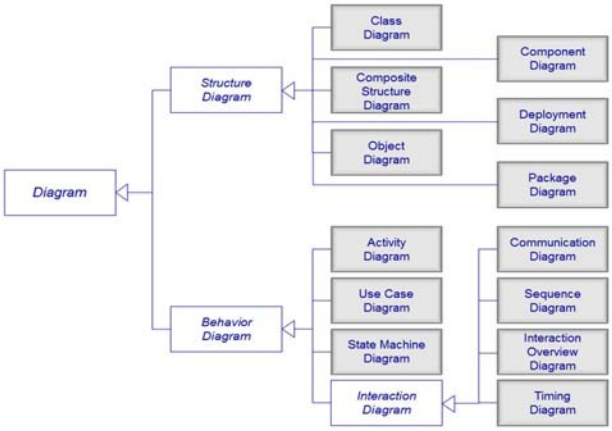
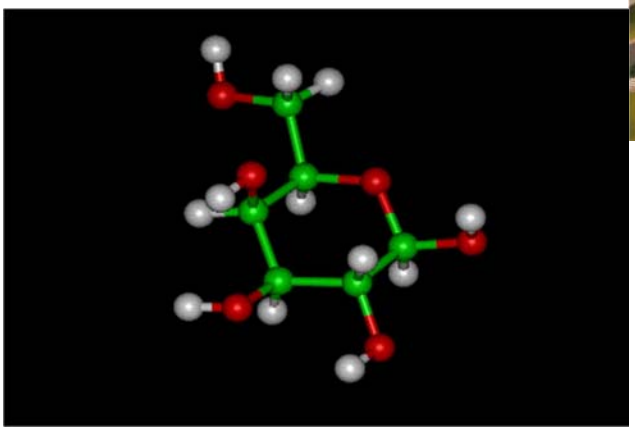
# Object-Oriented Analysis and Design using UML and Patterns

## Unified Modeling Language (UML)

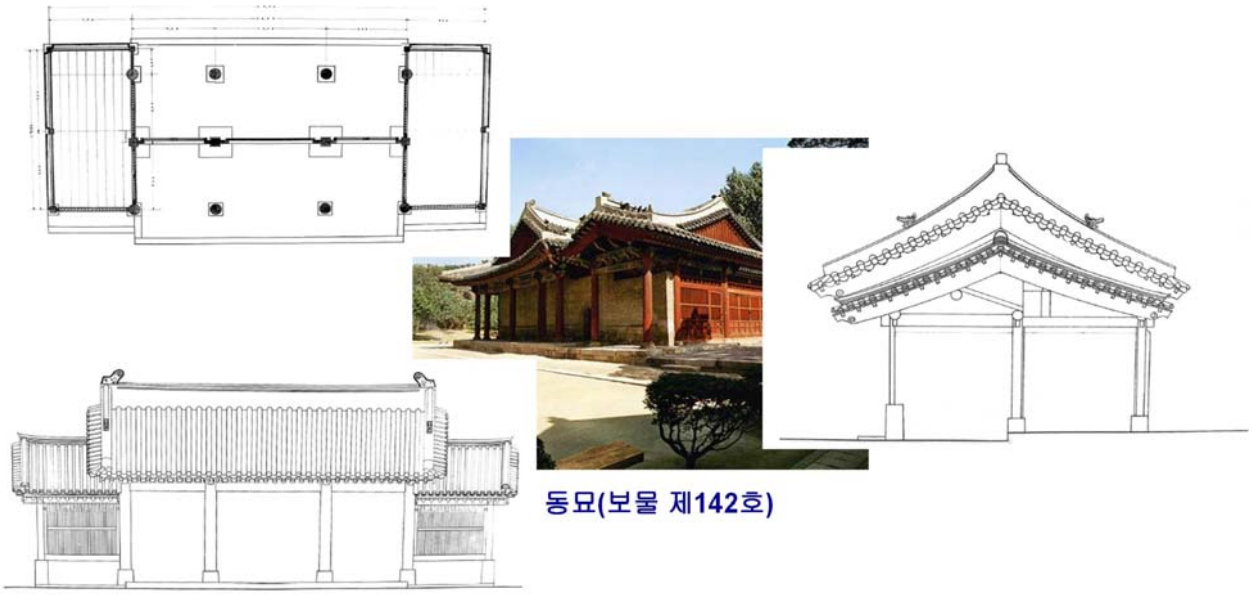

# What is a "model"?

**A model is a simplification of reality**



**Models capture the essential aspects of a system which are relevant to a given level of abstraction**

# Every system may be described from different aspects using different models

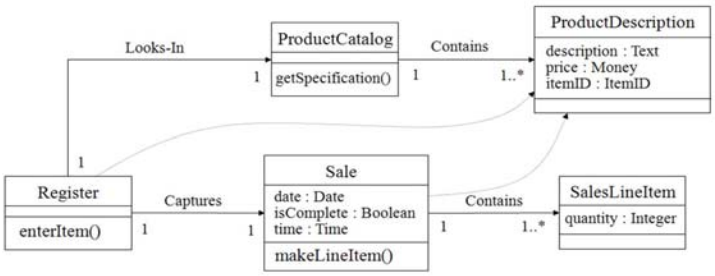


동묘(보물 제142호)

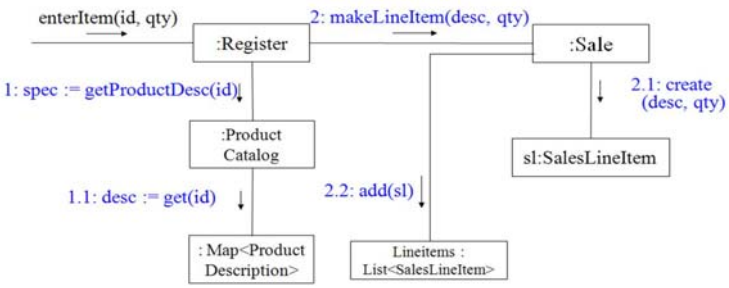


# A model may be structural or behavioral

**Static models:**
**describe a structural properties of a system**




**Dynamic models:**
**describe a behavioral properties of a system**



## We build models so that we can better understand the system we are developing

We build models of complex systems because we cannot comprehend such a system in its entirety

Through modeling, we achieve four aims:

- To visualize a system as it is or as we want it to be
- To specify the structure or behavior of a system
- To give a blueprint to construct a system
- To document the decisions we have made

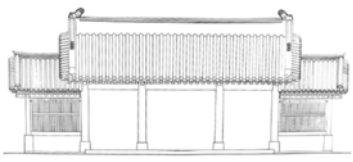


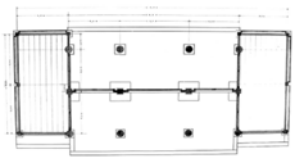
## Principles of modeling

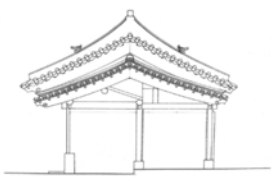
The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped

Every model may be expressed at different levels of precision

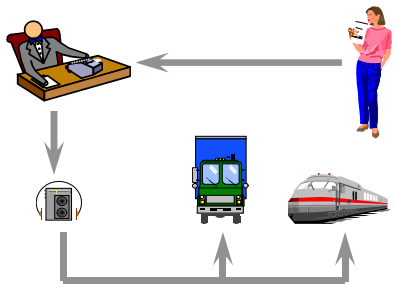
No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models





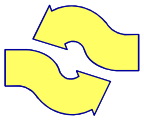


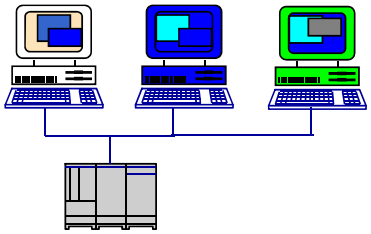
# What is visual modeling?



Business Process

*"Modeling captures essential parts of the system."*
*Dr. James Rumbaugh*





Computer System

***Visual modeling is modeling using standard graphical notations***



# UML is a standard visual modeling language

**Leading notations among > 50 ( ~ mid 90's):**

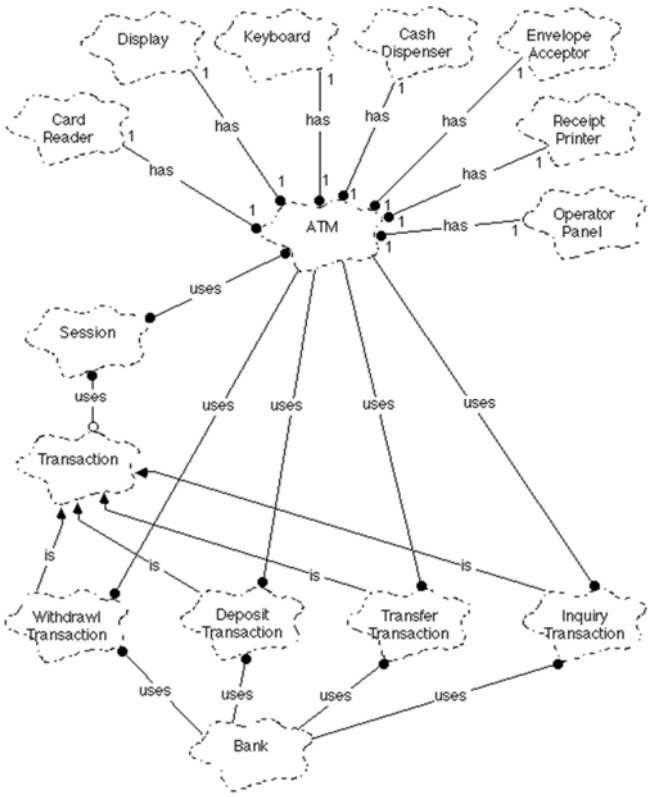
- **Booch**
- **OMT**

**New OMG standard (since 1997):**

- **Unified Modeling Language (UML)**
  - **Visual notation and semantics**
  - ***Process independent!***
  - ***www.omg.org***

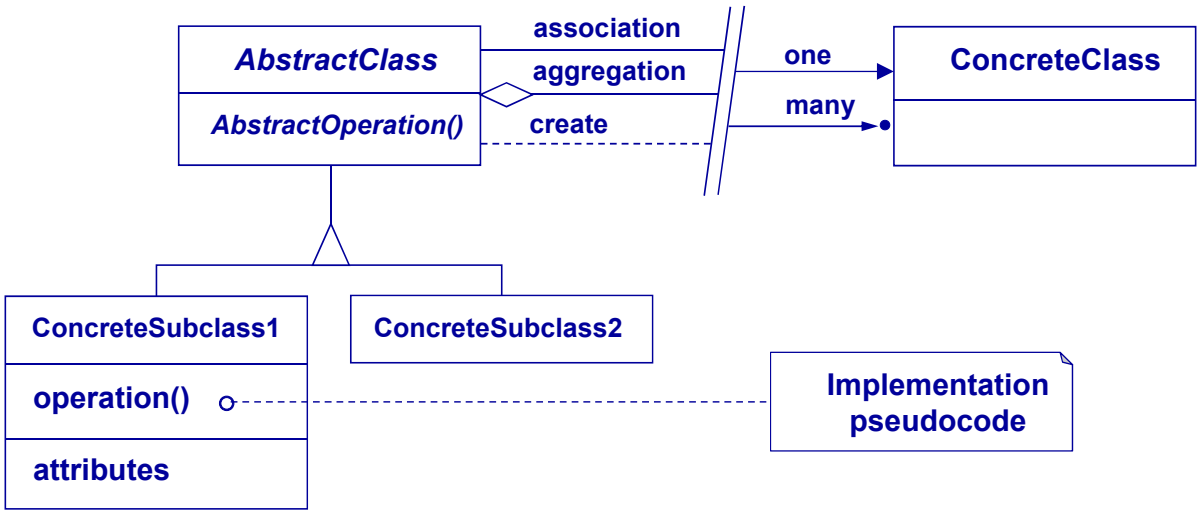
UNIFIED MODELING LANGUAGE

# Booch: Class Diagram

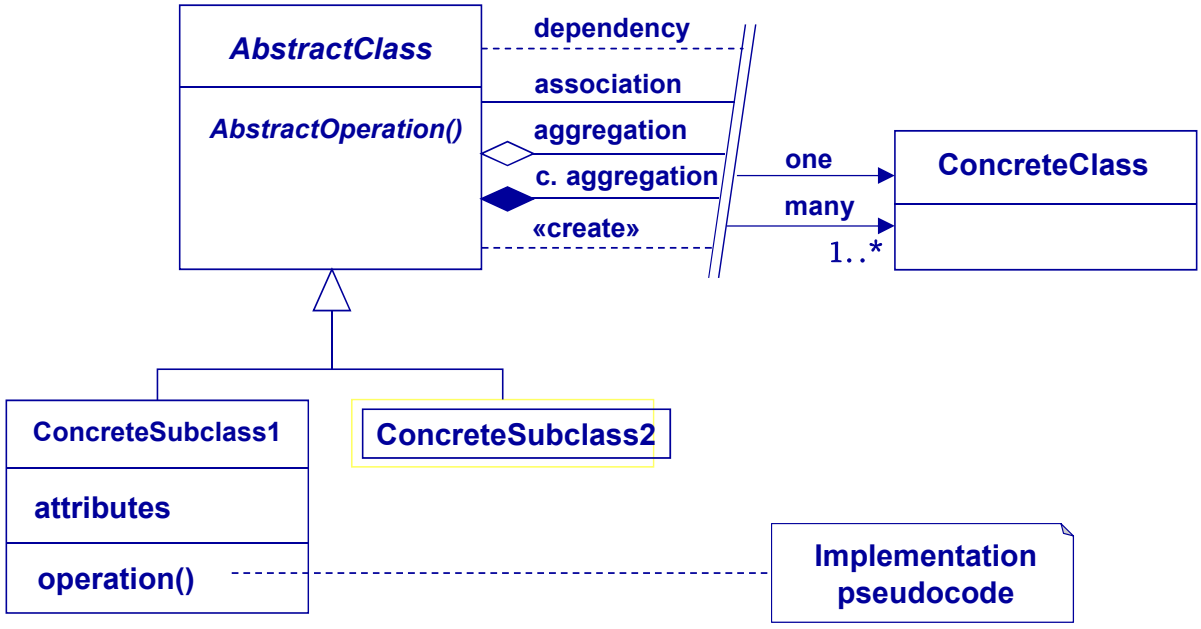





# OMT: Class Diagram



# UML: Class Diagram






# UML attempts in being unified across several different domains (not just historical)

**Development life cycle**

- from requirements engineering to implementation

**Application domains**

- from hard real-time embedded systems to management decision support systems

**Implementation languages and platforms**

- language and platform neutral

**Development processes**

- development process neutral

**Its own internal concepts**

- consistent and uniform in its application of small set of internal concepts

# Where can the UML be used?

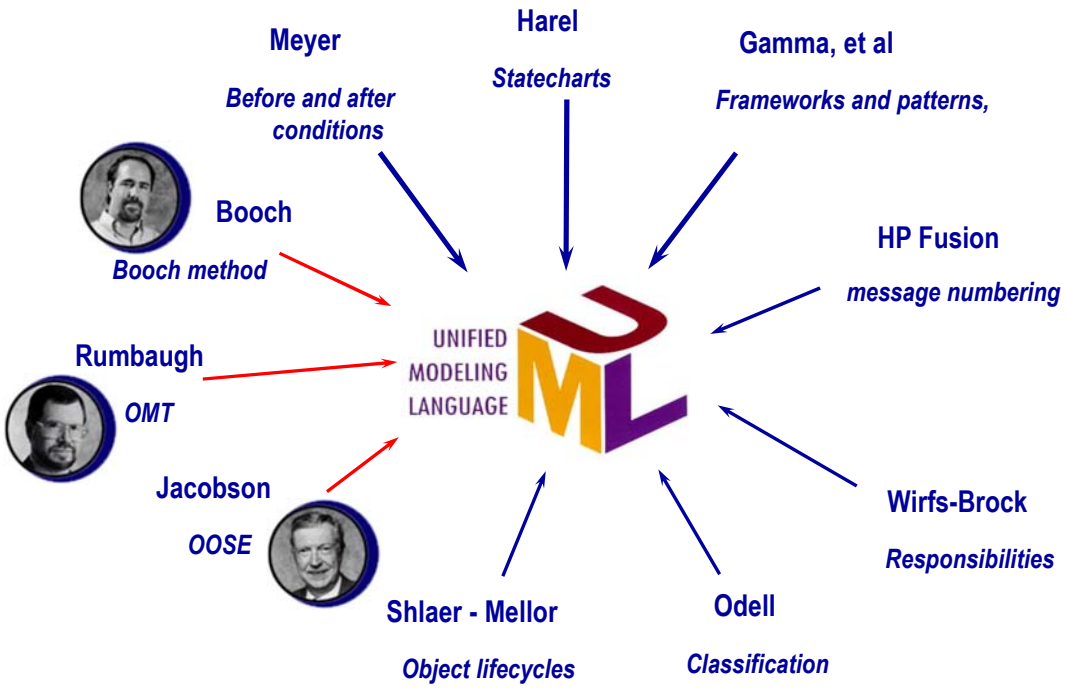
***The UML is primarily intended for software-intensive systems (oriented towards OO systems)***

- Enterprise information systems
- e-commerce
- Banking and insurance
- Computer games
- Command and control
- Telephony
- Defense/aerospace
- Medical electronics
- etc.

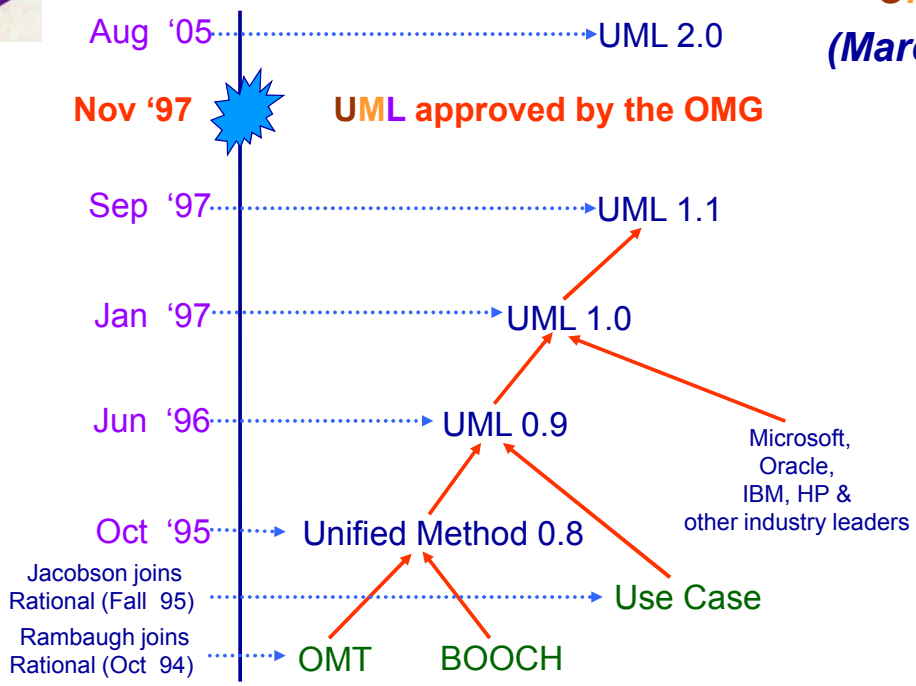
***However, UML can also be used to model non-software systems such as workflow.***



# Contributions to the UML



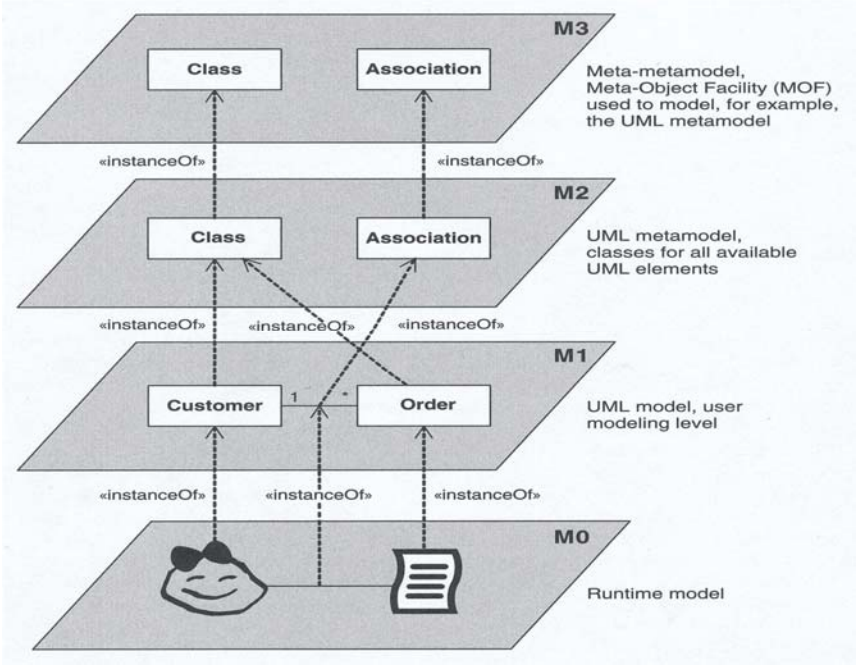
# History of UML



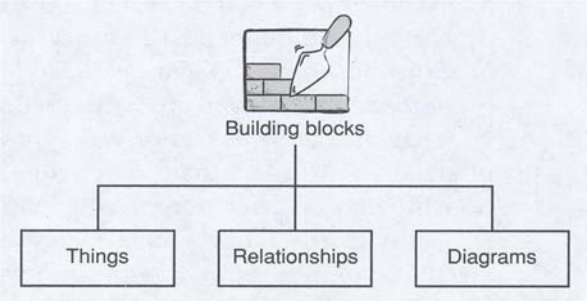

***Latest version***
***UML 2.4***
***(March 2011)***



# The Four layer Meta-model Hierarchy



# Basic Building Blocks




**Things** (aka, **modeling elements**)

- **Structural things**
  - nouns of UML models, static parts
- **Behavioral things**
  - verbs of UML models, dynamic parts
- **Grouping things**
  - organizational parts
- **Anotational things**
  - explanatory parts

**Relationships**

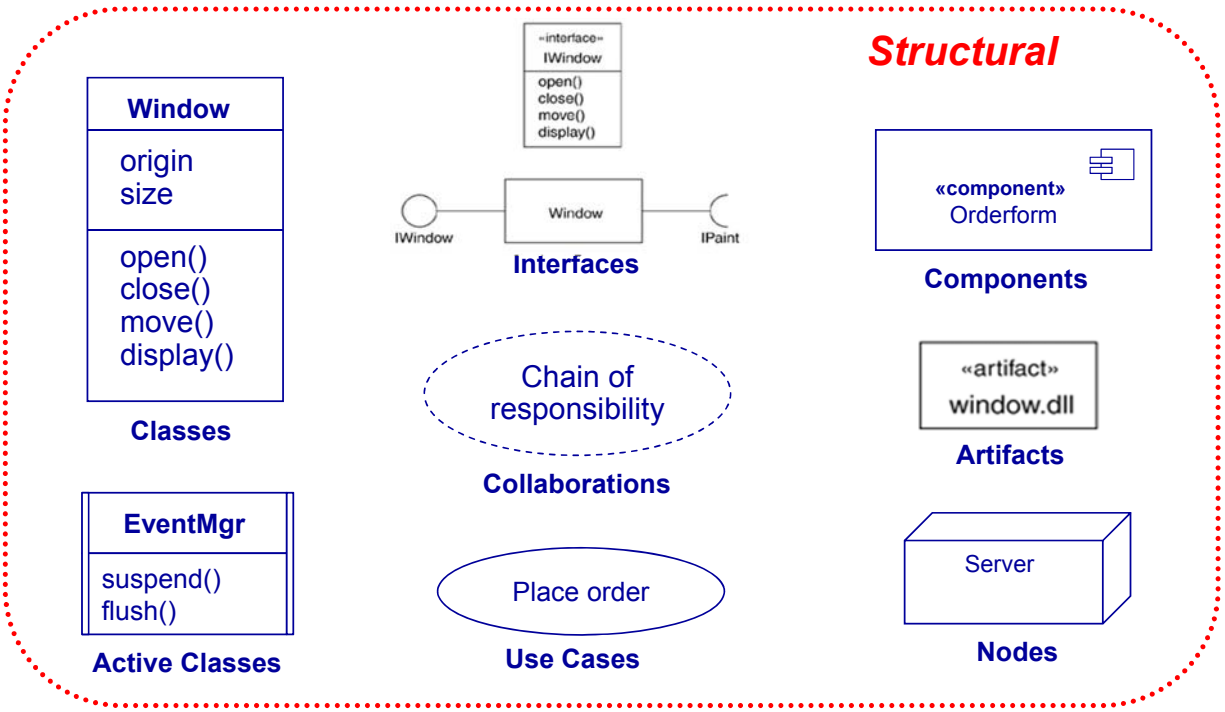
- How two or more things relate to each other

**Diagrams**

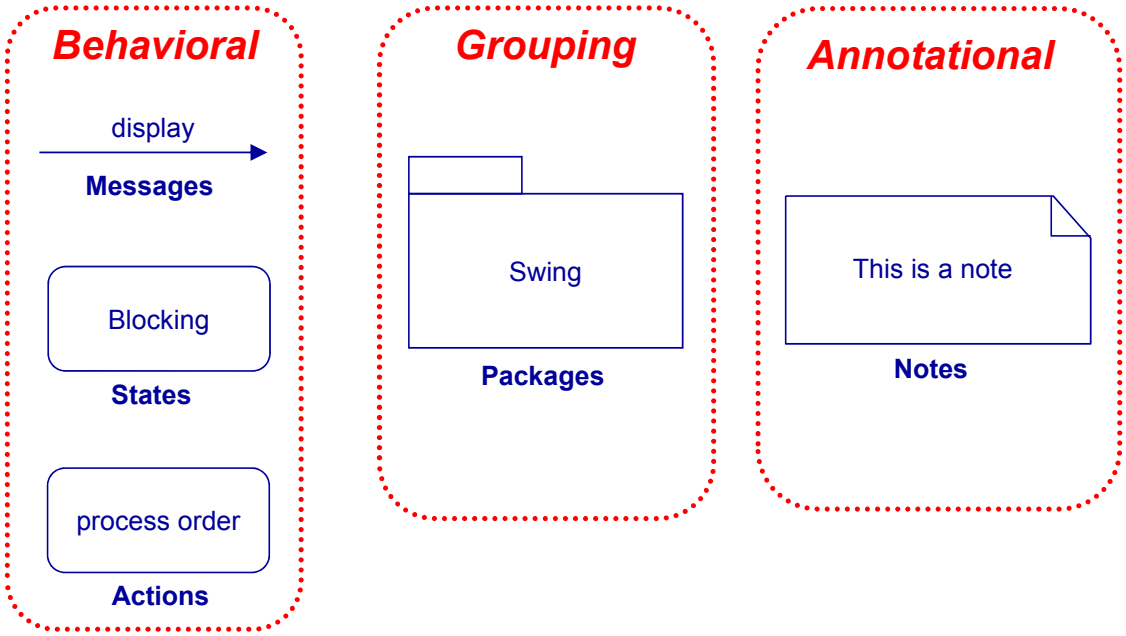
- Group interesting things together
- Only *views* into the model



# Things in UML



# Things in UML (Cont'd)






# Relationships

| | source → target | |
|---|---|---|
| **Dependency** | ----------> | The source element depends on the target element and my be affected by changes to it |
| **Association** | ———— | The description of set of links between elements |
| **Aggregation** | ◇———— | The target is a part of the source element |
| **Composition** | ◆———— | A strong (more constrained) form of aggregation |
| **Containment** | ⊕———— | The source element contains the target element |
| **Generalization** | ————▷ | The source element is a specialization of the more general target element and my be substituted for it |
| **Realization** | ----------▷ | The source element guarantees to carry out the contract specified by the target element |

# Extensibility Mechanisms of UML

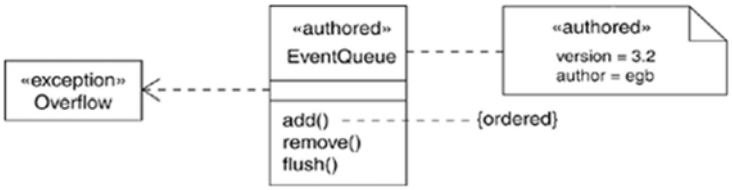
**Allows you to extend the language in controlled ways**

**Stereotypes (e.g., «subsystem», «utility»)**
- introduce new modeling elements derived from existing ones

**Tagged values (e.g., {author=kim}, {query})**
- add new properties to stereotype's specification

**Constraints (e.g., { radius > 0 })**
- add new rules or modify existing rules

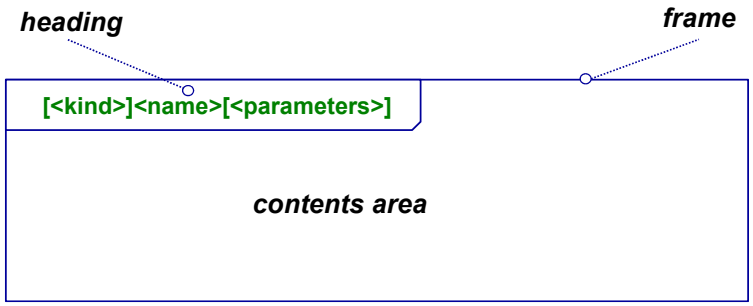
**UML profile (e.g., UML profile for CCM, EAI, MARTE etc.)**
- customize UML models for particular domains and platforms by defining a collection of constraints and stereotypes





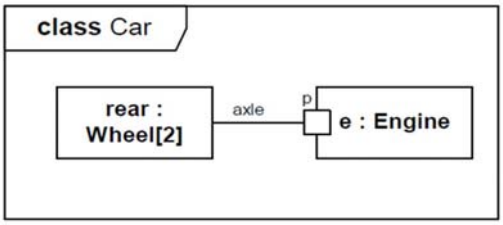

# Overview of Diagrams

**Diagrams are graphical representations of parts of UML models.**
**Each diagram has a *contents area* + optional *frame* and *heading***



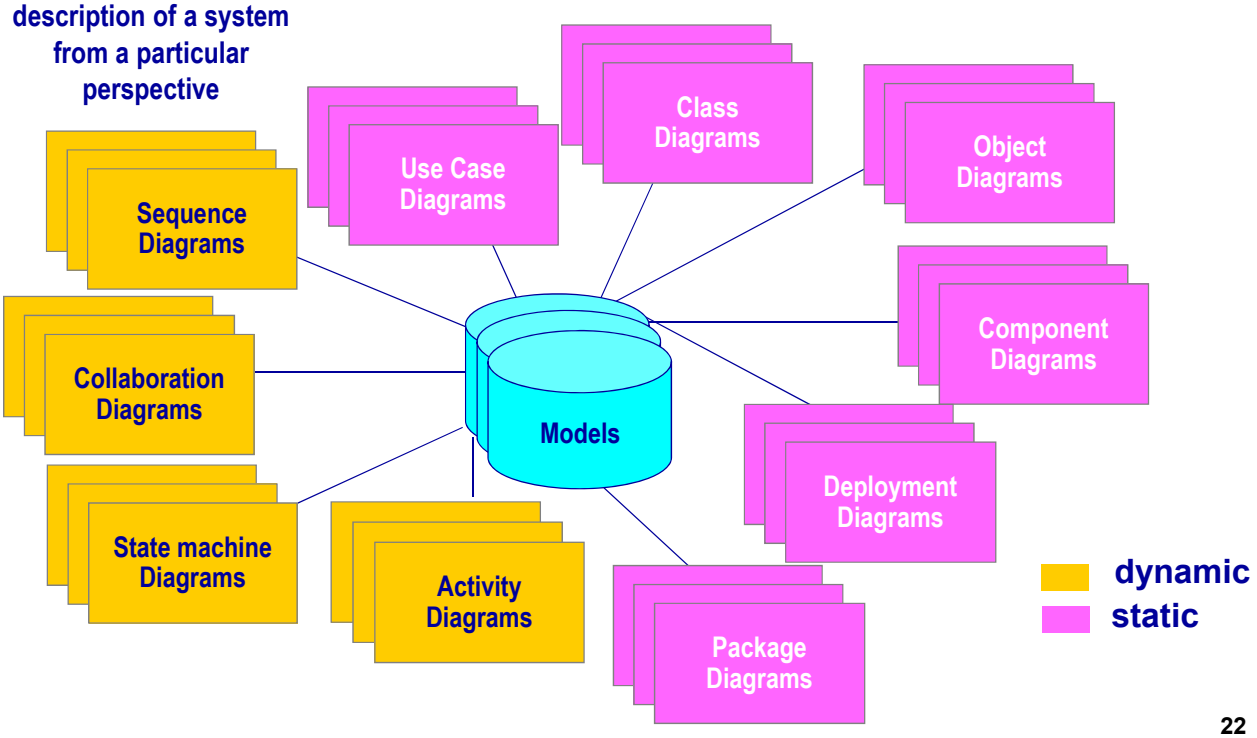

**<kind>**
**activity (act)**
**class**
**component (cmp)**
**deployment (dep)**
**interaction (sd)**
**package (pkg)**
**state machine (stm)**
**use case (uc)**

**Eg.) Composite Structure Diagram**



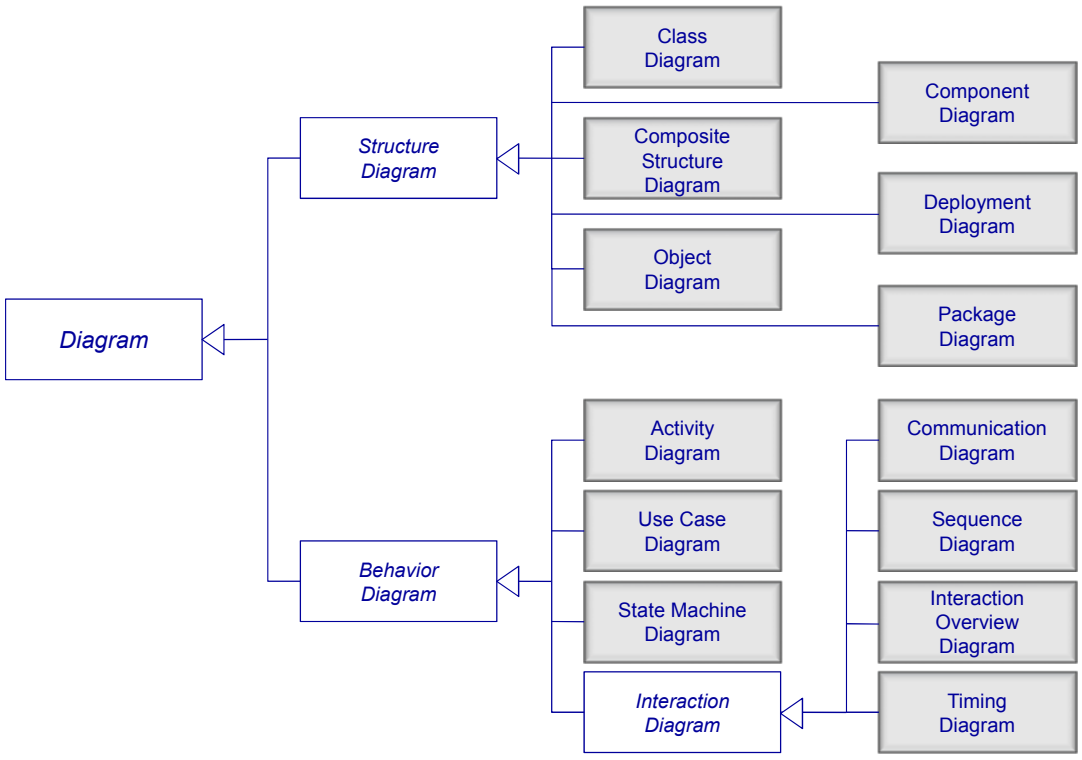
# Models and UML 1.x Diagrams

A *model* is a complete description of a system from a particular perspective



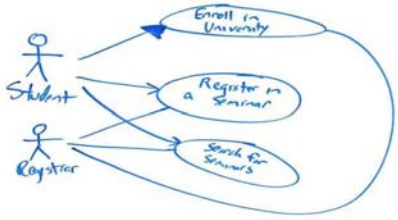



# Classification of UML 2.0 Diagrams



# Ways of Using UML

**UML as a Sketch**

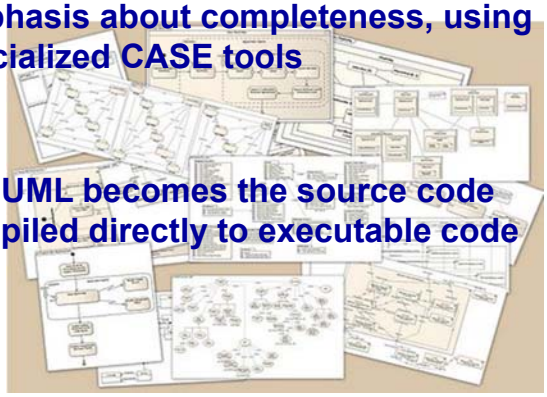



**Emphasis is on selective communication rather than complete specification**

**Developers use the UML to help communicate some aspects of a system using lightweight drawing tools**

**UML as a Blueprint**

**Emphasis about completeness, using specialized CASE tools**



**UML as a Programming Language**

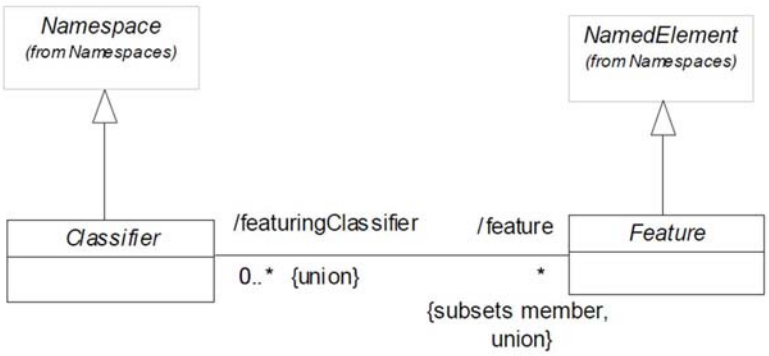
**The UML becomes the source code compiled directly to executable code**



# Classifiers
## (abstract metaclass)

**A classifier is a classification of instances – it describes a set of instances that have features in common**

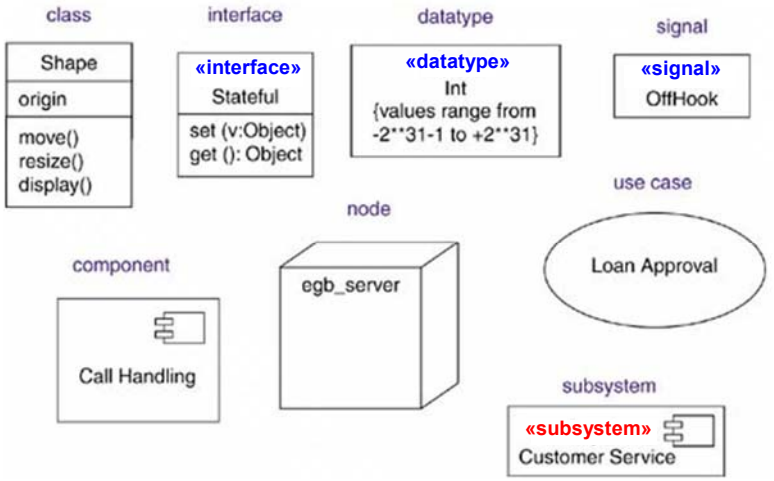
**A feature declares a behavior or structural characteristics of instances of classifiers**



# Concrete Subclasses of Classifier

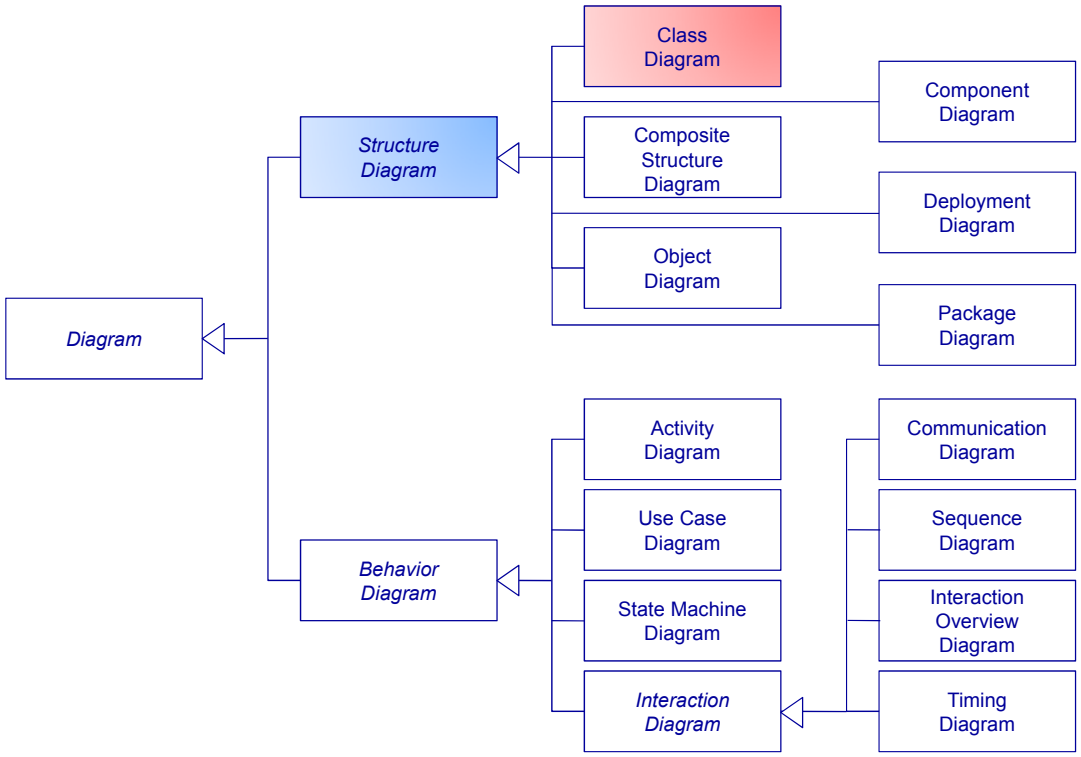
**Classifiers include classes, associations, interfaces, datatypes, signals, components, nodes, use cases, and subsystems**

**Icons**






# Class Diagram



# Class Diagram

*A class diagram shows the existence of classes (and interfaces) and their relationships in the logical view of a system*

**A class is a classifier whose features are attributes and operations**

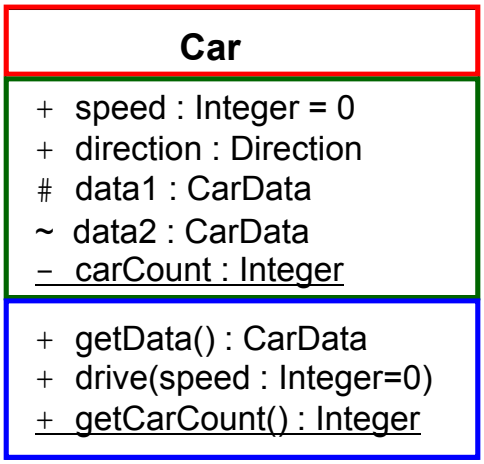
**UML modeling elements**

- Classes and Interfaces
- Association, Aggregation, Composition, Dependency, and Generalization relationships
- Role names, Multiplicity, Navigation indicators
- Stereotypes
- Tagged values



# Class Icon

**Class icon consists of *compartments***

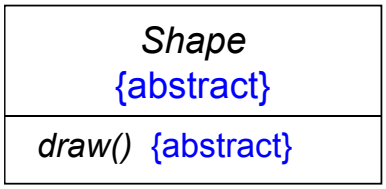



(a) Concrete class

```
class Car {
  public int speed;
  public Direction direction;
  protected CarData data1;
  CarData data2;
  static private int carCount;
  public CarData getData(){...}
  public void drive(int speed){...}
  static public int getCarCount(){...}
}
```

☞ visibility ::= {+|-|#|~}

# Class Icon (Cont'd)




(b) Abstract class

In Java:

```
abstract class Shape {
  public abstract void draw();
}
```

In C++:

```
class Shape {
public:
  virtual void draw() = 0;
};
```



# Tagged Values

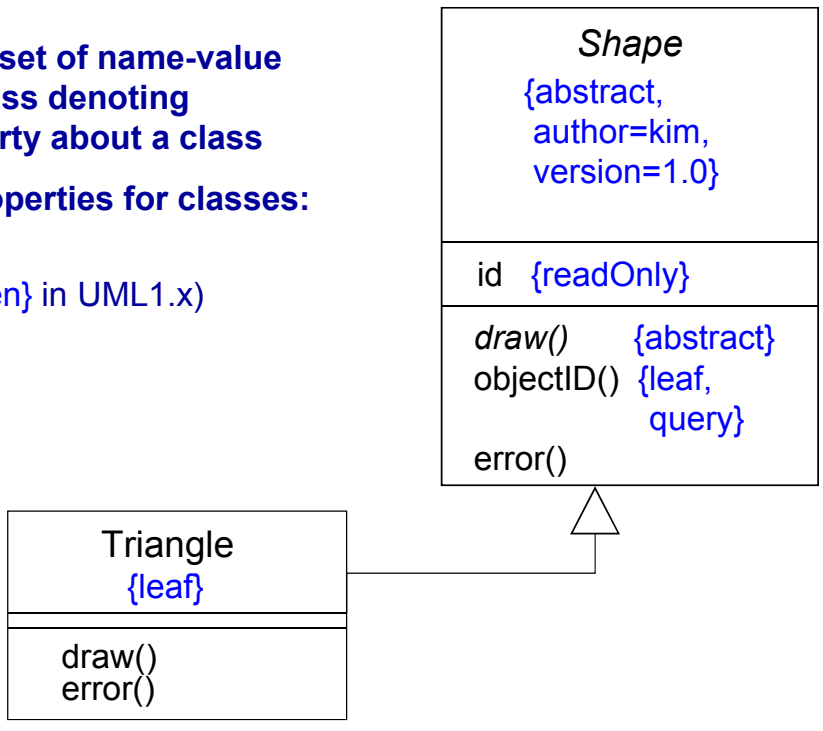
**Tagged values are a set of name-value associated with a class denoting information or property about a class**

**Some predefined properties for classes:**

{abstract}, {leaf}

{readOnly} ( {frozen} in UML1.x)

{query}



# Stereotypes

**What is a stereotype?**

A stereotype extends the vocabulary of UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem

It is drawn in **«guillemets»**

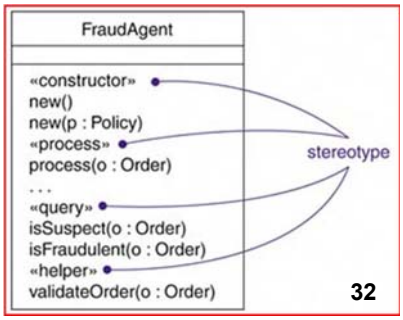
A class stereotype marks the class as having certain properties

**Some standard class stereotypes**

**«metaclass»** , **«stereotype»** , **«type»**, **«utility»**, **«powertype»**

**You can define your own stereotypes if you like.**

**«singleton»**, **«constructor»**

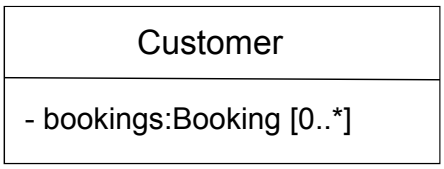


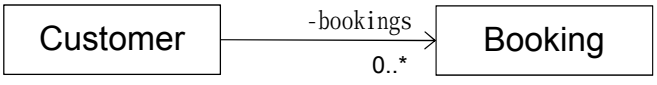



# Attributes

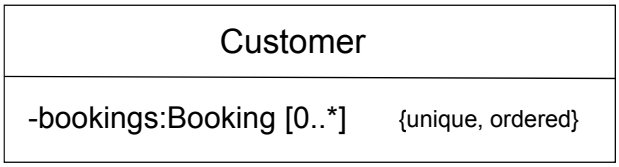
**Can be simple data types or relationships to other objects**

**Can be represented as inlined attributes or relationships between classes**




**Multiplicity, uniqueness, and ordering can also be specified**



# Relationships

**A class relationship might indicate some kind of *semantic connection* or some sort of *sharing***

- Association
- Aggregation
- Composition
- Generalization
- Dependency



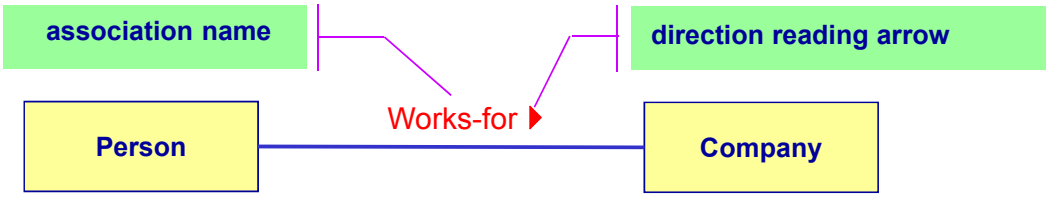
# Association

**An association is a structural relationship between classes that indicates some meaningful and interesting connection**

- "knows-of" relationship
- An association only denotes a semantic dependency between two classes, but it does not state the exact way in which one class relates to another
- Bi-directional unless otherwise specified *(More on this later!)*

**The most weaker form of structural relationship normally identified at analysis and early design phases**

- Turned into concrete class relationships as design and implementation continues

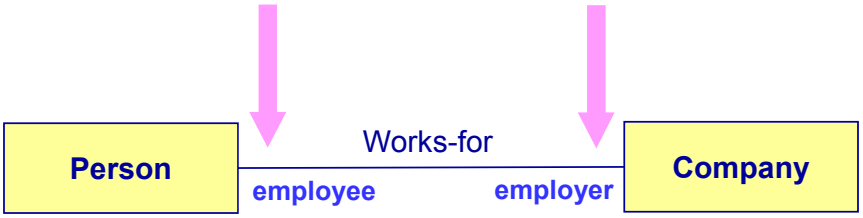


# Role Name

**Each end of an association is called an "Association End"**

A role name is a noun that describes the role that the class plays in the association

The role name is attached shown near the association end






# Association

**The multiplicity describes the number of instances of one class that is related to ONE instance of the other class *at any point in time***

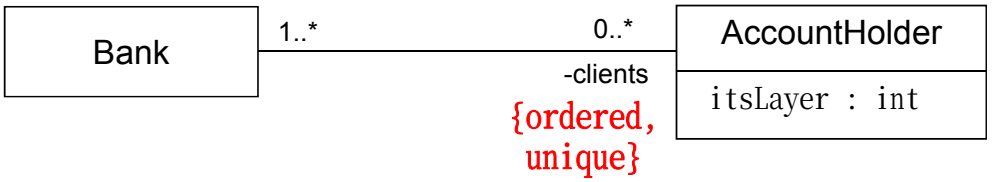
| | |
|---|---|
| * or 0..* | Zero to many |
| 1..* | One to many |
| 0..1 | Zero or One |
| 1 | One and only one |
| n..m | Where n and m are any two integers |


**If not explicitly specified, it is "undecided"**

# Properties

**There are several predefined properties for multiplicities greater than 1:**

| | |
|---|---|
| ordered | The elements are ordered into a list |
| unique | [Default], no duplicate elements |

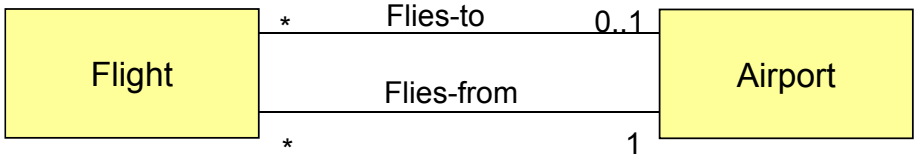


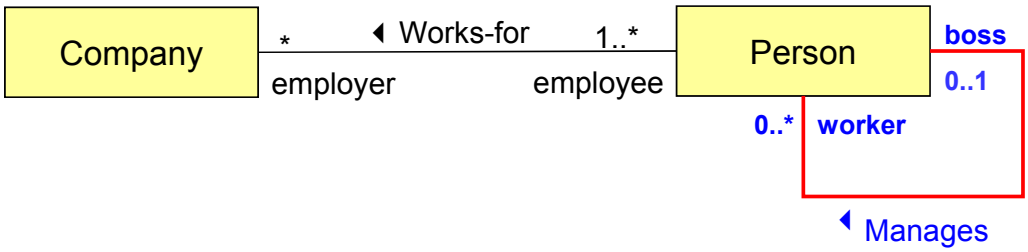

**Other properties for attributes can also be specified:**

eg. {readOnly}



# Multiple & Self Associations

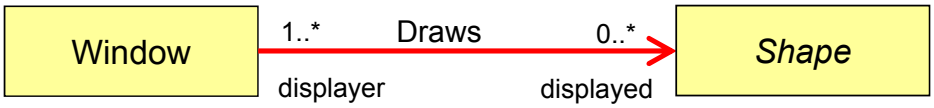


# Unidirectional Association

**Navigability is shown as an arrowhead on the association end pointing to the class that can be navigated to**

"Messages can only be sent in the direction of the arrow"

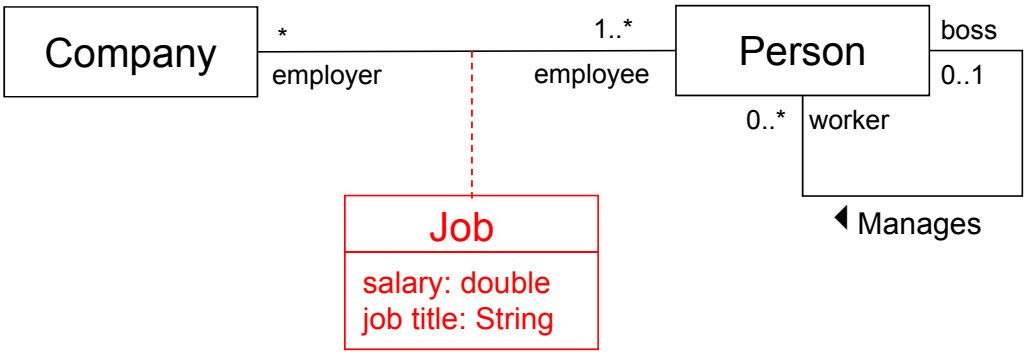





# UML 2 Navigability Idioms

| UML 2 navigability idioms | | | |
|---|---|---|---|
| UML 2 syntax | Idiom 1: Strict UML 2 navigability | Idiom 2: No navigability | Idiom3: Standard practice |
| A ↔ B | A to B is navigable<br>B to A is navigable | | |
| A → B | A to B is navigable<br>B to A is not navigable | | |
| A → B | A to B is navigable<br>B to A is undefined | | A to B is navigable<br>B to A is not navigable |
| A — B | A to B is undefined<br>B to A is undefined | A to B is undefined<br>B to A is undefined | A to B is navigable<br>B to A is navigable |
| A B | A to B is not navigable<br>B to A is not navigable | | |

# Association Class



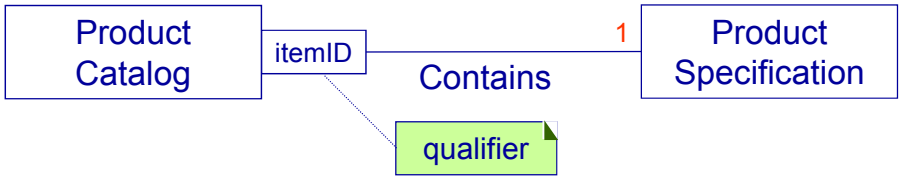

**It is useful to model an association as a class when it can have class-like properties, such as attributes, operations, and other associations**

**Association class can be used only when there is a *single unique link* between two objects at any point in time**



# Qualified Associations

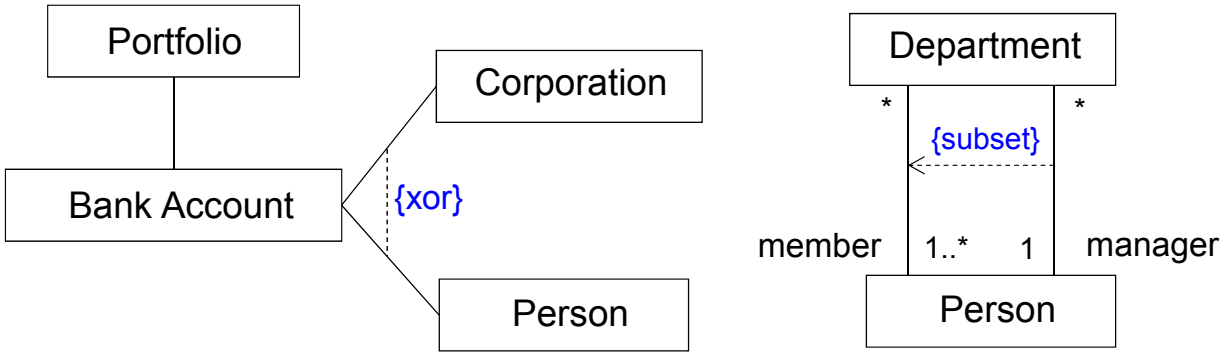
**A *qualifier* distinguishes the set of objects at the far end of the association based on the qualifier value. An association with a qualifier is a *qualified association***

# Constraints

**A constraint specifies a conditions that must be held true for the model to be well-formed**

**With constraints, you can add new semantics or change existing rules**






# Aggregation

**No semantic difference from "association"**

**Aggregation represents a "*part-whole* "or "*has-a*" relationship, i.e., the aggregate object (whole) is made up of other objects (parts)**

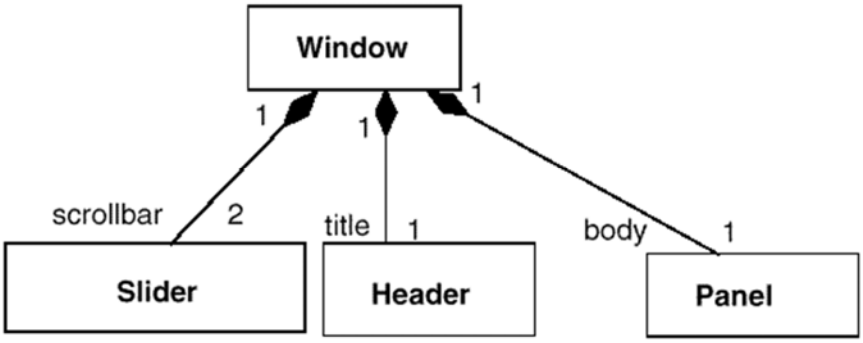

```
class Window {
public: ...
  void  addShape(Shape*); Shape* removeShape(Shape*);
private:
  vector<Shape*> itsShapes;
};
```

# Composition Aggregation

**A hard form of aggregation denoting *ownership***

**Composites control the lifetime of their constituents**

Ownership can be transferred, but cannot be shared






# Difference between Association and Aggregation

**Aggregation denotes part-whole relationship whereas associations do not**

**However, there is not likely to be much difference in the way the two relationships are implemented**

**Rule of thumb by three amigos (Rumbaugh, Booch, Jacobson):**

***" … if you don't understand [aggregation] don't use it."***

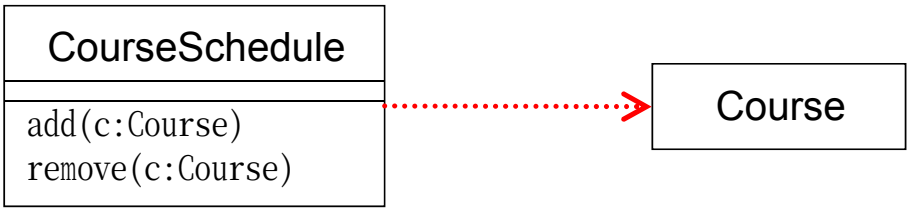
# Dependency

**A dependency denotes a *using* (*or client-supplier*) relationship, specifying a compile, link, or load time dependence**

**An object of a client class uses the services of the supplier class to provide its own service**

**Used when objects share very short term relationships:**

So short that they are not held in pointer or reference variables.




**Navigable associations, aggregations, and compositions are also forms of dependency**



# Dependency
## (Cont'd)

**Typically used to indicate the decision that**

Operations of the client class invoke operations of the supplier class, or

Have signatures whose return class or arguments are instances of the supplier class, or

Creates an instance of the supplier class as a local object

# Types of Dependencies

**Usage dependency**

Clients uses some of the services made available by the supplier to implement its own behavior

**«use»**, **«call»**, **«parameter»**, **«send»**, **«instantiate»**

**Abstraction dependency**

The supplier is more abstract than the client (e.g. analysis model *vs.* design model)

**«trace»**, **«substitute»**, **«refine»**, **«derive»**

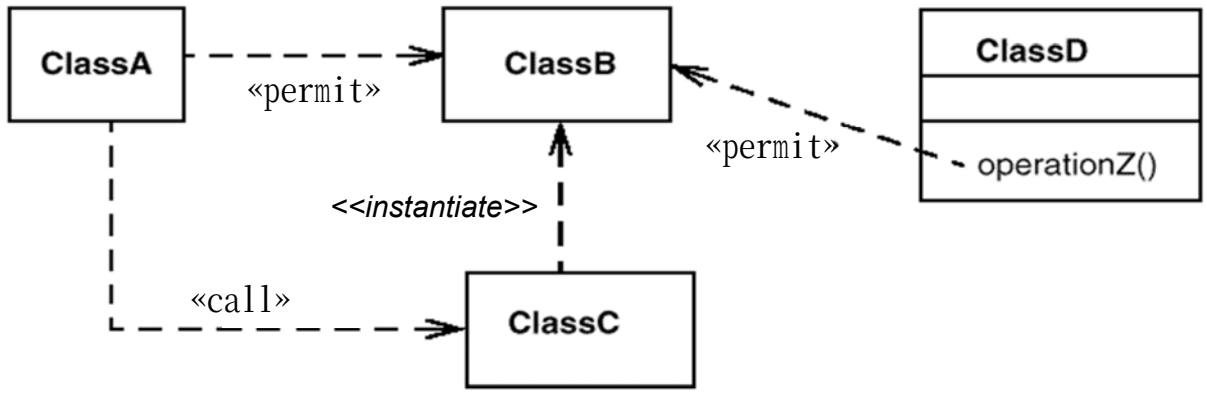
**Permission dependency**

The supplier grants some sort of permission for the client to access its contents

**«access»**, **«import»**, **«permit»**

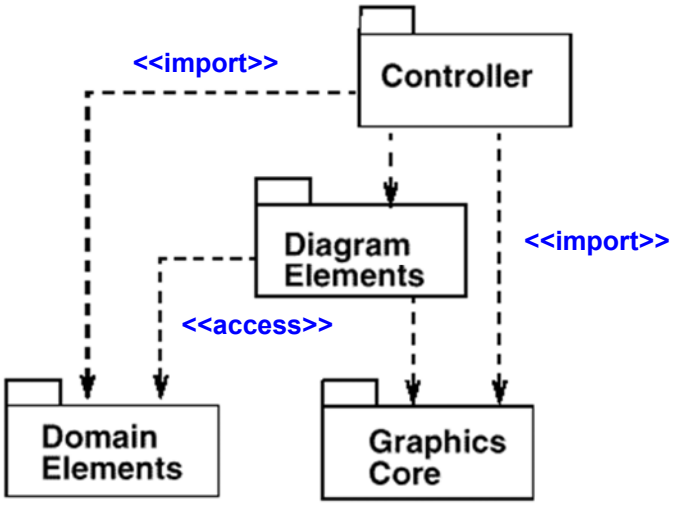


# Dependency Example (among Classes)




☞ **«permit» used to be «friend», finally dropped in UML 2**

# Dependency Example (among Packages)



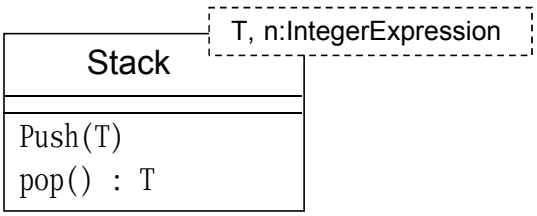



# Parameterized Class

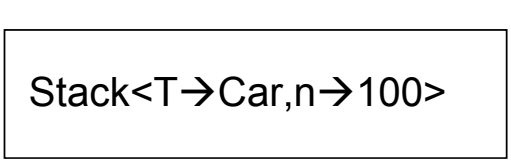
**A parameterized class denotes a family of classes whose structure and behavior are defined independently of their formal class parameters**

**Relationship between a parameterized class and its instantiated classes is also denoted as a dependency with «bind» stereotype.**

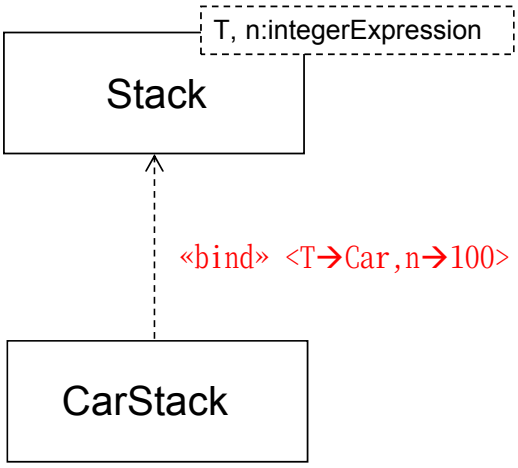
```
template<class T,int n>
class Stack {
public:
   void push(const T&);
   T pop();
..
}
```



# Instantiation of Template Classes




(a) Implicit binding




(b) Explicit binding



# Generalization

**Relationship between superclass and subclass**

**Generalization/specialization relationship**

***"is a"* relationship**

subclass *is a* superclass

Cat *is a* Mammal

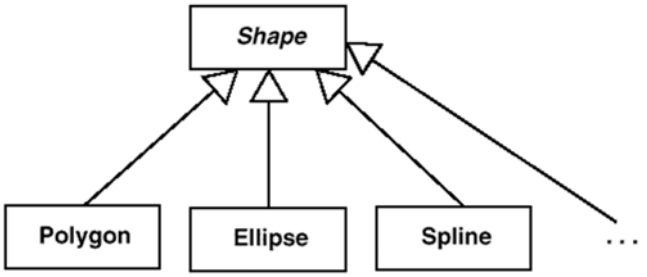
**Primary purpose of inheritance is for *subtyping***

Remember the Liskov substitution principle

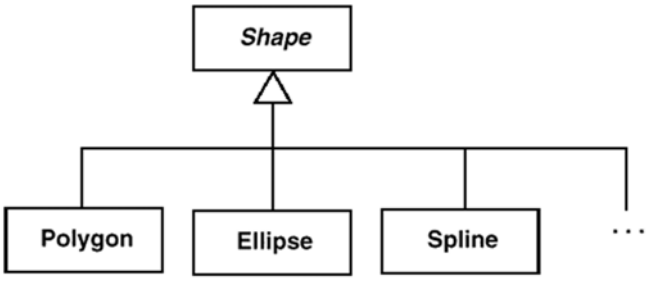
**Sometimes programmers use the inheritance to accomplish a code reuse by *subclassing* from a super class, which should be avoided whenever possible**

Use aggregation instead

# Inheritance



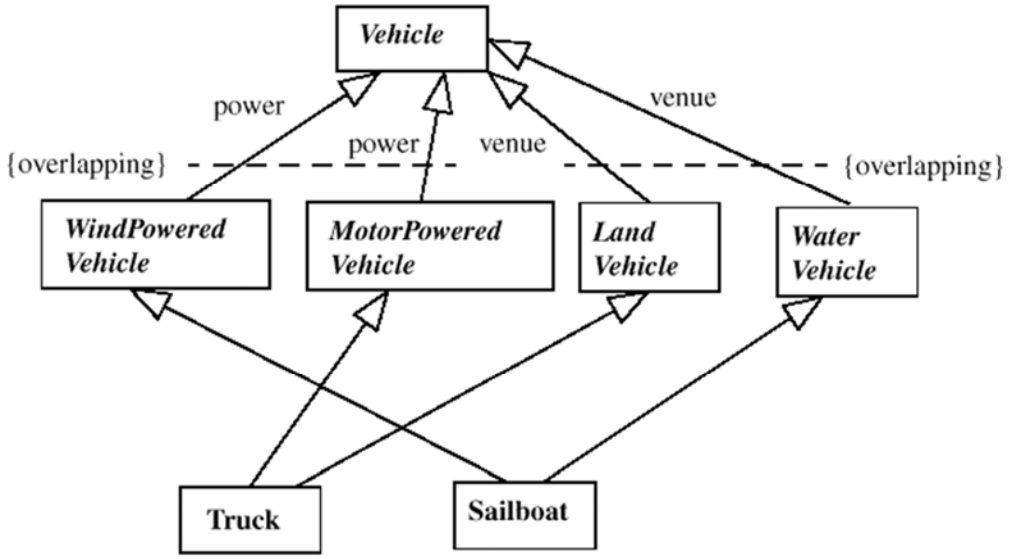

(a) Separate Target Style



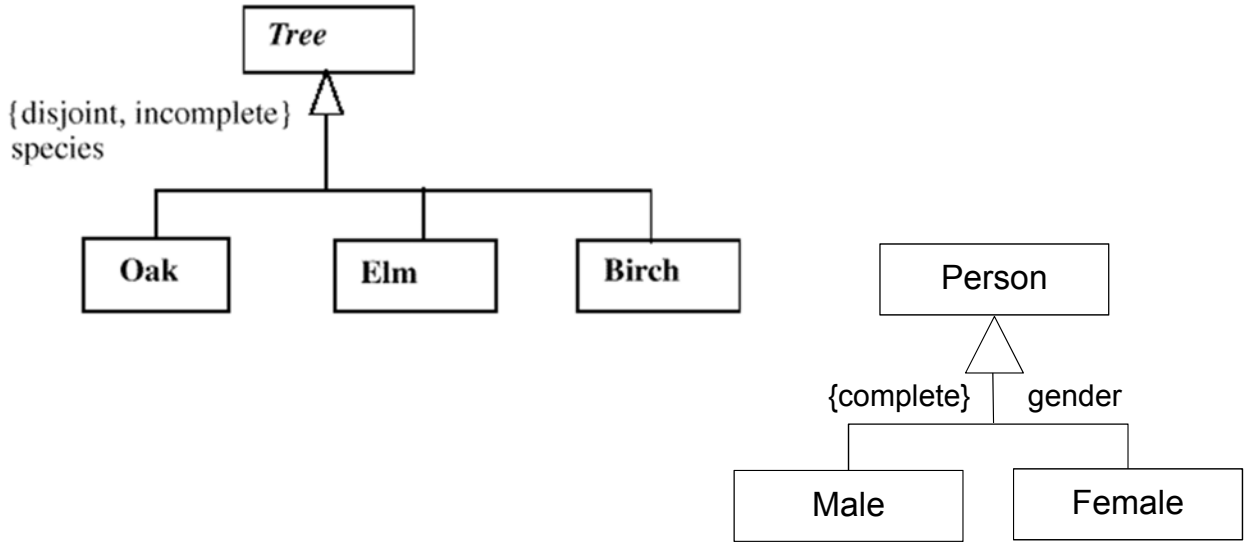

(b) Shared Target Style



# Inheritance
## (with generalization set names and constraints)



# Inheritance
## (with Constraints)





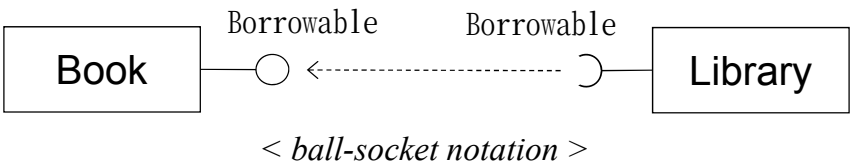

# Interfaces

**A collection of (abstract) operations that are used to specify a service (or contract) of a class or a component**

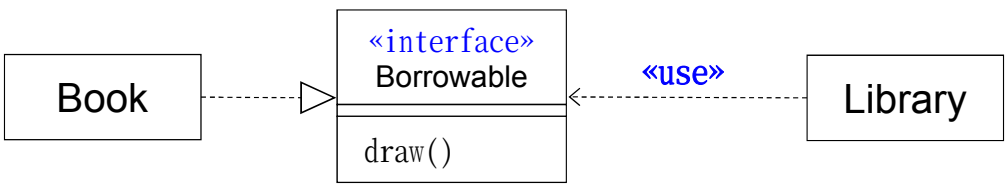
UML interface can also have attributes.

**An interface uses a classifier icon with «interface» keyword.**

**Provided interface *vs.* Required interface**

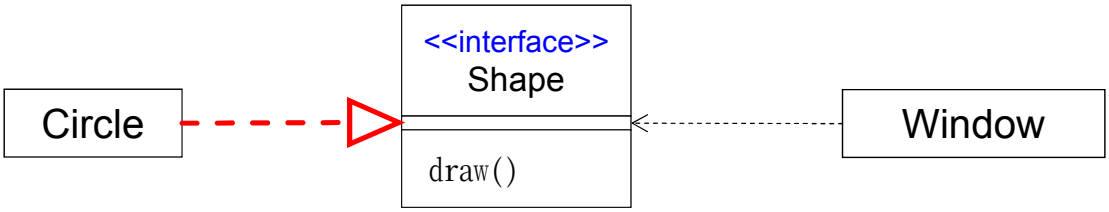



< *ball-socket notation* >



# Realization Relationship

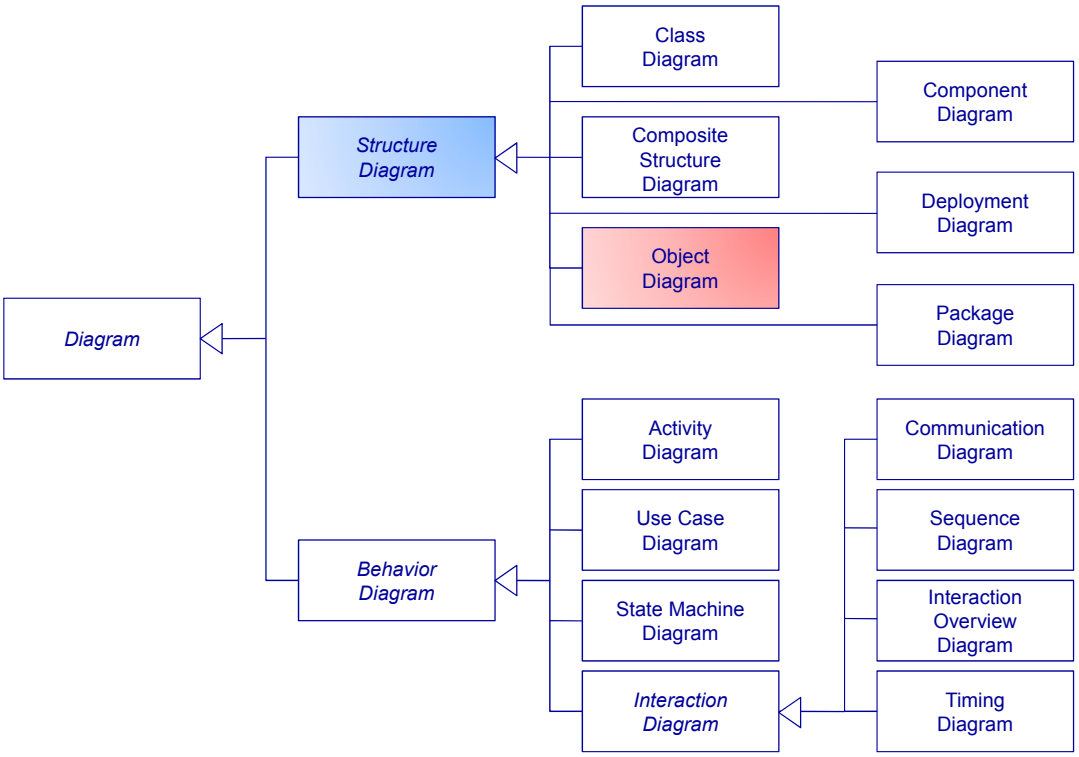

(a) Simple form




(b) Expanded form



# Object Diagram



# Object Diagram

*An object diagram is a graph of instances, including objects and data values.*

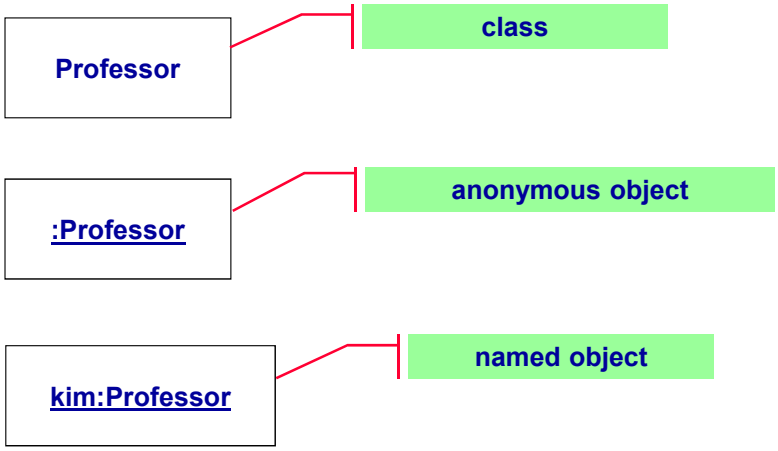
*It shows a snapshot of the detailed state of a system at a point in time.*

**UML modeling elements**

- Objects
- Links

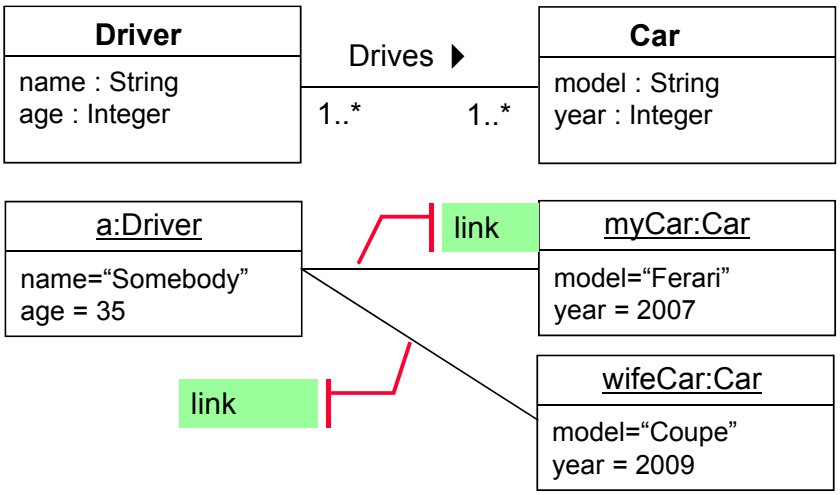


# UML Object Icons



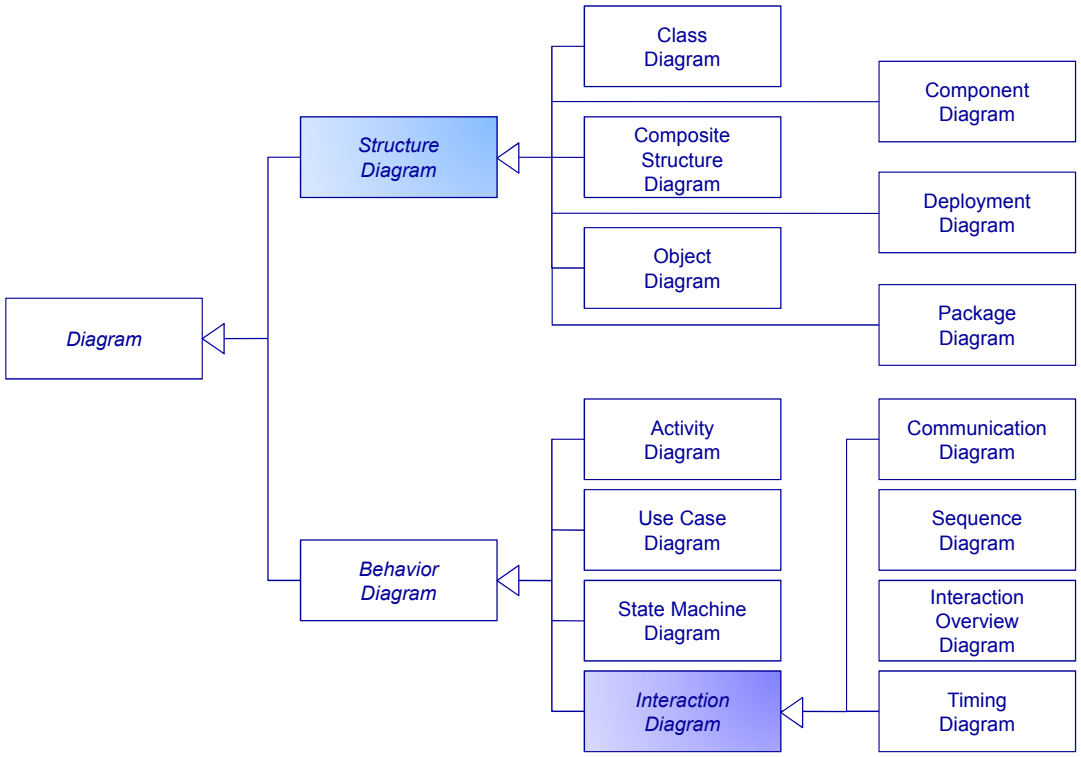
# Links

**A *link* is an instance of an association which denotes a path between two objects**






# Interaction Diagram



# Interaction Diagram

***Describes the communications between Lifelines for a particular scenario by showing Lifelines participating in the interaction and the messages that they exchange***

**Sequence diagram**

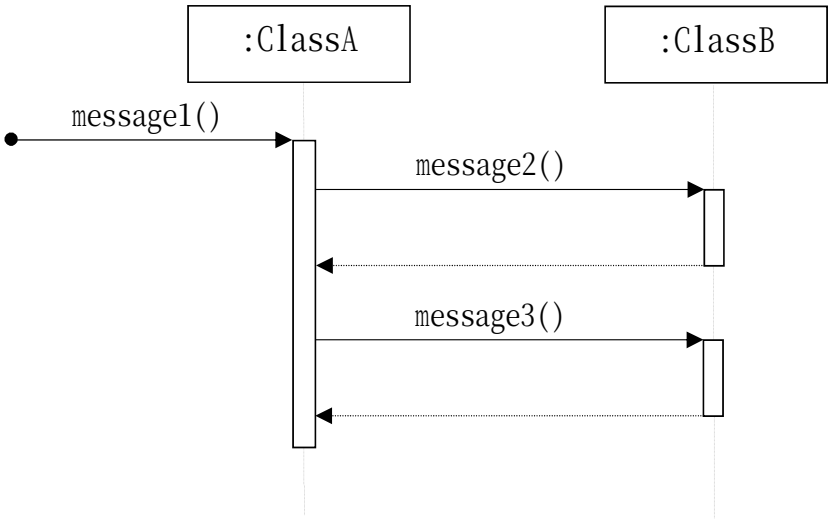
focuses on the ***time*** (i.e., order in which the messages are sent)

**Communication diagram (*was* Collaboration diagram)**

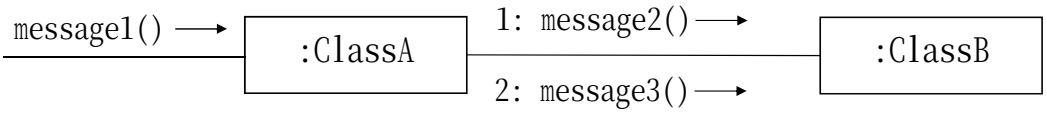
focuses on the ***space*** (relationships between Lifelines)



# Sequence Diagram



# Communication Diagram






# Lifeline

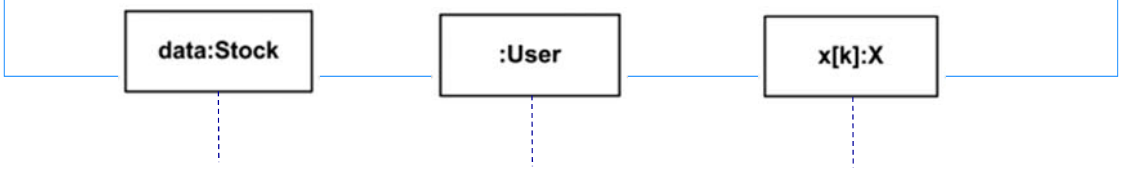
***Lifeline*** **denotes a connectable element which represents an** ***individual participant*** **in the interaction**

**Lifelines represent** ***only one*** **interacting entity**

**Must use a** ***selector*** **to specify only one specific element from multivalued connectable element (i.e., multiplicity > 1)**

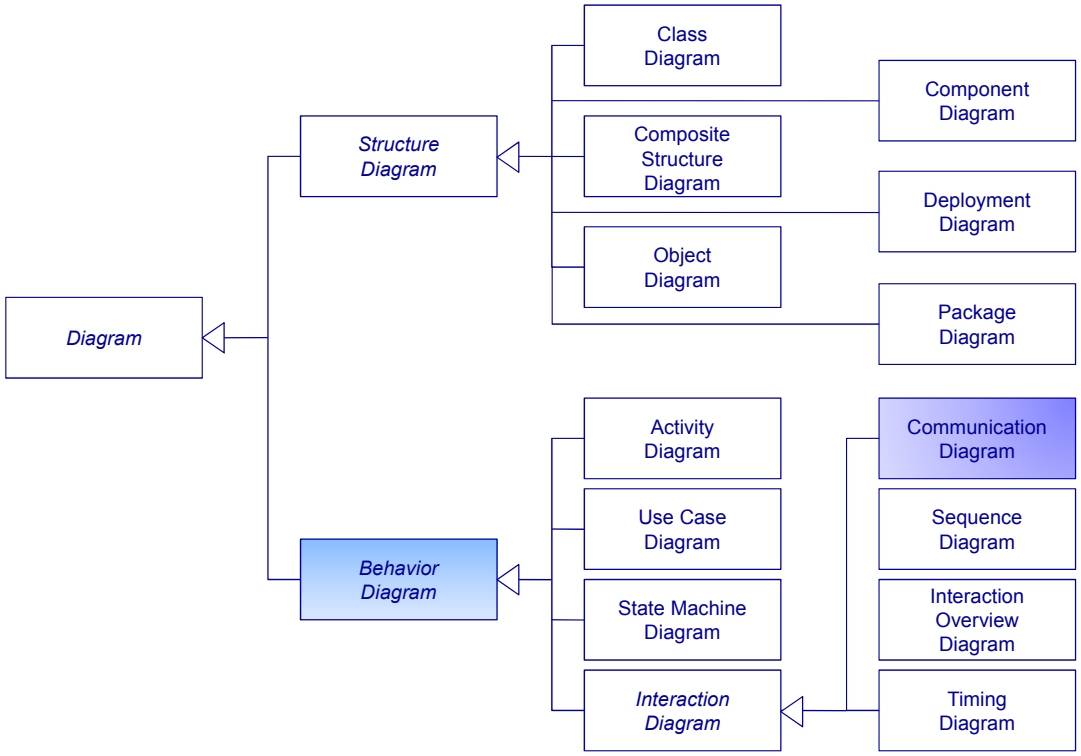
**A Lifeline is shown as a rectangle, called "head "**

Lifeline in sequence diagrams does have "tail" representing the **line of life** whereas "lifeline" in **communication diagram** has no tail




***[k] is a selector***

# Communication Diagram

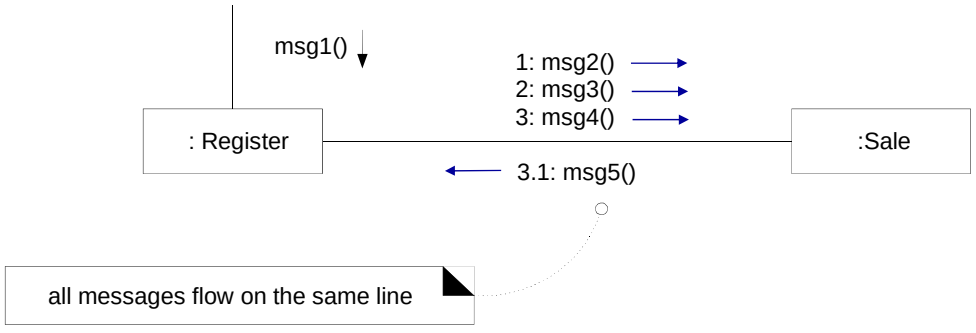





# Illustrating Messages

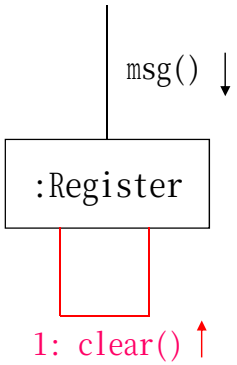
**A message is represented via a labeled arrow on a line**

**A sequence number is added to show the sequential order of messages in the current thread of control**



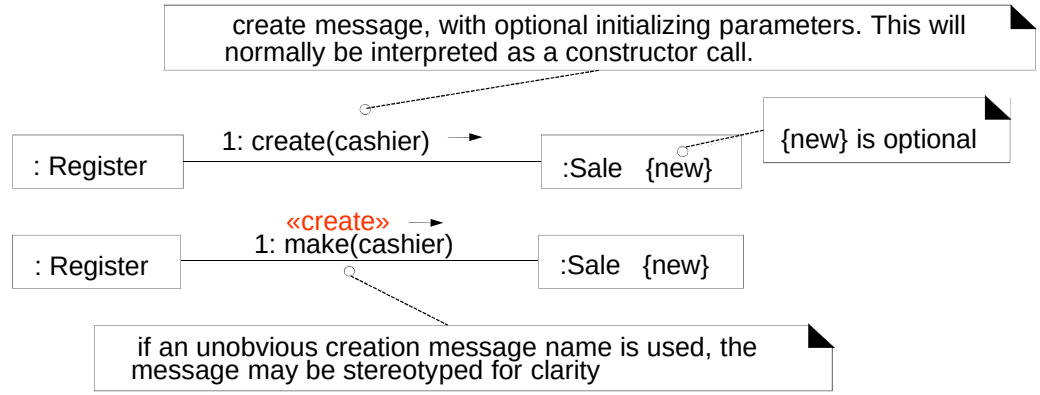
# Illustrating Messages to “self”

**A message can be sent from a Lifeline to itself**

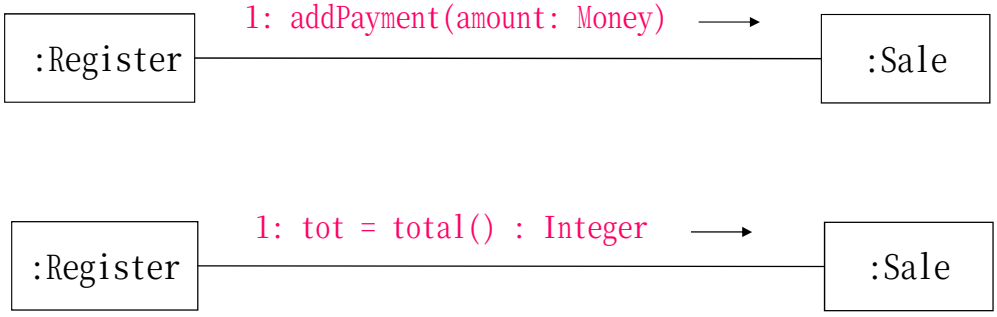





# Illustrating Object Creation & Deletion

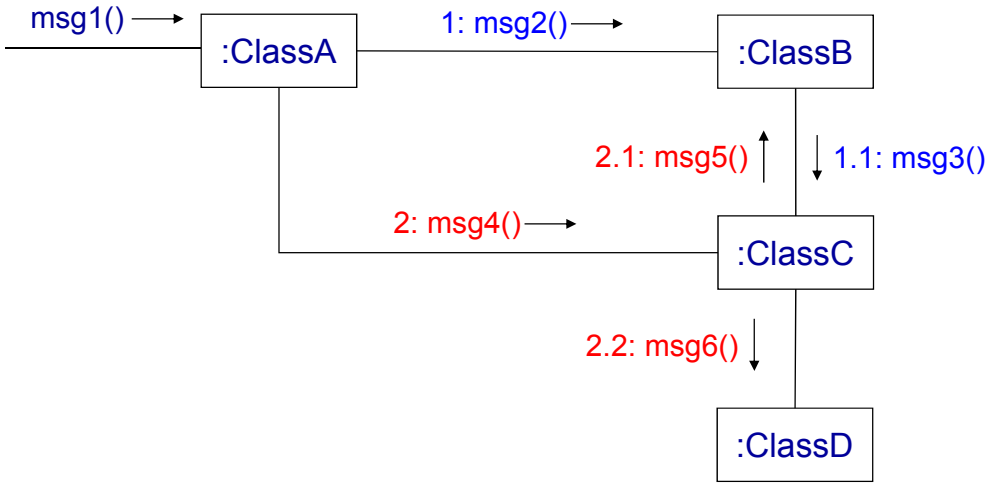



# Illustrating Parameters & Return Value

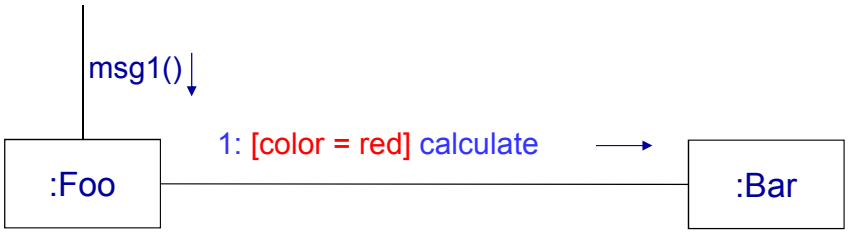





# Message Number Sequencing

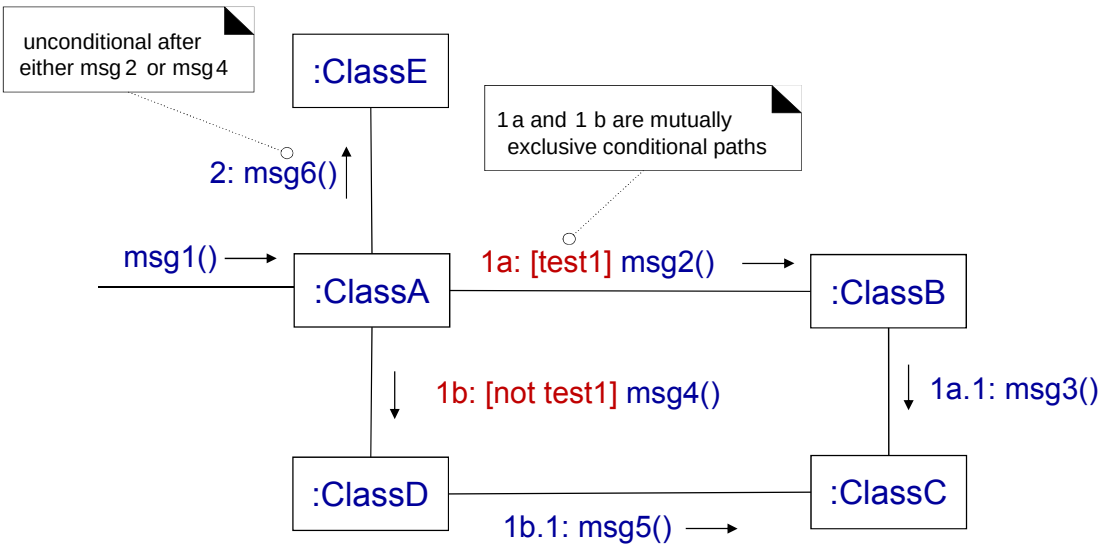


# Illustrating Conditional Messages

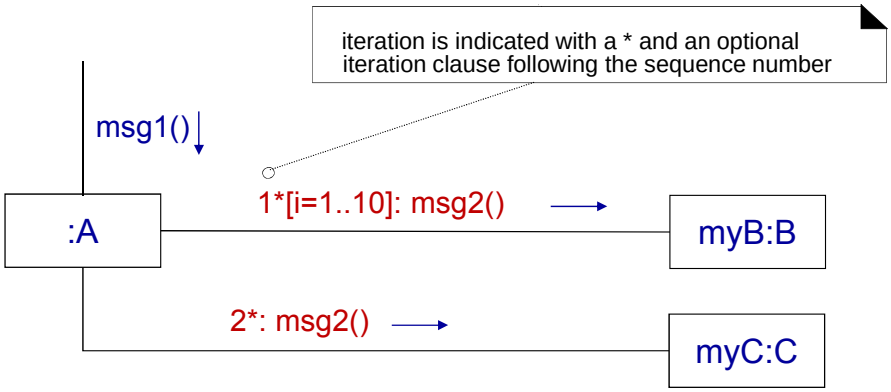





# Mutually Exclusive Conditional Paths

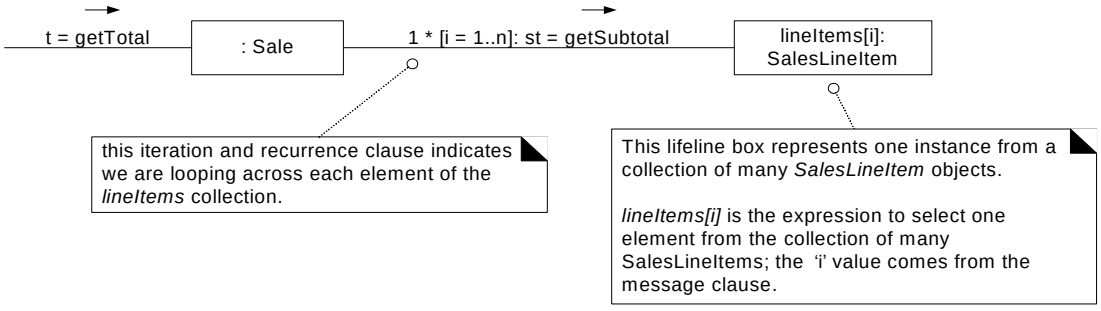


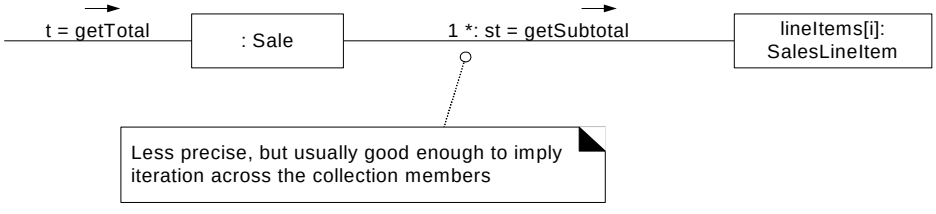
# Illustrating Iteration or Looping






# Illustrating Iterations

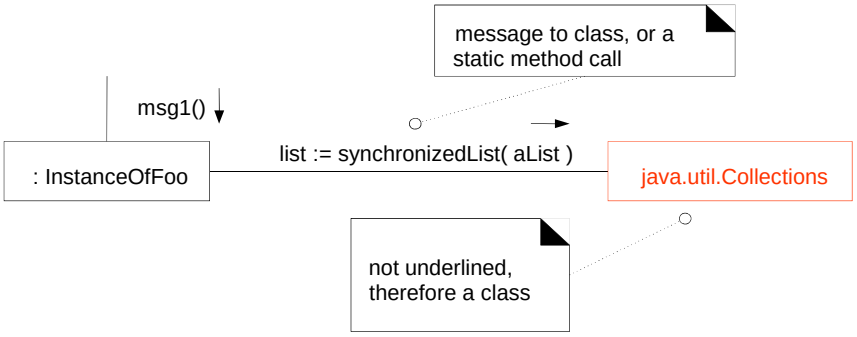


# Messages to a Class

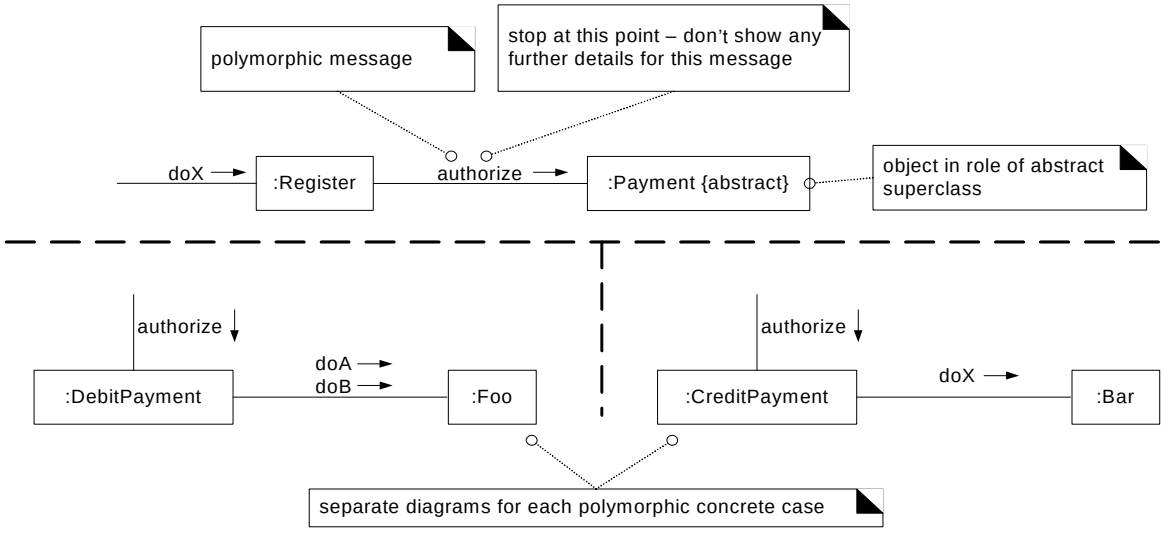
**Messages may be sent to a class itself, rather than an instance**

Class methods (aka, static methods) in Java and C++

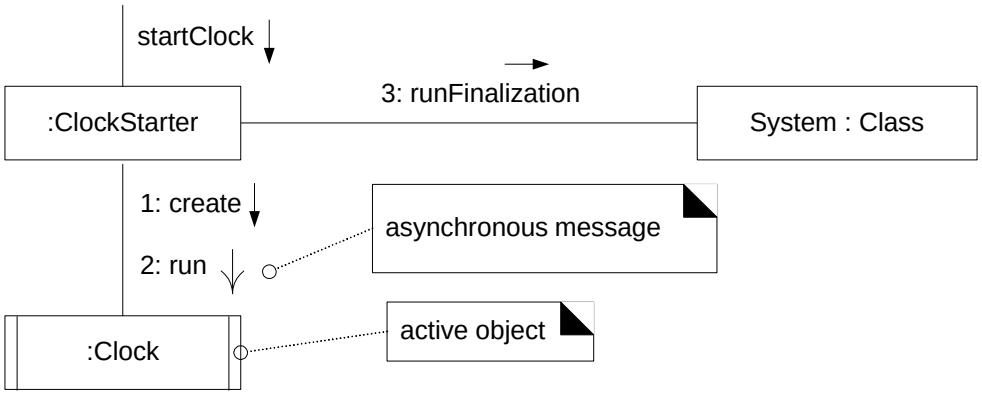





# Polymorphic Messages

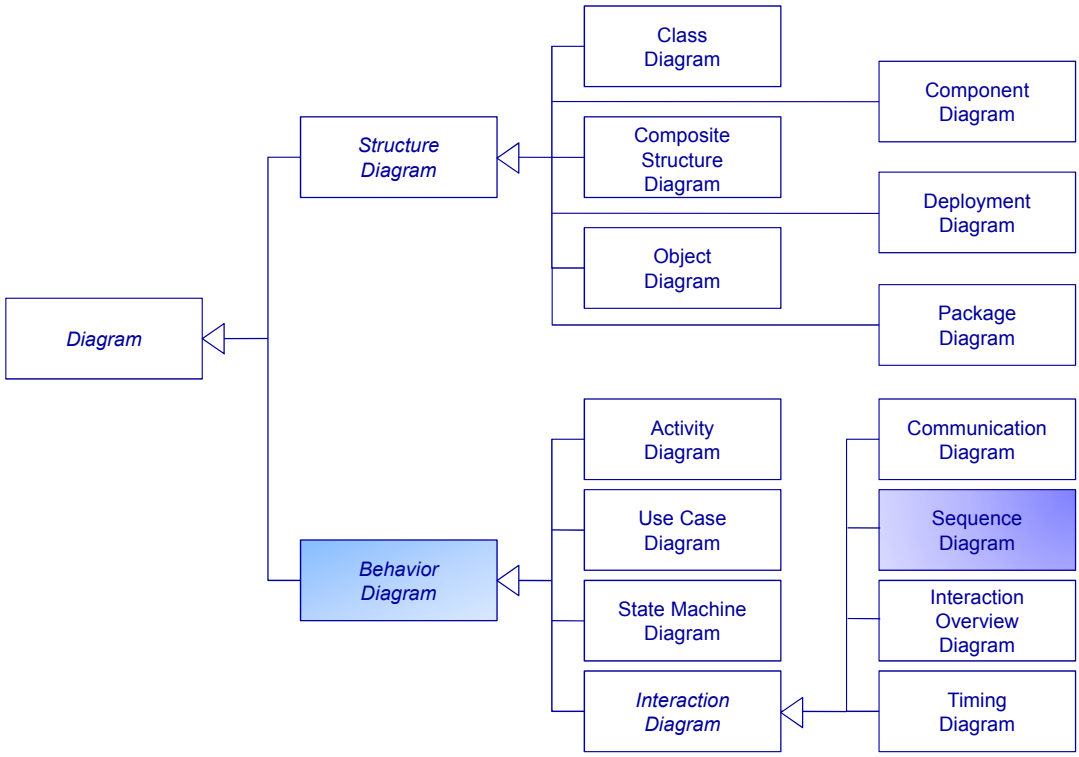


# Active Objects & Asynchronous Messages



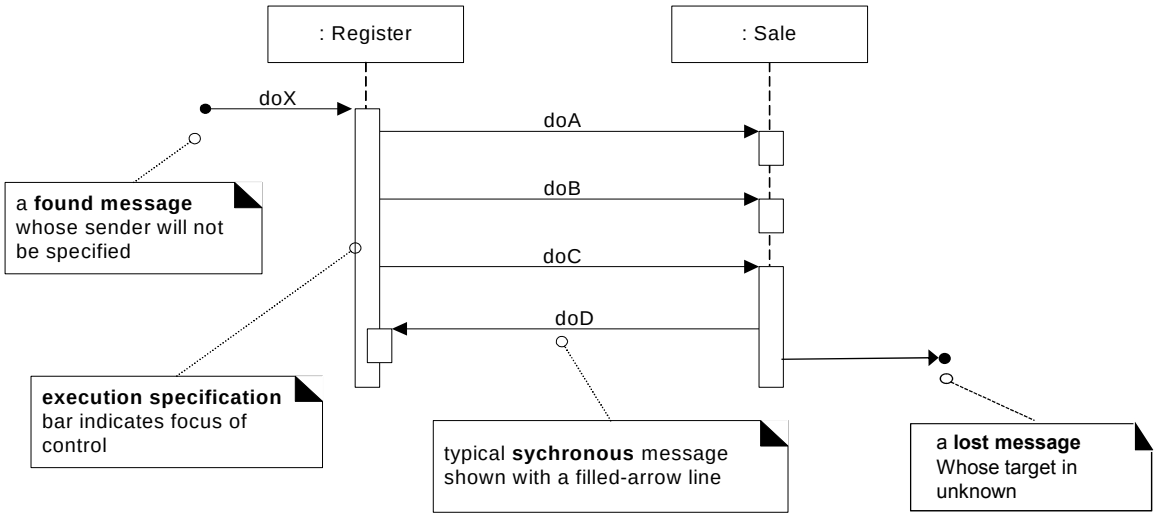



# Sequence Diagram



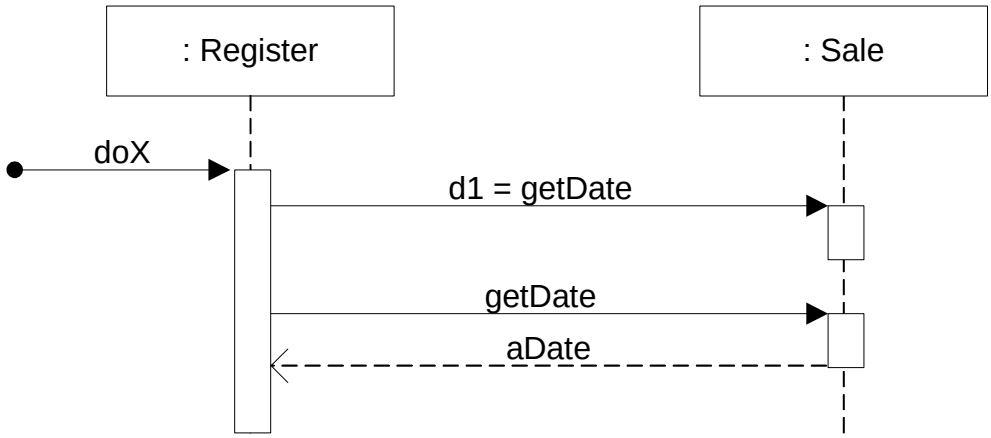
# Sequence Diagram

**A message is represented via a labeled arrow line between Lifelines**
**The time ordering is organized from top to bottom**

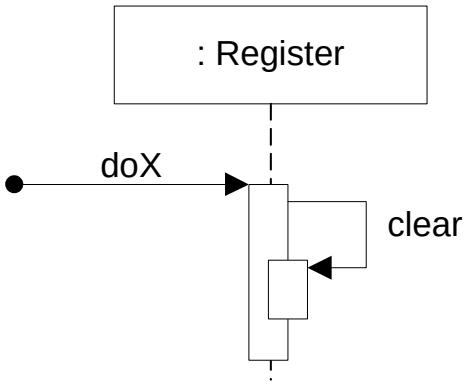





# Illustrating Returns

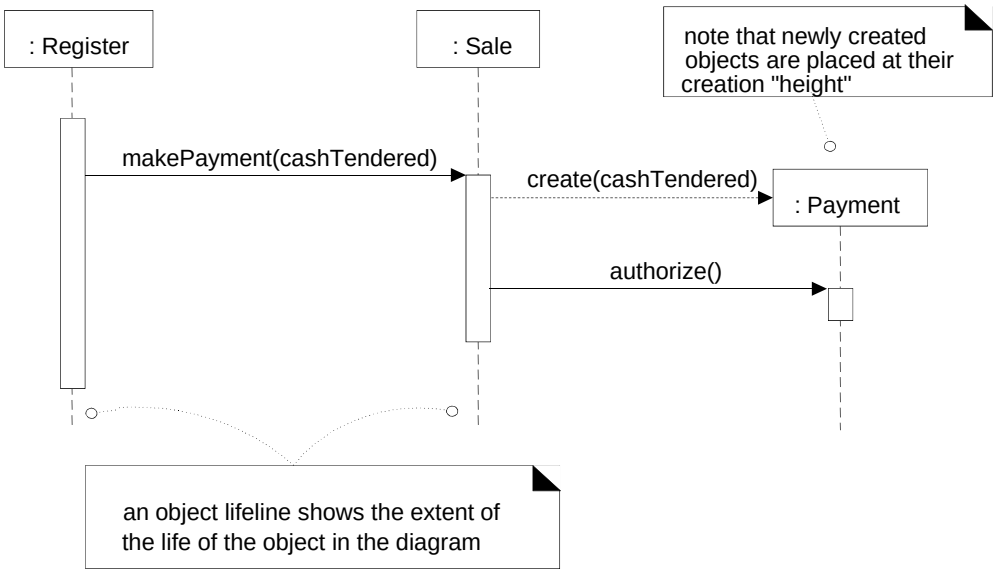


# Illustrating Messages to “self”

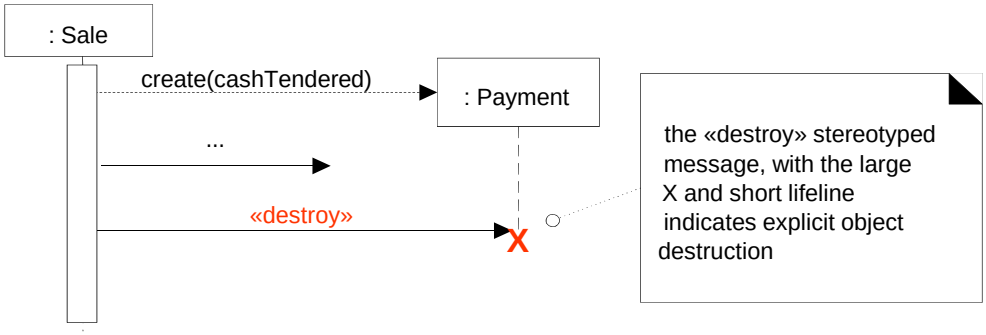





# Illustrating Object Creation



# Illustrating Object Destruction



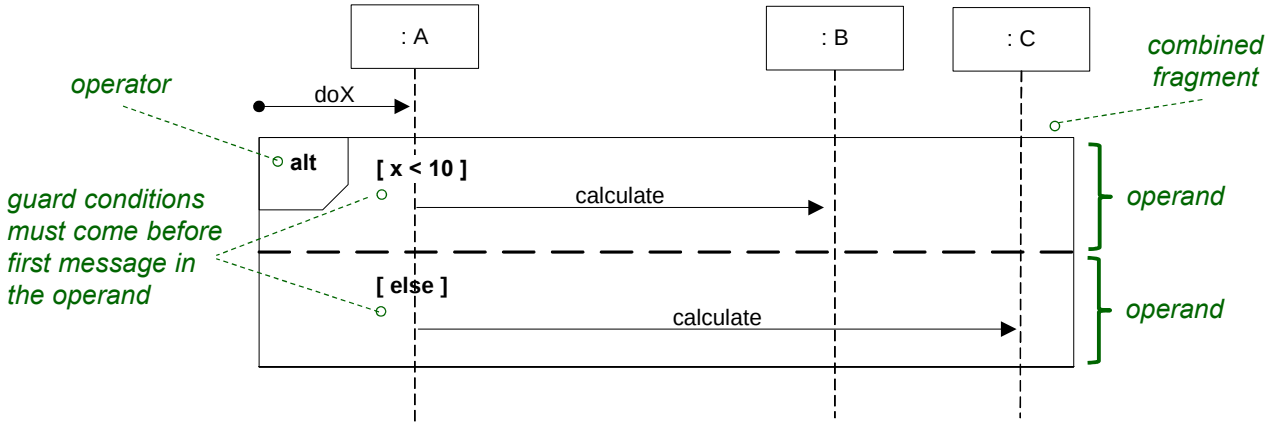



# Combined Fragment

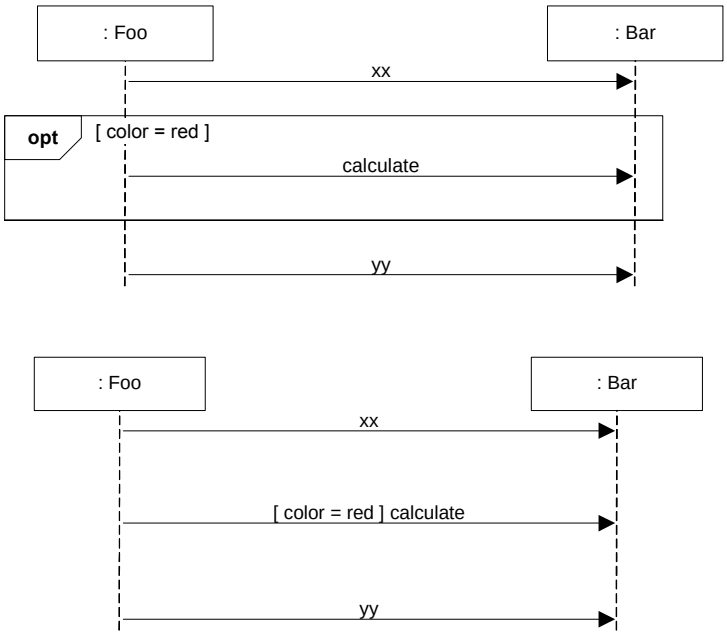
**A combined fragment has one *operator*, one or more *operands*, and zero or more *guard conditions***

The operator determines how its operands are executed

Guard conditions are Boolean expressions to determine whether their operands execute

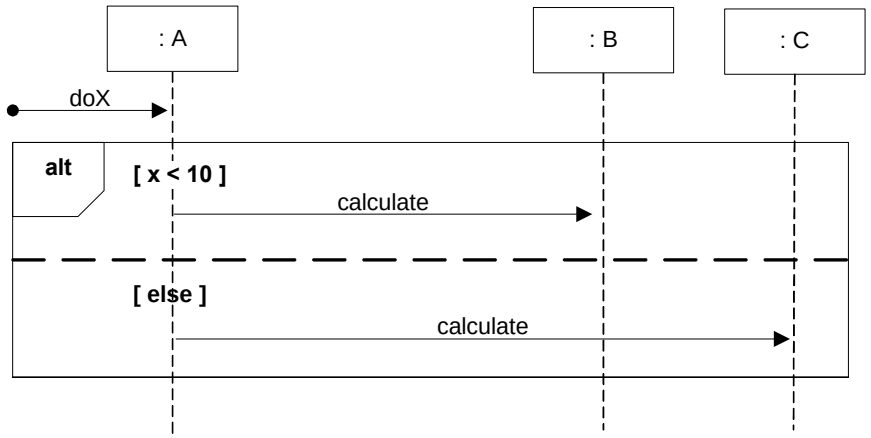


# Illustrating Conditional Messages

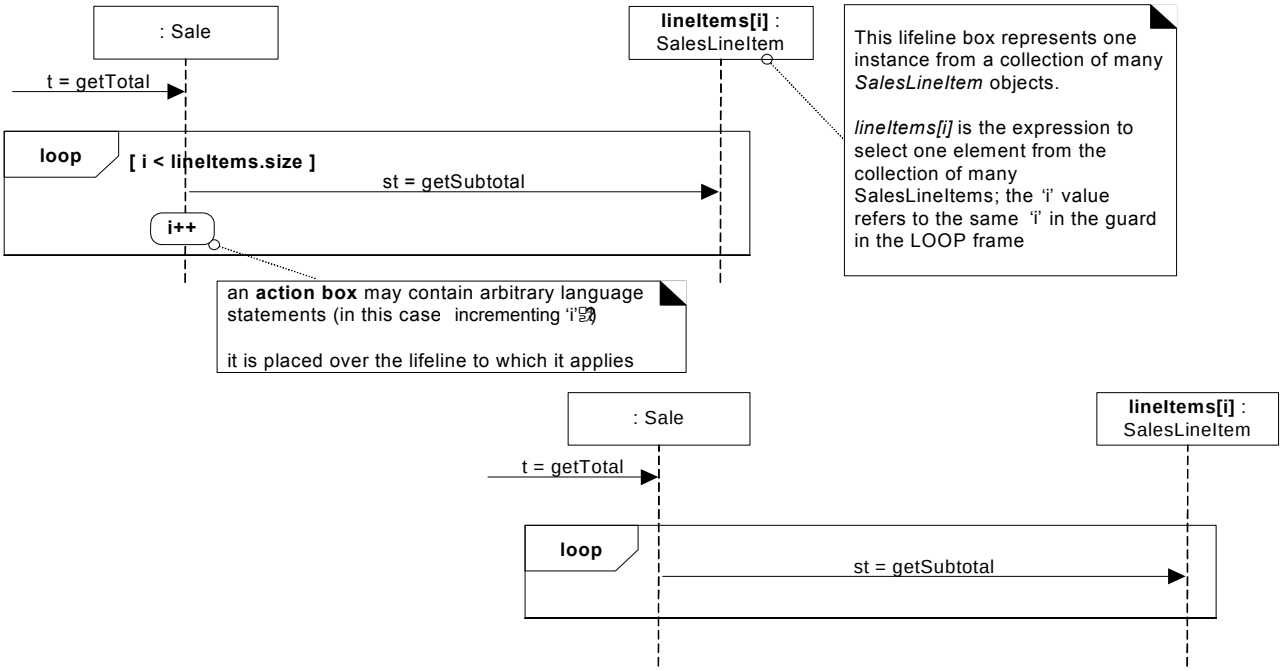





# Illustrating Conditional Messages
## (Cont'd)

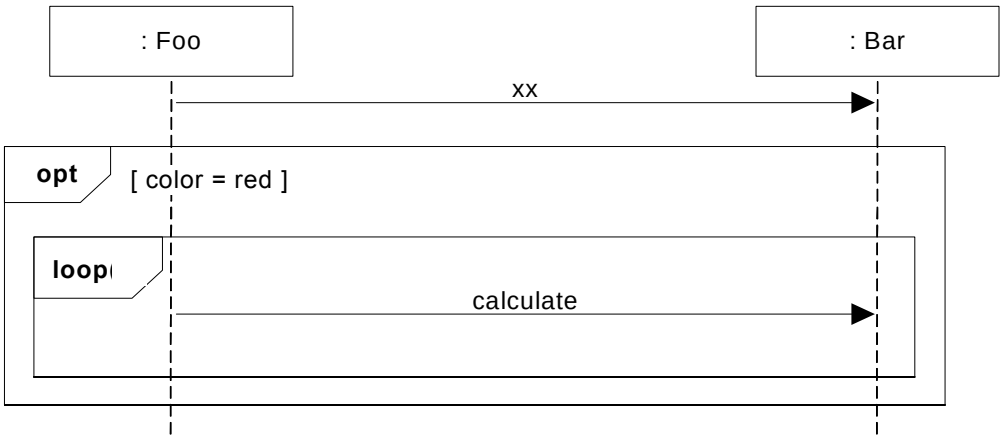


# Illustrating Iteration or Looping

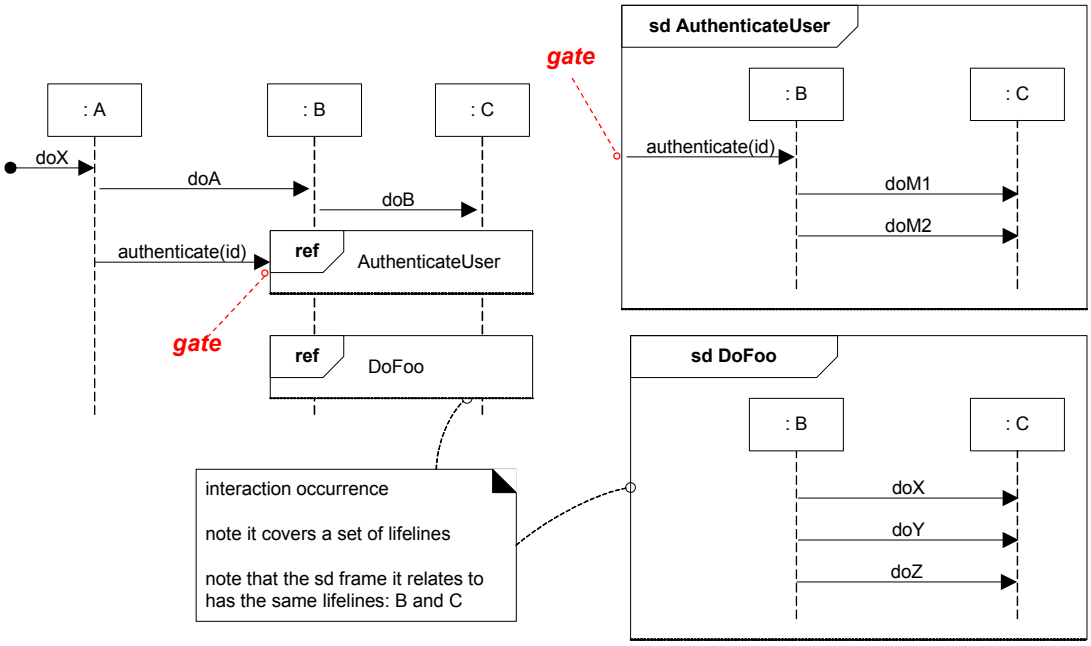





# Illustrating Condition & Iteration

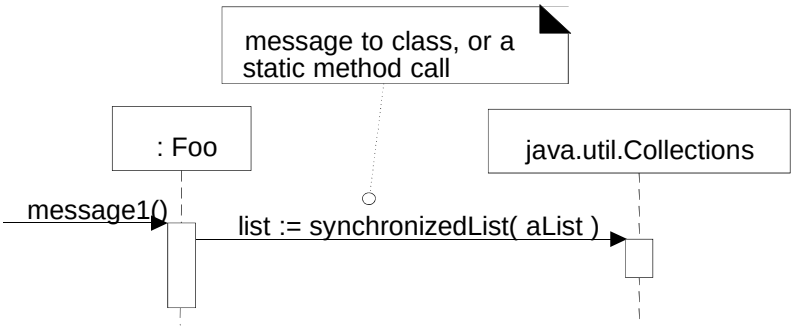


# Reference to Other SD





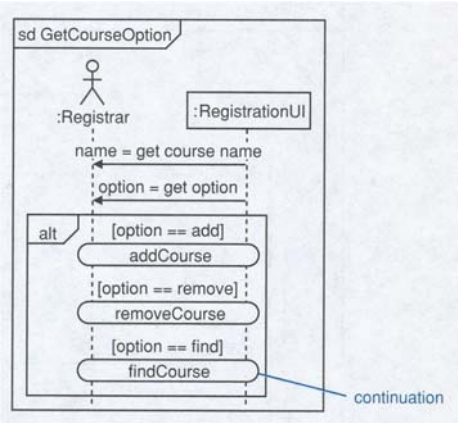

# Messages to a Class

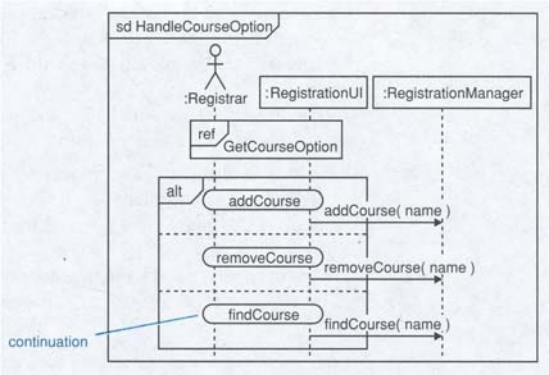


# Continuations

**Continuations terminate an interaction fragment so that it can be continued by another fragment.**

- Used first item in the fragment → will be continuing from another fragment
- Used last item in the fragment → the fragment terminates but may be continued by another fragment






# Operators for Combined Fragments

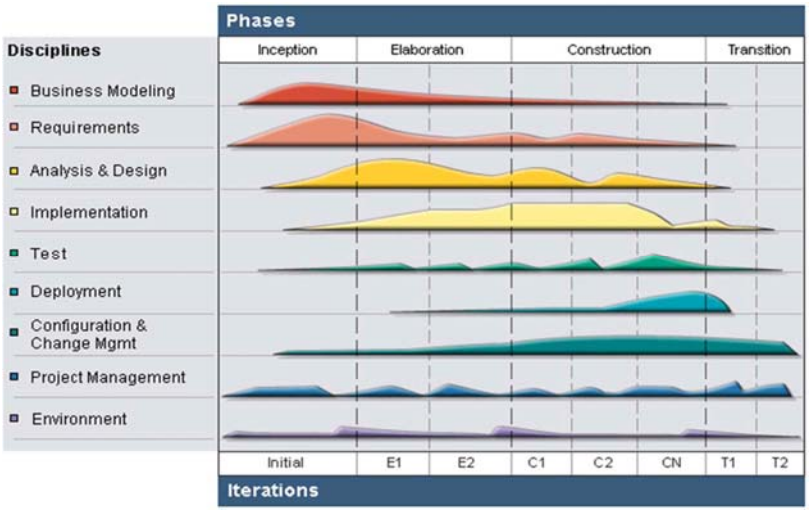
| Operator | Meaning |
|---|---|
| `alt` | Alternative multiple fragments; only the one whose condition is true will execute |
| `opt` | Optional; the fragment executes only if the supplied condition is true. Equivalent to an `alt` with only one trace |
| `par` | Parallel; each fragment is run in parallel. |
| `loop` | Loop; the fragment may execute multiple times, and the guard indicates the basis of iteration |
| `region` | Critical region; the fragment can have only one thread executing it at once. |
| `neg` | Negative; the fragment shows an invalid interaction. |
| `ref` | Reference; refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value. |
| `sd` | Sequence diagram; used to surround an entire sequence diagram, if you wish. |

# UML References


- Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide*, 2nd ed., Addison-Wesley, 2005.
- James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual, 2nd ed.,* Addison-Wesley, 2004.
- Ivar Jacobson, Grady Booch, James Rumbaugh, *The Unified Software Development Process,* Addison-Wesley, 1999.
- Dan Pilone et al, *UML 2.0 In a Nutshell,* O'Reilly, 2005.
- Martin Fowler, *UML Distilled, 3rd ed.,* Addison-Wesley, 2004.
- Tim Weilkens et al, *UML 2 Certification Guide,* 3rd ed., Morgan Kaufman Publishers, 2007.
- Bruce Powel Douglass, *Real-Time UML,* 3rd ed., Addison-Wesley, 2004.

# Object-Oriented Analysis and Design using UML and Patterns

## Unified Process (UP)




# Iterative and Evolutionary Development Process

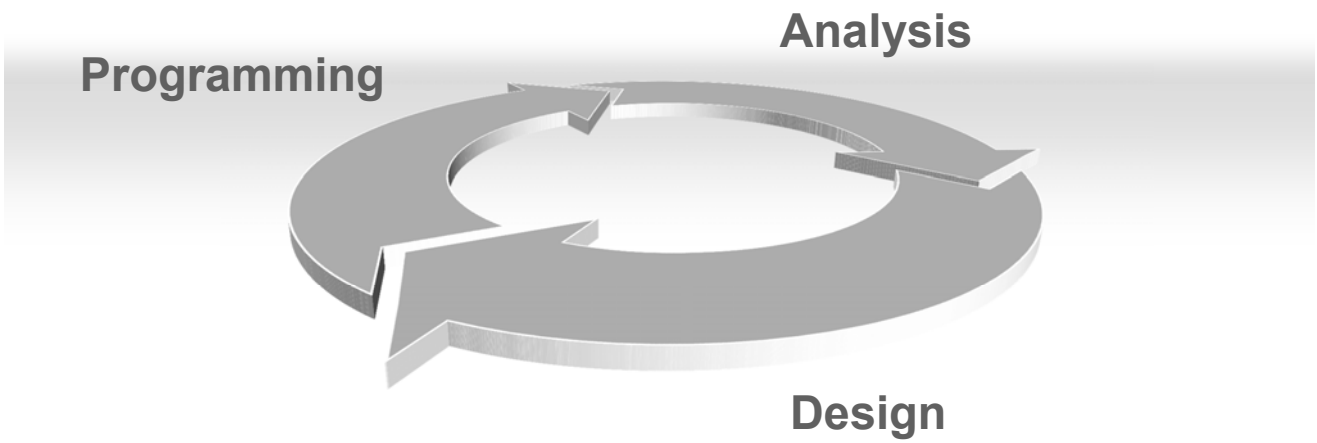
## Objectives

Define the software engineering process

Provide motivation for the iterative & incremental process

Compare traditional vs. modern approach

## Motivation for *Iterative & Evolutionary* Development Processes




## Stakeholders in Software Development

Customers

Users


Government agencies

Project Managers


Software developers

## Traits of Successful Software Products

Must *satisfy* the *stakeholder's requirements*
*- functional & non-functional*

Must be developed *on time* and *on budget*

Must be *resilient to change!*



## Engineering Analogy: Building a piece of D.I.Y *vs.* Building a Bridge

Up to now, the programs you've written are probably more like D.I.Y.

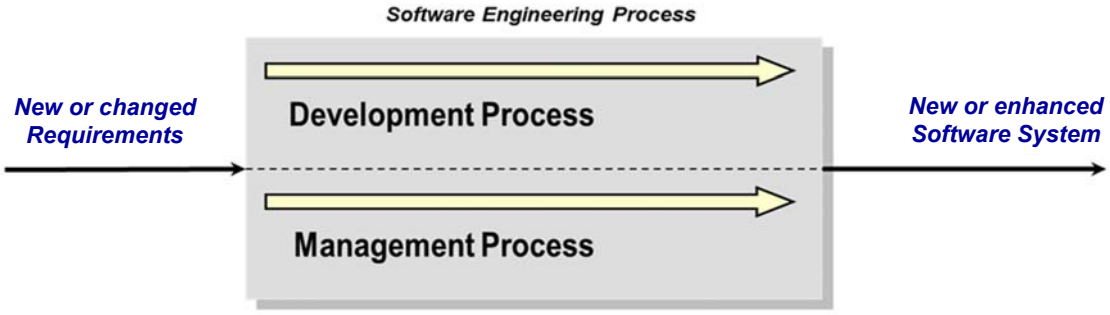
Software engineering is more like constructing bridges.

# Software Engineering Process

**A software engineering process consists of**

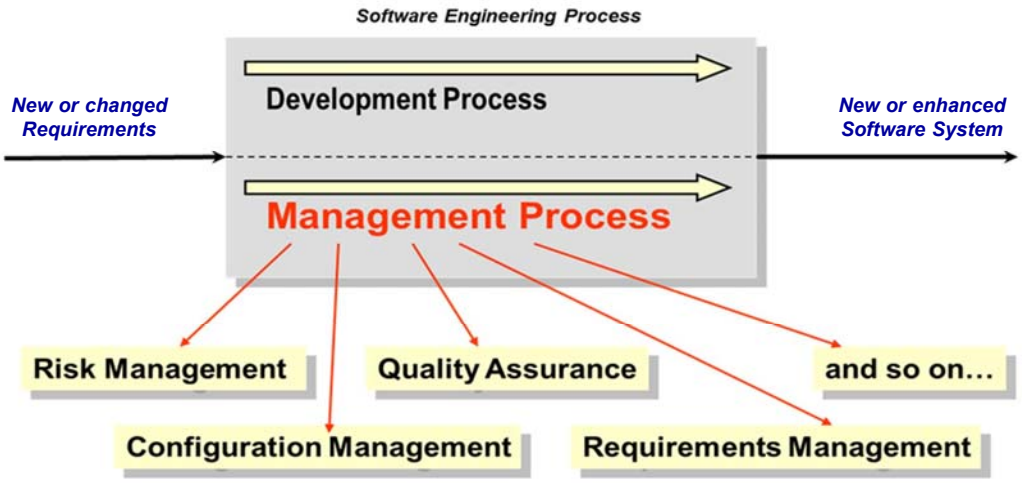
Development Process

Management Process



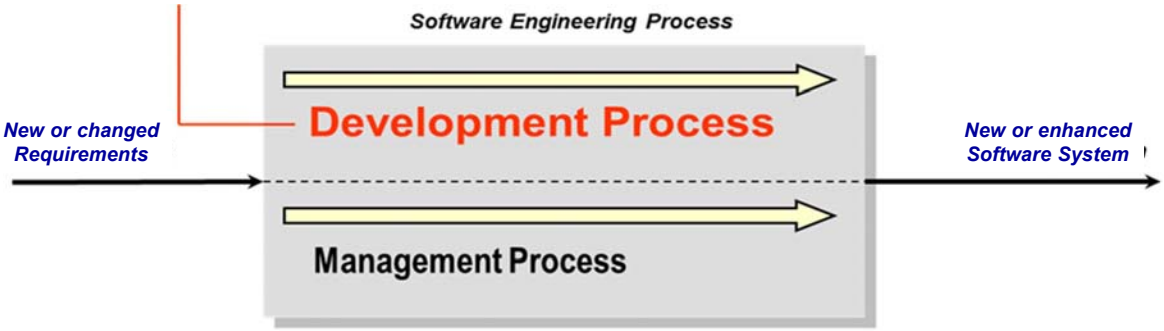



# Management Process



# Development Process

A development process defines who is doing what, when and how to reach a certain goal



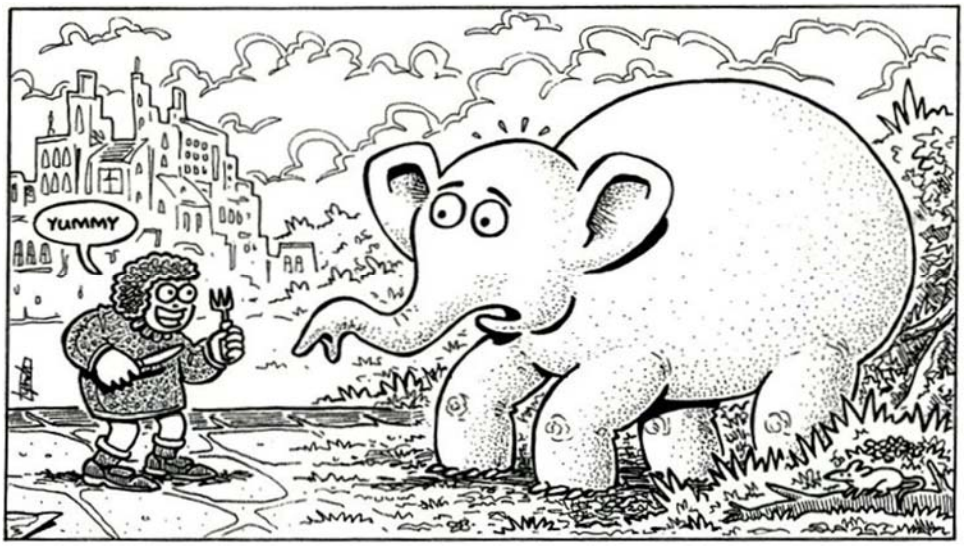

In software engineering, the goal is to build a software product or to enhance an existing one



# Basic disciplines in a process

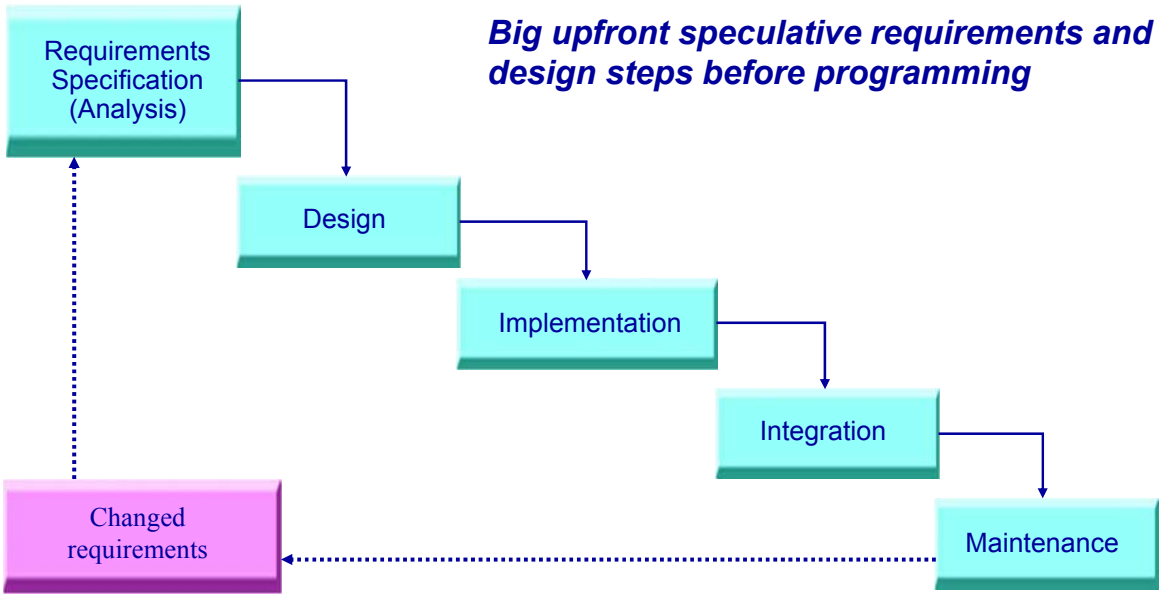
| Analysis | Design |
|---|---|
| What are the (functional/non-functional) requirements?<br>-Domain analysis<br>-Requirements gathering/analysis/spec. | How to devise a logical solution to fulfill the requirements?<br>-System Architecture, Internal Designs<br>-UI, Database designs etc. |
| **Implementation** | **Testing** |
| Coding of the logical solution | - Unit test, Integration test, Regression Test<br>- Acceptance Test<br>Does the system do what it was meant to do? |

# How to eat an elephant?



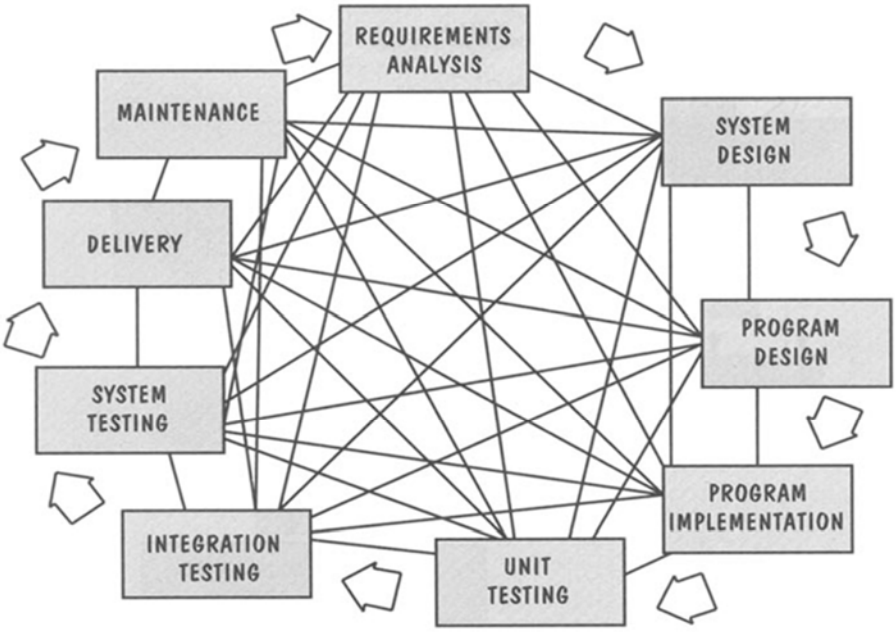



# Conventional Waterfall Model



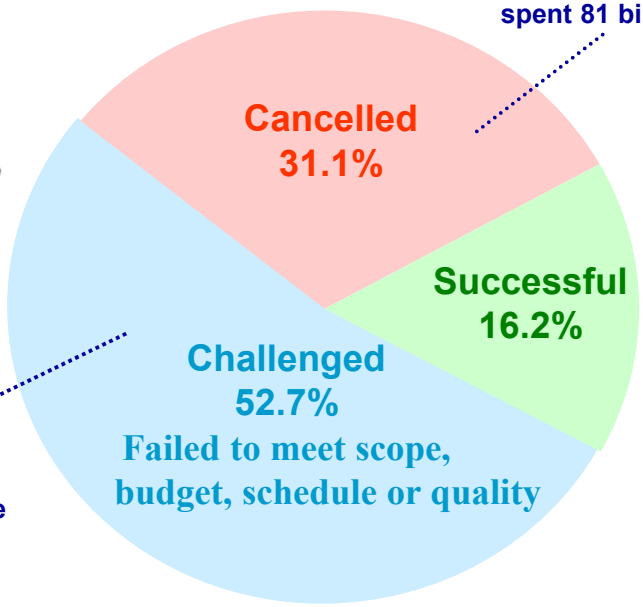

***Big upfront speculative requirements and design steps before programming***

# Waterfall Model In Reality






# Unpleasant Facts
## (All 8380 projects adopted Waterfall Model)




*Survey conducted by The Standish Group in 1995*

# Problems in Conventional Software Development

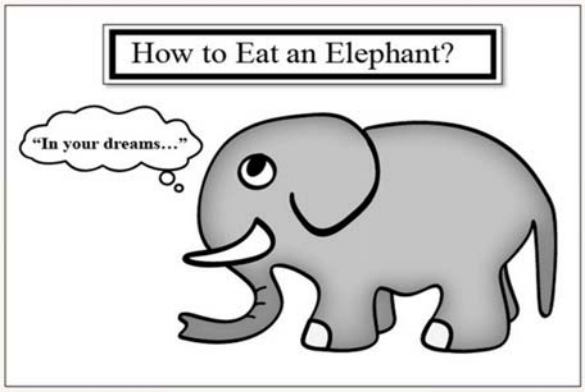
**Long delays**

**High development cost**

**High cancellation rate**

**Low quality**
**(*reliability*, *extensibility*, *maintainability* etc.)**

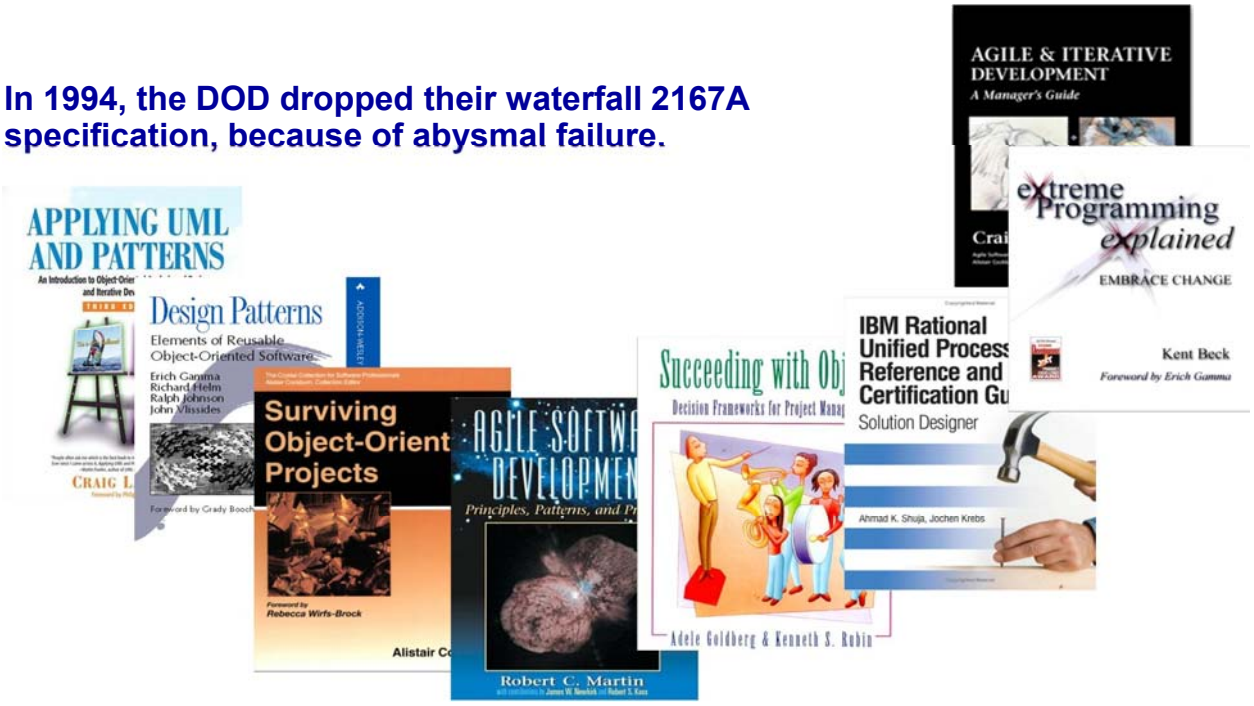
**High maintenance cost**





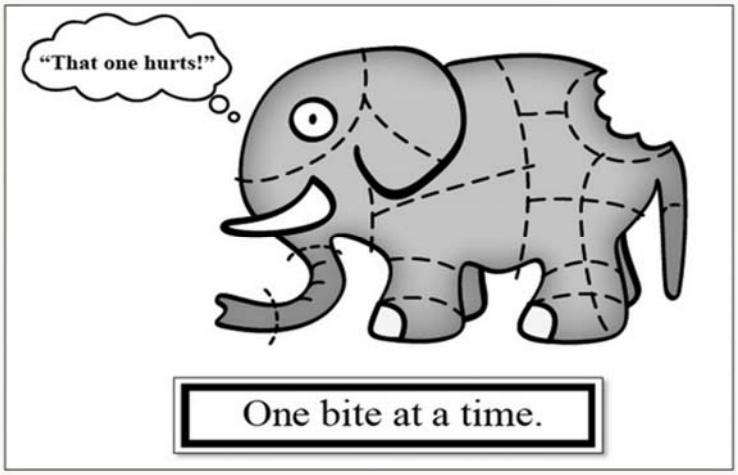

# Recent publications on software development process advocates replacing a waterfall with an iterative lifecycle

**In 1994, the DOD dropped their waterfall 2167A specification, because of abysmal failure.**



***You should use iterative development only on projects that you want to succeed!***

Martin Fowler




***Short quick development steps, feedback, and adaptation to clarify the requirements and design***

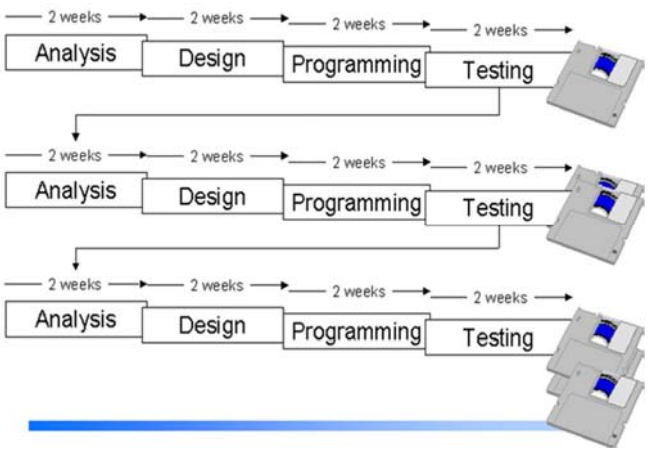
## Iterative & Incremental Process embraces Changes

**Iterative**

- Instead of building the entire system as one go, the project has a few or many builds
- A build includes only a subset of the entire functionality

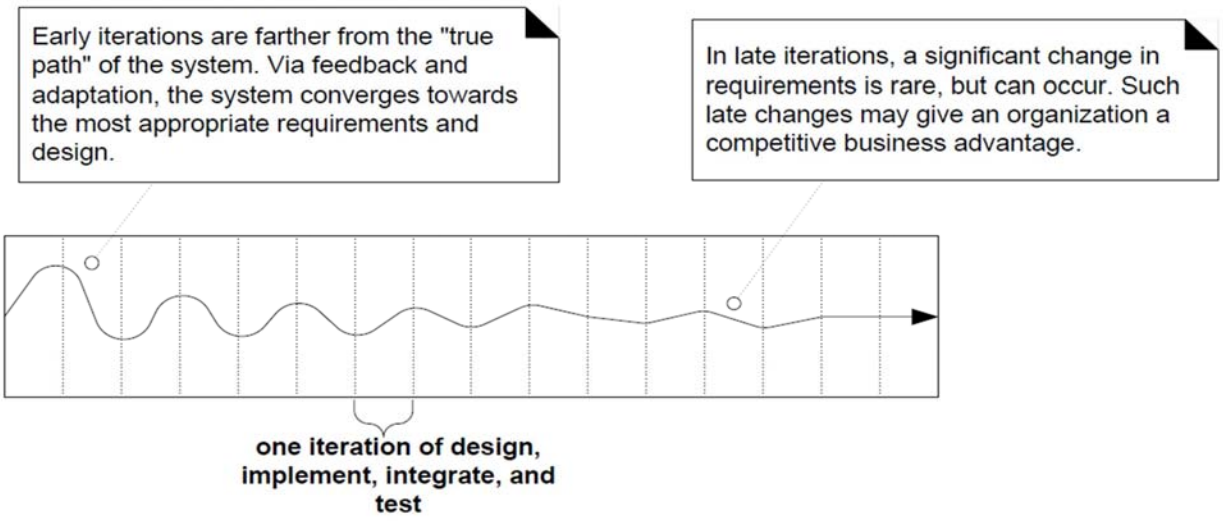
**Incremental**

- Software is developed on top of previous build
- Make small but noticeable improvements in each iteration

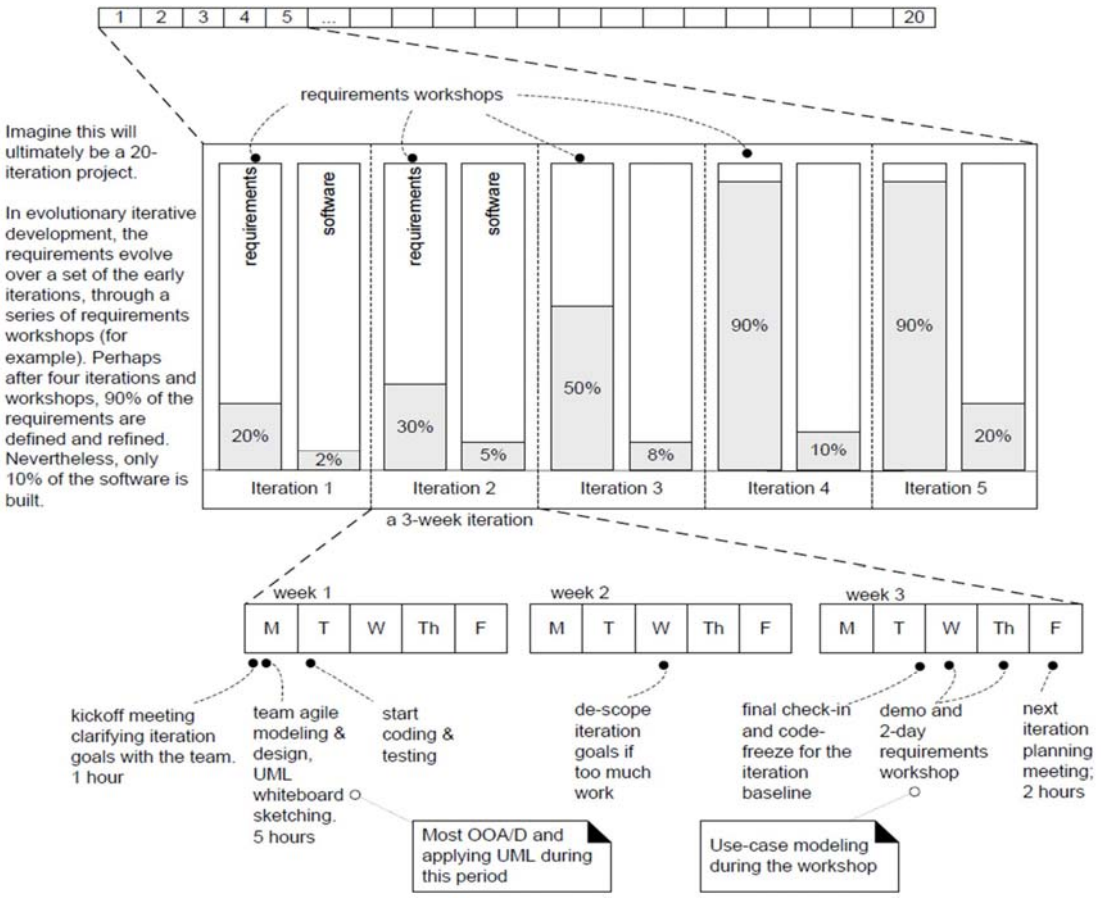



- Small steps, *feedback* and refinement and *adaptation*
- Time-boxed
- Aka. Evolutionary or spiral

# Iterations and Convergence





# Benefits of Iterative and Incremental Development Process

Early mitigation of high risks (technical, requirements, etc)

Early visible progress

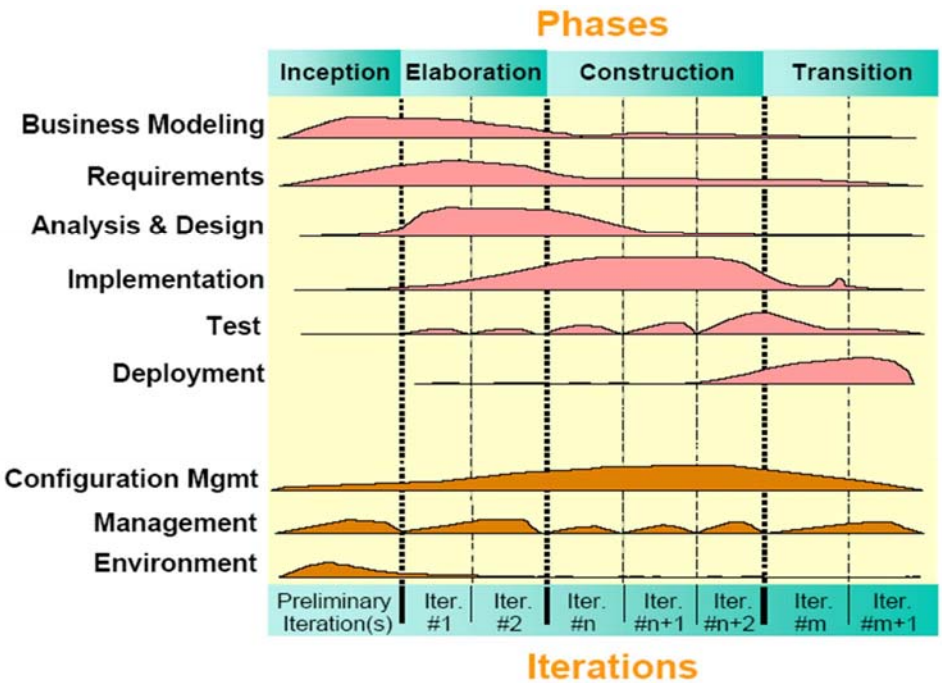
Early feedback, user engagement, and adaptation

Managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complexity steps

Can improve the process itself, iteration by iteration



# Example Process: Unified Process (UP)



## Example Process: Agile Methods




# OOAD and Unified Process

### Objectives

Define object-oriented analysis and design (OOA/D)

Illustrate a brief OOA/D example

Overview UP and define fundamental concepts in UP

Introduce our case study

# Analysis

Analysis emphasizes an *investigation*, *understanding*, and *discovery* of the problem domain and requirements

*what the problem is about* and *what a system must do*

Analysis does not concern how a logical solution is defined

All the vocabularies (e.g., class name, relationships, etc.) used in the analysis must come from the problem domain

Analysis requires domain knowledge and analyst expertise



# Requirements Analysis & Object Analysis

Requirements Analysis

Investigation of functional & non-functional requirements

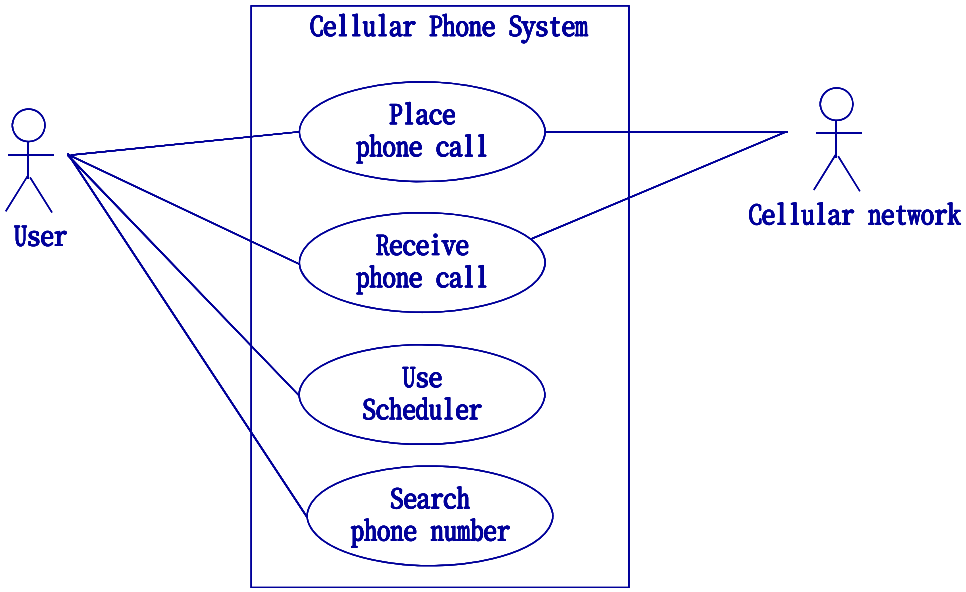
Functional requirements are captured by Use-Case Model

Object (or Domain) Analysis

Investigation of domain objects, i.e., emphasizing on finding and describing objects (or concepts), relationships among those concepts, and attributes of those concepts, in the problem domain

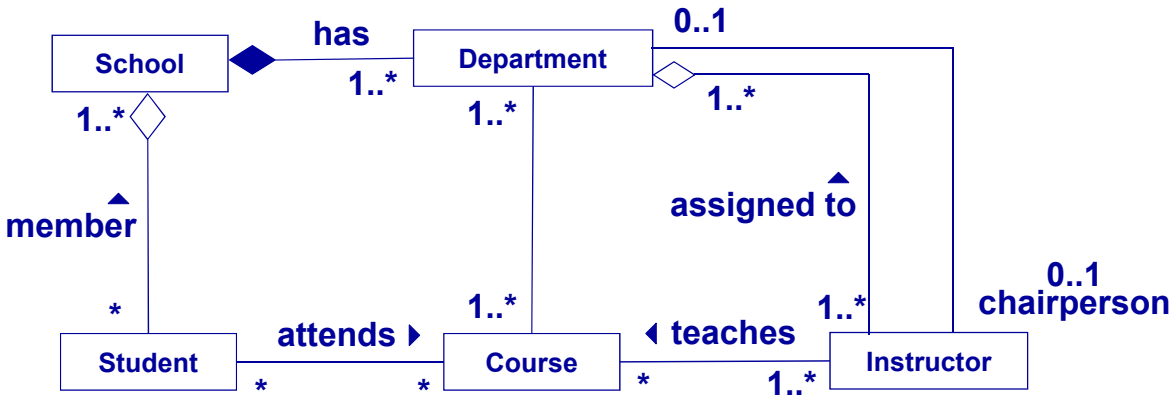
Captured by Domain Model

# Use-Case Diagrams






# Example: Domain Model



# Object-Oriented Design

**OO design (OOD) is primarily a process of *invention* and *adaptation of conceptual solution.***

**The development team defines software objects and how they collaborate to fulfill the system's behavioral requirements that are determined at requirements discipline.**

**OOD tends to be relatively independent of the language used.**

**e.g., design patterns help to transcend programming language-centric viewpoints**

**Obviously, the more consistent/related the OOP and OOD techniques, the easier they are to apply in real-life.**

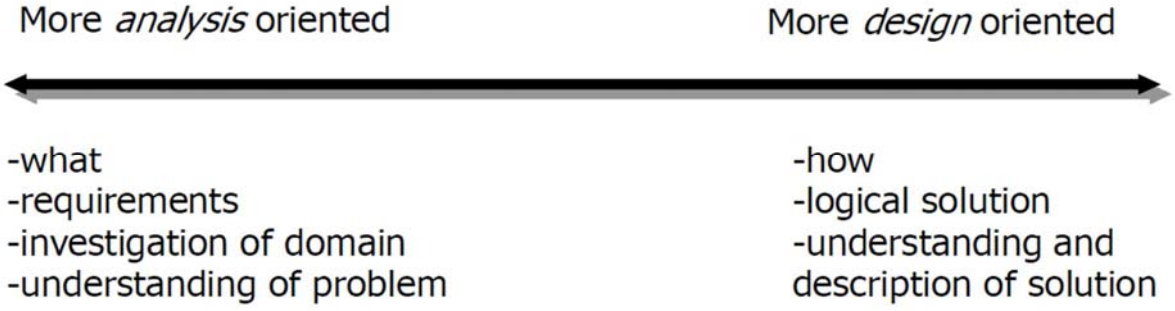


# OOA & OOD

**Division between OOA & OOD is fuzzy**

**OOA & OOD activities exist on a continuum**

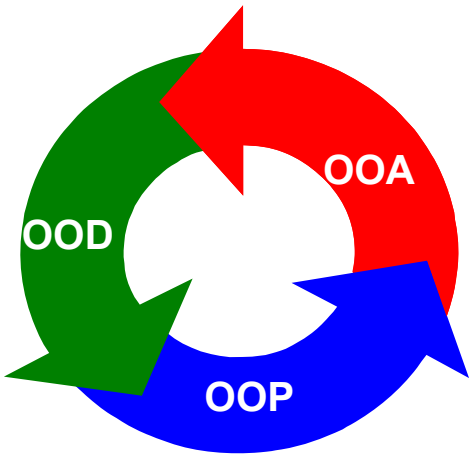
**Some practitioners can classify an activity as analysis while others put it into design category**



# Object-Oriented Programming

**This corresponds to the implementation discipline.**

**The classes and class operations are coded, tested, and integrated.**






# How Objects Are Used?

**During analysis:**

to promote understanding of the real world

**During design and programming:**

to provide a basis for logical solution and implementation

**Decomposition of a problem into objects depends on judgment and the nature of the problem.**

*There is no one correct representation!*

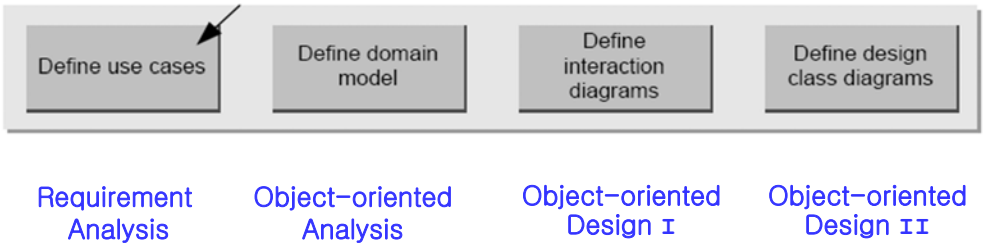
# A Simple Example

**Birds-eye view of Requirement Analysis and OOA/D**

**Example) A "dice game" in which a player rolls two die.**

- **If the total is seven, they win; otherwise, they lose.**

– **Four Steps**





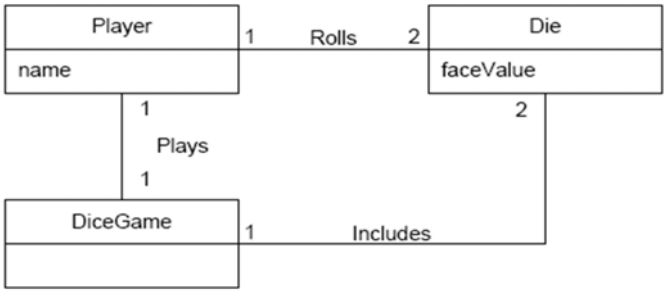

# A Simple Example (Cont'd)

1. **Define Use Cases (Requirement Analysis)**
   - **A description of related domain processes as *use cases*.**
   - **Play a Dice Game use case:**

     **Play a Dice Game: A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose.**

# A Simple Example (Cont'd)

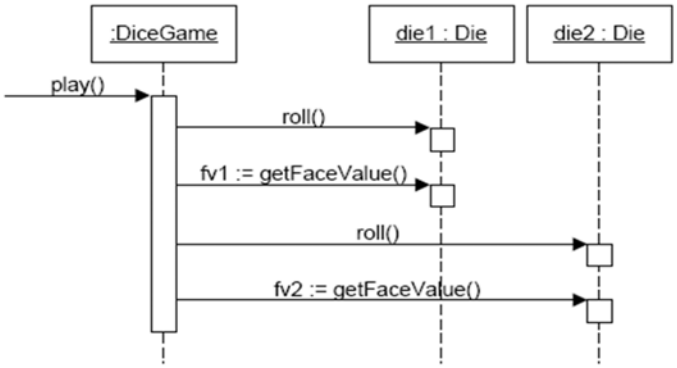
2. Define a Domain Model (OOA)
   - Creating a description of the domain from the perspective of classification by objects.
   - Domain model
     - A set of diagrams that show domain concepts or objects
     - Not a description of software objects





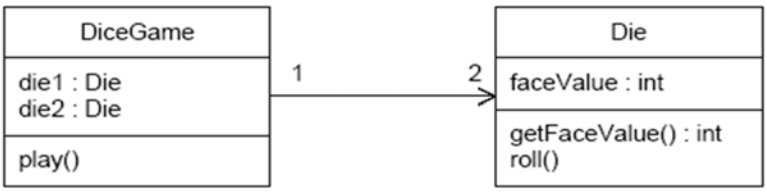

# A Simple Example (Cont'd)

3. Define Interaction Diagrams (OOD)
   - Defining software objects and their collaborations.
   - Interaction diagram (dynamic view of collaborating objects)
     - The flow of messages between software objects
     - The invocation of methods



# A Simple Example (Cont'd)

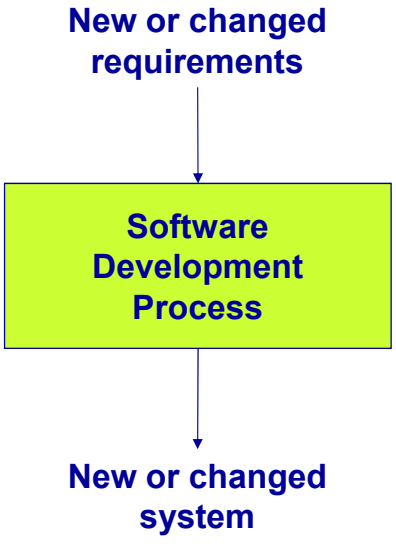
4. Define Design Class Diagrams (OOD)
   - A static view of the class definitions with a design class diagrams.
   - Design class diagram
     - The attributes and methods of the classes






**Review:** A development process defines **who** is doing **what**, **when** and **how** to reach a certain goal

In software engineering the goal is to build a software product or to enhance an existing one.



# Unified Process (UP) /Rational Unified Process (RUP)

**Developed by "*three amigos*" at Rational Software (IBM)**

*Grady Booch*
*(Booch Method)*

*Ivar Jacobson*
*(OOSE)*

*James Rumbaugh*
*(OMT)*

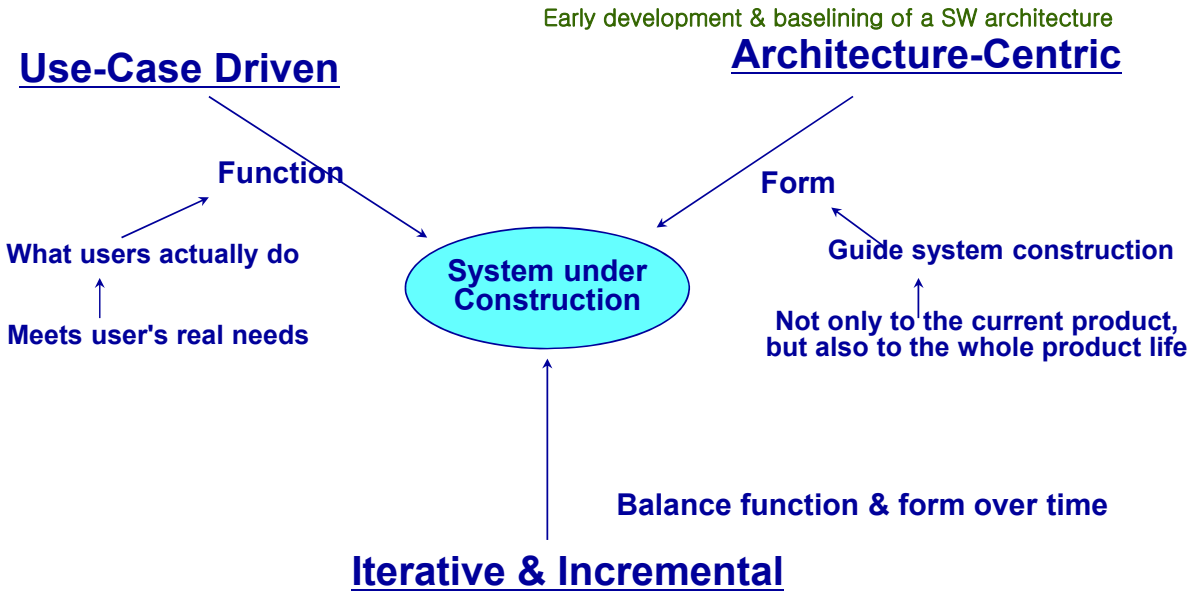
**Interestingly different from the traditional waterfall model**

**Unified Modeling Language (UML) is a set of graphical notations for modeling systems, not a process or method.**

**You don't have to use UP to use UML.**

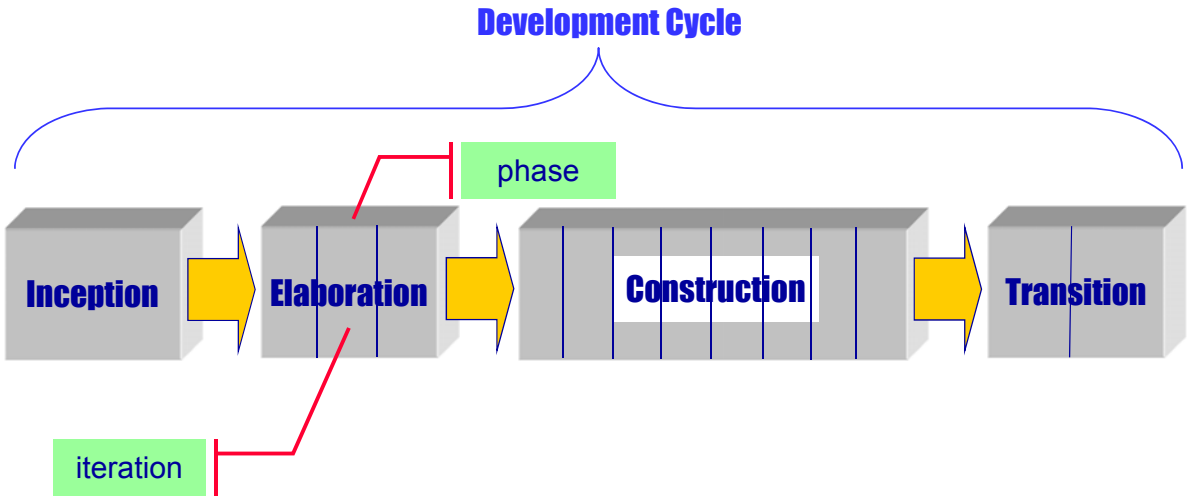


# Core of the Unified Process (UP)



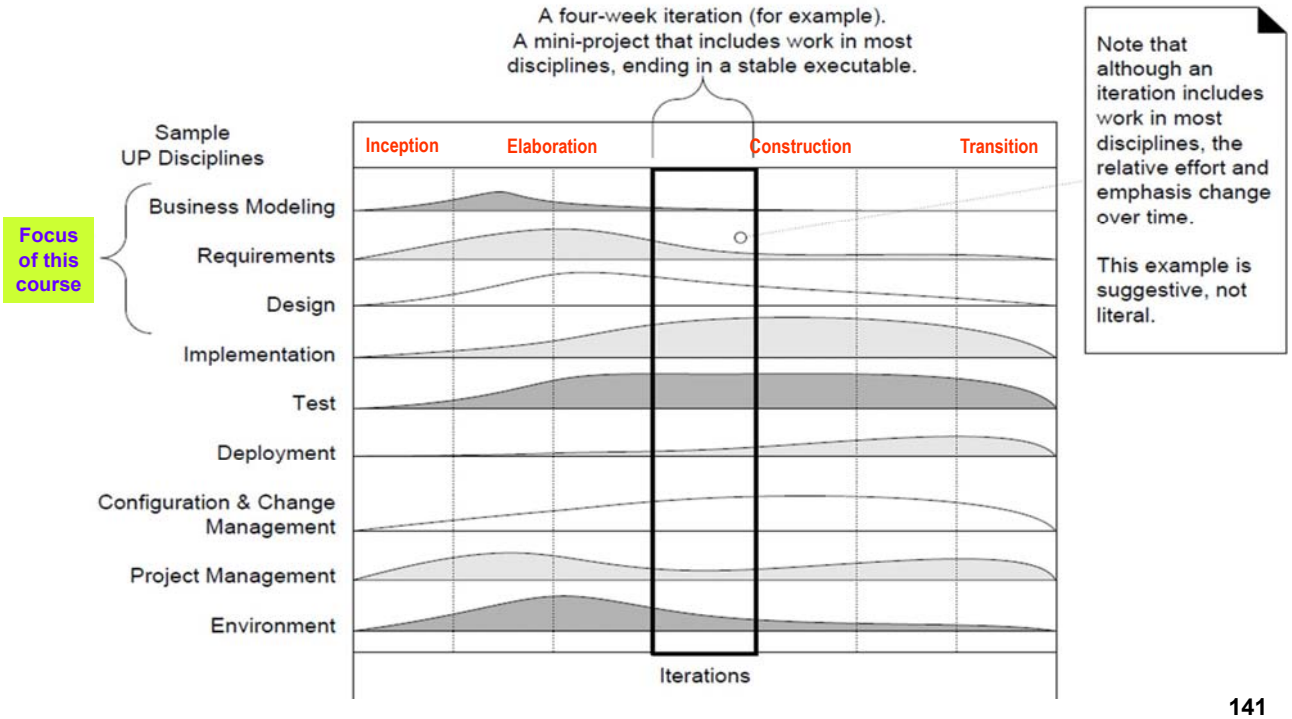
# Four Phases of Unified Process

(Phases are *not* the classical requirements/ design/coding/implementation activities)






# 2D View of Unified Process



# Inception Phase (Feasibility Phase)

*Envision the product scope, vision, and business case*


**A short initial step in which the following questions are explored:**

**What is the vision and business case for this project?**
**Feasible?**
**Buy and/or build?**
**Rough estimate of cost: Is it $10K-100K or in the millions?**
**Should we proceed or stop?**



# Elaboration Phase

*Define most requirements, build the core architecture, resolve the high-risk elements, and estimate overall schedule and resources*


**The majority of requirements are discovered and stabilized.**

Write most of the use cases and other requirements in detail, through a series of workshops, once per elaboration iteration.

**The major risks (in terms of techniques and/or business value) are mitigated or retired.**

**The core (or baseline) architecture is implemented and proven.**

**More realistic estimates and clear milestones are specified.**

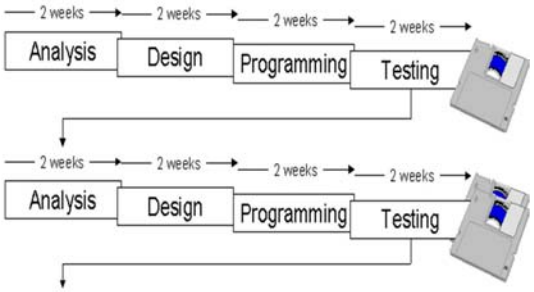
# Elaboration Phase (Cont'd)

**Elaboration consists of between 2 and 4 iterations; each iteration is recommended to be between 2 and 6 weeks, unless the team size is massive.**

**Each iteration is timeboxed, meaning its end date is fixed.**

*What do we have to do if we cannot meet the deadline?*

**At the end of each iteration, stable and tested production-quality portions of the final system must be released.**






# Construction and Transition Phases

**Construction Phase**


Iterative implementation of remaining lower risk & easier elements

**Transition Phase**


Beta Test, Performance Tuning

# Additional UP Best Practices

Tackle high-risk and high-value issues in early iterations.

Continuously engage users for evaluation, feedback and requirements.

Continuously verify quality; test early, often, and realistically.

Model software visually (with the UML).

Carefully manage requirements.

Practice change request and configuration management.



# UP Disciplines and Artifacts

A *discipline* is a set of activities (and related artifacts) in one subject area, such as activities in requirements analysis

An *artifact* is the general term used for any work product

We will focus on some artifacts in the following disciplines

| | | |
|---|---|---|
| Business modeling | ⇒ | Domain Model |
| Requirements | ⇒ | Use-Case Model |
| Design | ⇒ | Design Model |

# Unified Process Artifacts

| Discipline | Artifact<br>Iteration → | Incep.<br>I1 | Elab.<br>E1. .En | Const.<br>CL.Cn | Trans.<br>T1..T2 |
|---|---|---|---|---|---|
| Business Modeling | Domain Model | | s | | |
| Requirements | Use-Case Model | s | r | | |
| | Vision | s | r | | |
| | Supplementary Specification | s | r | | |
| | Glossary | s | r | | |
| Design | Design Model | | s | r | |
| | SW Architecture Document | | s | | |
| | Data Model | | s | r | |
| Implementation | Implementation Model | | s | r | r |
| Project Management | SW Development Plan | s | r | r | r |
| Testing | Test Model | | s | r | |
| Environment | Development Case | s | r | | |

s – start; r – refine



# Artifacts in Inception Phase

| Artifacts | Comments |
|---|---|
| Vision and Business Case | Describes high-level goals and constraints, the business case, and provides an executive summary. |
| **Use-Case Model** | Describes functional requirements, and related non-functional requirements. |
| Supplementary Specification | Describes other requirements. |
| Glossary | Key domain terminology. |
| Risk List & Risk Management Plan | Describes business, technical, resource, schedule risks, and ideas for their mitigation or response. |
| Prototypes and proof-of-concepts | To clarify the vision, and validate technical ideas. |
| Iteration Plan | Describes what to do in the first elaboration iteration. |
| Phase Plan & Software Development Plan | Low-precision guess for elaboration phase duration and effort, Tools, people, education, and other resources. |
| Development Case | A description of customized UP steps and artifacts for this project. In UP, one always customizes it for the project. |

# Artifacts that May Start in Elaboration

| Artifacts | Comments |
|---|---|
| **Domain Model** | This is a visualization of the domain concepts; it is similar to a static information model of the domain entities. |
| **Design Model** | This is the set of diagrams that describe the logical design. This includes software class diagrams, object interaction diagrams, package diagrams, and so forth. |
| Software Architecture Document | A learning aid that summarizes the key architectural issues and their resolution in design. It is a summary of the outstanding design ideas and their motivation in the system. |
| Data Model | This includes the database schemas, and the mapping strategies between object and non-object representations. |
| Test Model | A description of what will be tested, and how. |
| Implementation Model | This is the actual implementation – the source code, executables, databases, and so on. |
| Use-Case Storyboards, UI Prototypes | A description of the user interface, paths of navigation, usability models, and so forth. |



# Fitting a Process to a Project

**Software projects are greatly diverse in:**

**kind of system to build**
**technology to use**
**size & distribution of the team**
**nature of the risks**
**consequences of failure**
**working styles of the team**
**culture of the organization**

- ***No one-size-fits-all process that will work for all projects.***
- ***Adapt an appropriate process to fit your particular project environment.***

# The Development Case

**The choice of UP artifacts for a project may be written up in a short document called the Development Case (an artifact in the Environment discipline)**

**In the UP, one always customize the steps and artifacts (i.e., Development Case) for the project.**



# Agile UP

**Prefer a small set of UP activities and artifacts.**

*Focus on early programming, not early documentation*

**Requirements and designs emerge through a series of iterations, based on feedback.**

**Apply the UML with agile modeling practices.**

**There isn't a detailed plan for the entire project.**

**Phase Plan**: *estimates project duration and other major milestones*
**Iteration Plan**: *adaptively plans with greater detail one iteration in advance*

## What is Agile Modeling?

**Adopting an agile method does not mean avoiding any modeling**

**The purpose of modeling and models is primarily to support understanding and communication, not documentation**

**Don't model or apply the UML to all or most of the software design**

**Use the simplest tool possible**

Prefer "low energy" creativity-enhancing simple tools that support rapid input and change



## Two Desert Island Skills in OOA & OOD

**Assigning responsibilities to software components.**

**Finding suitable objects or abstractions.**

# Case Study:
# The NextGen POS System

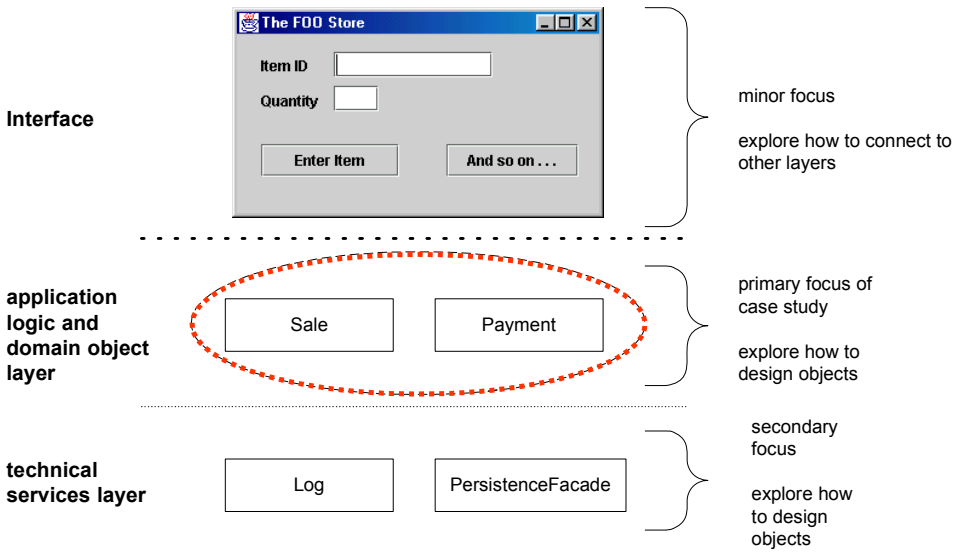
**The POS (Point-Of-Sale) system is a computerized system used to record sales and handle payments; primary goal of the system is**

- Quick checkout for the customer
- Fast and accurate sales analysis
- Automatic inventory control

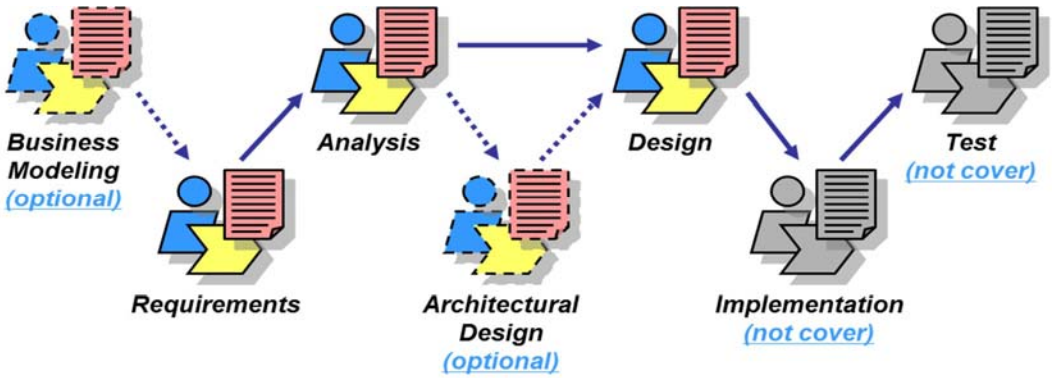
**Assume that we have been requested to create the software to run a POS system. Using an iterative-incremental development strategy, we are going to proceed through OO analysis, design, and implementation.**



# Architectural Layers



# Our Process


Business
Modeling
(optional)
Requirements
Analysis
Architectural
Design
(optional)
Design
Implementation
(not cover)
Test
(not cover)