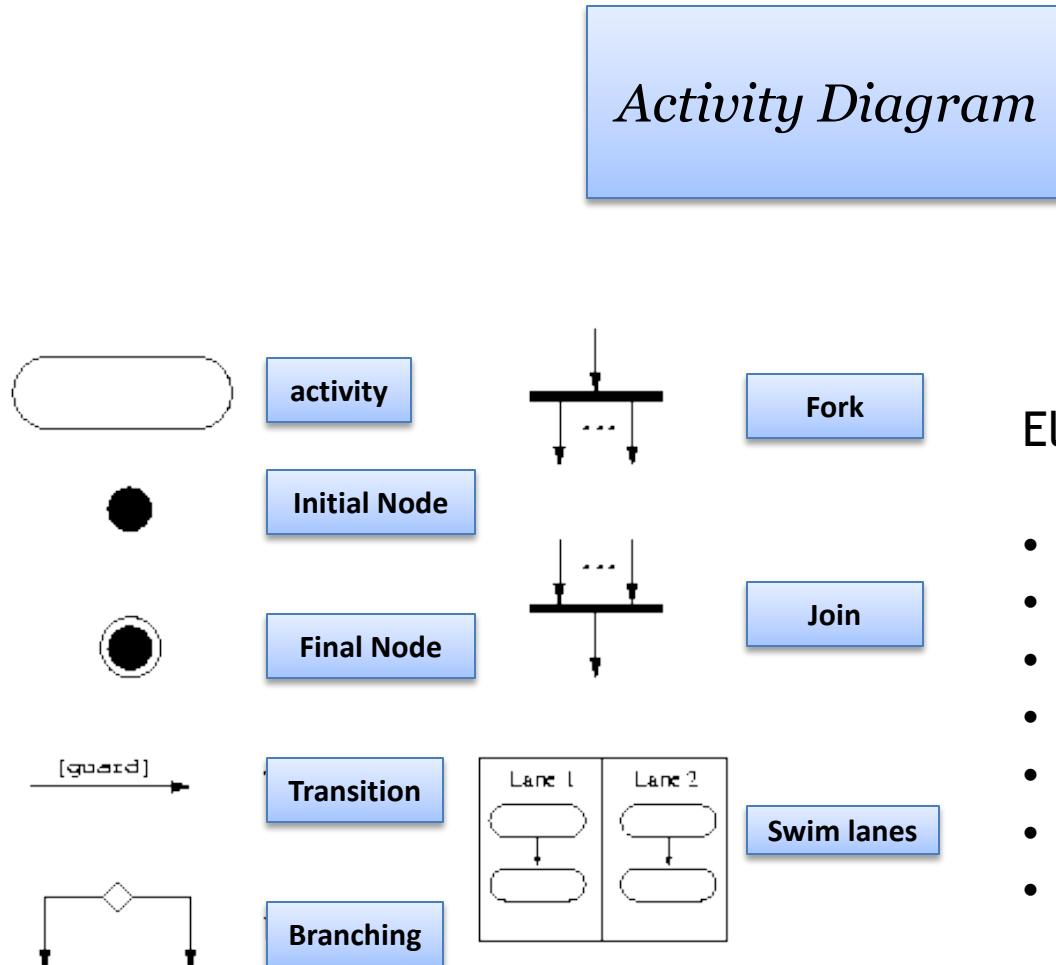
*Activity Diagram  
(Use Case to Activity Diagram)*

# Activity Diagram

---

- Activity diagrams represent the dynamic (behavioral) view of a system
  - Activity diagrams are typically used for business (transaction) process modeling and modeling the logic captured by a single use-case or usage scenario
  - Activity diagram is used to represent the flow across use cases or to represent flow within a particular use case
  - UML activity diagrams are the object oriented equivalent of flow chart and data flow diagrams in function-oriented design approach
  - Activity diagram contains activities, transitions between activities, decision points, synchronization bars, swim lanes and many more...
-

# Activity Diagram

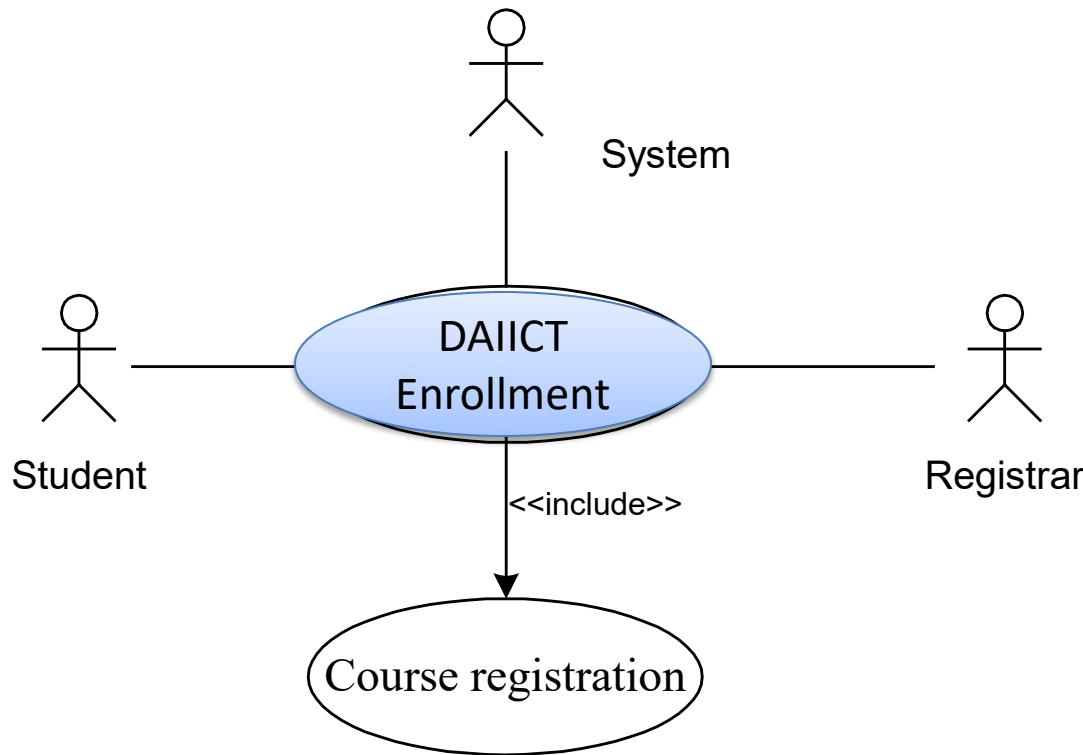


## Elements of Activity Diagram

- Initial Node
- Final Node
- Activity (Node)
- Control Flow (Edge)
- Decision (Branch)
- Fork/Join Node
- Swim lanes

# Student Enrollment in DAIICT

---



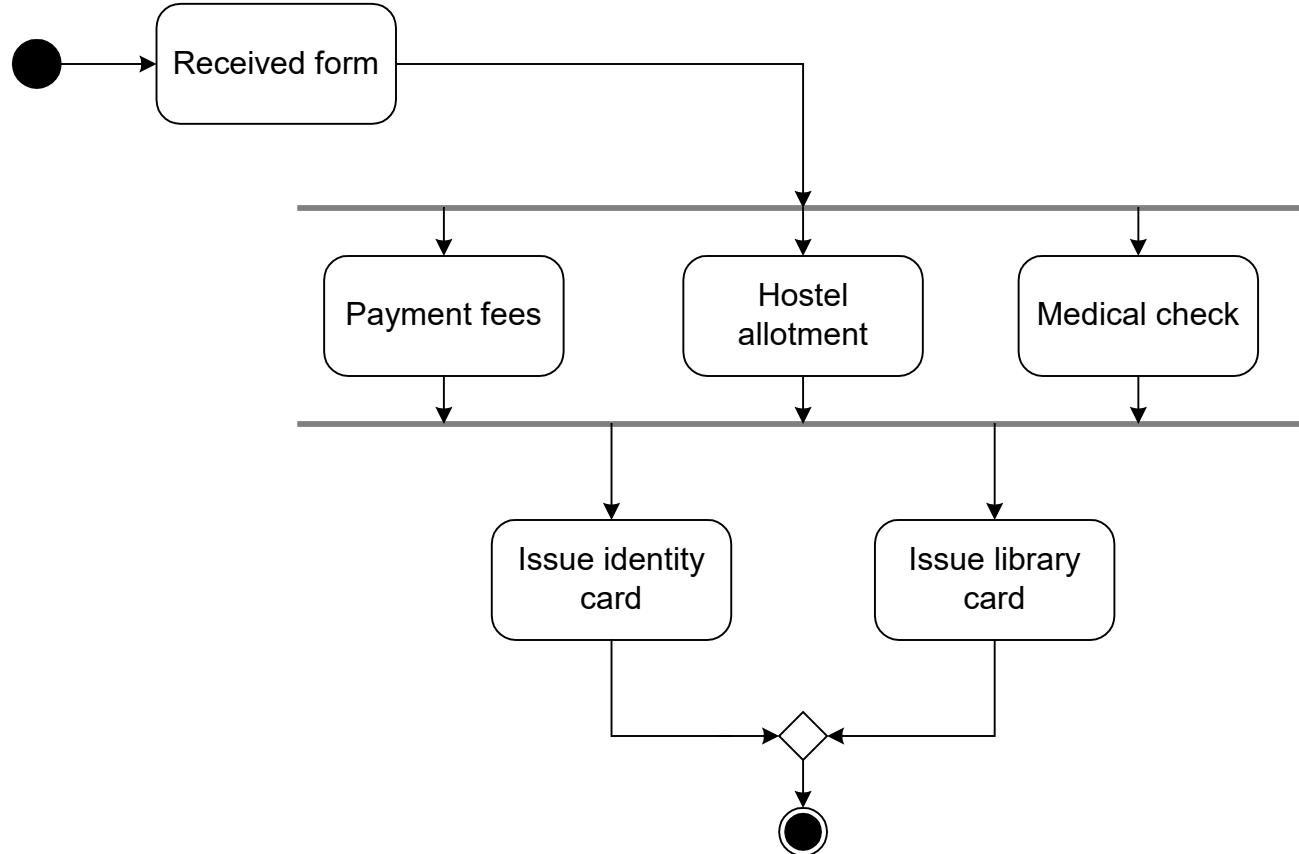


# SEDAIICT System

---

- Here different activities are:
  - Received enrollment form filled by the student
    - Registrar checks the form
    - Input data to the system
    - System authenticate the environment
  - Pay fees by the student
    - Registrar checks the amount to be remitted and prepare a bill
    - System acknowledge fee receipts and print receipt
  - Hostel allotment
    - Allot hostel
    - Receive hostel charge
    - Allot room
  - Medical check up
    - Create hostel record
    - Conduct medical bill
    - Enter record
  - Issue library card
  - Issue identity card

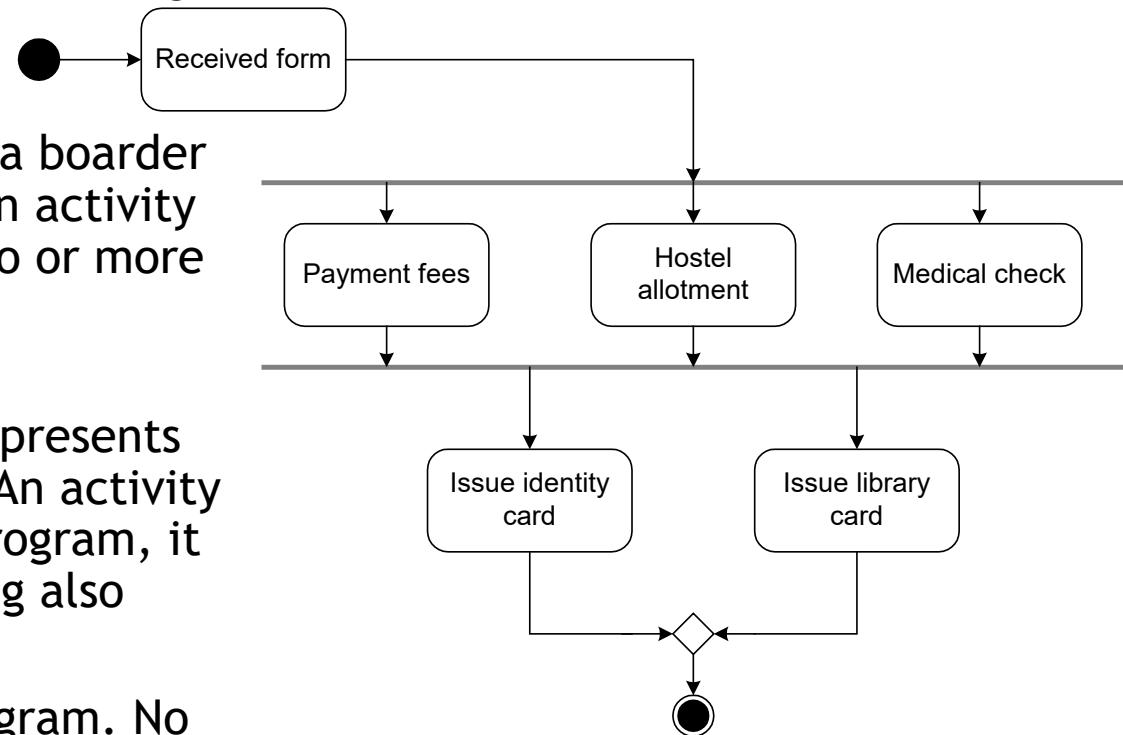
# Activity Diagram for the Use Case



# Basic Components in an Activity Diagram

- **Initial node**

- The filled circle is the starting point of the diagram



- **Final node**

- The filled circle with a border is the ending point. An activity diagram can have zero or more activity final state.

- **Activity**

- The rounded circle represents activities that occur. An activity is not necessarily a program, it may be a manual thing also

- **Flow/ edge**

- The arrows in the diagram. No label is necessary

# Basic Components in an Activity Diagram

- **Fork**

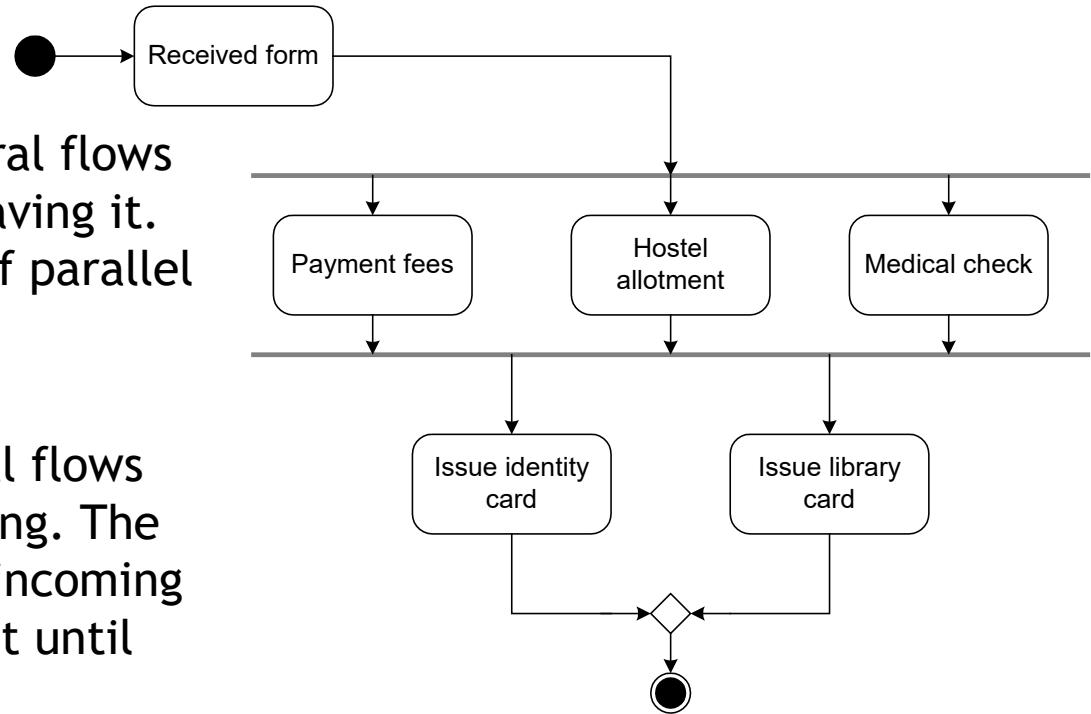
- A black bar ( horizontal/vertical) with one flow going into it and several leaving it. This denotes the beginning of parallel activities

- **Join**

- A block bar with several flows entering it and one leaving it. this denotes the end of parallel activities

- **Merge**

- A diamond with several flows entering and one leaving. The implication is that all incoming flow to reach this point until processing continues



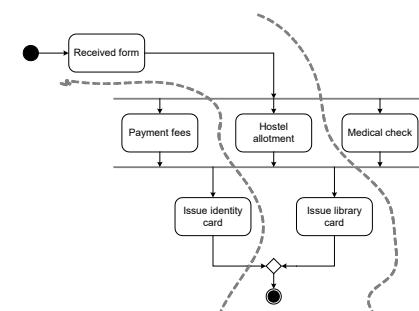
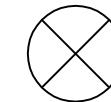
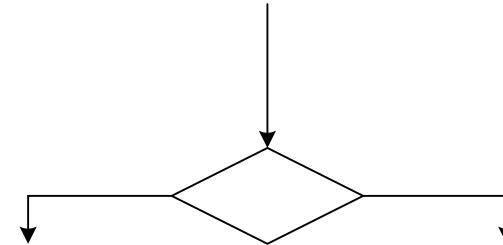
# Basic Components in an Activity Diagram

---

- Difference between Join and Merge
  - A join is different from a merge in that the join synchronizes two inflows and produces a single outflow. The outflow from a join cannot execute until all inflows have been received
  - A merge passes any control flows straight through it. If two or more inflows are received by a merge symbol, the action pointed to by its outflow is executed two or more times

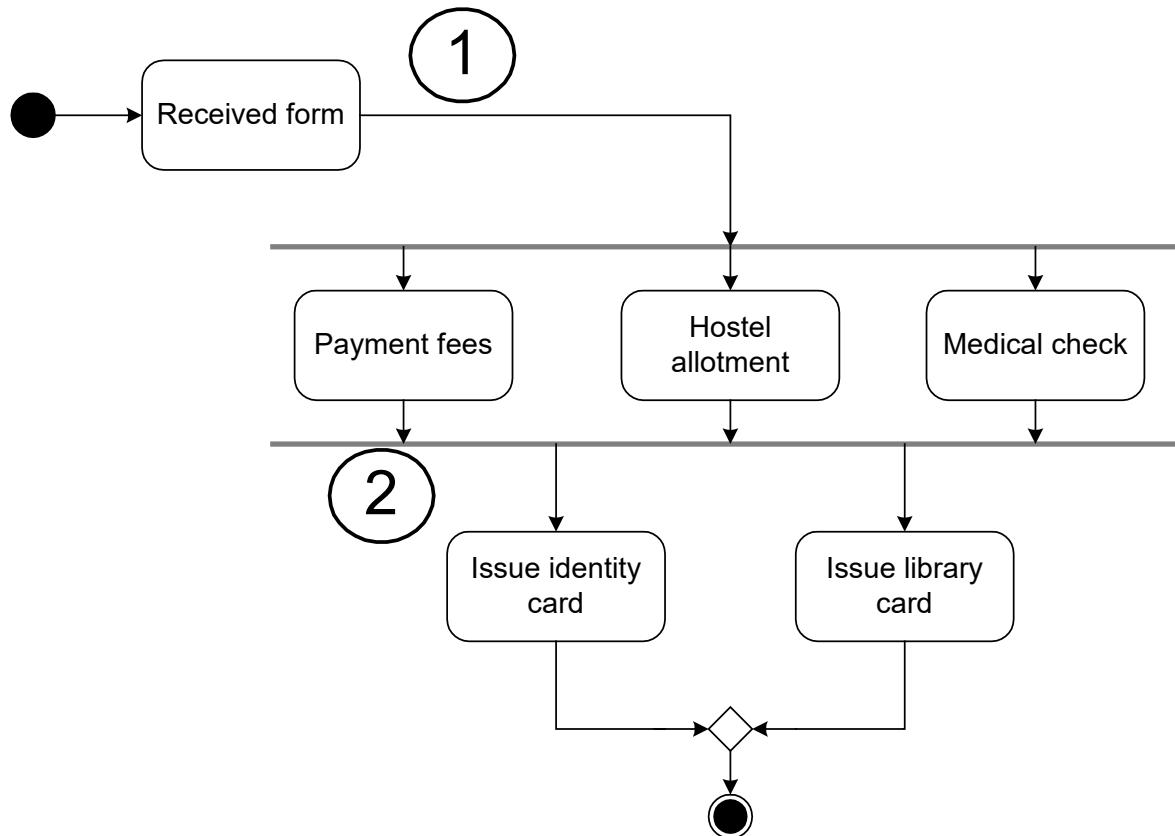
# Basic Components in an Activity Diagram

- **Decision**
  - A diamond with one flow entering and several leaving.  
The flow leaving includes conditions as yes/ no state
- **Flow final**
  - The circle with X though it.  
**This indicates that Process stop at this point**
- **Swim lane**
  - A **partition in activity diagram by means of dashed line, called swim lane.** This swim lane may be horizontal or vertical

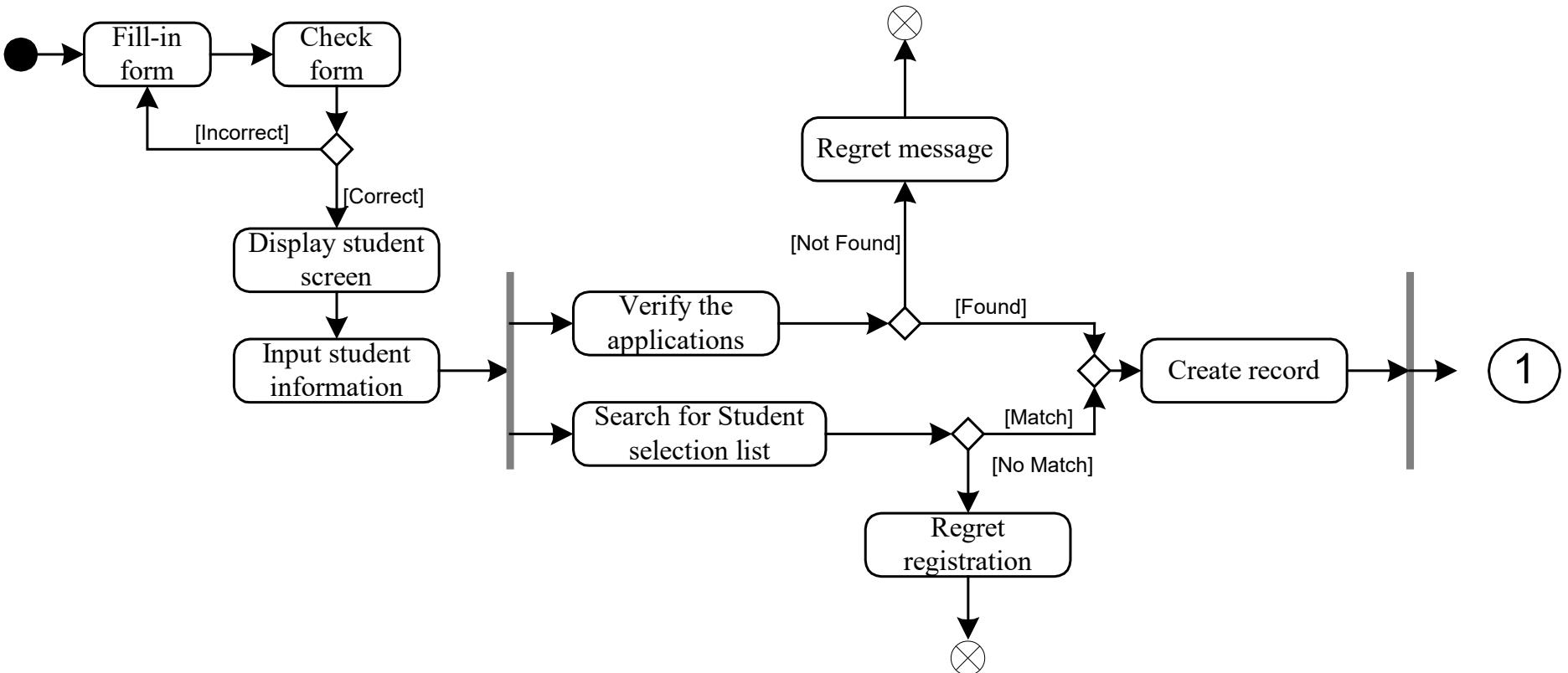


# Detailed Activity Diagram

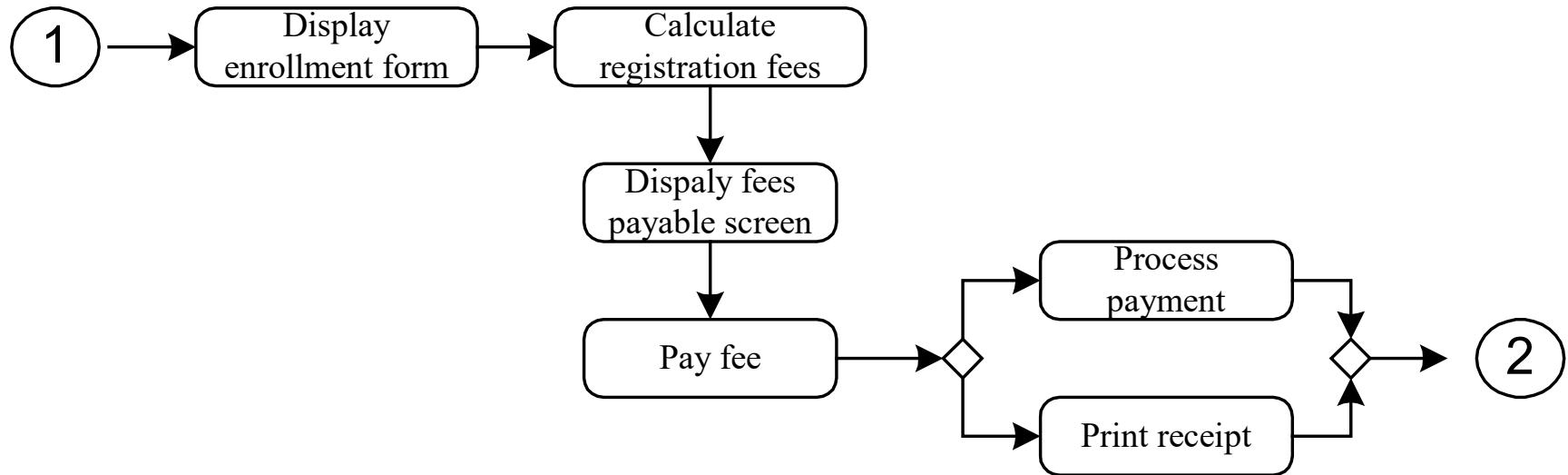
---



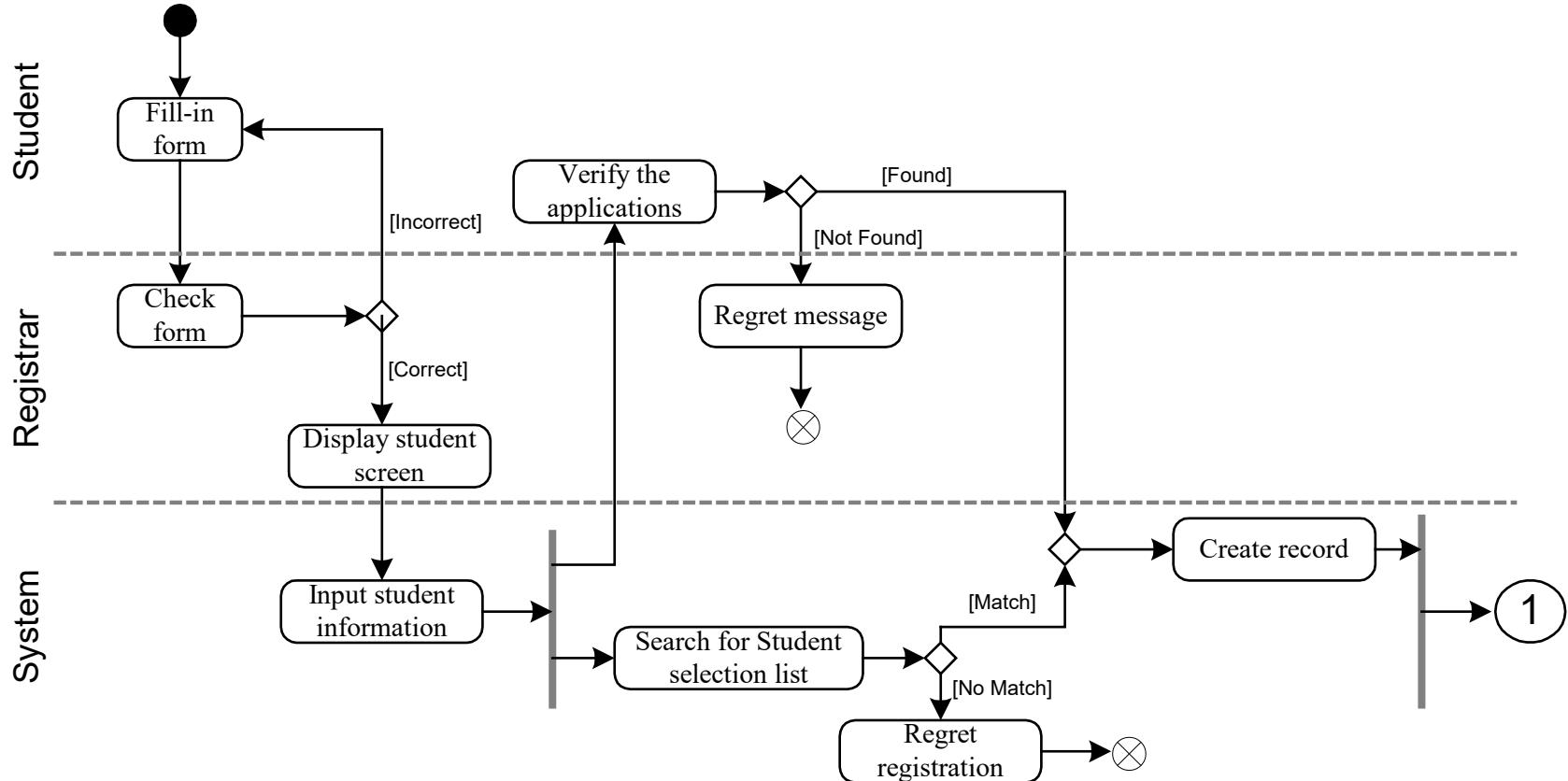
# Detailed Activity Diagram



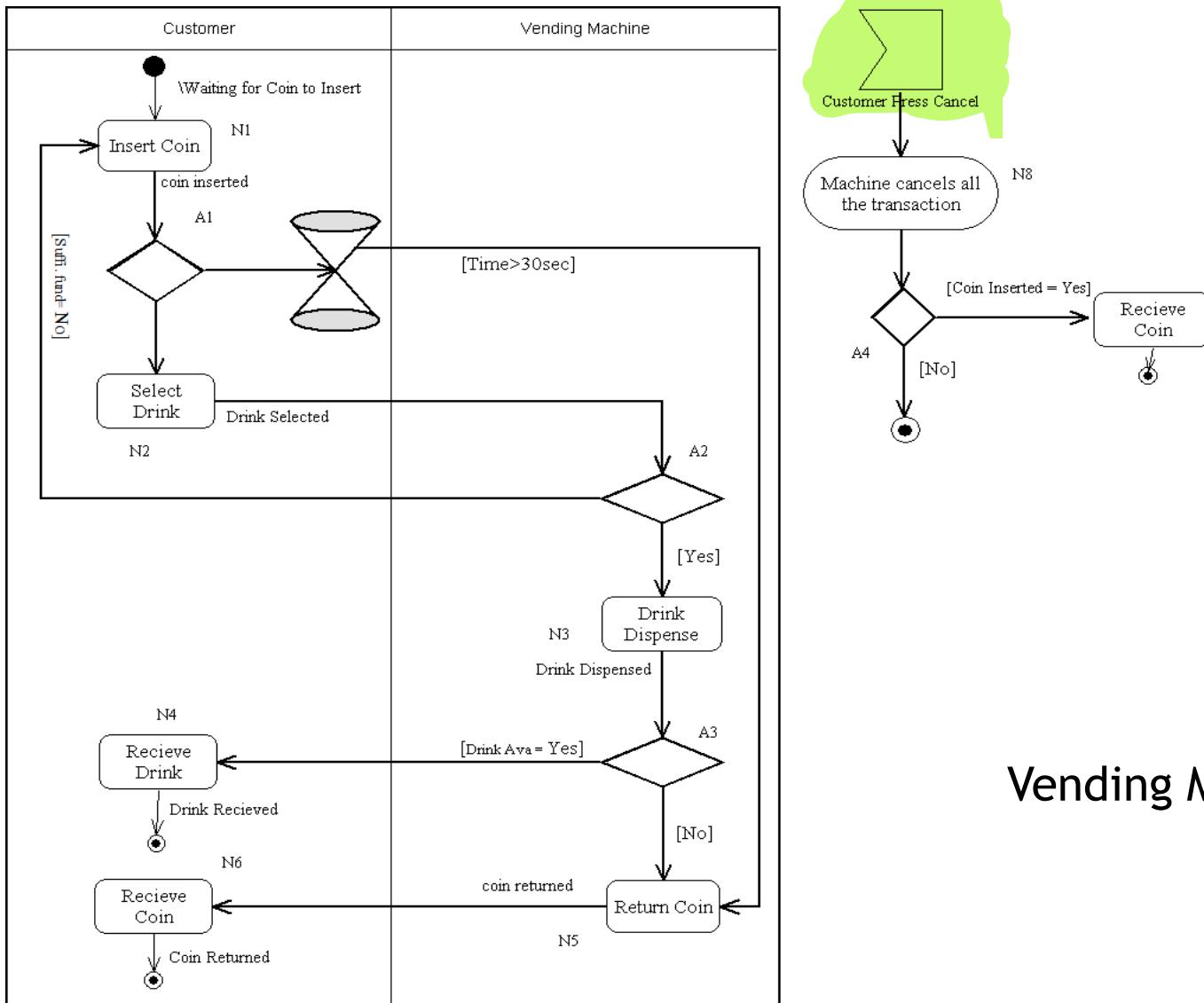
# Detailed Activity Diagram



# Activity Diagram with Swim Lane



# Activity Diagram with UML 2.0



Vending Machine



---

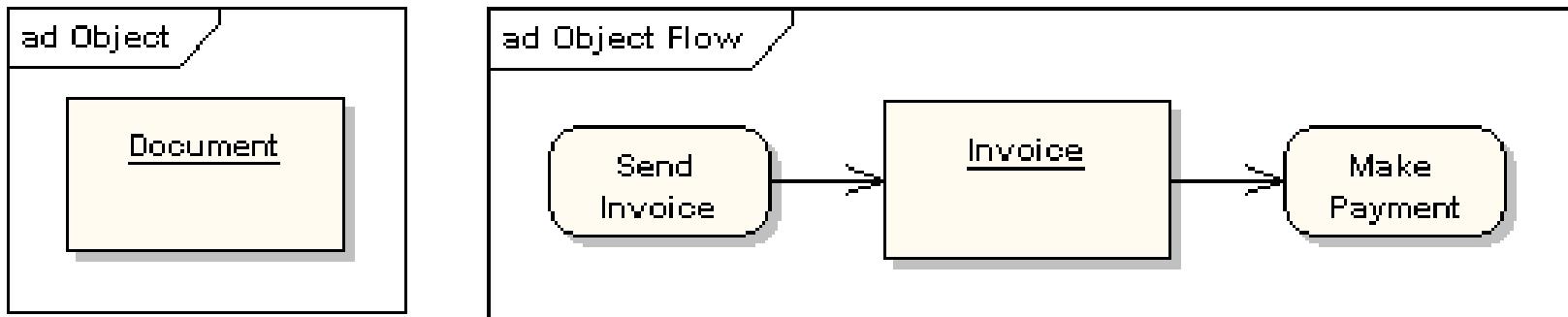
# Some more features in Activity Diagrams

---

# Object and Object Flow

---

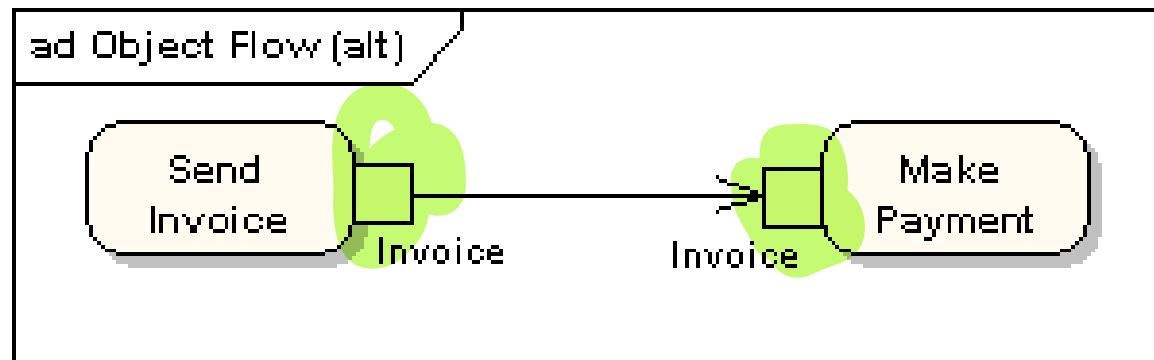
- An object flow is a path along which objects can pass. An object is shown as a rectangle
- An object flow is shown as a connector with an arrowhead denoting the direction the object is being passed.



# Input and Output Pin

---

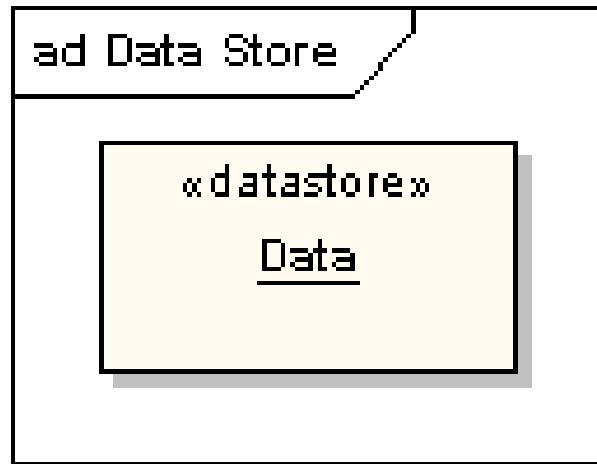
- An object flow must have an object on at least one of its ends. A shorthand notation for the above diagram would be to use input and output pins



# Data Store

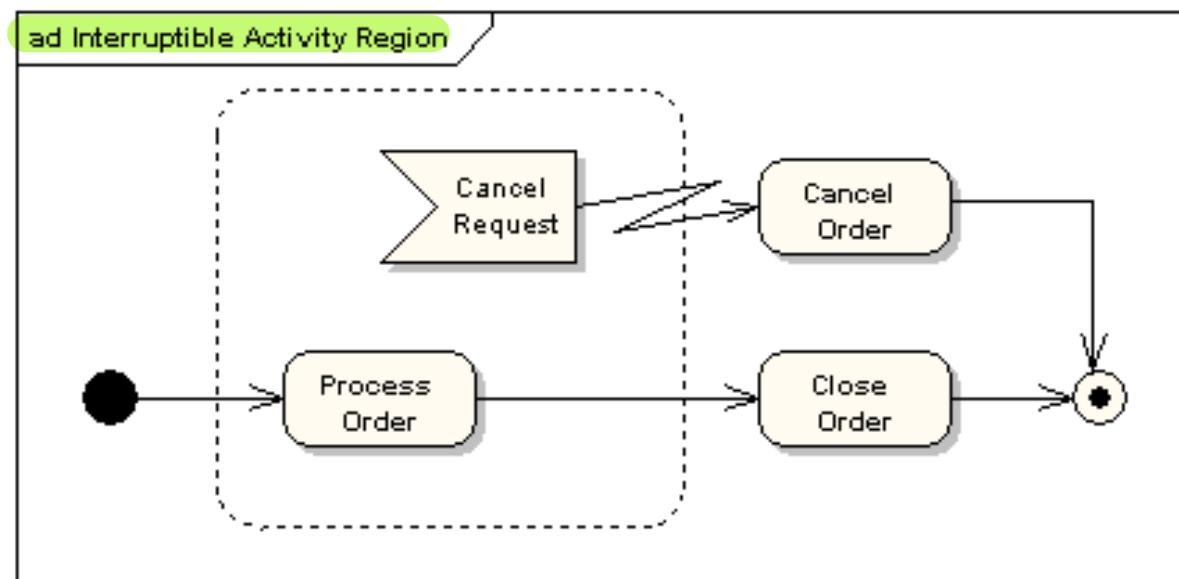
---

- A data store is shown as an object with the «datastore» keyword

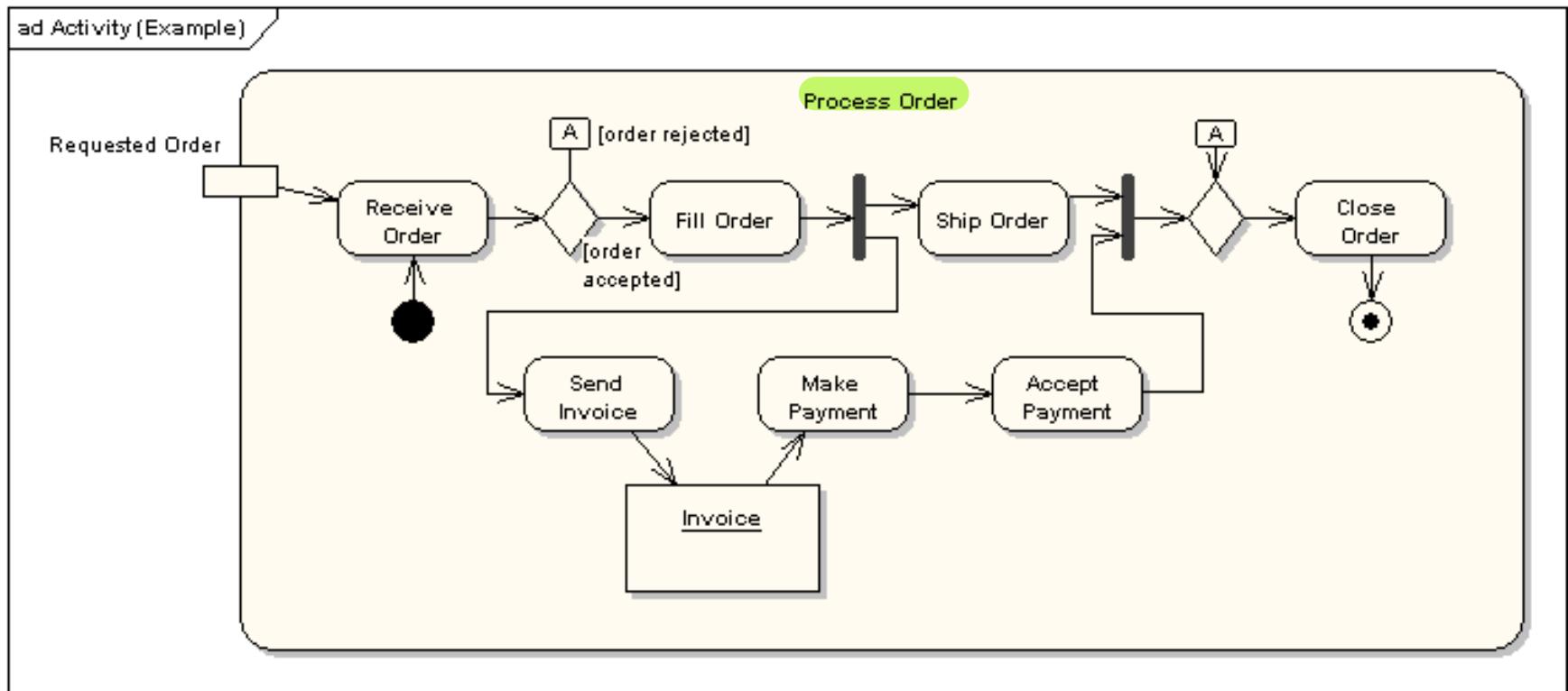


# Interruptible Activity Region

- An interruptible activity region surrounds a group of actions that can be interrupted. In the very simple example below, the Process Order action will execute until completion, when it will pass control to the Close Order action, unless a Cancel Request interrupt is received which will pass control to the Cancel Order action



# An Example



# Importance of Activity Diagram

---

- An activity diagram can depict a model in several ways
  - It can also depicts “Basic course of action” as well as “detailed courses”
  - Activity diagram can also be drawn that cross several use cases, or that address just a small portion of use case
  - Activity diagrams are normally employed in business process modeling. This is carried out during the initial stages of requirement analysis and specification
  - Activity diagrams can be very useful to understand the complex processing activities involving many components
  - The activity diagram can be used to develop interaction diagrams which help to allocate activities to classes
-



# Problems to Ponder

---

- How activity diagram related to flow chart? How it defers from flow chart?
  - How methods in classes and activities can be correlated?
-

# Exercises?

---

- Prepare an activity diagram for computing a restaurant bill. There should be a charge for each delivered item. The total amount should be subjected to a tax and a service charge of 18% for group of six or more. For smaller groups, there should be a blank entry for a gratuity according to the customer's discretion. Any coupons or gift certificates submitted by the customer should be subtracted.

## Activities

- Total items
- Add tax
- Credit coupons and certificates
- Customer determines gratuity [less than six]
- Add 18% [six or more]

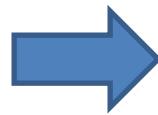
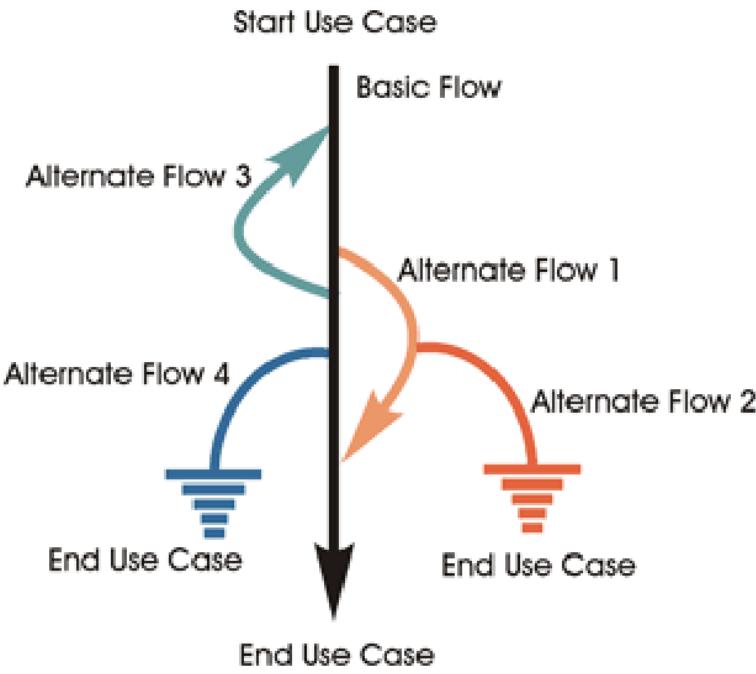
Activity diagram??

# Exercises?

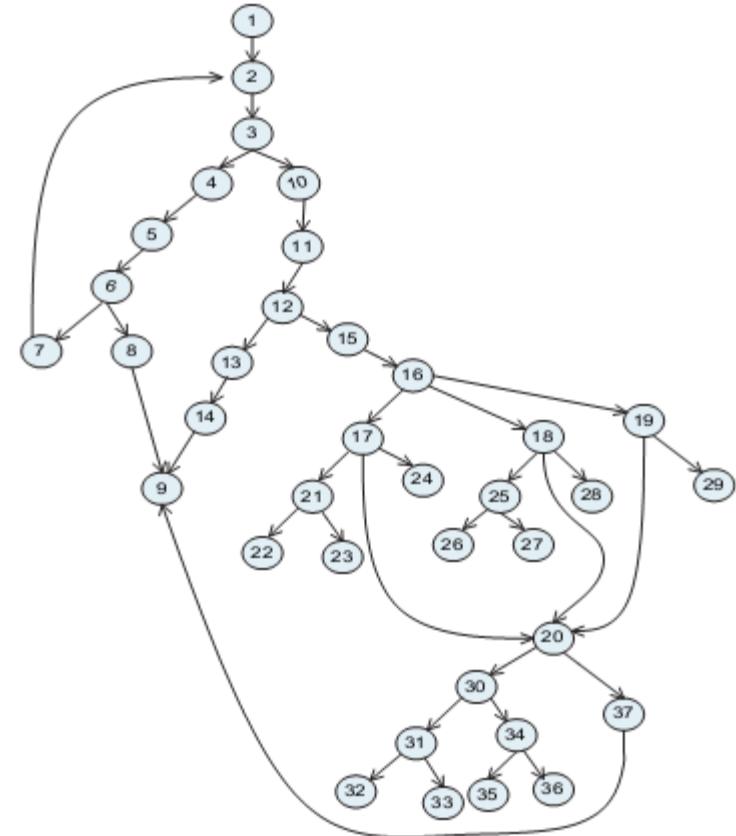
---

- Prepare an activity diagram that elaborates the details of logging into an email system. Note that entry of the user name and the password can occur in any order.
- Draw the activity diagrams for
  - Library Information System
  - Bank ATM

# Use Case to Activity: Graph - Example



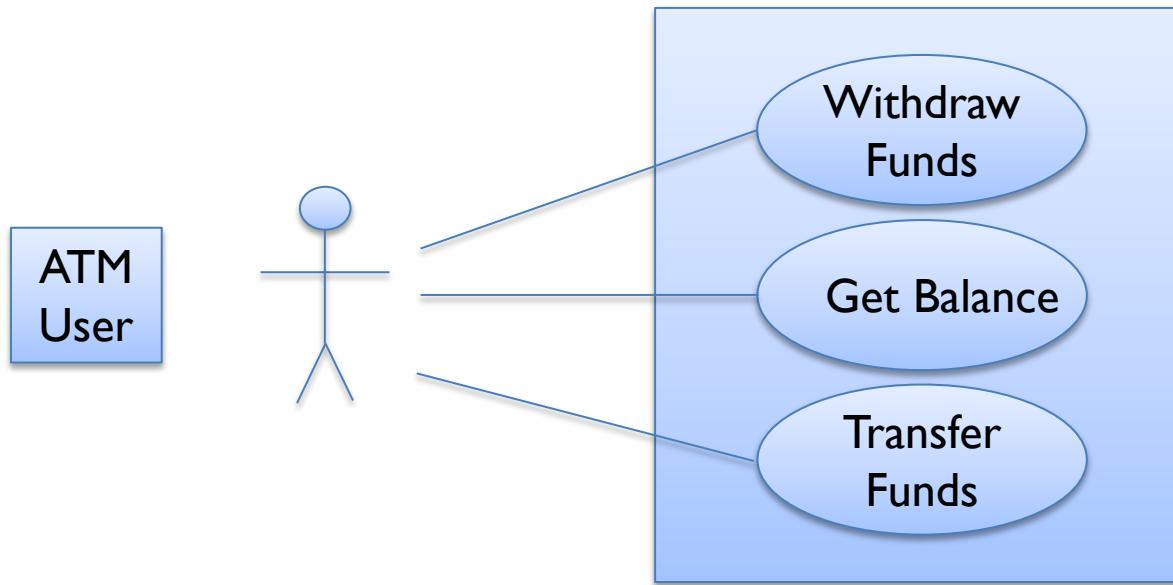
*Use Case Model*



*Graph*



# Simple Use Case Example



- **Actors** : Humans or software components that use the software being modeled
- **Use cases** : Shown as circles or ovals
- **Node Coverage** : Try each use case once ...

Use case graphs, by themselves, are not useful for testing

# Elaboration of ATM Use Case

---

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  - Customer inserts an ATM Card into the ATM Card Reader.
  - If the system can recognize the card, it reads the card number.
  - System prompts the customer for a PIN.
  - Customer enters PIN.
  - System checks the card's expiration date and whether the card has been stolen or lost.
  - If the card is valid, the system checks if the entered PIN matches the card PIN.
  - If the PINs match, the system finds out what accounts the card can access.
  - System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)



# Elaboration of ATM Use Case – 2/3

---

- Description (continued) :
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

# Elaboration of ATM Use Case – 3/3

---

- Alternatives :
    - If the system cannot recognize the card, it is ejected and the welcome message is displayed.
    - If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.
    - If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.
    - If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
    - If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.
    - If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.
    - If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.
    - If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.
    - If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.
  - Postcondition :
    - Funds have been withdrawn from the customer's account.
-

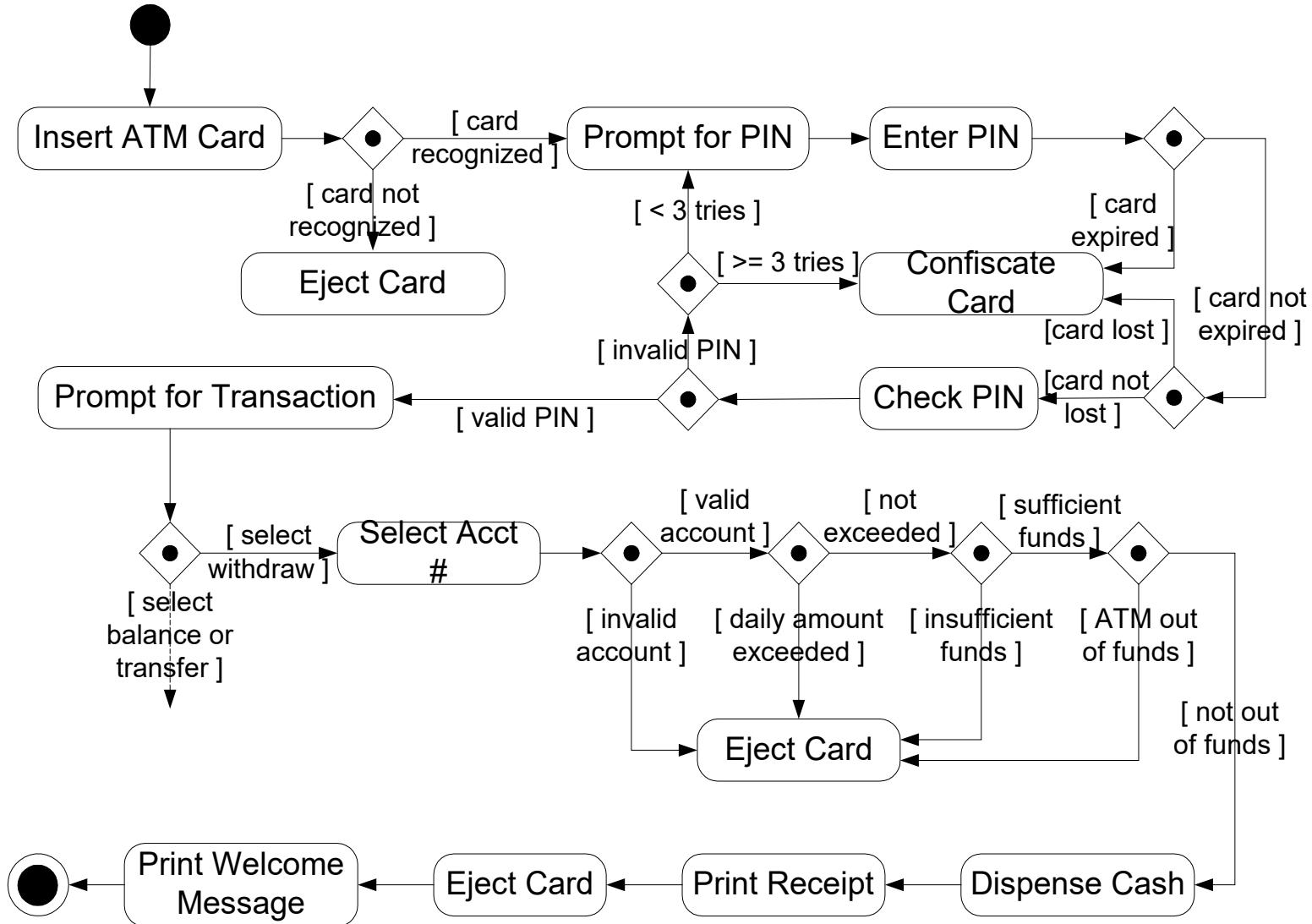


# Use Cases to Activity Diagrams

---

- Activity diagrams indicate **flow** among activities
- Activities should model **user level steps**
- Two kinds of nodes:
  - **Action states**
  - **Sequential branches**
- Use case descriptions become **action state nodes** in the activity diagram
- Alternatives are **sequential branch nodes**
- Flow among steps are **edges**
- Activity diagrams usually have some helpful characteristics:
  - Few loops
  - Simple predicates

# ATM Withdraw Activity Graph



# Exercise?

---

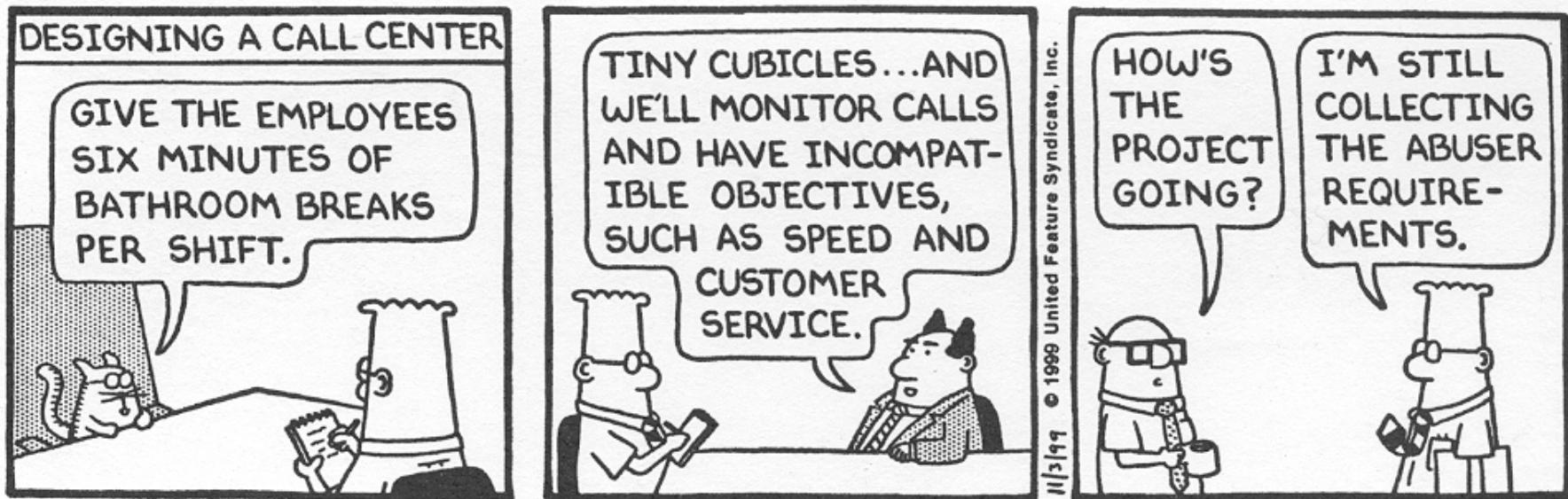
<b>Use case Name</b>	Process Sale	
<b>Actors</b>	Cashier, Catalog System, Inventory System	
	<b>Steps</b>	<b>Actions</b>
<b>Basic Flow</b>	1	Cashier starts a new sale.
	2	Cashier enters item identifier.
	3	POS System retrieve item information from the catalog system and, records sale line item and presents item description, price, and running total. Cashier repeats steps 2 until indicates done.
	4	POS System calculates and presents total price.
	5	Cashier tells Customer the total, and asks for payment.
	6	Customer pays and POS System handles payment
	7	POS System records completed sale and sends sale information to the external Inventory system for stock update.
	8	POS System prints receipt.
	9	Customer leaves with receipt and goods.
<b>Alternate Flow</b>	2.1	<b>Branching Actions</b> IF the item entered by the Cashier is Invalid identifier THEN POS System Indicate error and Cashier enters the item manually.
	7.1	If the items stock gets below a predefined minimum place a reposition order THEN Cashier deletes the item
	6.1	IF The Customer not have enough money THEN Customer asks the cashier to Cancel the transaction
	6.2	IF Customer says they intended to pay by cash but don't have enough cash THEN Customer uses an alternate payment method. The cashier tells customer to pay by card.
<b>Precondition</b>	Cashier is identified and authenticated.	
<b>Postcondition</b>	Sale is saved. Accounting and Inventory are updated. Receipt is generated. Payment authorization approvals are recorded.	

*Prepare an activity diagram??*

---

# Cartoon of the Day

---



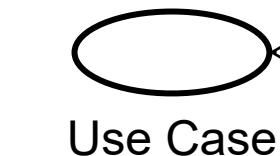
# Questions?

Next Lectures...

Sequence diagram: Identification of Domain Objects

# Use Case Realization

*Use Case Model*



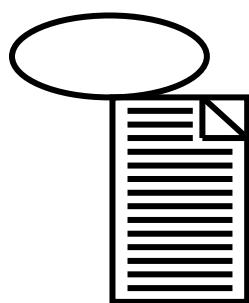
<<realizes>>

*Design Model*

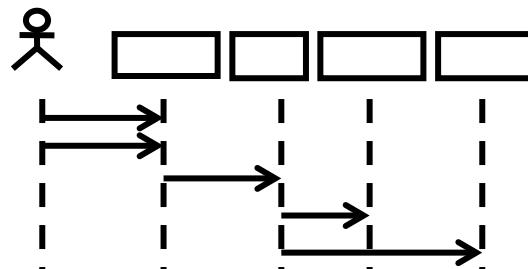


Use Case

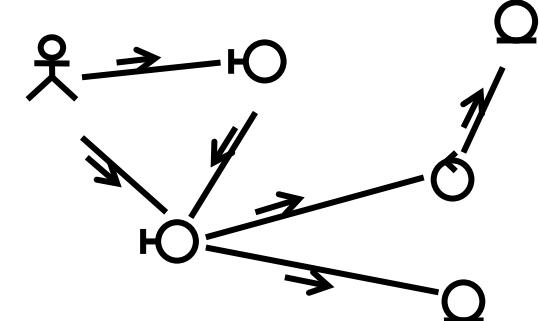
Use Case Realization



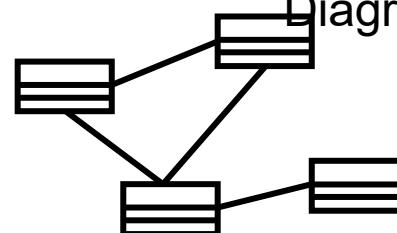
Use Case  
Realization  
Documentation



Sequence Diagrams



Collaboration  
Diagrams



Class Diagrams

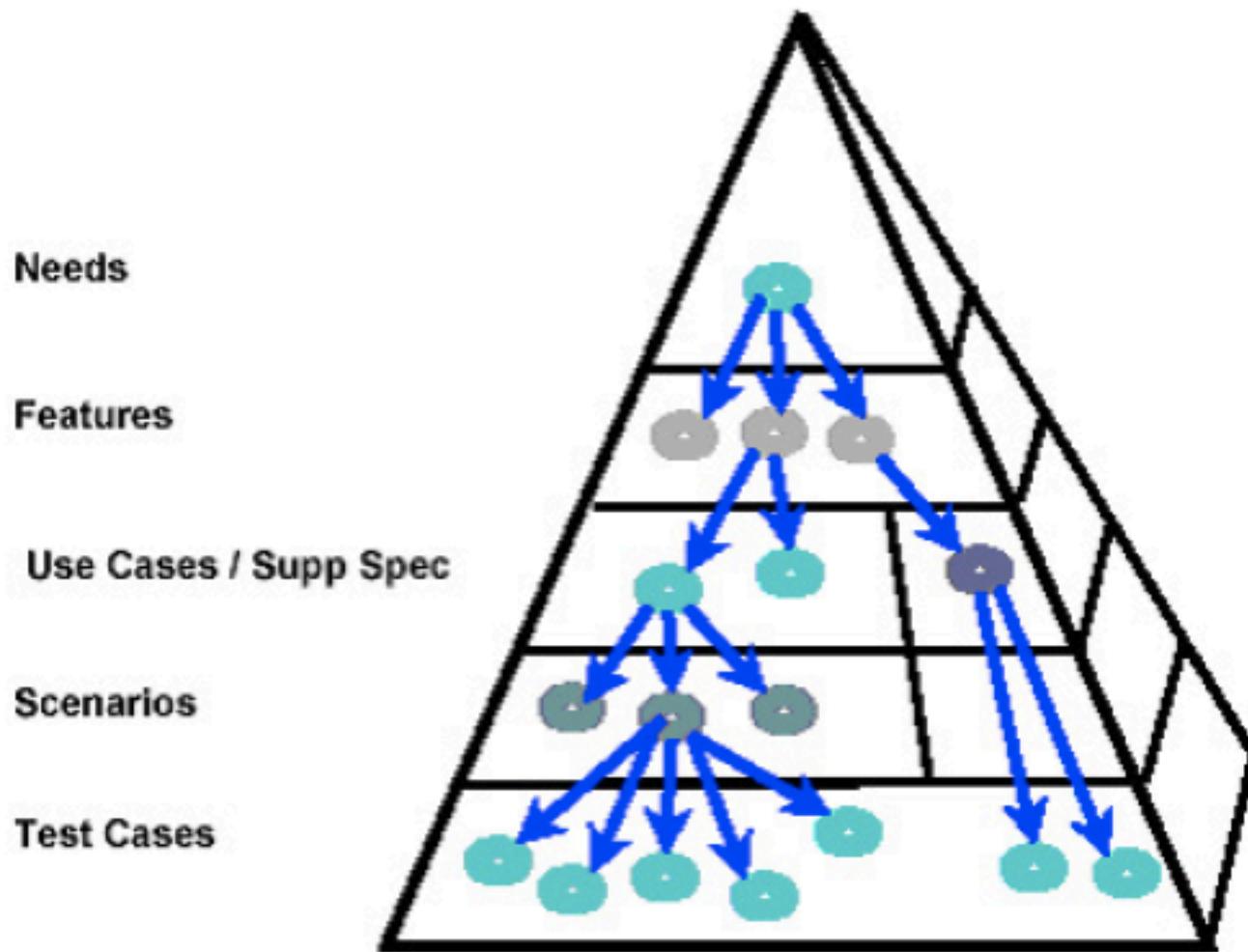


# Software Engineering

*Generating System-Level Test Cases from Use Cases*

# Requirements Pyramid

---



# Use Case Structure

## Use Case

### Name

1 Brief Description

2 Actors

3 Flow of Events

#### 3.1 Basic Flow

  3.1.1 Step 1

  3.1.2 Step 2...

#### 3.2 Alternative Flows

##### 3.2.1 Alternative Flow 1

    3.2.1.1 Step 1

    3.2.1.1 Step 2...

##### 3.2.1 Alternative Flow 2...

#### 3.3 Error Flows

##### 3.3.1 Error Flow 1...

    3.3.1.1 Step 1

    3.3.1.1 Step 2...

#### 3.4 Sub- Flows

##### 3.4.1 Sub-Flow 1...

    3.4.1.1 Step 1

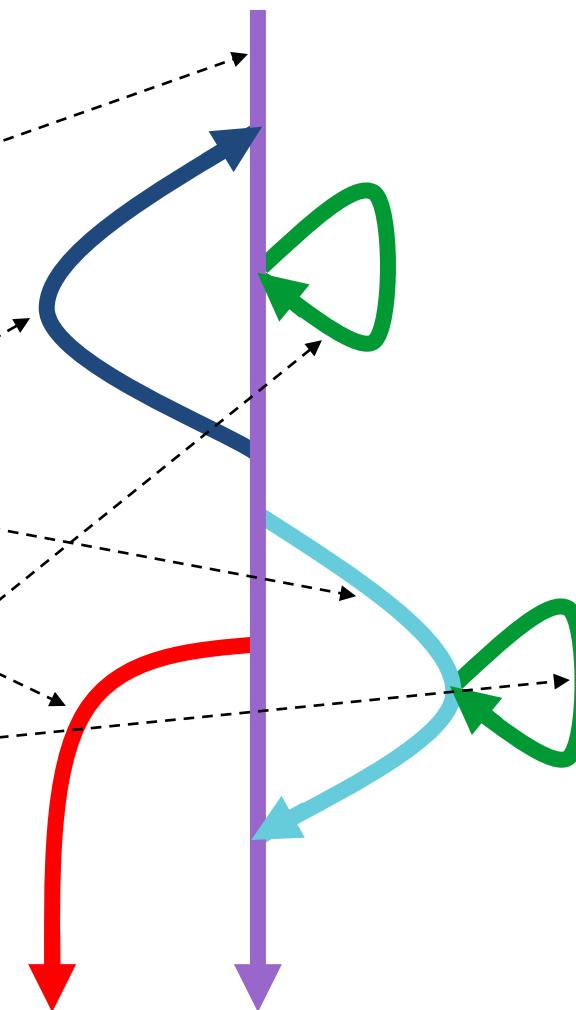
    3.4.1.1 Step 2...

4 Special requirements

5 Pre-Conditions

6 Post-Conditions

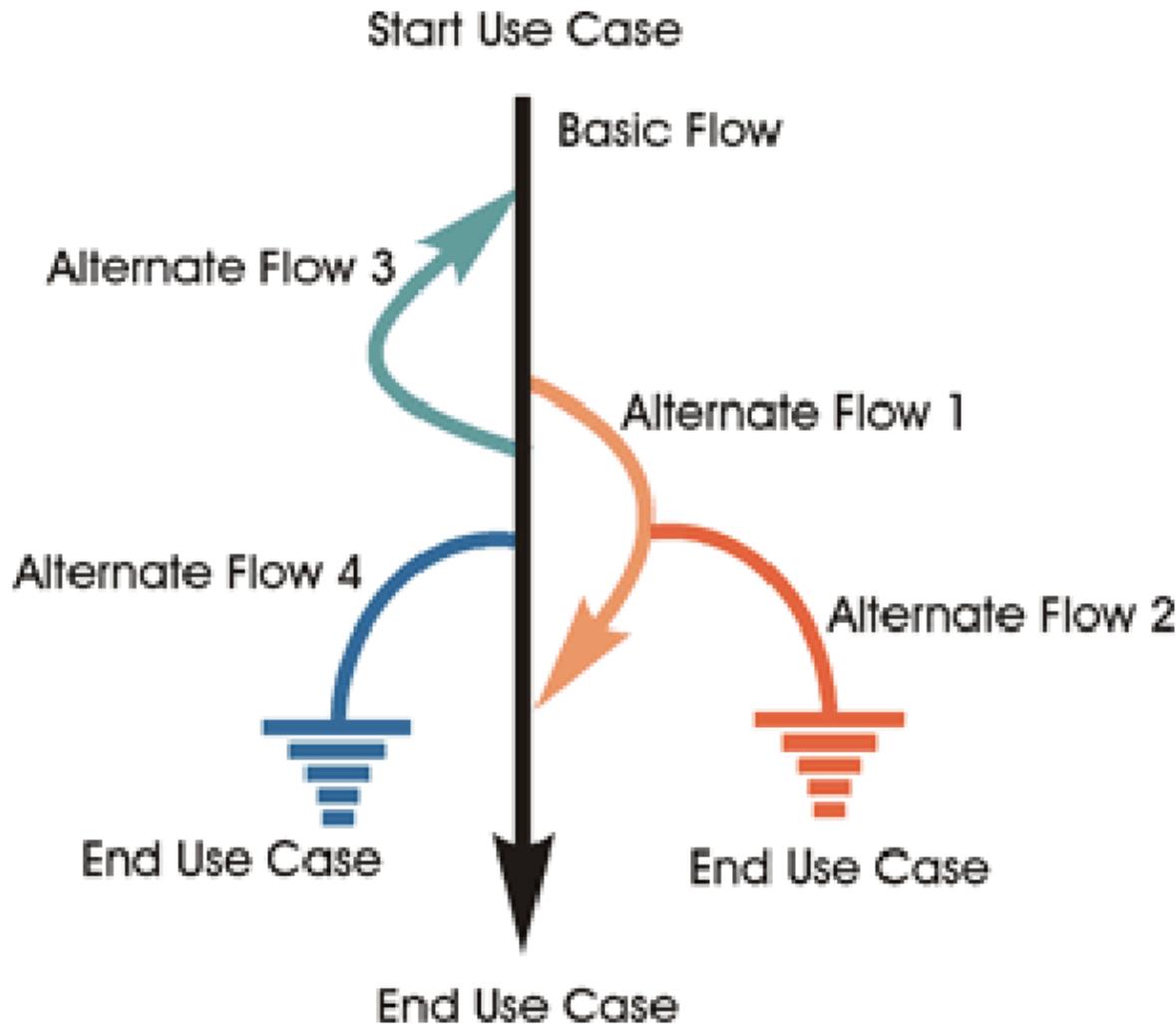
7 Extension Points



- Each path through the use case is a Use Case Scenario
- One use case may contain many Use Case Scenarios
- All Use Case Scenarios must be covered during development and testing but not necessarily in the same iteration

# Use Case Structure

---





# Test Case vs Test Scenarios

---

- A **TEST CASE** is a ***set of conditions or variables*** under which a tester (or developer) will determine whether a system under **test** satisfies requirements or works correctly.
- The process of developing **test cases** can also help find problems in the requirements or design of an application.
- Test Scenario gives the idea of what we have to test.
- Test Scenario answers “***What to be tested***”
- Test Case answers “***How to be tested***”

# Example

---

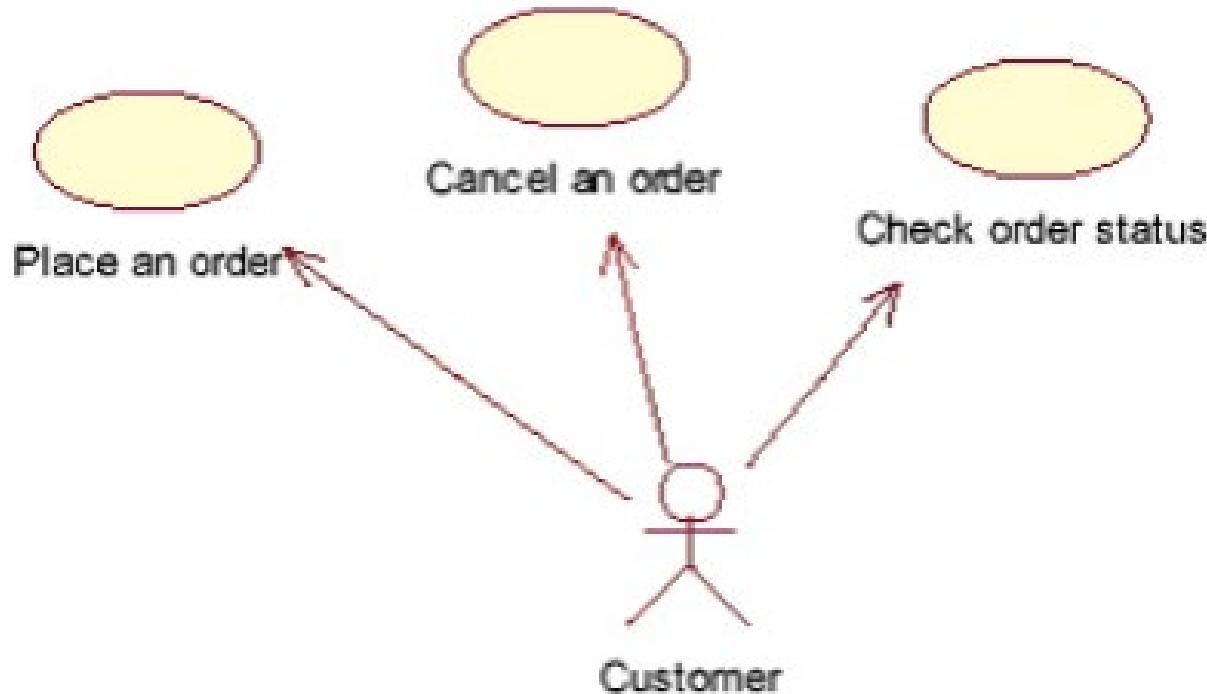
- Assume that we need to test the functionality of a login page of Gmail application.
- Test scenario for the Gmail login page functionality as follows:
- Test Scenario Example: **Verify the login functionality**
- Test Case Examples:
  - Test Case 1: Enter valid User Name and valid Password
  - Test Case 2: Enter valid User Name and invalid Password
  - Test Case 3: Enter invalid User Name and valid Password
  - Test Case 4: Enter invalid User Name and invalid Password

Test cases are derived from test scenarios

Test scenarios are derived from use cases

# Running Example

---



# Running Example

---

In our Online Bookstore project, the basic flow of the use case **place an order:**

1. B1 User enters web site address in the browser.  
System displays login page.
  2. B2 User enters an email address and a password.  
System confirms correct login, presents main page, and prompts for a search string.
  3. B3 User enters search string – partial name of a book.  
System returns all books matching search criteria.
  4. B4 User selects a book.  
System presents detailed information about a book.
  5. B5 User adds the book to a shopping cart.  
Shopping cart contents is presented to the user.
  6. B6 User selects "proceed to checkout" option.  
System asks for confirmation of a shipping address.
  7. B7 User confirms shipping address.  
System presents shipping options
-

# Running Example

---

8. B8 User selects shipping option.

Systems asks which credit card will be used.

9. B9 User confirms credit card that is stored in the system.

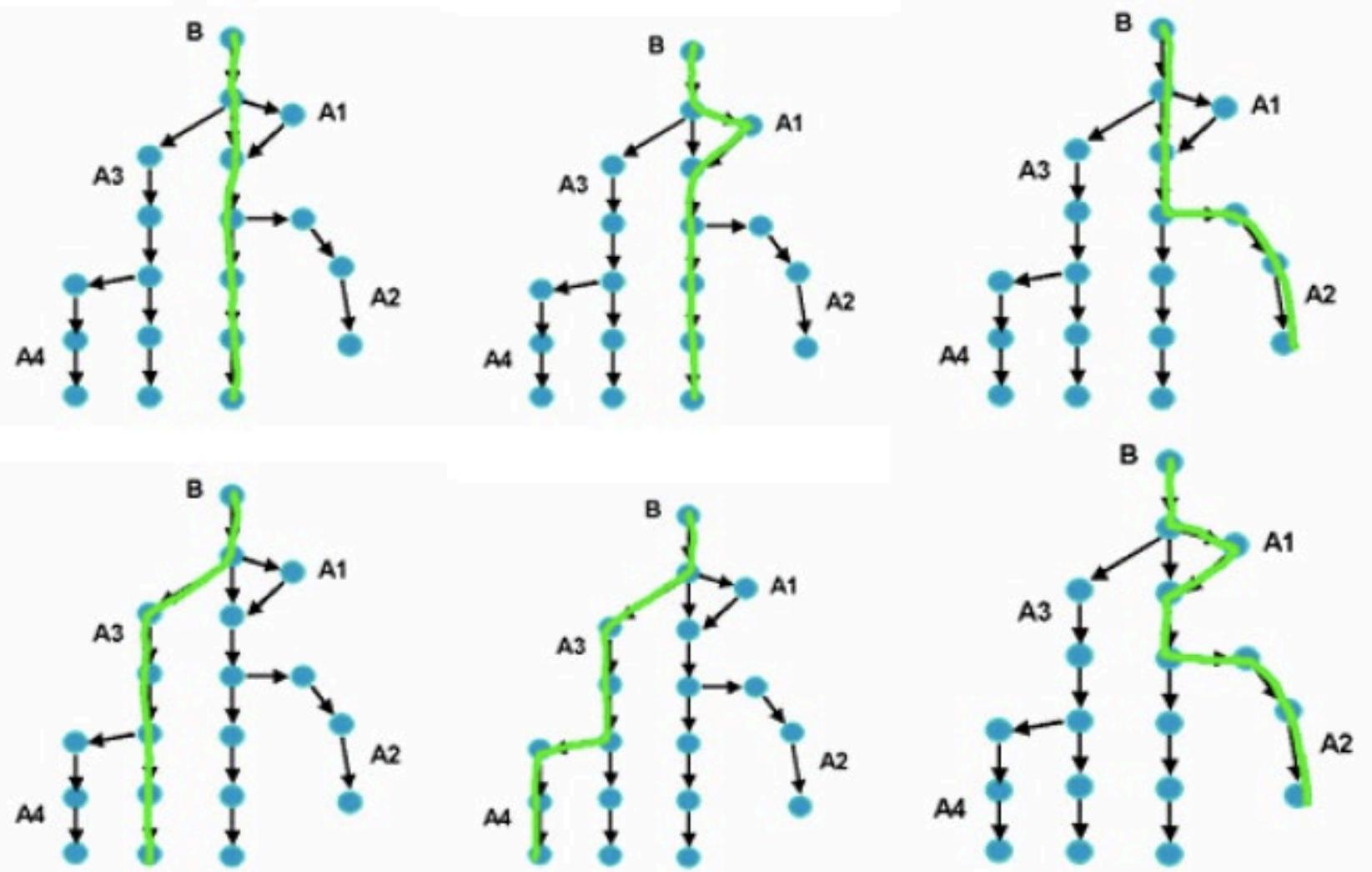
System asks for final confirmation to place an order.

10. B10 User places the order.

System returns a confirmation number.

A1	Unregistered user
A2	Invalid password
A3	No books matching search criteria were found
A4	Decline a book
A5	Continue shopping after storing a book in the shopping cart
A6	Enter a new address
A7	Enter a new credit card
A8	Cancel order

# Running Example – Finding Scenarios



# Running Example – Scenarios

---

Scenario 1 Basic Flow

Scenario 9 A8

Scenario 2 A1

Scenario 10 A1, A2

Scenario 3 A2

Scenario 11 A3, A4

Scenario 4 A3

Scenario 12 A4, A5

Scenario 5 A4

Scenario 13 A3, A5

Scenario 6 A5

Scenario 14 A6, A7

Scenario 7 A6

Scenario 15 A7, A8

Scenario 8 A7



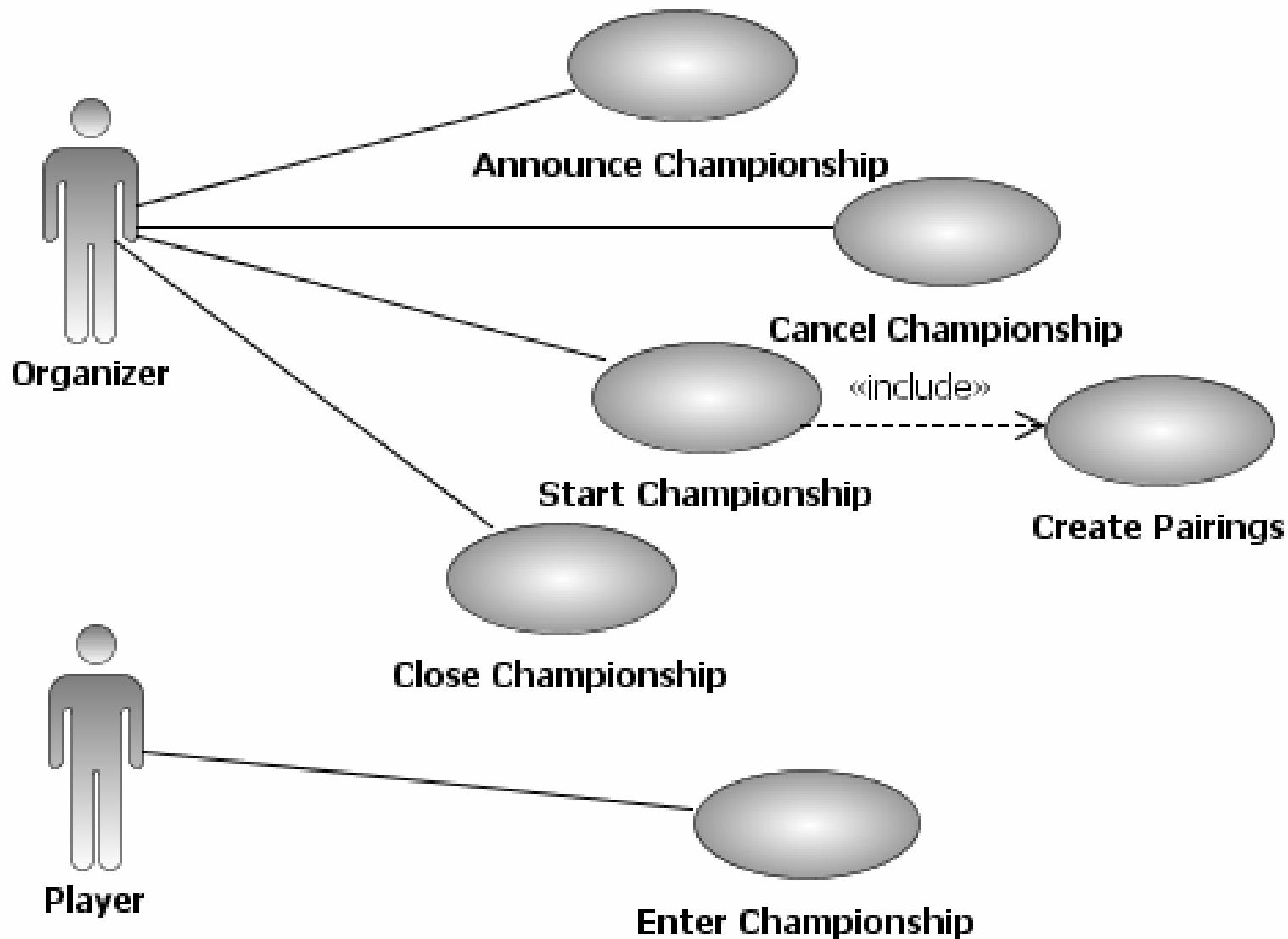
# Deriving Test Cases from Scenarios

---

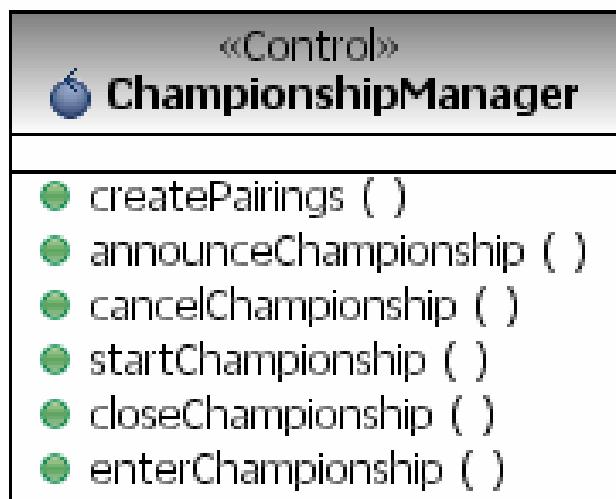
Now that you have all the scenarios, you need to get the test cases. There are four steps to do that:

- 1.Identify variables for each use case step
- 2.Identify significantly different options for each variable
- 3.Combine options to be tested into test cases
- 4.Assign values to variables

# Championship Management

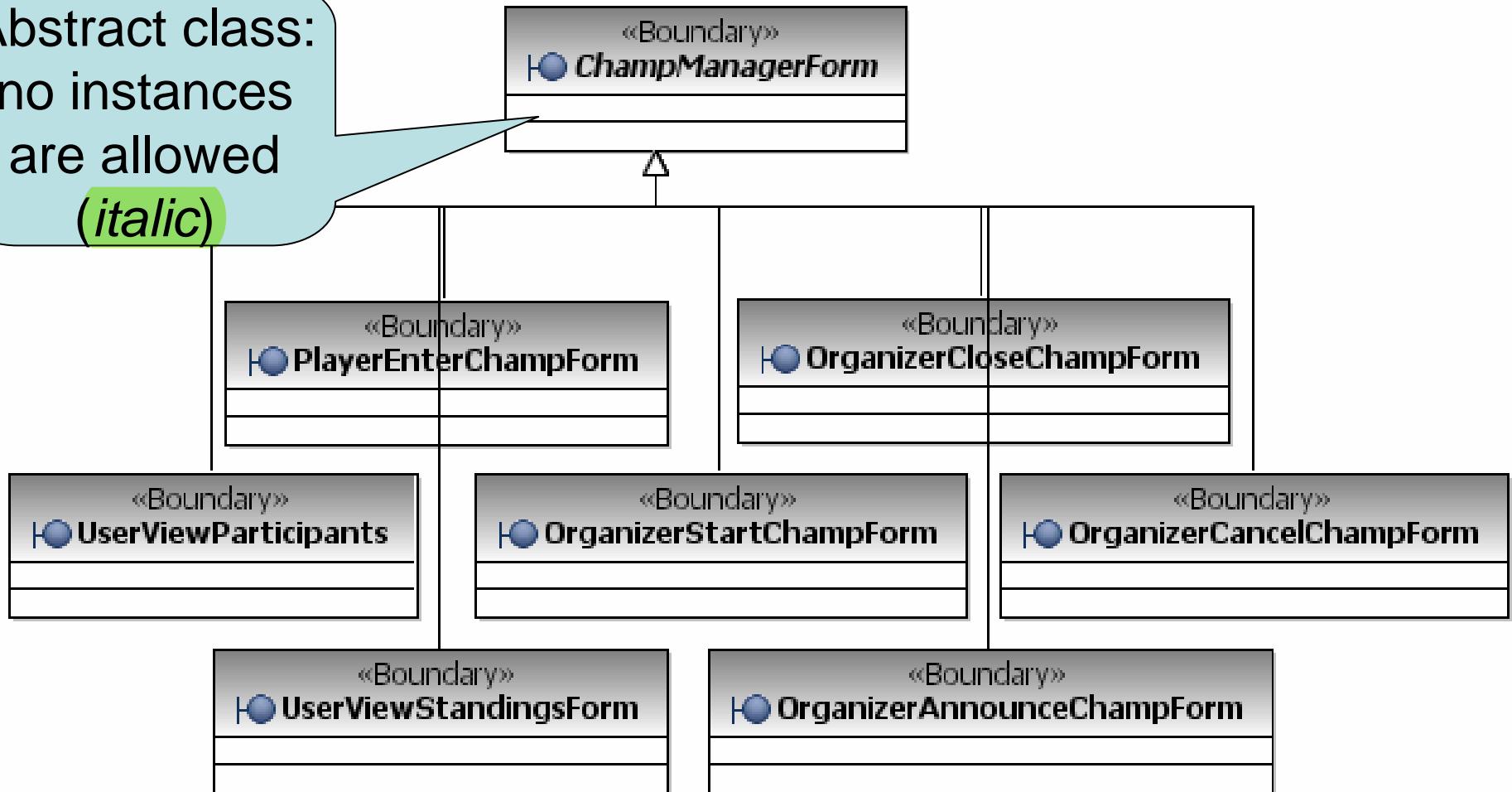


# Control and Entity Classes for Championship Management



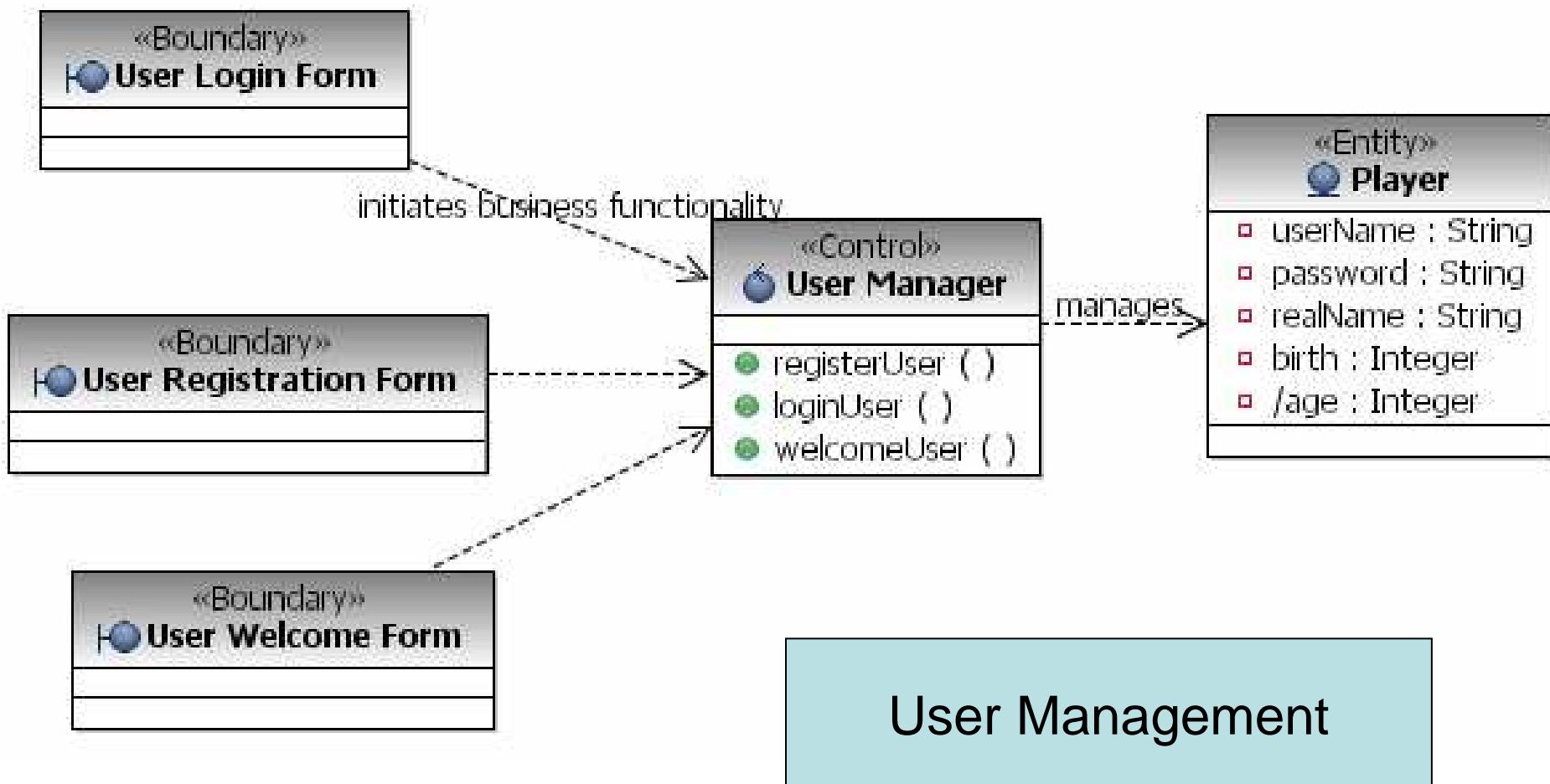
# Boundary Classes for Championship Management

Abstract class:  
no instances  
are allowed  
*(italic)*



Detailed design of boundary classes will come later

# Relationship between Analysis Classes





# IT 314: Software Engineering

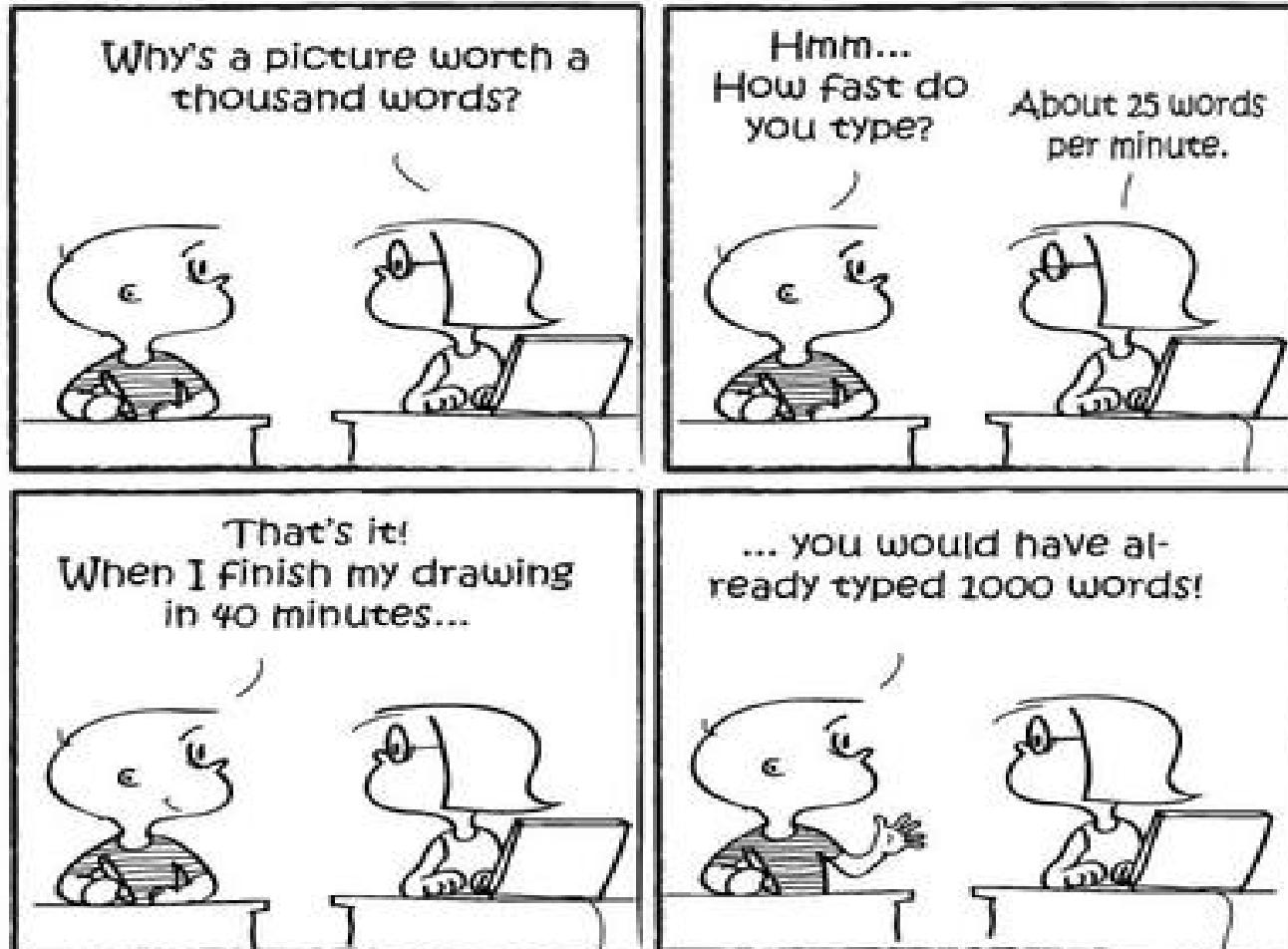
*UML - Introduction*



---

## Communication

### giggleBites



© 2009 cartoosh.com

---

# UML History

- OO languages appear mid 70's to late 80's (cf. Budd: communication and complexity)
  - Between '89 and '94, OO methods increased from 10 to 50.
  - Unification of ideas began in mid 90's.
    - Rumbaugh joins Booch at Rational '94
    - v0.8 draft Unified Method '95
      - Jacobson joins Rational '95
    - UML v0.9 in June '96
    - UML 1.0 offered to OMG in January '97
    - UML 1.1 offered to OMG in July '97
      - Maintenance through OMG RTF
    - UML 1.2 in June '98
    - UML 1.3 in fall '99
    - UML 1.5 <http://www.omg.org/technology/documents/formal/uml.htm>
    - UML 2.0 underway <http://www.uml.org/>
- } pre-UML
- } UML 1.x
- } UML 2.0
- IBM-Rational now has *Three Amigos*
  - Grady Booch - Fusion
  - James Rumbaugh – Object Modeling Technique (OMT)
  - Ivar Jacobson – Object-oriented Software Engineering: A Use Case Approach (Objectory)
  - ( And David Harel - StateChart)
- Rational Rose <http://www-306.ibm.com/software/rational/>

# Unified Modeling Language

---

- An effort by IBM (Rational) – OMG to standardize OOA&D notation
- Combine the best of the best from
  - Data Modeling (Entity Relationship Diagrams);  
Business Modeling (work flow); Object Modeling
  - Component Modeling (development and reuse - middleware, COTS/GOTS/OSS/...:)
- Offers vocabulary and rules for **communication**
- ***Not*** a process but a language

*de facto* industry standard

# UML is for Visual Modeling

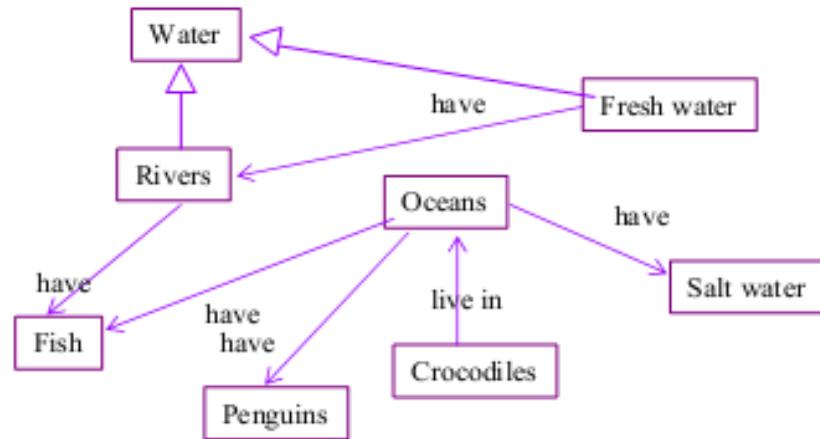
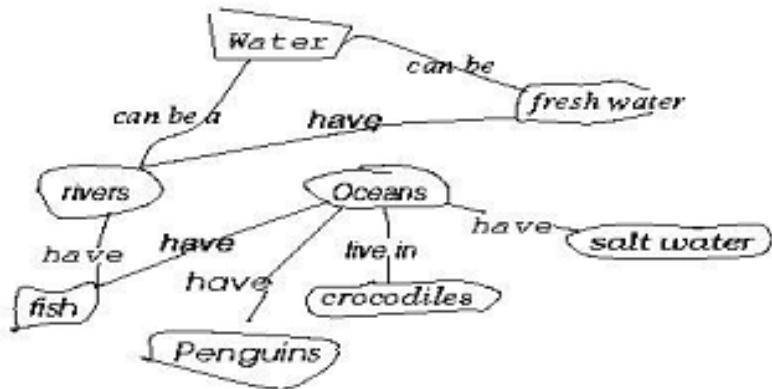
*A picture is worth a thousand words!*

- standard graphical notations: Semi-formal
- for modeling enterprise info. systems, distributed Web-based applications, real time embedded systems,  
...



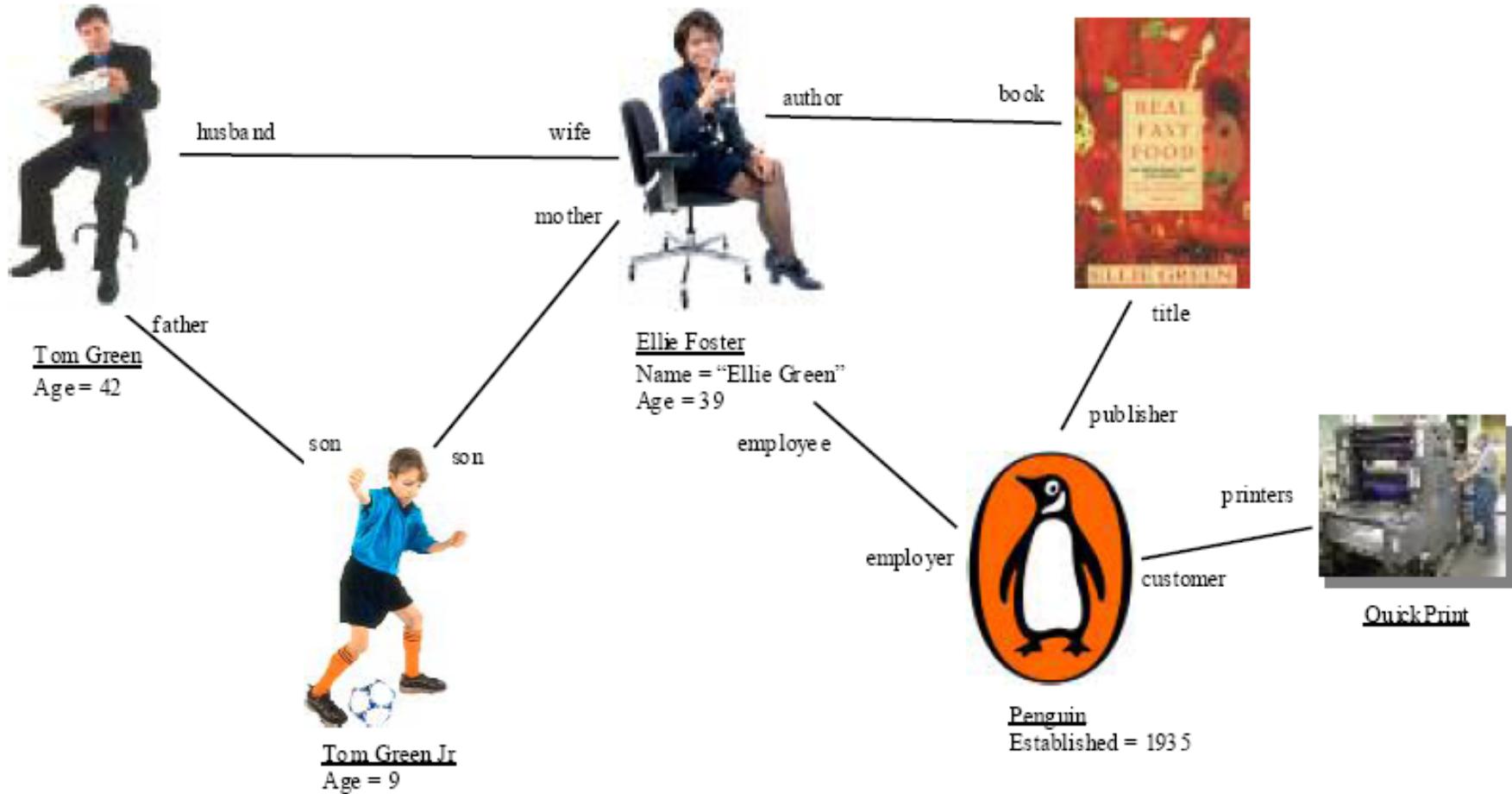
- **Specifying & Documenting:** models that are precise, unambiguous, complete
  - UML symbols are based on well-defined syntax and semantics.
  - analysis, architecture/design, implementation, testing decisions.
- **Construction:** mapping between a UML model and OOPL.

# Three (3) basic BUILDING blocks of UML



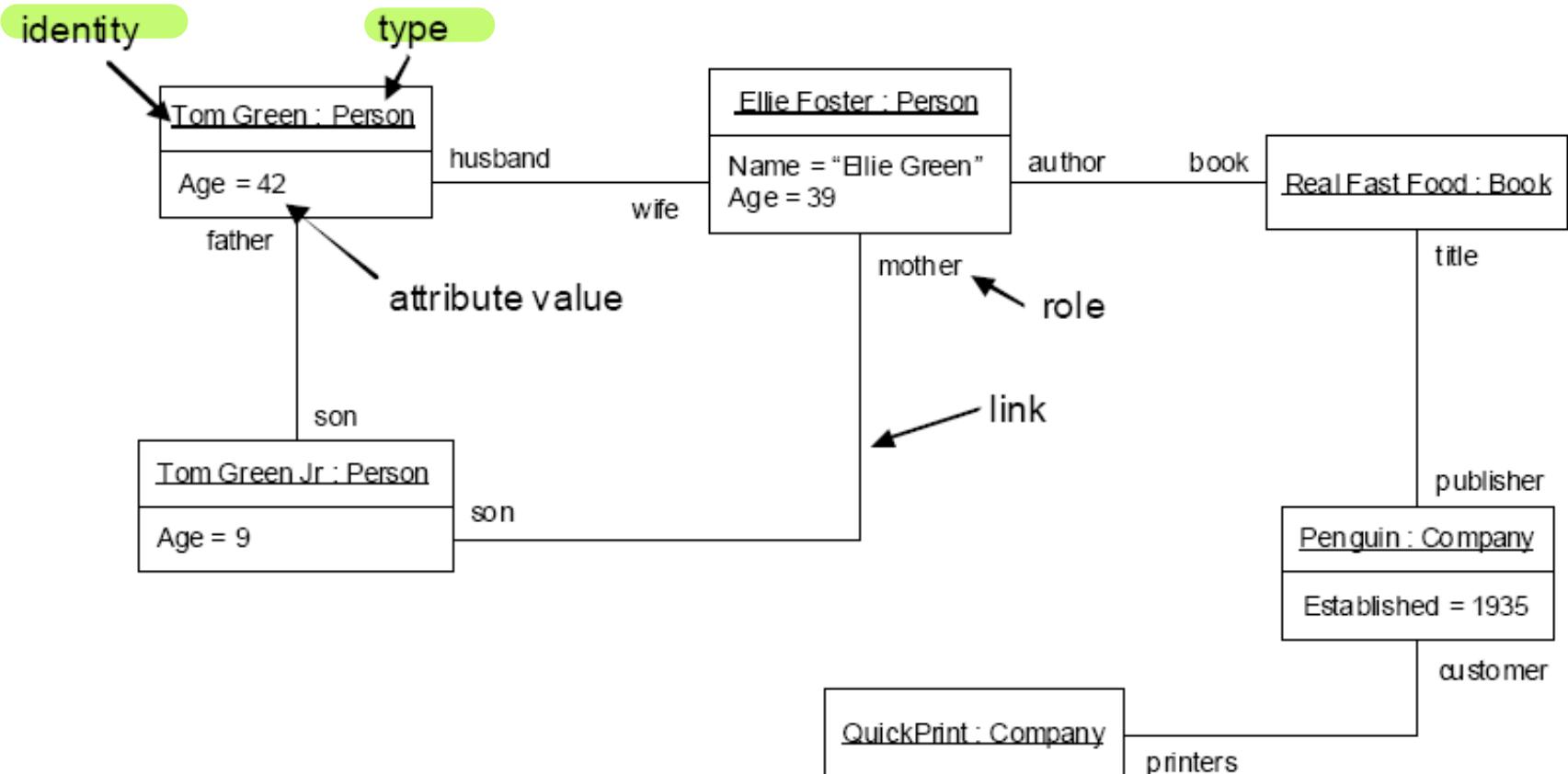
- Things - important modeling concepts
  - Relationships - tying individual things
  - Diagrams - grouping interrelated collections of things and relationships
- Just glance thru  
for now*

# Relationships between objects at some point



# Object Diagrams

More formally in UML



A connected graph: Vertices are things; Arcs are relationships/behaviors.

### UML 1.x: 9 diagram types.

#### Structural Diagrams

Represent the *static* aspects of a system.

- Class;
- Object
- Component
- Deployment

#### Behavioral Diagrams

Represent the *dynamic* aspects.

- Use case
- Sequence;
- Collaboration
- Statechart
- Activity

### UML 2.0: 12 diagram types

#### Structural Diagrams

- Class;
- Object
- Component
- Deployment
- Composite Structure
- Package

#### Behavioral Diagrams

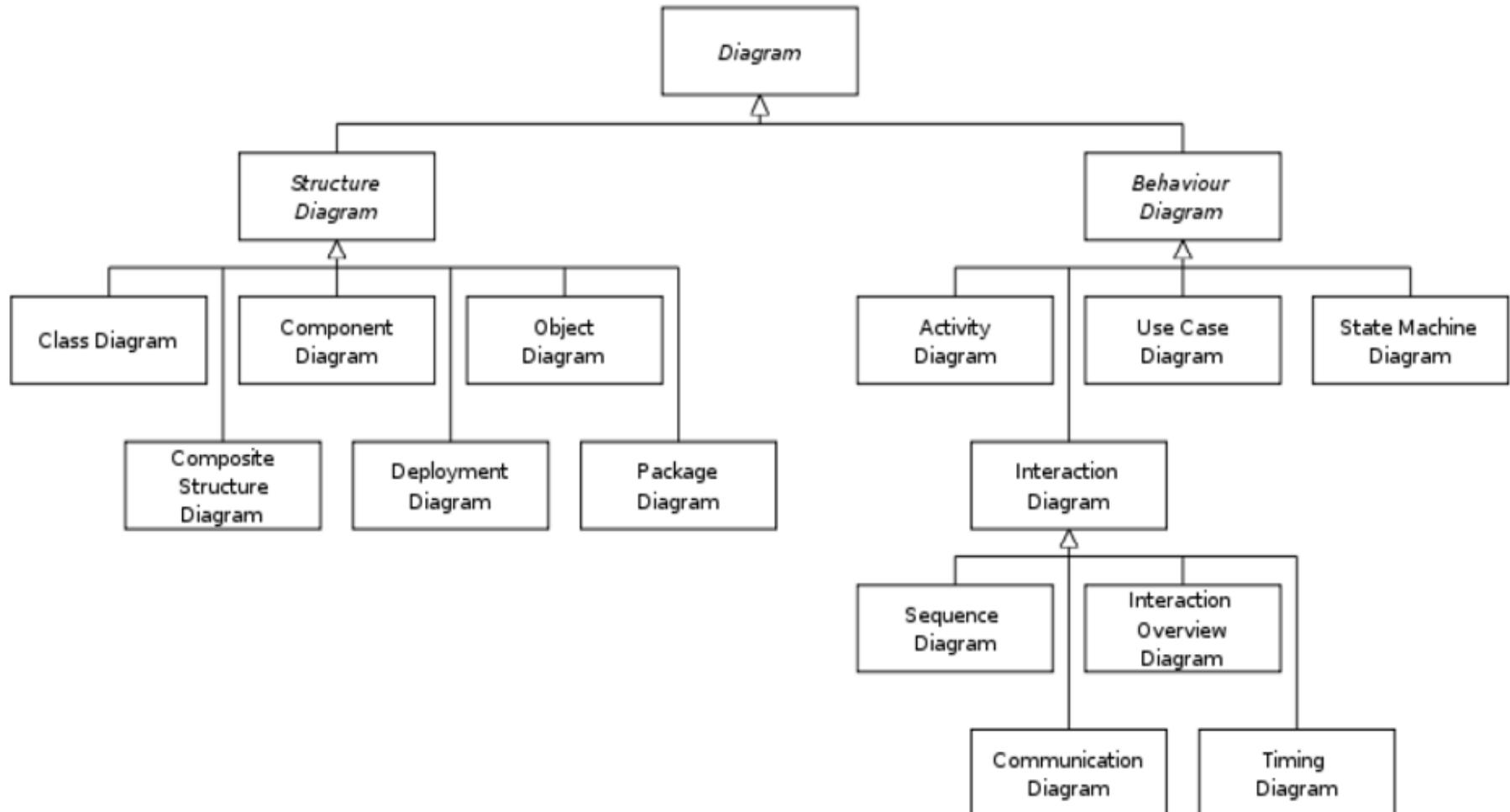
- Use case
- Statechart
- Activity

#### Interaction Diagrams

- Sequence;
- Communication
- Interaction Overview
- Timing

# ■ Unified Modeling Language (UML) - Diagrams

---



Source: Wikipedia

---

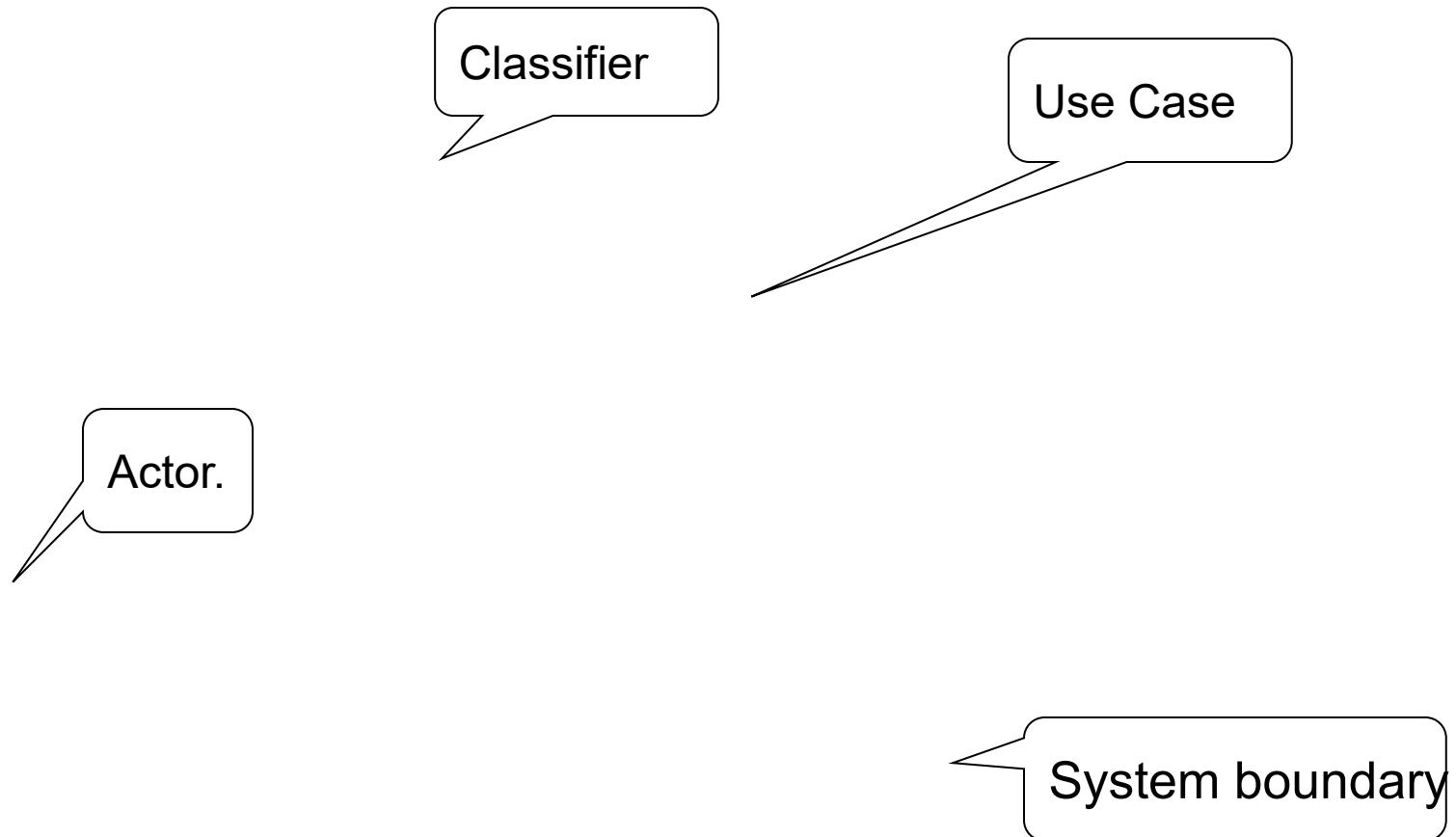
# UML Core Conventions

---

- All UML Diagrams denote graphs of nodes and edges
    - Nodes are entities and drawn as rectangles or ovals
    - Rectangles denote classes or instances
    - Ovals denote functions
  - Names of Classes are not underlined
    - SimpleWatch
    - Firefighter
  - Names of Instances are underlined
    - myWatch:SimpleWatch
    - Joe:Firefighter
  - An edge between two nodes denotes a relationship between the corresponding entities
-

# UML first pass: Use case diagrams

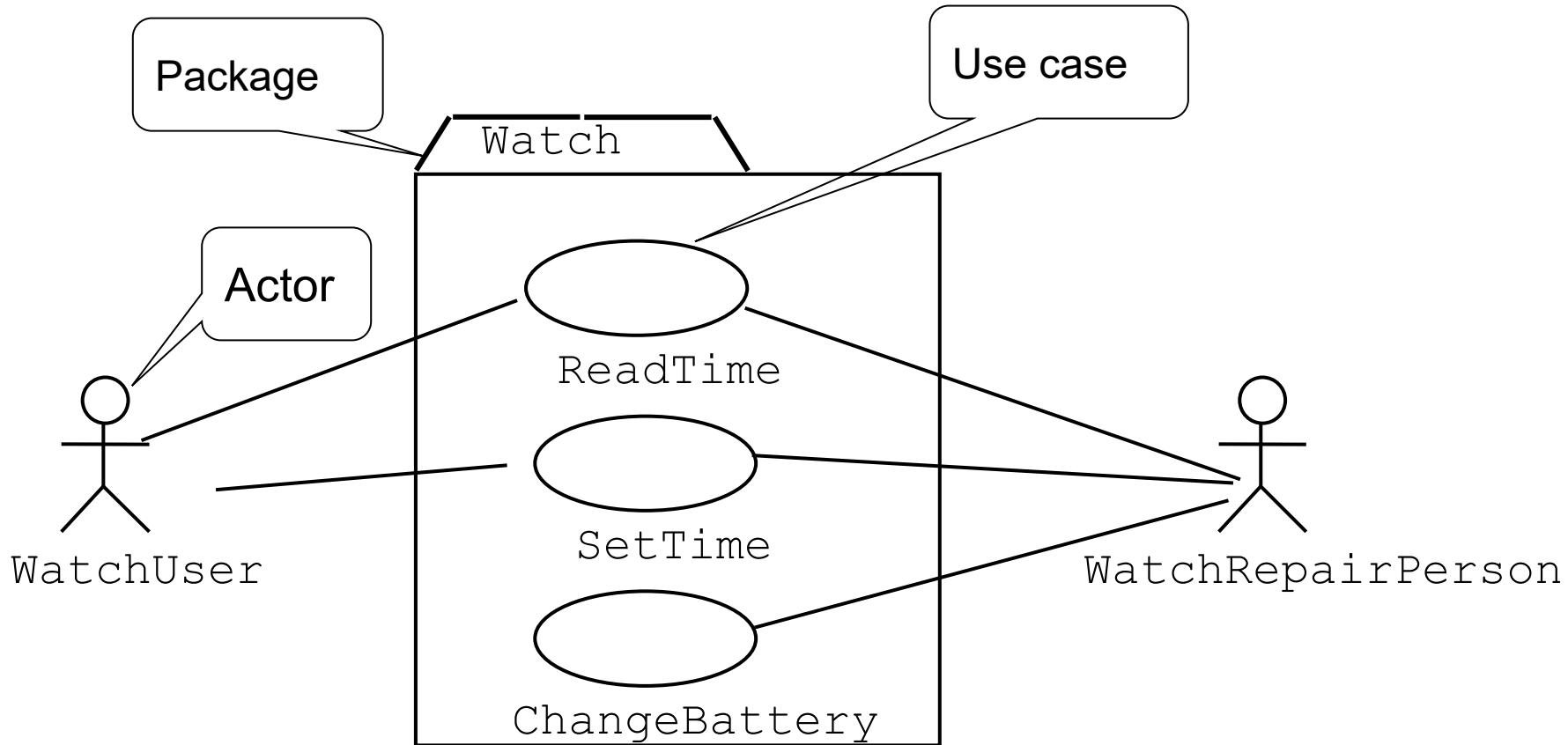
---



Use case diagrams represent the functionality of the system  
from user's point of view

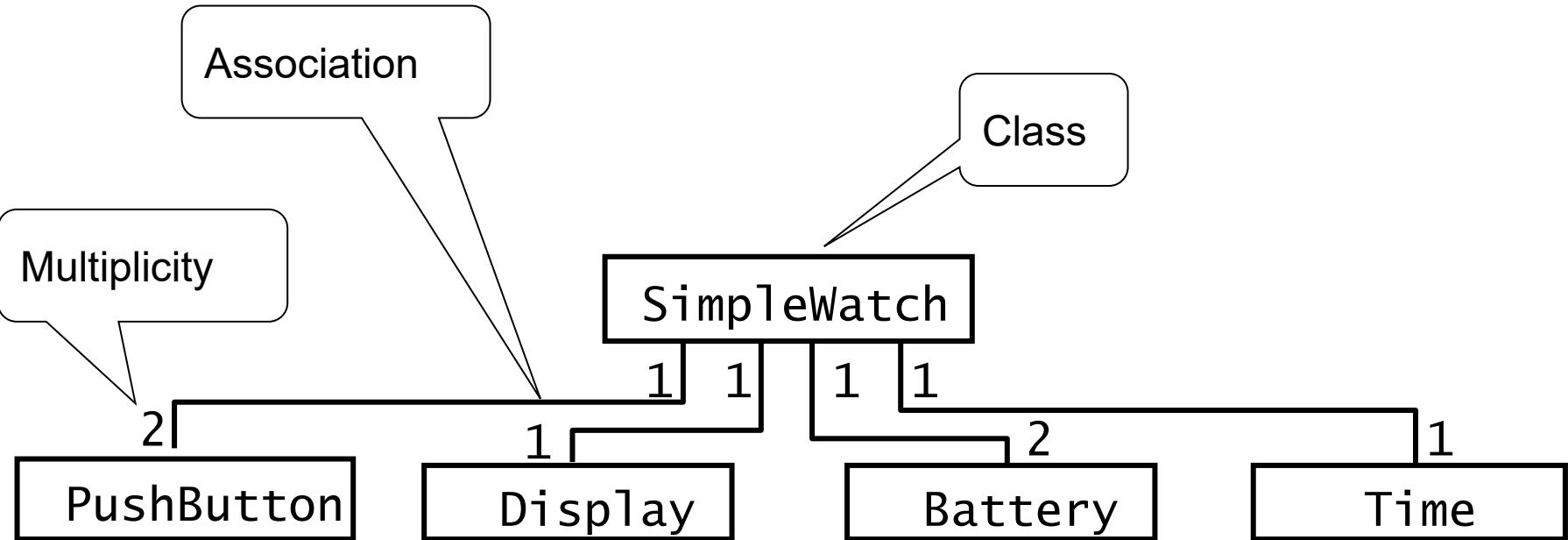
---

# Historical Remark: UML 1 used packages



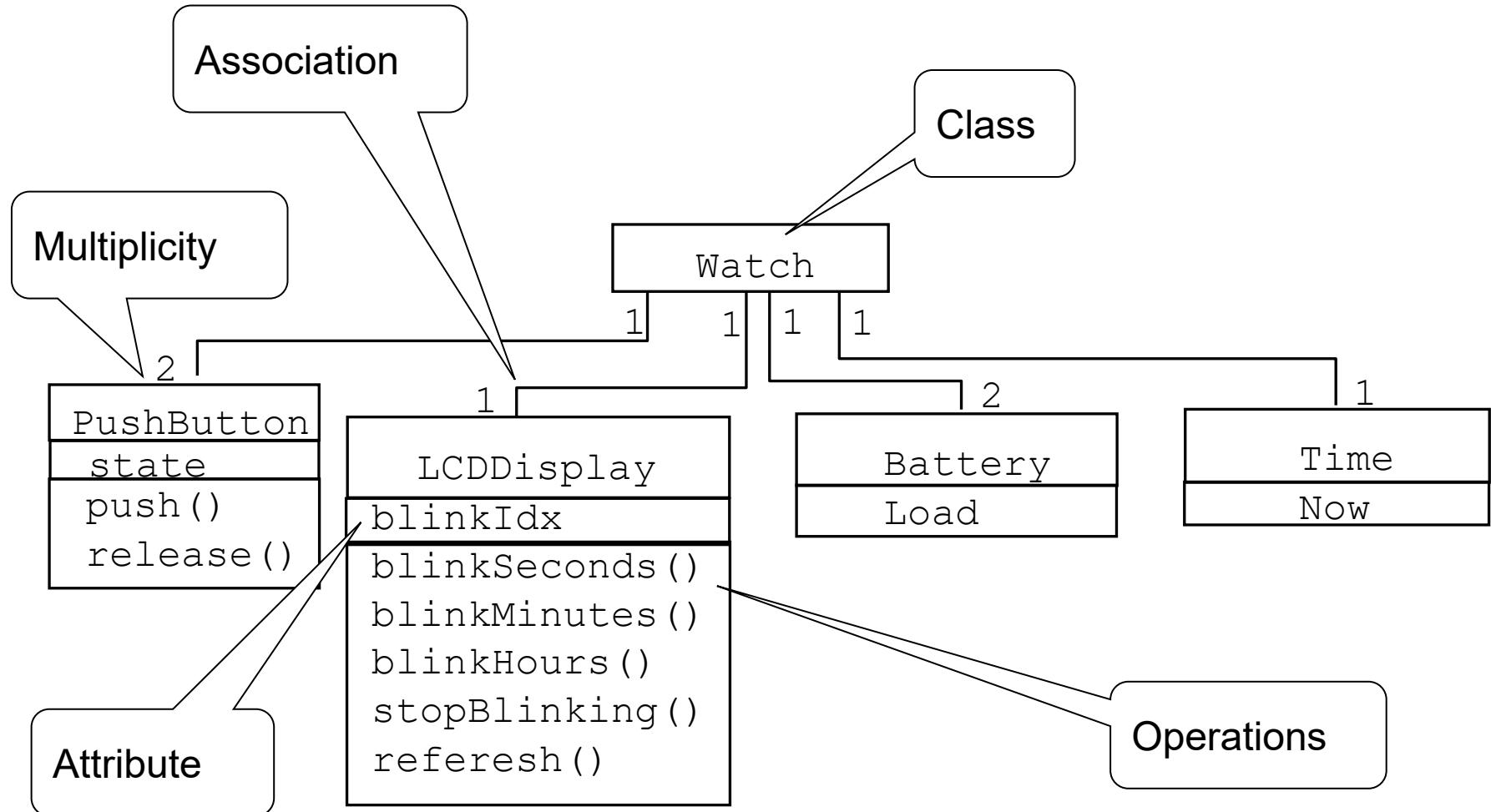
Use case diagrams represent the functionality of the system from user's point of view

# UML first pass: Class diagrams

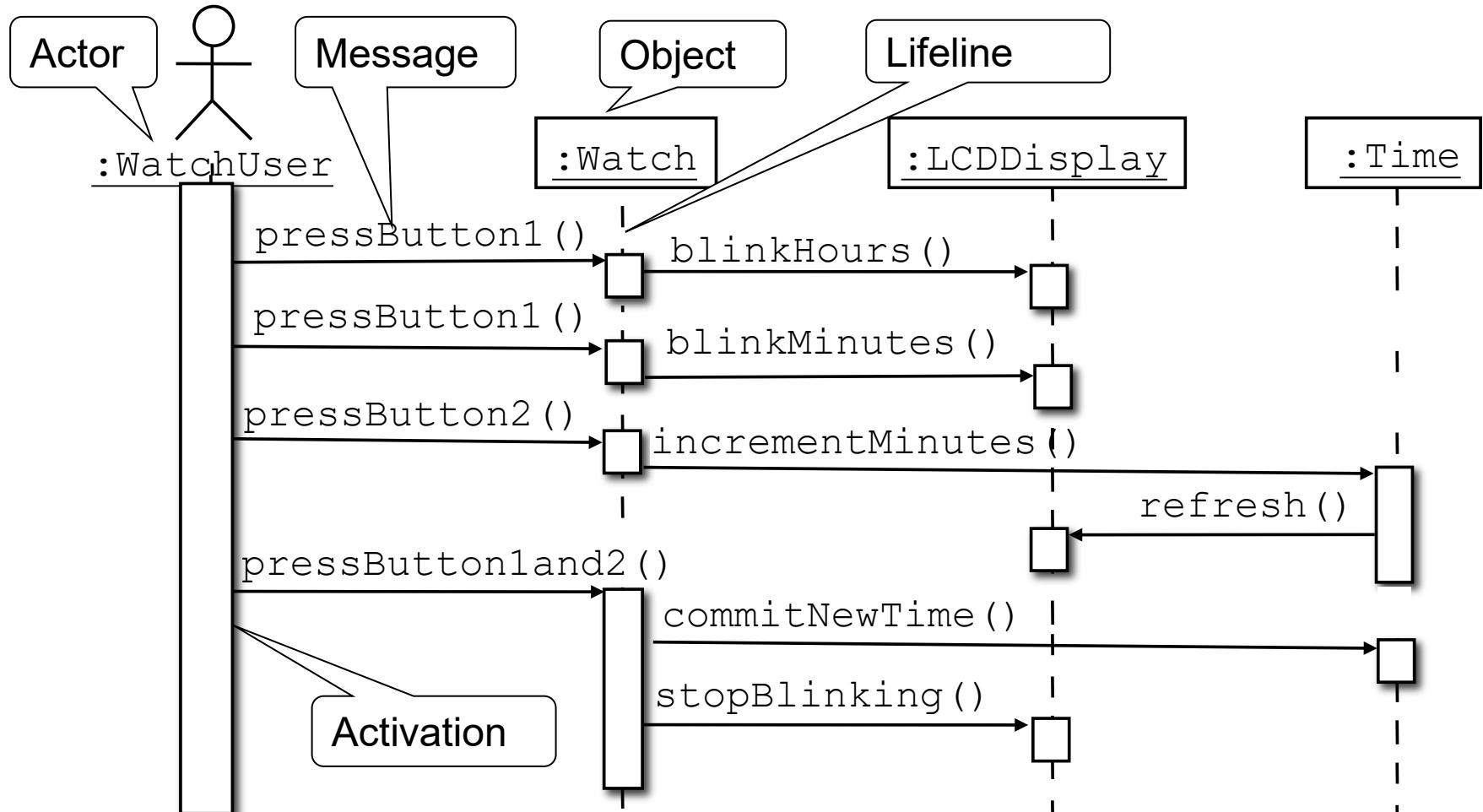


Class diagrams represent the structure of the system

# UML first pass: Class diagrams

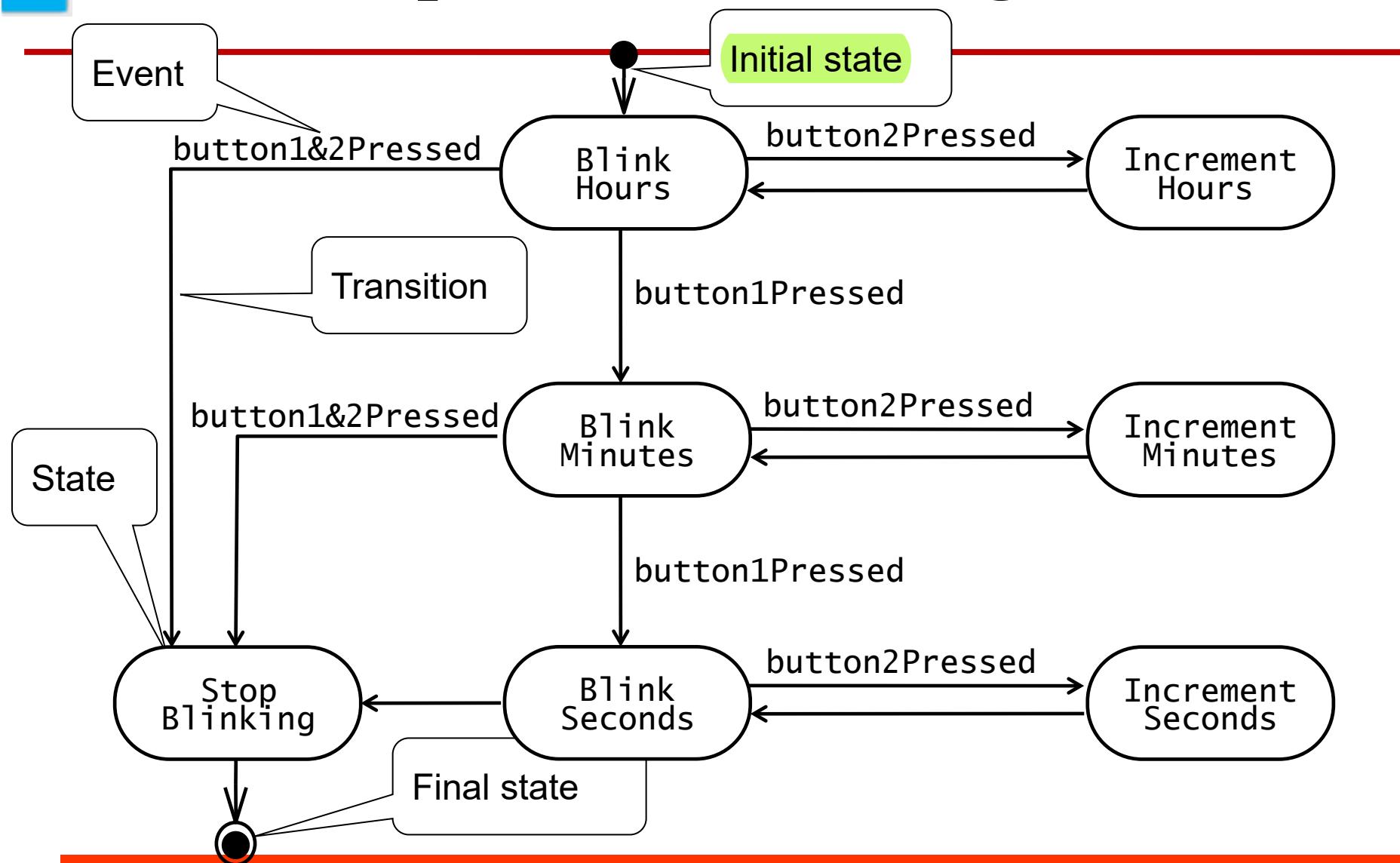


# UML first pass: Sequence diagram



Sequence diagrams represent the behavior of a system as messages (“interactions”) between *different objects*

# UML first pass: Statechart diagrams



Represent behavior of a *single object* with interesting dynamic behavior.



# UML Basic Notation Summary

---

- UML provides a wide variety of notations for modeling many aspects of software systems
- Today we concentrated on a few notations:
  - Functional model: Use case diagram
  - Object model: Class diagram
  - Dynamic model: Sequence diagrams, state chart

# Diagrams in UML

---

The UTD wants to computerize its registration system

- The Registrar sets up the curriculum for a semester
- Students select 3 core courses and 2 electives
- Once a student registers for a semester, the billing system is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- Professors use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users of the registration system are assigned passwords which are used at logon validation

***What's most important?***

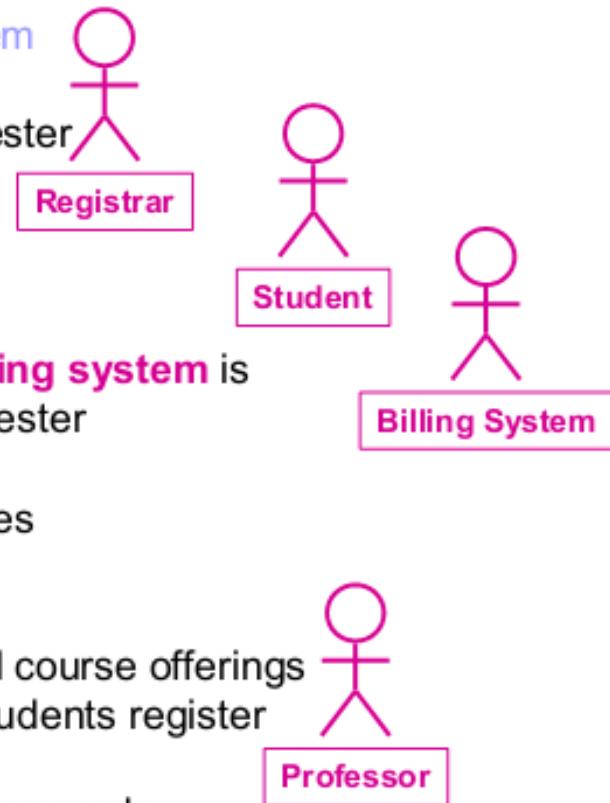
---

# Diagrams in UML – Actors in Use case

- An **actor** is someone or some thing that must interact with the system under development

The UTD wants to computerize its registration system

- The **Registrar** sets up the curriculum for a semester
- Students** select 3 core courses and 2 electives
- Once a student registers for a semester, the **billing system** is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- Professors** use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users** of the registration system are assigned passwords which are used at logon validation

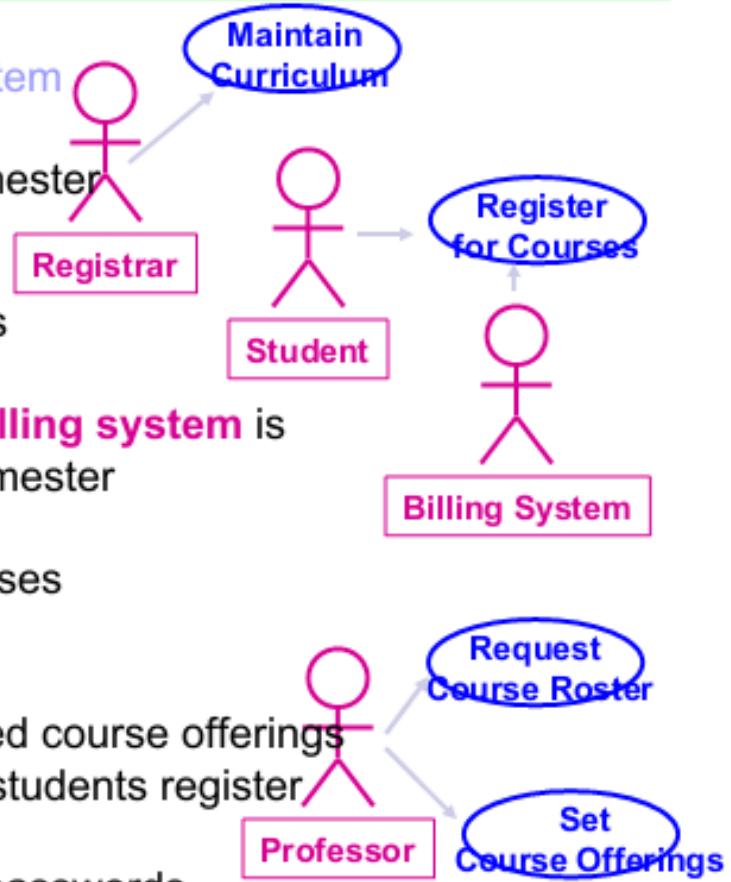


# Diagrams in UML – Use cases in Use case diagram

- A **use case** is a sequence of interactions between an actor and the system

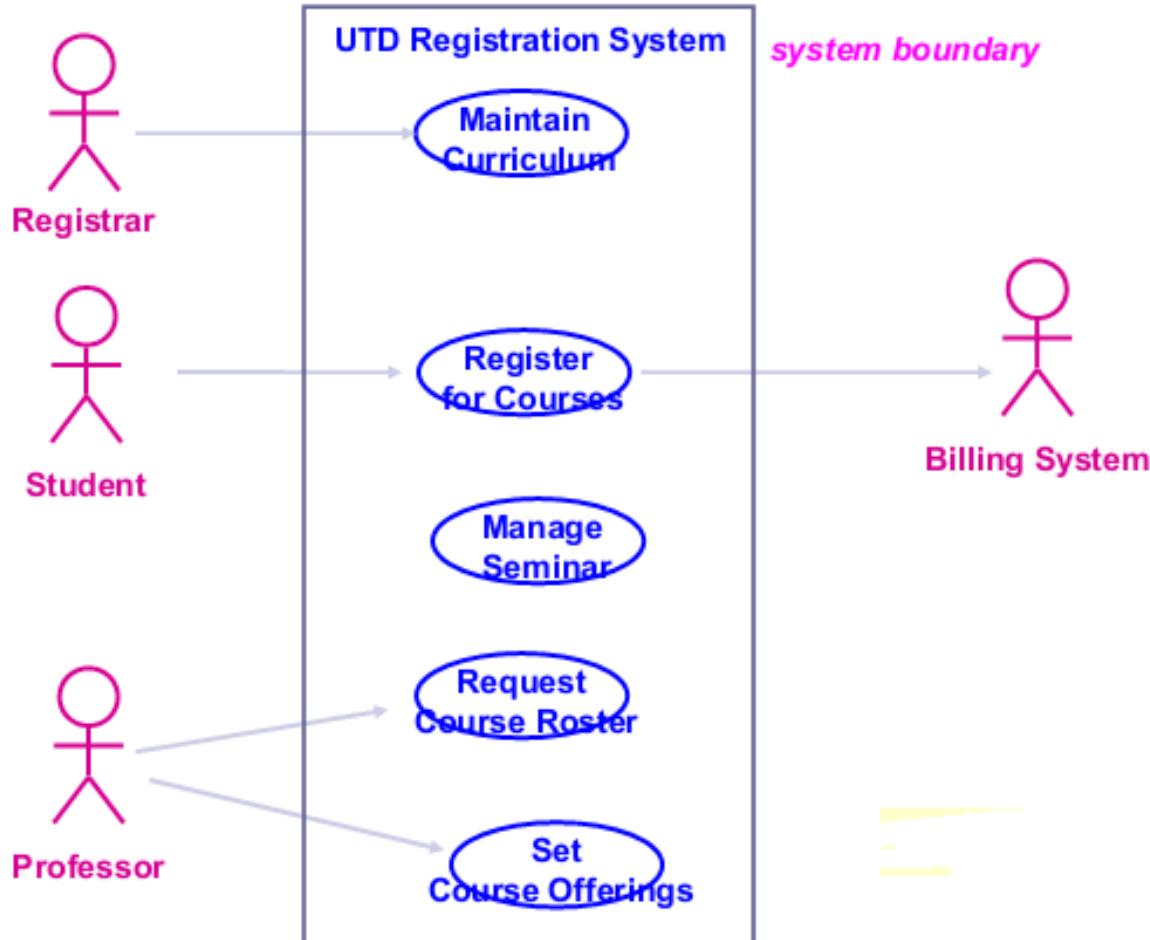
The UTD wants to computerize its registration system

- The **Registrar** sets up the curriculum for a semester
- **Students** select 3 core courses and 2 electives
- Once a student registers for a semester, the **billing system** is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- **Professors** use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users of the registration system are assigned passwords which are used at logon validation



# Diagrams in UML – Use case Diagram

- Use case diagrams depict the relationships between actors and use cases

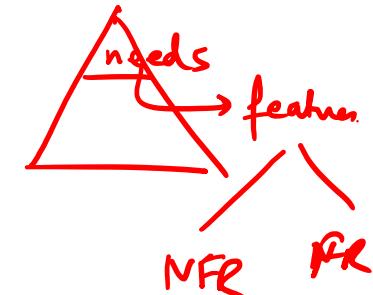




---

## **Next Lecture:** We Discuss

- Use Case Diagram
  - Sequence Diagram
  - Collaboration Diagram
  - Statechart
  - Activity Diagram
  - Class Diagram
  - Package Diagram
  - Component/Deployment Diagram
-



## IT 314: Software Engineering

*Software Effort/Cost Estimation*  
Function Point – Object Point

- Fugent → 1.
- Mapens → ① TRF
- fenz → ① SRF
- Certigen
- Combin

→ Overhead →

## **Bus Facility to and fro between Campus and Railway station, some city points**



# Motivation

---

- How much effort is required to complete the project?
- How much calendar time is needed to complete the project?
- What is the total cost?
- Project estimation and scheduling (project -> activities -> tasks  
-> roles)

Resources (employees, stakeholders (users, developers testers...))

Time

Effort

Cost





# Software Cost Components

---

- Effort costs (dominant factor in most projects)
    - salaries
    - Social and insurance & benefits
  - Tools costs: Hardware and software for development
    - Depreciation on relatively small # of years
  - Travel and Training costs (for particular client)
  - Overheads(OH): Costs must take overheads into account
    - costs of building, air-conditioning, heating, lighting
    - costs of networking and communications (tel, fax, )
    - costs of shared facilities (e.g. library, staff restaurant, etc.)
    - depreciation costs of assets
  - Others...
-



# Issues with Effort/Cost Estimation

---

- Ambiguous area (Impossible to estimate the unknown)
- Perfectionist & Confident Estimation (usually optimistic)
- A lot of time is spent on things that are not estimated
- Estimates do not consider productivity variations between programmers
- Changes in requirements do not reflect on estimations
- Wrong people do the estimates
- Others...

# Software Estimation

---

Predicting the resources required for a software development process

## Objectives:

- To introduce the fundamentals of software costing and pricing
- To describe
  - LOC model
  - COCOMO ‘Constructive Cost Model’
    - Object-point model
    - Function points model



# Programmer productivity

---

- A measure of the rate at which individual engineers involved in software development produce software and associated documentation
- Not quality-oriented although quality assurance is a factor in productivity assessment
- Essentially, we want to measure useful functionality produced per time unit



# Productivity measures

---

- Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.
- Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure



# Lines Of Code (LOC)

---

## What's a line of code?

- The measure was first proposed when programs were typed on cards with one line per card
- How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line

What programs should be counted as part of the system?

Assumes linear relationship between system size and volume of documentation

---

# Productivity comparisons

---

- The lower level the language, the more productive the programmer
  - The same functionality takes more code to implement in a lower-level language than in a high-level language
- The more verbose the programmer, the higher the productivity
  - Measures of productivity based on lines of code suggest that programmers who write verbose code are more productive than programmers who write compact code

# System Development Times

---

	<b>Analysis</b>	<b>Design</b>	<b>Coding</b>	<b>Testing</b>	<b>Documentation</b>
Assembly code	3 weeks	5 weeks	8 weeks	10 weeks	2 weeks
High-level language	3 weeks	5 weeks	8 weeks	6 weeks	2 weeks
	<b>Size</b>	<b>Effort</b>	<b>Productivity</b>		
Assembly code	5000 lines	28 weeks	714 lines/month		
High-level language	1500 lines	20 weeks	300 lines/month		

# The COCOMO Cost model (Constructive Cost Model)

---

- COCOMO II is a 3-level model that allows increasingly detailed estimates to be prepared as development progresses
- **Early prototyping level**
  - Estimates based on object-points and a simple formula is used for effort estimation
- **Early design level**
  - Estimates based on function-points that are then translated to LOC
  - Includes 7 cost drivers
- **Post-architecture level**
  - Estimates based on lines of source code or function point
  - Includes 17 cost drivers



# Object-Points (for 4GLs)

---

- Object-points are an alternative function-related measure to function points **when 4GLs** or similar languages are used for development
- Object-points **are NOT** the same as object classes
- The number of object-points in a program is considered as a weighted estimate of 3 elements:
  - The number of separate **screens** that are displayed
  - The number of **reports** that are produced by the system
  - The number of **3GL modules** that must be developed to supplement the 4GL code



# Object-Points (for 4GLs)

---

- Object-points are an alternative function-related measure to function points **when 4GLs** or similar languages are used for development
- Object-points **are NOT** the same as object classes
- The number of object-points in a program is considered as a weighted estimate of 3 elements:
  - The number of separate **screens** that are displayed
  - The number of **reports** that are produced by the system
  - The number of **3GL modules** that must be developed to supplement the 4GL code

# Object-Points Weighting

---

Object Type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
Each 3GL Module	10	10	10

# Object-Points Complexity Levels

---

Object point complexity levels for screens

Number of Views Contained	Number and sources of data tables		
	Total < 4	Total < 8	Total 8+
<3	simple	simple	medium
3-7	simple	medium	difficult
8+	medium	difficult	difficult

Object point complexity levels for reports

Number of Sections Contained	Number and source of data tables		
	Total < 4	Total < 8	Total 8+
0-1	simple	simple	medium
2-3	simple	medium	difficult
4+	medium	difficult	difficult

# Object-Points Estimation

---

Object-points are easy to estimate

- simply concerned with screens, reports and 3GL modules

At an early point in the development process:

- Object-points can be easily estimated
- It is very difficult to estimate the number of lines of code in a system

# Productivity Estimates

---

## LOC productivity

Real-time embedded systems: 40-160 LOC/P-month

Systems programs: 150-400 LOC/P-month

Commercial applications: 200-800 LOC/P-month

## Object-points productivity: PROD

measured 4 - 50 object points/person-month

depends on tool support and developer capability

Developer's experience and Capability / ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD: Productivity Object-point per person-month	4	7	13	25	50

# Object Point Effort Estimation

---

**Effort in p-m = NOP / PROD**

NOP = number of OP of the system

Example:

An application contains 840 Object-points (NOP=840) & Productivity is very high (= 50 object points/person-month )

Then, Effort =  $840/50 = (16.8) = 17$  p-m

# Adjustment for % of Reuse

---

% reuse: the % of screens, reports, & 3GL modules reused from previous applications, pro-rated by degree of reuse

- Adjusted NOP = NOP \* (1 - % reuse / 100)
- Adjusted NOP: New NOP
- Example: An application contains 840 OP, of which 20% can be supplied by existing components.

$$\text{Adjusted NOP} = 840 * (1 - 20/100) = 672 \text{ OP "New OP"}$$

$$\text{Adjusted effort} = 672/50 = (13.4) = 14 \text{ p-m}$$

# OP Estimation Procedure

---

1. Assess **object-counts** in the system: number of screens, reports, & 3GL.
2. Assess complexity level for each object (use table): simple, medium and difficult.
3. Calculate “**NOP**” the **object-point count** of the system: add all weights for all object instances
4. Estimate the % of reuse and compute the **adjusted NOP “New Object Points”** to be developed
5. Determine the productivity rate PROD (use metrics table)
6. Compute the **adjusted effort PM = adjusted NOP / PROD**

# OP Estimation Example

---

- Assessment of a software system shows that:
- The system includes
  - 6 screens: 2 simple + 3 medium + 1 difficult
  - 3 reports: 2 medium + 1 difficult
  - 2 3GL components
- 30 % of the objects could be supplied from previously developed components
- Productivity is high
- Compute the estimated effort PM ‘Person-months’ needed to develop the system

# OP Estimation Example: Solution

---

## Object counts:

2 simple screens	x 1 =	2
3 medium screens	x 2 =	6
1 difficult screen	x 3 =	3
2 medium reports	x 5 =	10
1 difficult report	x 8 =	8
2 3GL components	x 10 =	20

NOP

49

# OP Estimation Example: Solution

---

$$\begin{aligned}\text{Adjusted NOP 'New NOP'} &= \text{NOP} * (1 - \% \text{ reuse} / 100) \\ &= 49 * (1 - (30/100)) \\ &= (34.3) \\ &= 35\end{aligned}$$

For high productivity (metric table): PROD = 25 OP/P-M

$$\begin{aligned}\text{Estimated effort Person-Month} &= \text{Adjusted NOP} / \text{PROD} \\ &= 35 / 25 \\ &= 1.4 \text{ P-M}\end{aligned}$$

---



---

# Function Points

---



# Function Points

FR | NFR

---

A **Function Point (FP)** is a unit of measurement to express the amount of business functionality, an information system (as a product) provides to a user.

FPs measure software size.

FPs are a standard unit of measure that represent the functional size of a software application

They are widely accepted as an industry standard for functional sizing.

---

# Function Point Analysis

---

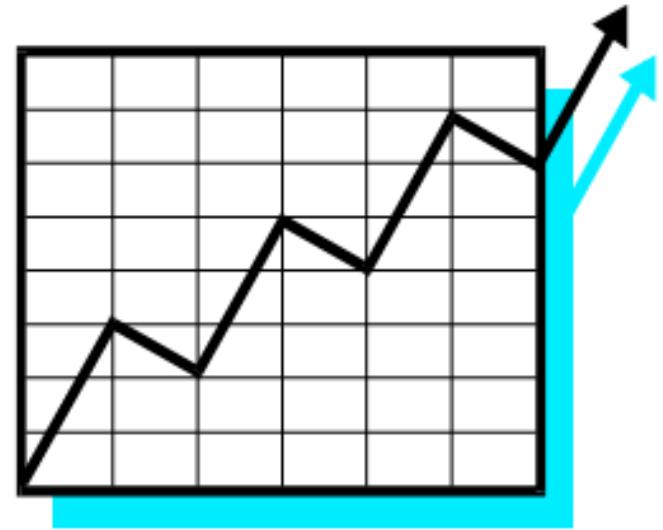
Most practitioners of Function Point Analysis (FPA) will probably agree that there are three main objectives within the process of FPA:

- Measure software by quantifying the functionality requested by and provided to the customer.
- Measure software development and maintenance independently of technology used for implementation.
- Measure software development and maintenance consistently across all projects and organizations.

# ■ Why use Function Points?

---

- Size of Requirements
- Changes to Requirements
- Estimation Based on Requirements
- Measuring and Improving Productivity and Quality



# Estimating Examples

## Function Point Size

Project A – 100 FPs

## Project Variables

- On-line/database
- New development
- C++
- Highly experienced development staff

Project B – 100 FPs

- Batch
- Enhancement
- Cobol
- Average experienced development staff

## Project Estimate Based on Historical Data and/or Vendor Tool

Effort = 5 months

Schedule = 3 months

Cost (@ \$5K) = \$25,000

KLOC = 6

Delivered Defects = 25

Productivity Rate = 20 FP/Month.

Effort = 20 months

Schedule = 6 months

Cost (@ \$5K) = \$100,000

KLOC = 10

Delivered Defects = 100

Productivity Rate = 5 FP/Month



# Function Points vs. LOC

---

- Technology and platform independence
  - Available from early requirements phase
  - Consistent and objective unit of measure throughout the life cycle
  - Objectively defines software application from the customer perspective
  - Objectively defines a series of software applications from the customer's, not the technician's perspective
  - Is expressed in terms that users can readily understand about their software
-

# What is Wrong with LOC?

---

- There is no standard for a line of code
- Lines of Code measure components, not completed products
  - Don't measure the panels produced; measure the number of cars assembled
- Measuring lines of code
  - Rewards profligate design
  - Penalizes tight design
- Positively misleading?



# Classic Productivity Paradox

---

Lines of Code	10,000	3,000
Function Points	25	25
Total Months effort	25	15
Total Costs	\$125,000	\$75,000
Cost per Source Line	\$12.50	\$25.00
Lines per Person month	400	200
FPs per Person month	1.2	2
Cost per FP	\$5,000	\$3,000

# Function Points (5 Characteristics)

---

Based on a combination of program 5 characteristics

- The number of : user inputs
  - External (user) inputs: input transactions that update internal files
  - External (user) outputs: reports, error messages
  - User interactions: inquiries
  - Logical internal files used by the system: Example a purchase order logical file composed of 2 physical files/tables Purchase\_Order and Purchase\_Order\_Item
  - External interfaces: files shared with other systems

# FPs Standard

---

For sizing software based on FP, several recognized standards and/or public specifications have come into existence. As of 2013, these are -

## ISO Standards

COSMIC - ISO/IEC 19761:2011 Software engineering. A functional size measurement method.

FiSMA - ISO/IEC 29881:2008 Information technology - Software and systems engineering - FiSMA 1.1 functional size measurement method.

IFPUG - ISO/IEC 20926:2009 Software and systems engineering - Software measurement - IFPUG functional size measurement method.

Mark-II - ISO/IEC 20968:2002 Software engineering - Ml II Function Point Analysis - Counting Practices Manual.

NESMA - ISO/IEC 24570:2005 Software engineering - NESMA function size measurement method version 2.1 - Definitions and counting guidelines for the application of Function Point Analysis.

---

# Another View of FP

---

FP is a standard method for quantifying the software deliverable based upon the user view, where:

- User is any person or thing that communicates or interacts with the software at any time
  - User View is the Functional User Requirements as seen by the user
  - Functional user requirements describe what the software shall do, in terms of tasks and services.
  - It is also important to keep in mind that function points look at the logical view, not the physical. So things like coding algorithms, database structure, screenshots of transactions are not counted.
-



# FPs - Artifacts

---

Things that are counted within the Function Points methodology:

- Input files and input transactions (batch interfaces)
  - Screens (adds, changes, deletes)
  - Control Information
  - Internal Logical Files (tables, files with data, control files)
  - External tables and referenced files from other applications
  - Output files and transactions (batch interfaces)
  - Other outputs - reports, files, dvd's, views, notices, warnings
-

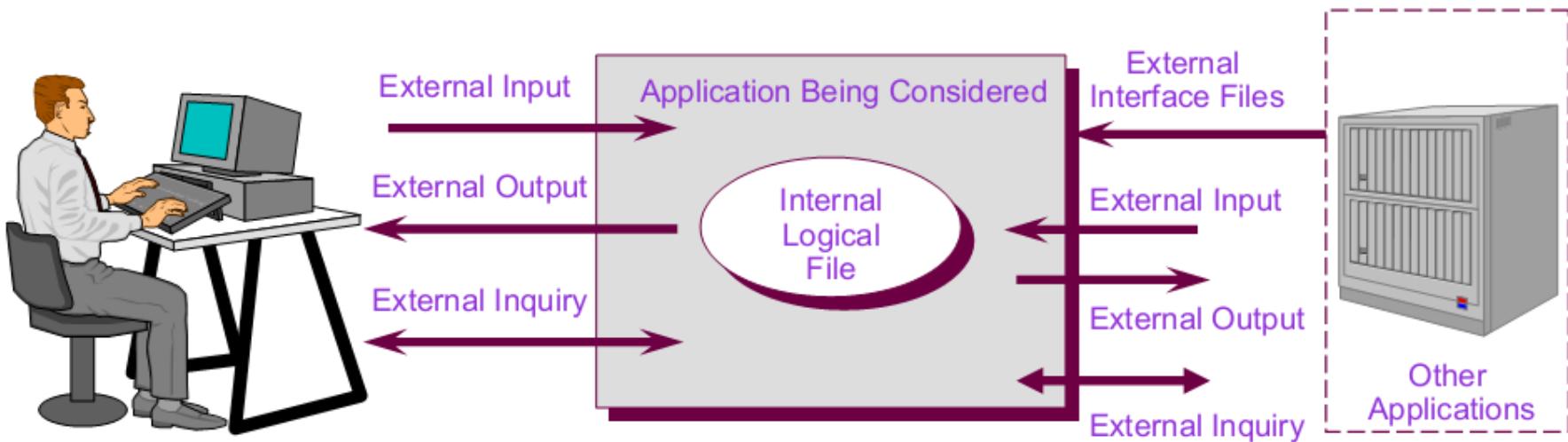
# FPs

---

A weight is associated with each of the above 5 characteristics

- Weight range:
  - from 3 for simple feature
  - to 15 for complex feature
- The function point count is computed by multiplying each raw count by the weight and summing all values.

# FPs are a Unit of Measure



- Functionality as viewed from the user's perspective



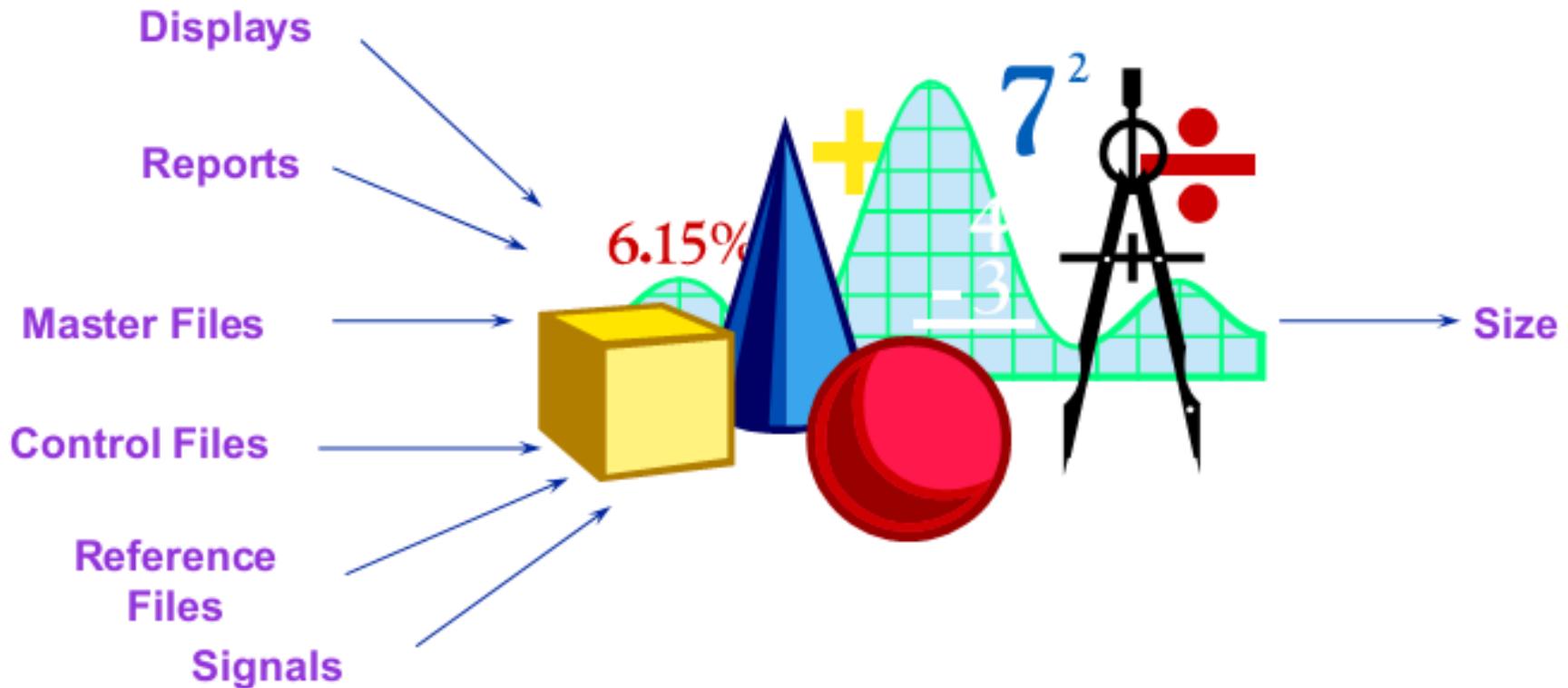
---

# **How to Count Function Points???**

---

# How to Count Function Points?

---



# How to Count Function Points?

---

## Internal Logical File (ILF)

Logical group of data maintained by the application (e.g., Employee file)

## External Interface File (EIF)

Logical group of data referenced but not maintained (e.g., Global state table)

## External Input (EI)

Maintains ILF or passes control data into the application

## External Output (EO)

Formatted data sent out of application with added value (e.g., calculated totals)

## External Query (EQ)

Formatted data sent out of application without added value

---

# How to Count Function Points?

---

Function Type	Low	Average	High
EI	x 3	x 4	x 6
EO	x 4	x 5	x 7
EQ	x 3	x 4	x 6
ILF	x 7	x 10	x 15
EIF	x 5	x 7	x 10

# Function Points - Calculation

<u>measurement parameter</u>	<u>count</u>	<u>weighting factor</u>			=	<input type="text"/>
		simple	avg.	complex		
number of user inputs	<input type="text"/>	X 3	4	6	=	<input type="text"/>
number of user outputs	<input type="text"/>	X 4	5	7	=	<input type="text"/>
number of user inquiries	<input type="text"/>	X 3	4	6	=	<input type="text"/>
number of files	<input type="text"/>	X 7	10	15	=	<input type="text"/>
number of ext.interfaces	<input type="text"/>	X 5	7	10	=	<input type="text"/>
count-total	<hr/>					<input type="text"/>
complexity multiplier						<input type="text"/>
function points	<hr/>					<input type="text"/>

# Adjusted FPs Count Complexity:14 Factors Fi (cont.)

---

14 factors: Each factor is rated on a scale of:

**Zero**: not important or not applicable

**Five**: absolutely essential

1. Backup and recovery
  2. Data communication
  3. Distributed processing functions
  4. Is performance critical?
  5. Existing operating environment
  6. On-line data entry
  7. Input transaction built over multiple screens
-



# **Adjusted FPs Count Complexity:14 Factors Fi (cont.)**

---

8. Master files updated on-line
  9. Complexity of inputs, outputs, files, inquiries
  10. Complexity of processing
  11. Code design for reuse
  12. Are conversion/installation included in design?
  13. Multiple installations
  14. Application designed to facilitate change by the user
-

## Adjusted FPs Count Complexity:14 Factors $F_i$ (cont.)

$$AFPC = UFPC * [ 0.65 + 0.01 * \sum_{i=1}^{i=14} F_i ]$$

AFPC: Adjusted function point count

UFPC: Unadjusted function point count

$$0 \leq F_i \leq 5$$

# Estimated LOC Approach

---

Assuming

- Estimated project LOC = 33200
- Organisational productivity (similar project type) = 620 LOC/p-m
- Burdened labour rate = 8000 \$/p-m

Then

- Effort =  $33200/620 = (53.6) = 54$  p-m
- Cost per LOC =  $8000/620 = (12.9) = 13$  \$/LOC
- Project total Cost =  $8000 * 54 = 432000$  \$

# Estimated FPs Approach

	A	B	C	D	E	F	G
1	Info Domain	Optimistic	Likely	Pessim.	Est Count	Weight	FP count
2	# of inputs	22	26	30	26	4	104
3	# of outputs	16	18	20	18	5	90
4	# of inquiries	16	21	26	21	4	84
5	# of files	4	5	6	5	10	50
6	# of external interfaces	1	2	3	2	7	14
7	UFC: Unadjusted Function Count						342
8		Complexity adjustment factor					1.17
9						FP	400

# Estimated FPs Approach: Complexity Factor

Complexity factor: Fi	value=0	value=1	value=2	value=3	value=4	value=5	Fi
Backup and recovery	0	0	0	0	1	0	4
Data communication	0	0	1	0	0	0	2
Distributed processing functions	0	0	0	0	0	0	0
Is performance critical?	0	0	0	0	1	0	4
Existing operating environment	0	0	0	1	0	0	3
On-line data entry	0	0	0	0	1	0	4
Input transaction built over multiple screens	0	0	0	0	0	1	5
Master files updated on-line	0	0	0	1	0	0	3
Complexity of inputs, outputs, files, inquiries	0	0	0	0	0	1	5
Complexity of processing	0	0	0	0	0	1	5
Code design for re-use	0	0	0	0	1	0	4
Are conversion/installation included in design?	0	0	0	1	0	0	3
Multiple installations	0	0	0	0	0	1	5
Application designed to facilitate change by the user	0	0	0	0	0	1	5
						Sigma (F)	52
Complexity adjustment factor	$0.65 + 0.01 * \text{Sigma (F)} =$				1.17		

## Estimated FPs Approach

---

Assuming  $\sum_i F_i = 52$

$$FP = UFC * [ 0.65 + 0.01 * \sum_i F_i ]$$

$$FP = 342 * 1.17 = 400$$

Complexity adjustment factor = 1.17

---

# FPs Approach

---

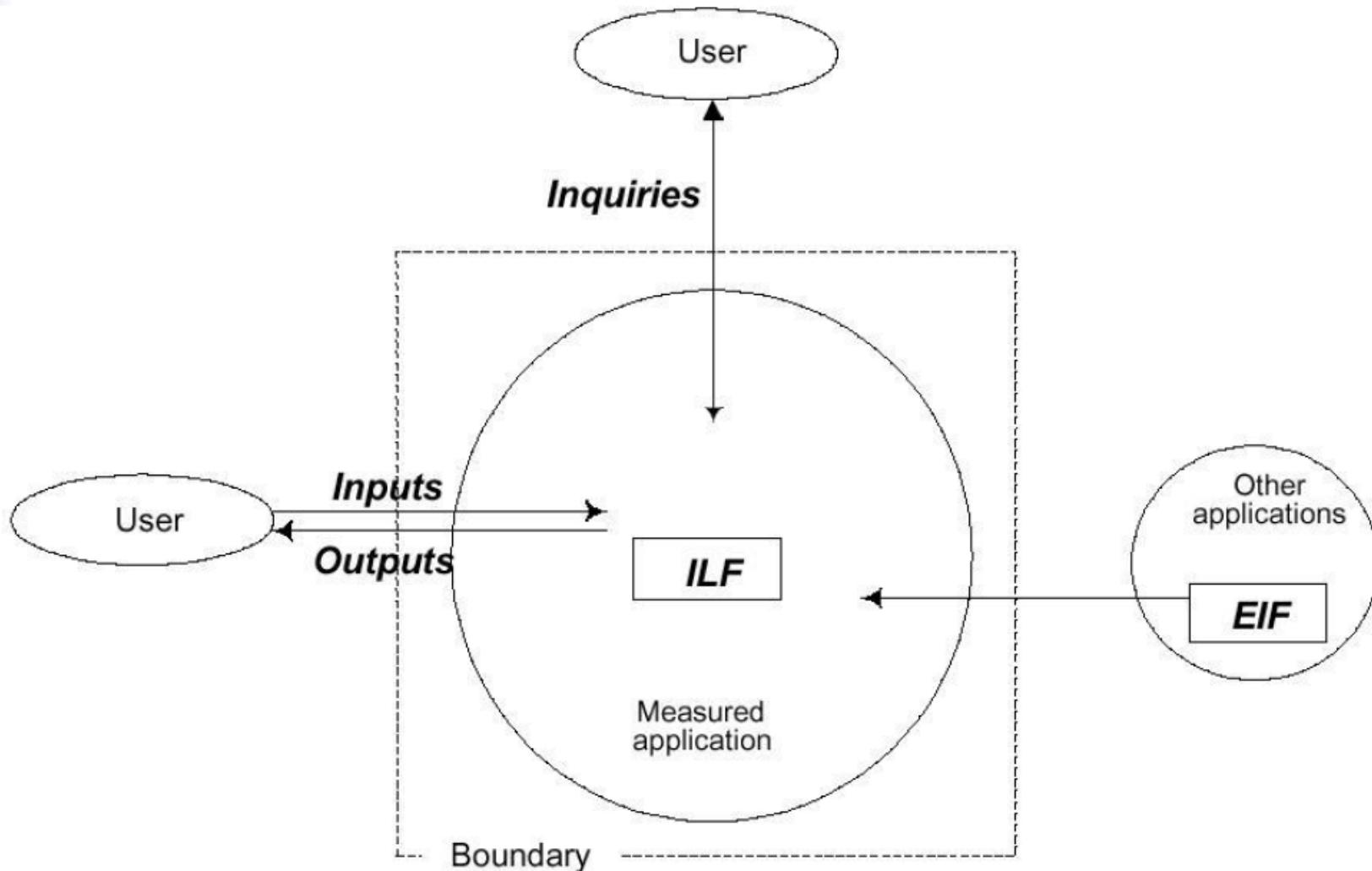
Assuming

- Estimated FP = 400
- Organisation average productivity (similar project type) = 6.5 FP/p-m (person-month)
- Burdened labour rate = 8000 \$/p-m

Then

- Estimated effort =  $400/6.5 = (61.53) = 62$  p-m
- Cost per FP =  $8000/6.5 = 1231$  \$/FP
- Project cost =  $8000 * 62 = 496000$  \$

# FP Components



# FP Example

---

**STOCK CONTROL SYSTEM** - estimating the time needed to develop application

*Let's imagine a company which sells goods on the phone - if agents call the customers, customers call the agents, and so on - business operates successfully, but there comes a time for putting the whole in order. There occurs a need for developing a system able to control the whole stock, from orders to payments. Our thing is to estimate how complex such system can be and - after that - try to predict how long it would take to develop it.*

# FP Example

---

- **External Inputs** - customer, order, stock, and payment details. There are four things we need to consider.
  - **External Outputs** - customer, order, and stock details, and credit rating. Once again, there are four things to consider.
  - **External Inquiries** - the system is requested for three things, which are customer, order, and stock details.
  - **External Interface Files** - there's no EIFs to consider.
  - **Internal Logical Files** - finally, the four elements belong to the last group. Customer, and good files, and customer, and good transaction files.
-

# FP Example

---

Category	Multiplier	Weight
EI	4	3
EO	4	4
EQ	3	3
ILF	3	7

$$4*3+4*4+3*3+3*7=58 \text{ [Function Points]}$$

Omit additional technical complexity factors, so the only thing left to do is to check how long it takes to produce 58 function points. Some sources prove that one function point is an equivalent of eight hours of work in C++ language.

$$58*8=464 \text{ [hours]}$$

The estimate for developing the application would take about 464 hours of work.

---



# IT 314: Software Engineering

*Object Design*



---

## Object Design: Completing the Puzzle

- The pieces found during object design are:
    - New solution objects
    - Off-the-self-components and their adjustments
    - Design Patterns
    - Specification of subsystem interfaces and classes
-

# Application domain vs. solution domain objects

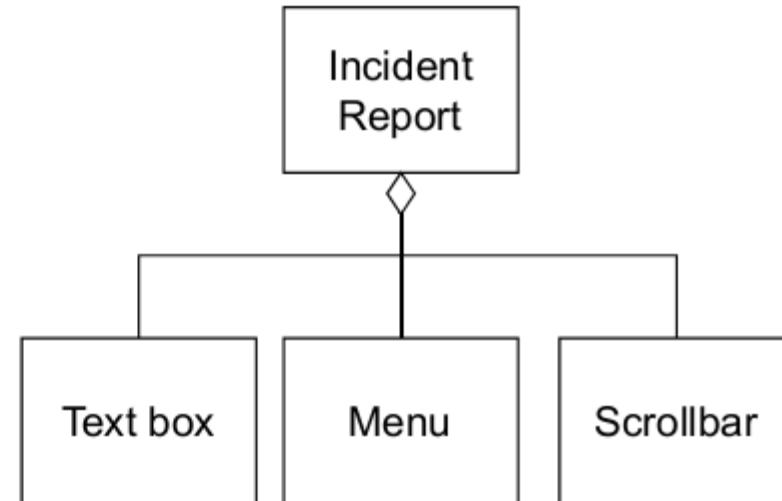
- Application objects (also called domain objects) represent concepts of the domain that are relevant to the system
  - They are identified by the application domain specialists and by the end users
- Solution objects represent concepts that do not have a counterpart in the application domain,
  - They are identified by the developers
  - Examples: Persistent data stores, connection objects, user interface objects, data structures, middleware

# Application Domain vs Solution Domain Objects

**Requirements Analysis**  
(Language of Application Domain)



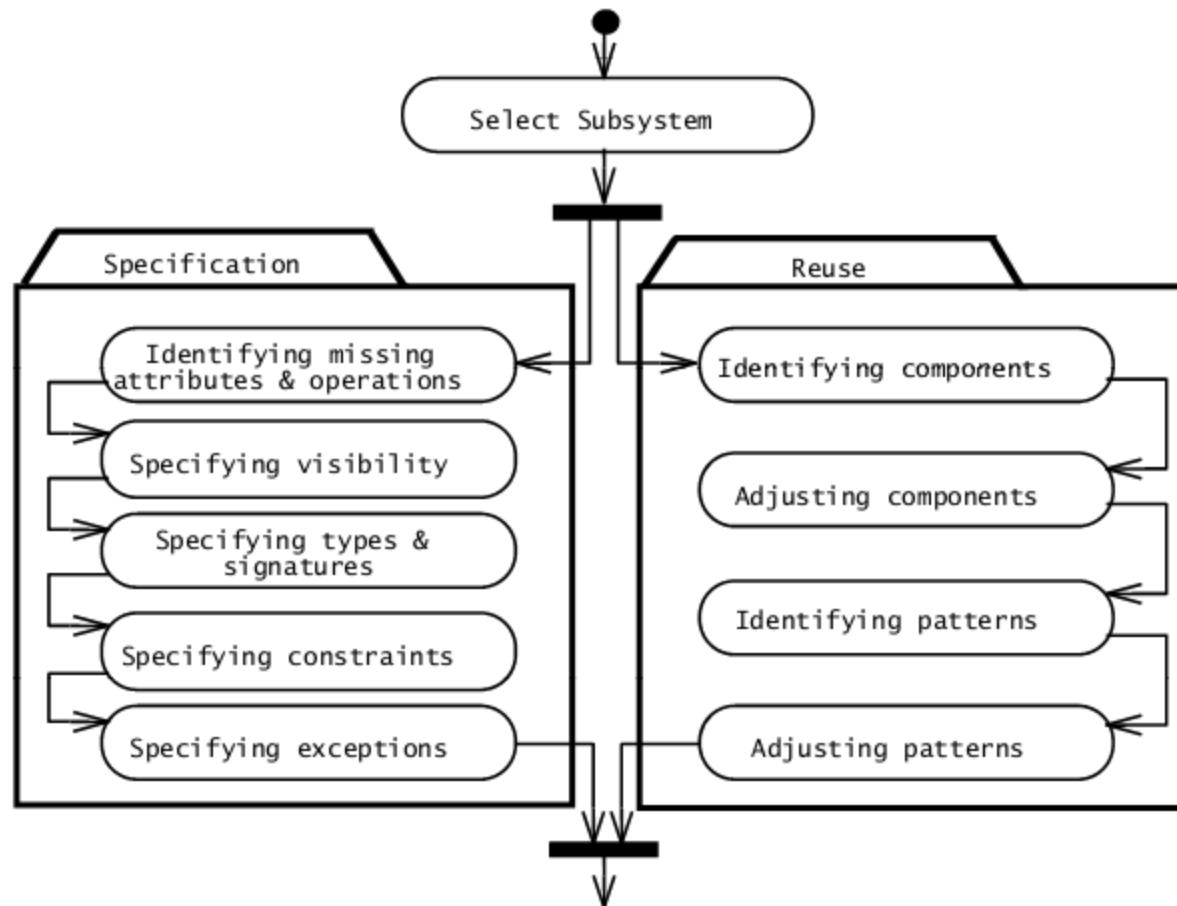
**Object Design**  
(Language of Solution Domain)



# Implementation of Application Domain Classes

- New objects are often needed during object design:
  - The use of design patterns introduces new classes
  - The implementation of algorithms may necessitate objects to hold values (Data Structures)
  - New low-level operations may be needed during the decomposition of high-level operations
- Example: The eraseArea() operation in a drawing program.
  - Conceptually very simple
  - Implementation
    - getArea() represented by pixels
    - repair () cleans up objects partially covered by the erased area
    - redraw() draws objects uncovered by the erasure
    - draw() paints pixels in background color not covered by other objects

# Object Design Activities



# Design Activities

## 1. Reuse: Identification of existing solutions

- Use of inheritance
- Off-the-shelf components and additional solution objects
- Design patterns

Component/  
Object  
Design

## 2. Interface specification

- Describes precisely each class interface

## 3. Component/Object model restructuring

- Transforms the object design model to improve its understandability and extensibility

Mapping  
Models to Code

## 4. Component/Object model optimization

- Transforms the object design model to address performance criteria such as response time or memory utilization.

# Component Selection

- Select existing
  - off-the-shelf class libraries
  - frameworks or
  - components
- Adjust the class libraries, framework or components
  - Change the API if you have the source code.
  - Use the adapter or bridge pattern if you don't have access
- Create a new Component (Architecture Driven Design)

# Reuse

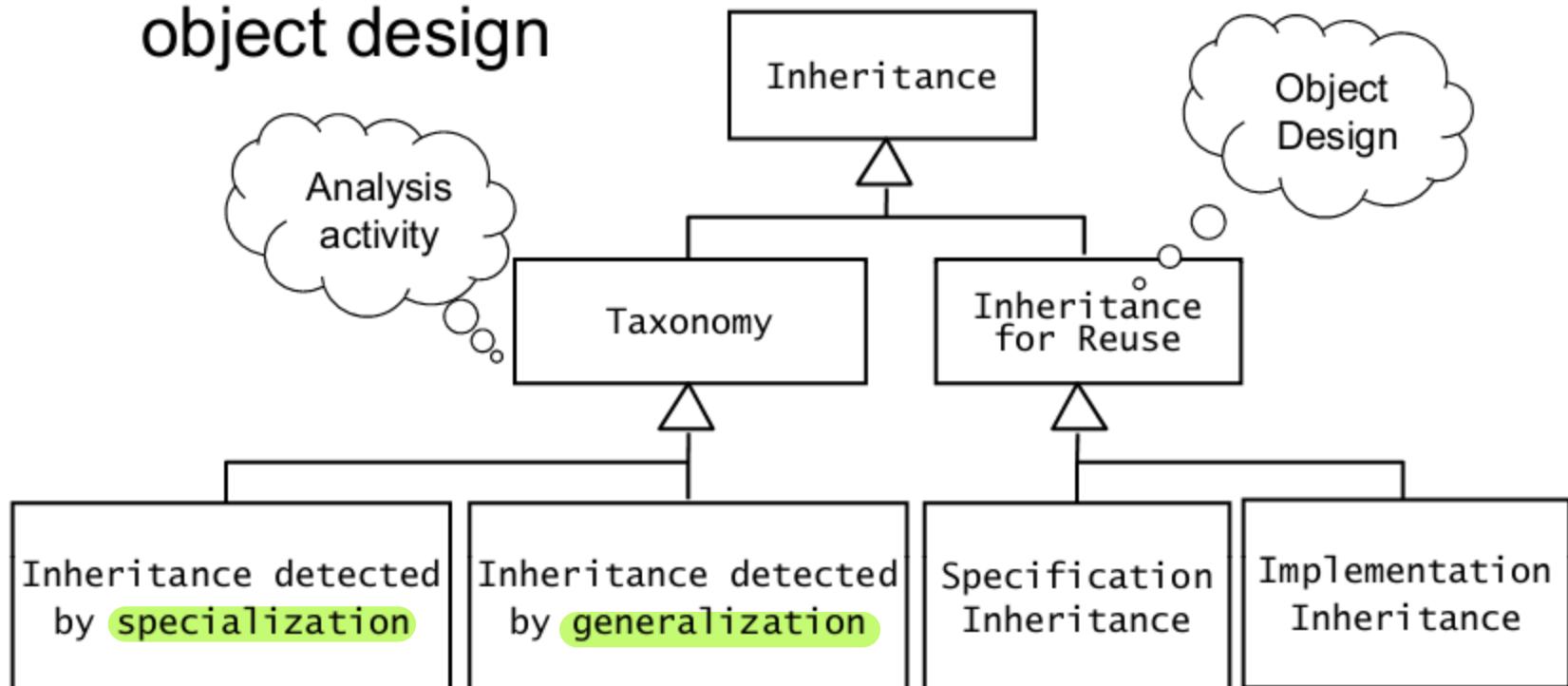
- Main goal:
  - Reuse knowledge from previous experience to current problem
  - Reuse functionality already available
  - Save resources
- Composition (also called Black Box Reuse)
  - New functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing components
- Inheritance (also called White-box Reuse)
  - New functionality is obtained by inheritance.
- Four ways to get new functionality:
  - Implementation inheritance
  - Interface inheritance
  - Delegation
  - Aggregation

# The use of inheritance

- Inheritance is used to achieve two different goals
  - Description of Taxonomies
  - Interface Specification
- Identification of taxonomies
  - Used during requirements analysis.
  - Activity: identify application domain objects that are hierarchically related
    - Goal: make the analysis model more understandable
- Service specification
  - Used during object design
  - Activity: identify solution domain objects to enhance reuse
    - Goal: increase reusability, enhance modifiability and extensibility

# Metamodel for Inheritance

- Inheritance is used during analysis and object design





---



Lecture on  
Design Patterns ?



Many design patterns use a combination of inheritance and delegation

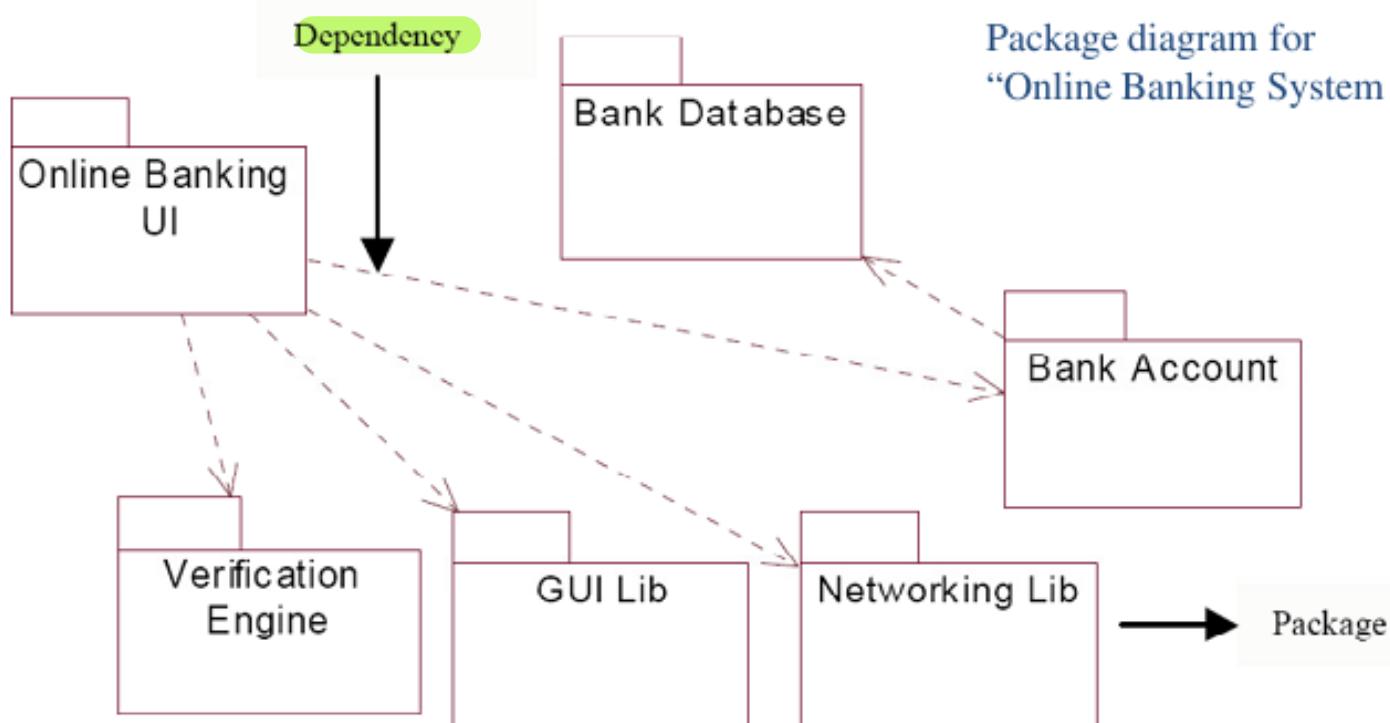


# Package Diagram

---

- Structured organization of Code
- Grouping of related classes to help the software engineer to identify and to understand dependencies
- When to use?
  - Program Comprehension
  - Change Management

# Package Diagram: An Example



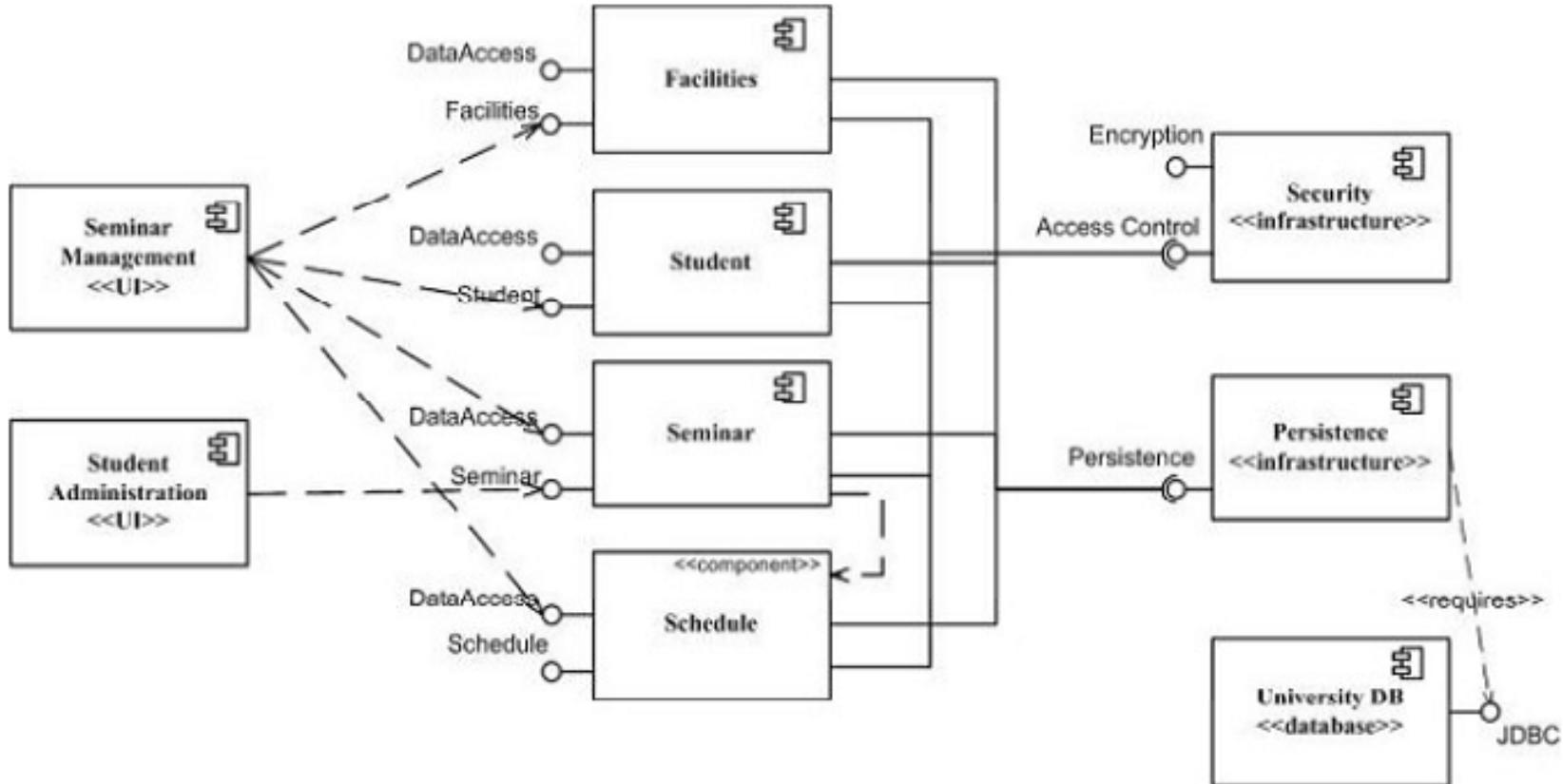


# Component Diagram

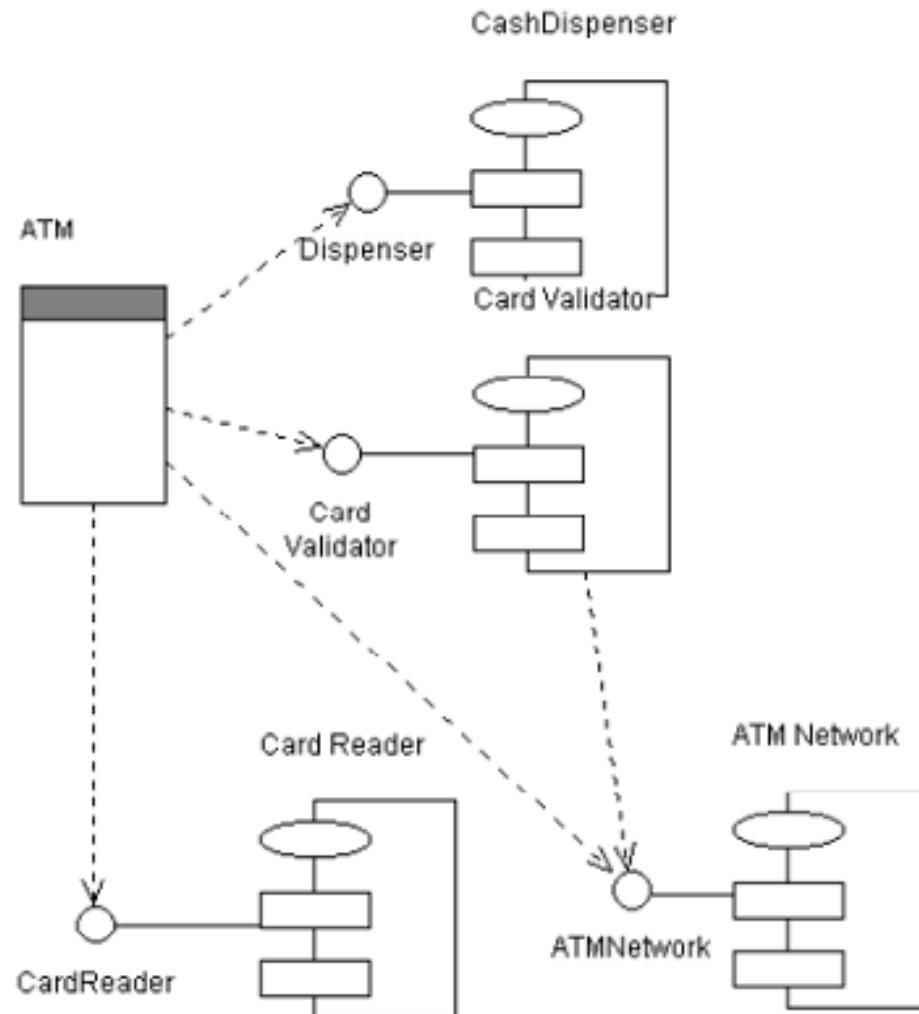
---

- Depicts how components are **wired** together to form bigger component or system
  - Component interacts with each other through **interfaces**
  - Connect the **required interface** of one component with the **provided interface** of another component.
  - Designed with an eye towards **deployment**
-

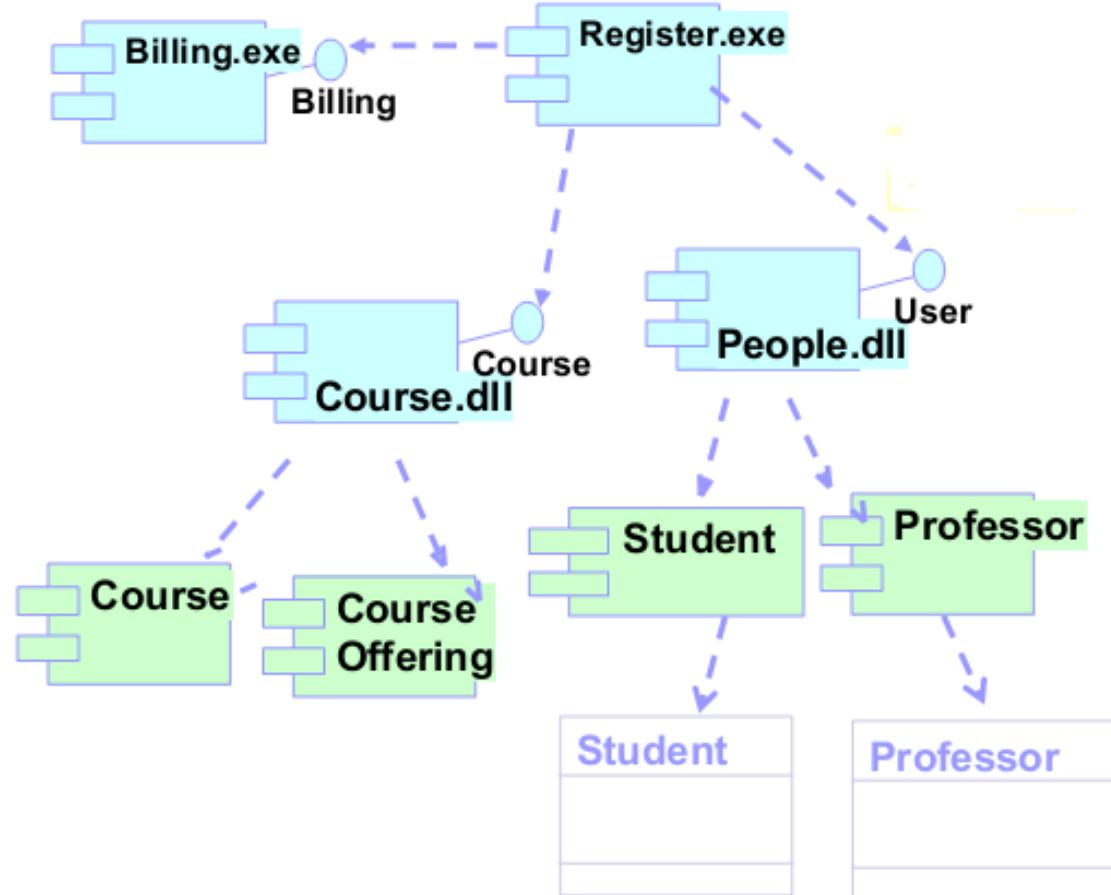
# Component Diagram



# Component Diagram

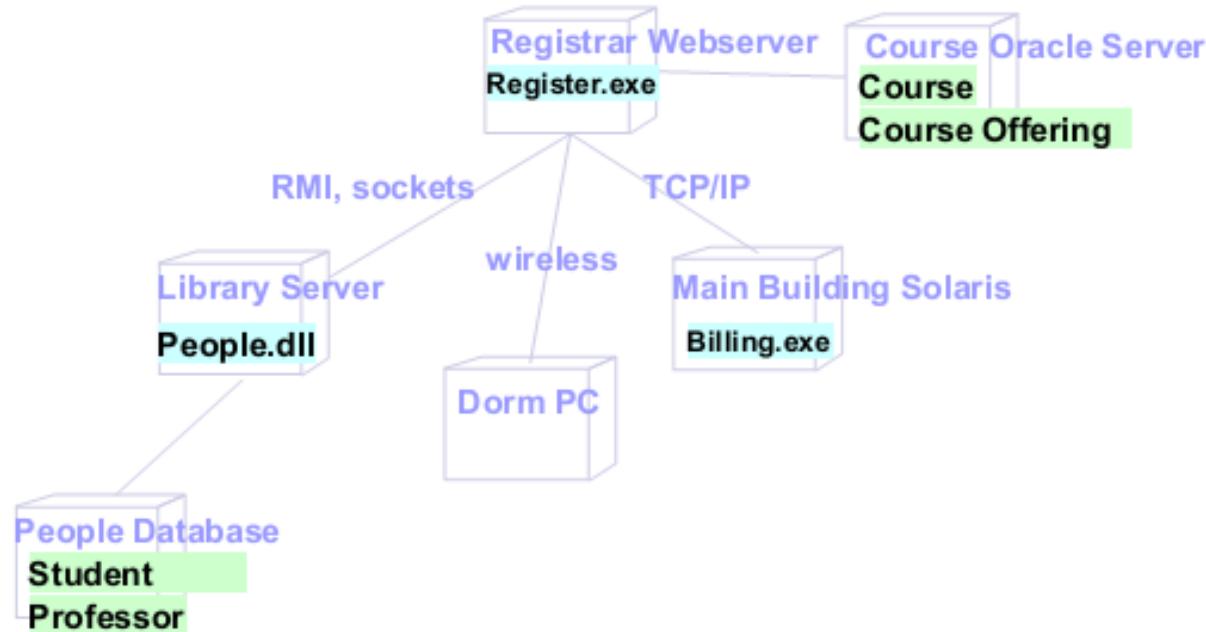


# Component Diagram



# Deployment Diagram

- shows the configuration of run-time processing elements and the software processes living on them.
- visualizes the distribution of components across the enterprise.





# Summary

---

Design is the process of adding details to the requirements analysis and making implementation decisions

- An evolutionary activity
- Consists of
  - Sub-system Design (Choosing an Architecture)
  - Object Design (Solution domain)

---

Next Lectures...  
Design Patterns



DA-IICT

## IT 314: Software Engineering

### *Introduction to Testing*

Saurabh Tiwari

1

## Spectacular Software Failures

- MARINER 1, the first U.S. attempt to send a spacecraft to Venus, failed minutes after launch in 1962. The guidance instructions from the ground stopped reaching the rocket due to a problem with its antenna, so the onboard computer took control. However, there turned out to be a bug in the guidance software, and the rocket promptly went off course, so the Range Safety Officer destroyed it. Although the bug is sometimes claimed to have been an incorrect FORTRAN DO statement, it was actually a transcription error in which the bar (indicating smoothing) was omitted from the expression "R-dot-bar sub n" (nth smoothed value of derivative of radius). This error led the software to treat normal minor variations of velocity as if they were serious, leading to incorrect compensation.
- Software problems in the automated baggage sorting system of a major airport in February 2008 prevented thousands of passengers from checking baggage for their flights. It was reported that the breakdown occurred during a software upgrade, despite pre-testing of the software. The system continued to have problems in subsequent months.

## Spectacular Software Failures

- AT&T long distance network crash (January 15, 1990), in which the failure of one switching system would cause a message to be sent to nearby switching units to tell them that there was a problem. Unfortunately, the arrival of that message would cause those other systems to fail too - resulting in a 'wave' of failure that rapidly spread across the entire AT&T long distance network. [Wrong BREAK statement in C-Code](#)
- The Northeast blackout of 2003 was a widespread power outage that occurred throughout parts of the Northeastern and Midwestern United States and Ontario, Canada on Thursday, August 14, 2003, just before 4:10 p.m. The blackout affected an estimated 10 million people in Ontario and 45 million people in eight U.S. states. [A software bug known as a race condition existed in General Electric Energy's Unix-based XA/21 energy management system.](#)

## Spectacular Software Failures

- The European Space Agency's Ariane 5 Flight 501 was destroyed 40 seconds after takeoff (June 4, 1996). The US\$1 billion prototype rocket self-destructed due to a [bug in the on-board guidance software](#).
- NASA Mars Polar Lander was destroyed because its flight software mistook vibrations due to atmospheric turbulence for evidence that the vehicle had landed and shut off the engines 40 meters from the Martian surface (December 3, 1999). Its sister spacecraft Mars Climate Orbiter was also destroyed, but [due to human error and not, as is sometimes reported, due to a software bug](#).

Software bugs can potentially cause monetary and even loss of life

## Today's Software Market

- ☐ is much bigger
- ☐ is more competitive
- ☐ more users

*Emerging Software!!*

Example...

- APPS
- API's
- Games
- Security
- Safety

*Android Market  
Open Source Software*

Embedded Systems



*Challenges*

How to test??

## Most Costly Software Failures...

2002 : NIST report, "The Economic Impacts of Inadequate Infrastructure for Software Testing" costs the US alone between \$22 and \$59 billion annually

Huge losses due to web application failures

- Financial services \$6.5 million per hour (in USA)
- Credit card sales applications: \$2.4 million per hour (in USA)

London stock exchange shut down

- Sept 08 - 7 hours
- 25 Feb 11 about 4 hours

*Industries are going through a revolution in what testing means to the success of software products*

## What Does This Mean?



Software testing is getting more important

## Testing in the 21st Century

- More **safety** critical, **real-time** software
- Embedded software is ubiquitous ... check your pockets
- Enterprise applications means bigger programs, more users
- Paradoxically, free software **increases** our expectations !
- **Security** is now all about software faults
  - **Secure** software is **reliable** software
- The **web** offers a new deployment platform / APPS too...
  - Very **competitive** and very **available** to more users
  - Web apps are distributed
  - **Web apps** must be highly reliable

Industry desperately needs our inventions !

## Why Testing?

- Uncover as many as errors as possible in a given timeline.
- Demonstrate a given software product matching its requirements specification.
- Validate the quality of a software.

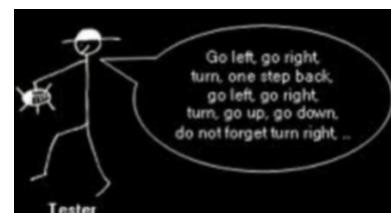
Software testing is arguably the least understood part of the development process. Through a four-phase approach, the author shows why eliminating bugs is tricky and why testing is a constant trade-off [Whittaker]

## Who Tests the Software?



### Developer

Understands the system but,  
will test "gently" and, is driven  
by "delivery"

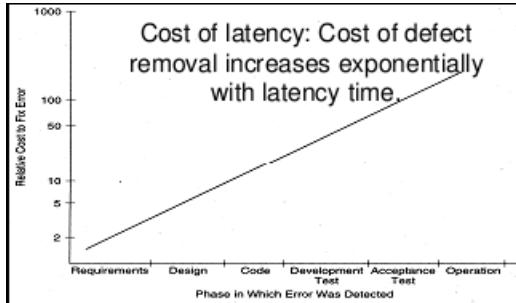


### Independent Tester

Must learn about the  
system, but, will attempt to  
break it and, is driven by  
quality

## Testing Challenges

- When to start testing? (cost of late testing)
- When to stop testing? How much testing is enough?
- Which and how many testing techniques/types?
- Test automation
- How will you know the program passed or failed test? (test output evaluation)



## Testing Purpose

1. Avoid Redundancy
  - Source Code
  - Requirement/Design/Maintenance/Change Management
2. Reducing Cost
  - Cost of detection, Cost of Prevention
  - Internal Failure (at the time of development or before you release the software product)
  - External Failure (finding bugs after deploying the software product)
3. Correctness CONFIDENCE
4. Quality Assurance

## Course Coverage

1. The Psychology and economics of Program Testing
2. Testing Techniques
3. Unit Testing, Integration Testing, System Testing, Acceptance Testing, Regression Testing
4. Functional and Non-functional Testing
5. Model-based Testing (especially, UML)
6. Test Case Design, Selection, Prioritization, Execution
7. GUI Testing
8. Analyzing Software Quality
  - a. Testing Maturity Models
  - b. McCall's Quality Factors and Criteria
  - c. ISO 9126 Quality Characteristic

Use of Open Source Tools like Junit Testing Framework, Selenium, MuJava, Static Analysis tools and many more ...

## What's this Course All About?

### Expected Outcomes:

- Understand the effectively strategies of testing, the methods and technologies of software testing
- Design test plan and test cases for the given requirement specification, small-scale programs and large-scale projects
- Perform automatic testing ??
- Clearly and correctly report the software bugs and defectives (6) asses the software product correctly
- Analyze the Quality of Software Product being developed

## Classroom Etiquette

---

- Come on time to both class and labs
  - Talking, cell phones, etc. will not be tolerated
- 

## Suggested books

---

- **Main text:**  
*“Software Testing and Quality Assurance: Theory and Practice”*,  
Kshirasagar Naik And Priyadarshi Tripathy, A John Wiley & Sons, Inc.,  
Publication
  - “*Software Quality Assurance: From theory to implementation*”, Daniel Galin, Pearson Education Limited 2004
  - “*Software Quality Engineering Testing, Quality Assurance, and Quantifiable Improvement*”, Jeff Tian, IEEE Computer Society
  - “*The Art of Software Testing*”, G.J. Myers, Second Edition, John Wiley & Sons, New York, 1976.
  - “*Lessons Learned in Software Testing*”, C Kaner, J. Bach, B. Pettichord Wiley, 2001.
-

## Grade Breakdown

---

- Exams (60%): weighted 15%, 15%, 30% (final)
  - Lab work (15%) and Assignments/Presentation/Open Source tool evaluation (25%)
    - Lab attendance is mandatory
    - ZERO marks for copying or allowing someone to copy
- 

## Taxonomy of Bugs

---

Error?

*Verification?*

*Validation?*

Defects ? Doesn't work as defined

Fault?

Failure?- Non functioning of system

Debugging?

*Testing?*

Bugs? – error in the system

Quality?

## Error, Defects, Fault and Failure

### Example:

You are driving a car and you are on road while on driving now there is two way on the road

1. left--> Mumbai
2. right--> Delhi

Now you have to go to Delhi, it means you have to turn the steering to the right, but by **mistake** you turn the steering to the left, from that position that is called **as "Error"**

and now Fault is there till you will not reach the Mumbai, but when you reach Mumbai that is a final stage which is called **"Failure"** because you had to reach Delhi but now you are in Mumbai.

**A mistake in coding is called Error, error found by tester is called defect, defect accepted by development team then it is called bug, build does not meet the requirements then it Is failure.”**

## Coding: Error, Defects, Fault and Failure

The program is required to add two numbers

```

1 #include<stdio.h>
2
3 int main ()
4 {
5     int value1, value2, ans;
6
7     value1 = 5;
8     value2 = 3;
9
10    ans = value1 - value2;
11
12    printf("The addition of 5 + 3 = %d.", ans);
13
14    return 0;
15 }
```

5 + 3 should be 8, but the result is 2. There could be various reasons as to why the program displays the answer 2 instead of 8. For now we have detected a *failure*.

As the failure has been detected a **defect** can be raised.

The program is required to add two numbers

```

1 #include<stdio.h>
2
3 int main ()
4 {
5     int value1, value2, ans;
6
7     value1 = 5;
8     value2 = 3;
9
10    ans = value1 - value2; // Bug, Defect
11
12    printf("The addition of 5 + 3 = %d.", ans);
13
14    return 0;
15 }
```

Now lets go back to the program and analyze what was the fault in the program.

There is a '-' sign present instead of '+' sign. So the fault in the program is the '-' sign.

Error is the mistake I made by typing '-' instead of '+' sign.

## Error, Defects, Fault and Failure

A tester does not necessarily have access to the code and may be just testing the functionality of the program. In that case the tester will realize the output is faulty and will raise a defect.

- **Error:** A mistake made by a programmer
  - Example: Misunderstood the requirements.
- **Defect/fault/bug:** Manifestation of an error in a program.

<i>Example:</i> Incorrect code: if ( $a < b$ ) {foo( $a, b$ );} Correct code: if ( $a > b$ ) {foo( $a, b$ );}
---

- **Failure:** Manifestation of one or more faults in the observed program behavior

## Testing & Debugging

**Testing:** Finding inputs that cause the software to fail

- Finding and localization of a defect
- Done by Testing team
- Intention behind is to find as many as defects possible

**Debugging:** The process of finding a fault given a failure

- Fixing that defect
- Done by development team
- Intention is to remove those defects

*“...testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.”*

<http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD0340.html> Edsger Dijkstra

## Trivial example

- Try to move a file from folder A to another folder B
- What are the possible scenarios?

## Possible Solutions

---

- Trying to move the file when it is open
  - You do not have the security rights to paste the file in folder B
  - Folder B is on a shared drive and storage capacity is full
  - Folder B already has a file with the same name
- 

## Another Scenario

---

- Suppose you have 15 input fields each one having 5 possible values.
- How many combinations to be tested?

$$5^{15} = 30517578125!!!$$

---

*Software Testing is a process, or a series of processes, designed to make sure computer code does what it is designed to do and that it does not do anything unintended.*

## A Self Assessment Test

Write a set of test cases - specific set of data - to properly test a relatively Simple program.

Create a set of test data for the program .

The program reads three integer values from an input dialog. The three value represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.

Test Action and Data	Expected Result
2, 5, 10	??
...	...

## A Self Assessment Test

The function triangle takes three integer parameters that are interpreted as the lengths of the sides of a triangle. It returns whether the triangle is equilateral (three lengths equal), isosceles (two lengths equal), scalene (no lengths equal), or invalid (impossible lengths).

```
final int EQUILATERAL = 0;
final int ISOSCELES = 1;
final int SCALENE = 2;
final int INVALID = 3;
int triangle(int a, int b, int c)
{
    if (a >= b+c || b >= a+c || c >= a+b)
        return(INVALID);
    if (a == b && b == c)
        return(EQUILATERAL);
    if (a == b || a == c || b == c)
        return(ISOSCELES);
    return(SCALENE);
}
```

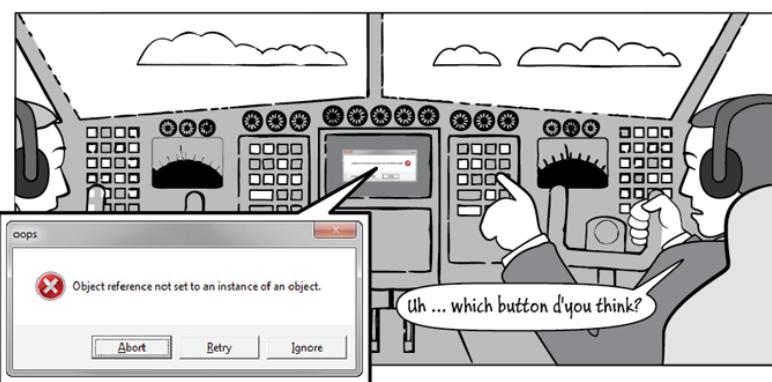
## Solution?

Tester Action and Data	Expected Result
Erroneous input partition a, b, or c a non-number	an error message
Scalene triangle partition 2, 3, 4 1, 2, 3	scalene an error message
Isosceles triangle partition 2, 2, 1 2, 2, 0	isosceles an error message
Equilateral triangle partition 1, 1, 1 1, 1, -1	equilateral an error message
User interface tests Try a leading space for a, b, c	an error message
Many more...	

## Summary: Why Do We Test Software ?

A tester's goal is to eliminate faults as early as possible

- Improve quality
- Reduce cost
- Preserve customer satisfaction



Software should be predictable and consistent, offering no surprises to users.

Source: Microsoft Academia



Software should be predictable and consistent, offering no surprises to users.

### The Future of Software Testing

Year: 2031  
Location: Undisclosed office interview room  
Reason: Interviewing for a Software Engineer/Developer role

So tell me, why  
are you applying  
for this role?

It's the easiest way to get into a  
development department. Once I  
get my foot in the door I hope I  
might be able to impress and  
become a tester in the testing team.

AG

## The Mind of a TESTER

---

Four different kinds of thinking exhibited by a GOOD TESTER  
*[Kaner, Bach, Pettichord]*

1. Technical Thinking: *the ability to model technology and understand causes and effects*
  2. Creative Thinking: *the ability to generate ideas and see possibilities*
  3. Critical Thinking: *the ability to evaluate ideas and make inferences*
  4. Practical Thinking: *the ability to put ideas into practice*
- 

## Example to TEST YOUR Mind

---

An example of these kinds of thinking is found in a fable called  
“*The King’s Challenge*”

*The King’s Challenge (a fable)*

*Once upon a time, a mighty king wanted to determine which of his three court wizards was the most powerful.*

*So,*

*He put the three court wizards in the castle dungeon and declared whoever escaped from his respective dungeon first was the most powerful wizard in all the kingdom.*

*(If you are one of three wizard, what you would do.)*

---

## Example to TEST YOUR Mind...

The **first wizard** immediately started chanting musical poems to open his cell door.

The **second wizard** immediately started casting small polished stones and bits of bone on the floor to learn how he might open his cell door

The **third wizard** sat down across from his cell door and thought about the situation for a minute. Then he got up, walked over to the cell door and pulled on the door handle. The cell door swung open because it was closed but not locked.

Thus, the third wizard escaped his cell first and became known as the most powerful wizard in all the kingdom.

## Example to TEST YOUR Mind...

What kind of “tester” thinking did the third wizard exercise in solving the king’s puzzle?

- Creative thinking: *the ability to see the possibility that the door was not locked in the first place.*
- Practical thinking: *the ability to decide to try the simplest solution first.*

## Another Example to TEST YOUR Mind...

---

### BUYING A CAR

-Go for a test drive & What you are supposed to do?

- Take test drive to BREAK THE CAR

OR

- To improve the CAR's DESIGN

Objectives of TEST DRIVE are:

- To validate affordability
  - To validate attractiveness
  - To validate comfort
  - To validate usefulness
  - To validate performance
- 

When you have clear testing objectives, then we choose approaches that best validate the car against those Objectives.

Testing Approaches include:

- Examine the sticker price and sale contract
- Trying out the radio, the air conditioner and the lights
- Trying acceleration, stopping and cornering

These testing approaches are referred to by fairly common terminology in the testing industry.

*How??*

---



---

**Examine = Static Testing**

(observe, read, review without actually driving the car)

**Try out = Functional and structural testing**

(work different features of the car without actually driving the car)

**Try = Performance Testing**

(work different features of the car by actually driving the car)

---



## The Psychology of Testing

---

One of the primary causes of POOR program testing is - most of the programmers begin with a FALSE definition of the term:

Such as,

- “*Testing is the process of demonstrating that error are not present*”
- “*The purpose of testing is to show that a program performs its intended functions correctly*”
- “*Testing is the process of establishing confidence that a program does what it is supposed to do*”

Testing is the process of executing a program with the intent of finding errors.

---

**Questions??**



DA-IICT

## IT 314: Software Engineering

*Psychology of Testing  
&  
Testing Principles*

Saurabh Tiwari

1

*Software Testing is a process, or a series of processes, designed to make sure computer code does what it is designed to do and that it does not do anything unintended.*

- Suppose you have a 15 input fields each one having 5 possible values.
- How many combinations to be tested?

$$5^{15} = 30517578125!!$$

Education means considerably more than just teaching a student to read, write, and manipulate numbers. Computers, the Internet, and advanced electronic devices are becoming essential in everyday life and have changed the way information is gathered. How this new technology is utilized in the curriculum and managed by teachers will have an important role to play in widening the resource and knowledge base for all students. Technology affects the way teachers teach and students learn. To make the best use of information technology (IT), schools need a workable plan to fully integrate it into all aspects of the curriculum so students are taught how, why, and when to use technology to further enhance their learning. If a school does not have a clear plan of how and why it wishes to implement IT, then it runs the risk of wasting money. In schools today, nearly all classrooms have access to a computer. However, many schools mistake this as incorporating information technology into the curriculum. School staff needs to research what IT is available and what would best serve the school's purpose, not simply purchase the latest equipment. There should be a policy stating how IT is going to assist pupils' development and what teachers want pupils to achieve. Staff members need to be clear about what they want IT to do for them before they can start incorporating it into their lessons.

## Psychology of Testing

### The Mind of a TESTER

Four different kinds of thinking exhibited by a GOOD TESTER  
[Kaner, Bach, Pettichord]

1. Technical Thinking: *the ability to model technology and understand causes and effects*
2. Creative Thinking: *the ability to generate ideas and see possibilities*
3. Critical Thinking: *the ability to evaluate ideas and make inferences*
4. Practical Thinking: *the ability to put ideas into practice*

## Example to TEST YOUR Mind

An example of these kinds of thinking is found in a fable called “The King’s Challenge”

*The King’s Challenge (a fable)*

*Once upon a time, a mighty king wanted to determine which of his three court wizards was the most powerful.*

*So,*

*He put the three court wizards in the castle dungeon and declared whoever escaped from his respective dungeon first was the most powerful wizard in all the kingdom.*

*(If you are one of three wizard, what you would do.)*

## Example to TEST YOUR Mind...

The **first wizard** immediately started chanting musical poems to open his cell door.

The **second wizard** immediately started casting small polished stones and bits of bone on the floor to learn how he might open his cell door

The **third wizard** sat down across from his cell door and thought about the situation for a minute. Then he got up, walked over to the cell door and pulled on the door handle. The cell door swung open because it was closed but not locked.

Thus, the third wizard escaped his cell first and became known as the most powerful wizard in all the kingdom.

## Example to TEST YOUR Mind...

What kind of “tester” thinking did the third wizard exercise in solving the king’s puzzle?

- Creative thinking: *the ability to see the possibility that the door was not locked in the first place.*
- Practical thinking: *the ability to decide to try the simplest solution first.*

## Another Example to TEST YOUR Mind...

### BUYING A CAR

-Go for a test drive & What you are supposed to do?

- Take test drive to BREAK THE CAR  
OR
- To improve the CAR's DESIGN

Objectives of TEST DRIVE are:

- To validate affordability
- To validate attractiveness
- To validate comfort
- To validate usefulness
- To validate performance

When you have clear testing objectives, then we choose approaches that best validate the car against those Objectives.

Testing Approaches include:

- *Examine the sticker price and sale contract*
- *Trying out the radio, the air conditioner and the lights*
- *Trying acceleration, stopping and cornering*

These testing approaches are referred to by fairly common terminology in the testing industry.

*How??*

**Examine = Static Testing**  
(observe, read, review without actually driving the car)

**Try out = Functional and structural testing**  
(work different features of the car without actually driving the car)

**Try = Performance Testing**  
(work different features of the car by actually driving the car)

## The Psychology of Testing

One of the primary causes of POOR program testing is - most of the programmers begin with a FALSE definition of the term:

Such as,

- “*Testing is the process of demonstrating that error are not present*”
- “*The purpose of testing is to show that a program performs its intended functions correctly*”
- “*Testing is the process of establishing confidence that a program does what it is supposed to do*”

Testing is the process of executing a program with the intent of finding errors.

## Testing Terms

- **Test case** - A set of *Inputs, execution preconditions, and expected outcomes* for testing an specific aspect of CUT
- **Test Suite** - A collection of test cases for the CUT
- **Test Criterion** - A set of test requirements
- **Effectiveness** - *Fault detection capability*
- **Efficiency** - *The average testing cost (i.e. effort) to identify a fault in the program*

## General Testing Principles

### The Seven Key Principles

#### 1. Testing shows the presence of Defects

( it is not a ghost. It is a fact)

- We test to find Faults (as known as Defects)
- As we find more defects, the probability of undiscovered defects remaining in a system reduces (decreasing nature).
- However Testing cannot prove that there are no defects present

## Why Testing is necessary

### Testing Pearls of Wisdom



*"The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that program"*

- *"Do not plan a test effort under the tacit assumption that no errors will be found"*
- *"A good test is one that has a high probability of detecting an as yet undiscovered error"*
- *"A successful test is one that detects an as-yet undiscovered error"*

To test a program is to try to make it fail

## General Testing Principles

### The Seven Key Principles

#### 2. Exhaustive Testing is Impossible!

- We have learned that we cannot test **everything** (i.e. all combinations of inputs and pre-conditions).
- That is we must Prioritise our testing effort using a Risk Based Approach.

## Why Testing is necessary

### Why don't we test everything ?

System has 20 screens  
 Average 4 menus / screen  
 Average 3 options / menu  
 Average of 10 fields / screen  
 2 types of input per field  
 Around 100 possible values

**Approximate total for exhaustive testing**  
 $20 \times 4 \times 3 \times 10 \times 2 \times 100 = 480,000$  tests

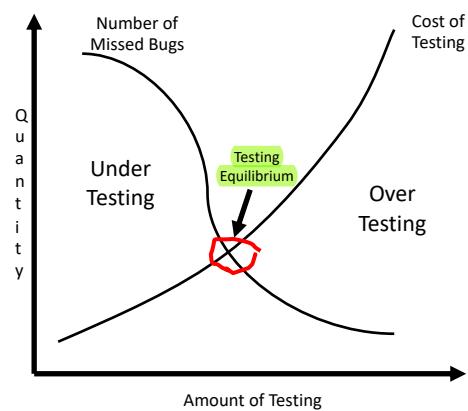
Test length = 1 sec then test duration = 17.7 days  
 Test length = 10 sec then test duration = 34 weeks  
 Test length = 1 min then test duration = 4 years  
 Test length = 10 mins then test duration = 40 years!



It is not a matter of time. But, time is money ( salary is taken by hour. So second is valuable for software houses)

## Urgency of Equilibrium

- If you test too little, the probability of software failure increases
- If you try to test too much, the development cost becomes unaffordable
- So, we need to conduct some sort of equilibrium



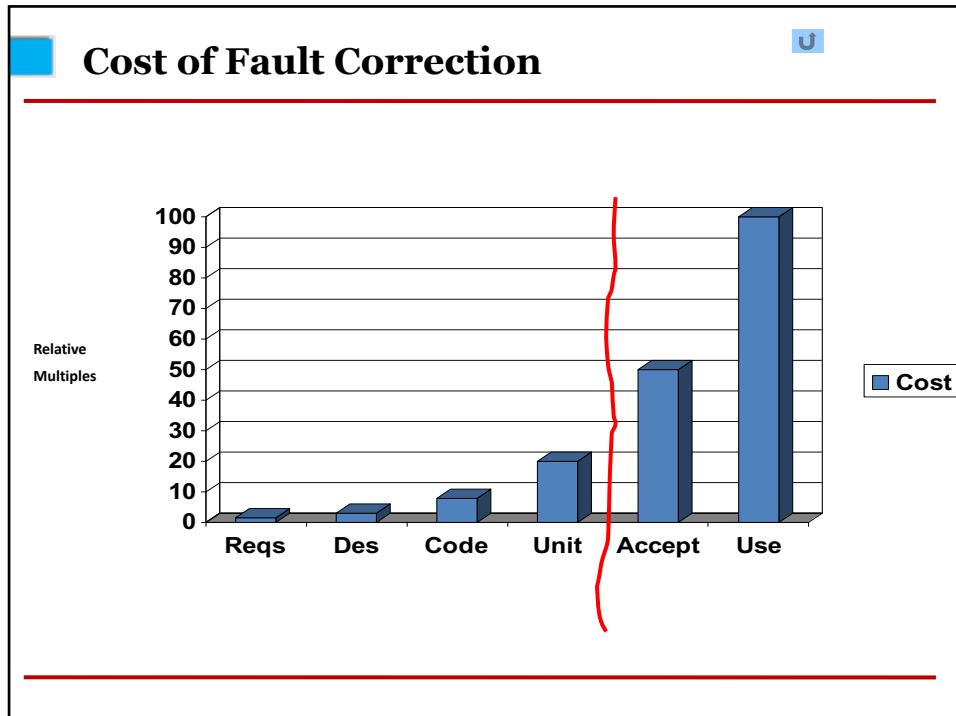
## General Testing Principles

### The Seven Key Principles

#### 3. Early testing

- Testing activities should start as early as possible in the development life cycle
- These activities should be focused on defined objectives - outlined in the Test Strategy
- Remember from our Definition of Testing, that Testing doesn't start once the code has been written!





## General Testing Principles

### The Seven Key Principles

#### 4. Defect Clustering

- Defects are not **evenly distributed** in a system
- They are '**scattered**' and may get '**clustered**'
- In other words, **most defects found during testing are usually confined to a small number of modules** (**80% of uncovered errors focused in 20% modules of the whole application**)
- Similarly, **most operational failures of a system are usually confined to a small number of modules**
- An important consideration in test prioritisation!

## General Testing Principles

### The Seven Key Principles



#### 5. The Pesticide Paradox

- Testing identifies bugs, and programmers respond to fix them
- As bugs are eliminated by the programmers, the software improves
- As software improves the effectiveness of previous tests erodes
- Therefore we must learn, create and use new tests based on new techniques to catch new bugs (i.e. It is not a matter of repetition. It is a matter of learning and improving)

## General Testing Principles

### The Seven Key Principles

#### 6. Testing is Context (background) Dependent

- Testing is done differently in different contexts
- For example, safety-critical software is tested differently from an e-commerce site
- Whilst, Testing can be 50% of development costs, in NASA's Apollo program (it was 80% testing)
- 3 to 10 failures per thousand lines of code (KLOC) typical for commercial software
- 1 to 3 failures per KLOC typical for industrial software
- 0.01 failures per KLOC for NASA Shuttle code!
- Also different industries impose different testing standards

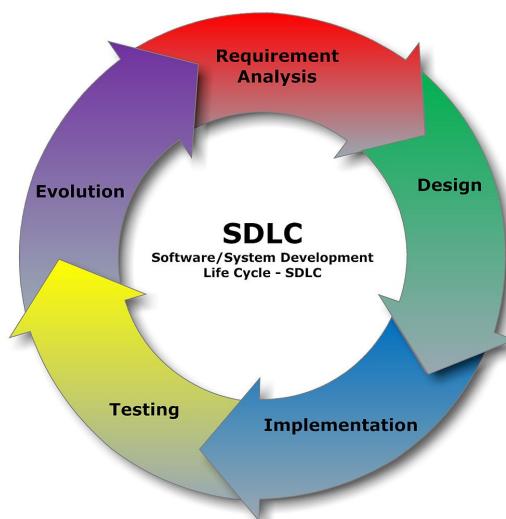
## General Testing Principles

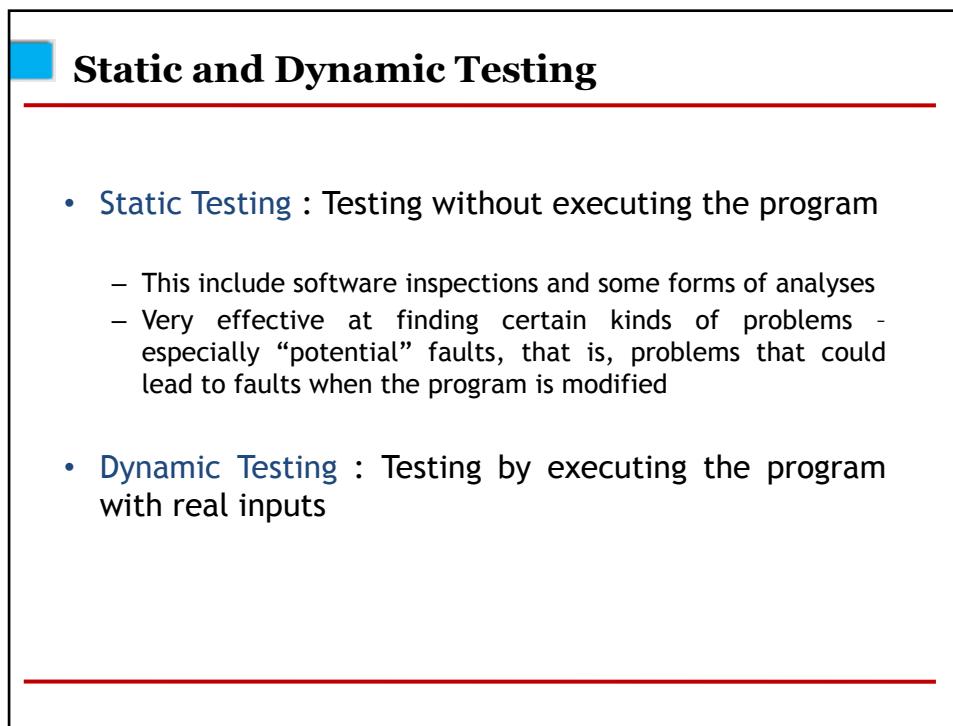
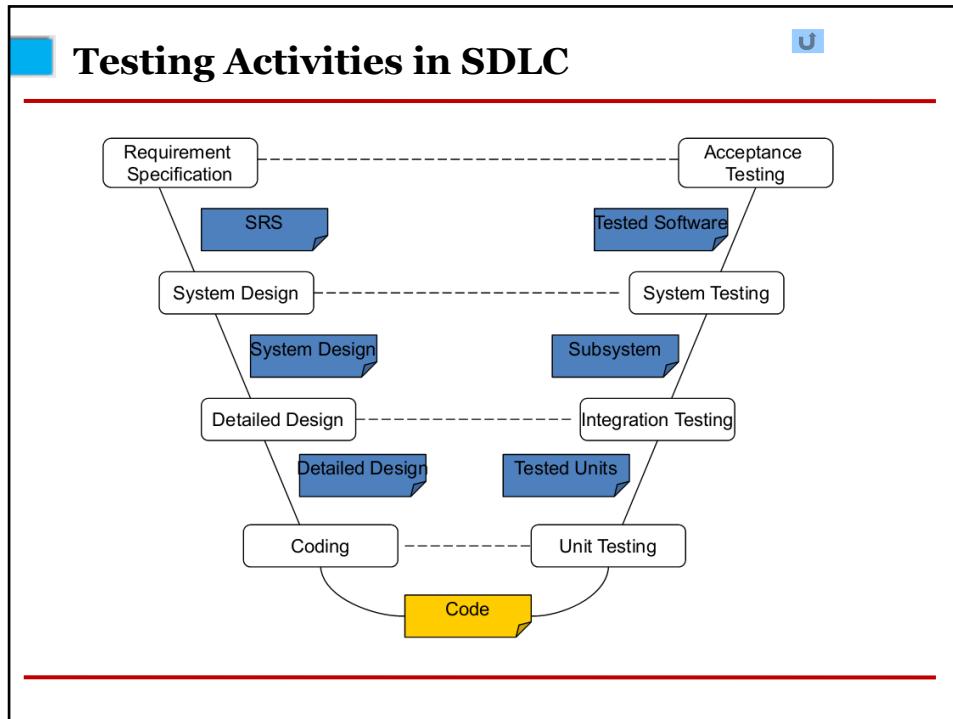
### The Seven Key Principles

#### 7. Absence of Errors Fallacy

- If we build a system and, in doing so, find and fix defects ....  
It doesn't make it a good system
- Even after defects have been resolved, it may still be unusable  
and/or does not fulfil the users' needs and expectations

## SDLC: Where Testing FITS?





**Questions??**