Object Oriented Analysis and Design

- Introduction

Questions

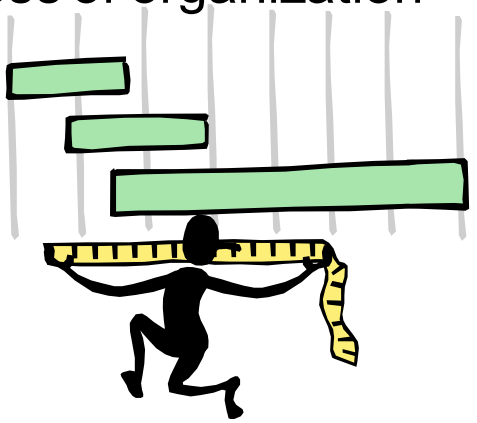
- From user's point of view, criticize a hardware or software system that has a flaw that especially annoys you. Describe the system, the flaw, how it was overlooked, and how it could have been avoided with a bit more thought during design.
- What major problems have you encountered during past software projects? How much time you have spend on analysis, design, coding and testing/debugging/fixing. How do you go about estimating how much effort a project will require?
- Recall a past system that you created. Briefly describe it. What obstacles did you encounter in the design? What software engineering methodology, if any, did you use? What were your reasons for choosing or not choosing a methodology? Are you satisfied with the system as it exists? How difficult is it to add new features to the system? Is it maintainable?
- Describe a recent large software system that was behind schedule, over budget, or failed to perform as expected. What factors were blamed? How could the failure have been avoided?

System/Software Life Cycle (SLC)

- Life cycle is the series of steps that software undergoes from concept exploration through retirement

- Intended to develop information systems in a very **deliberate**, **structured** and **methodical** way, reiterating each stage of the life cycle.

- Maturity of the process is some gauge of success of organization



Systems Development Life Cycle (SDLC)

Life-Cycle Phases

System Concept Development

Defines the scope or boundary of the concepts. Includes Systems Boundary Document. Cost Benefit Analysis. Risk Management Plan and Feasibility Study.

Planning

Develops a Project Management Plan and other planning documents. Provides the basis for acquiring the resources needed to achieve a soulution.

Requirements Analysis

Analyses user needs and develops user requirements. Create a detailed Functional Requirements Document.

Design

Transforms detailed requirements into complete, detailed Systems Design Document Focuses on how to deliver the required functionality

Development

Converts a design into a complete information system Includes acquiring and installing systems environment; creating and testing databases preparing test case procedures; preparing test files, coding, compiling, refining programs; performing test readiness review and procurement activities.

Integration and Test

Demonstrates that developed system conforms to requirements as specified in the Functional Requirements Document. Conducted by Quality Assurance staff and users. Produces Test Analysis Reports.

Implementation

Includes implementation preparation, implementation of the system into a production environment, and resolution of problems identified in the Integration and Test Phases

Operations & Maintenance

Describes tasks to operate and maintain information systems in a production environment. includes Post-Implementation and In-Process Reviews.

Disposition

Describes end-of-system activities, emphasis is given to proper preparation of data.

Importance of Lifecycle Models

- Provide guidance for project management
  - what major tasks should be tackled next? milestones!
  - what kind of progress has been made?
- The necessity of lifecycle models
  - characteristics of software development has changed
    - early days: programmers were the primary users
    - modest designs; potential of software unknown
  - more complex systems attempted
    - more features, more sophistication → greater complexity, more chances for error
    - heterogeneous users

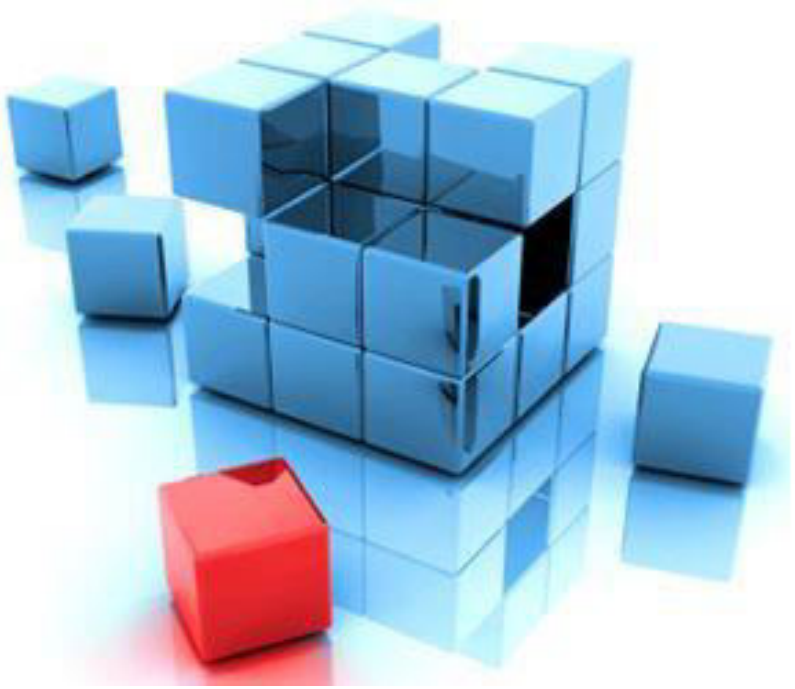
Object Oriented Approach

- Software development based on modeling objects from the real world and then use the models to build a language independent design organized around those objects.
- Promote better understanding of requirements, cleaner designs, and more maintainable systems.
- Use language-independent graphical notation for analyzing problem requirements, design a solution to the problem and then implement the solution.
- Same concept and notation throughout the software development process.
- Object oriented concepts throughout the software life cycle, from analysis through design to implementation.

Cont.

- Coding - last stage in the process of development.
- A good design technique defers implementation details until later stages of deign to preserve flexibility.
- Mistakes in the front of development process have a large impact on the ultimate product and on the time needed to finish.
- OOT is a way of thinking abstractly about a problem using real world concepts, rather than a computer concepts.
- Graphical notations helps developer to visualize a problem without prematurely resorting to implementation

Why Modeling..?



Modeling

- A model is a simplification of reality
- A hypothetical description of a complex entity or process
- A good model includes those elements that have broad effect and omits minor elements
- A model of a system is not the system!
- Modeling is used in many disciplines – architecture, aircraft building, …

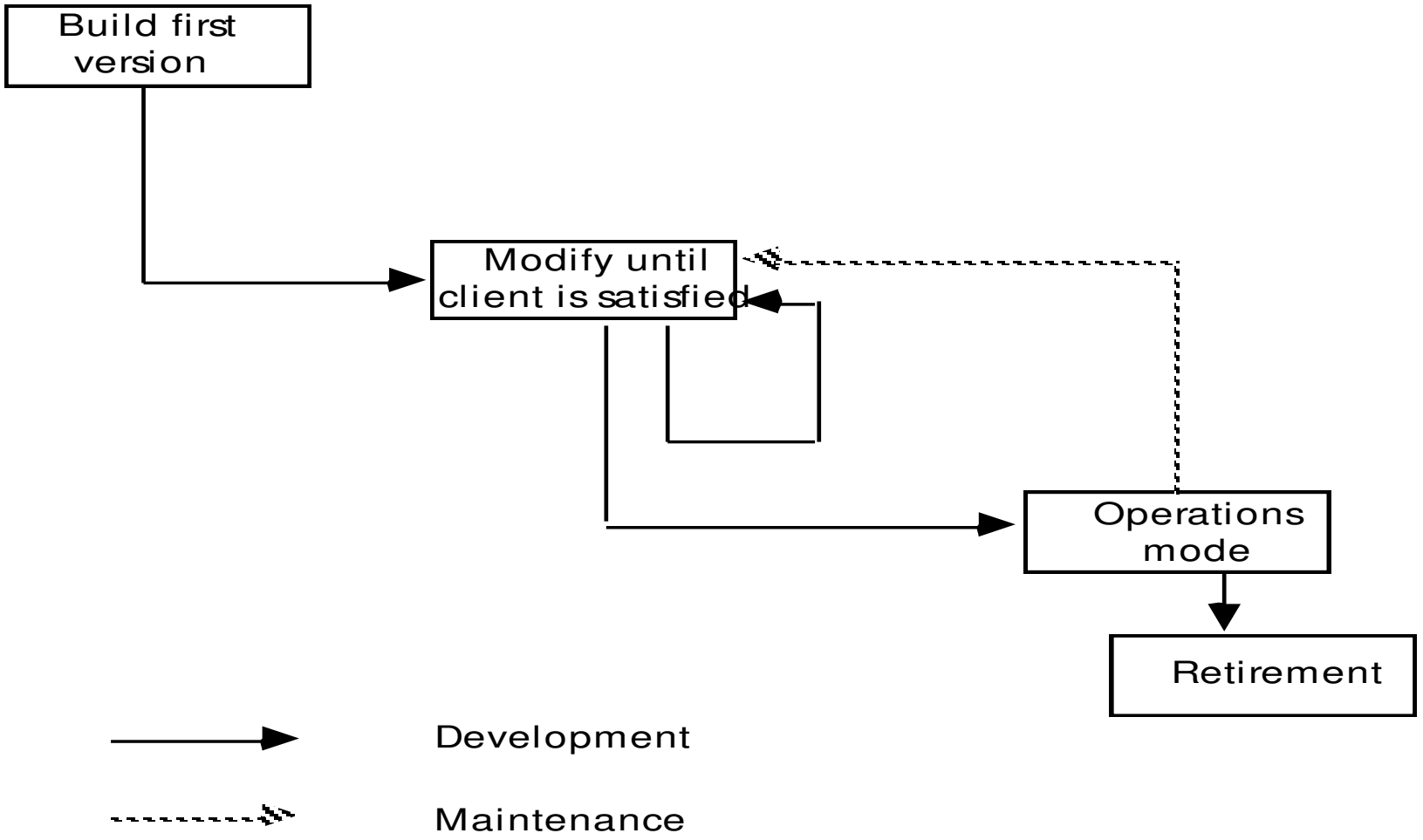
Why Modeling..?

- ✓ Models help us to **visualize** a system as it is or as we want it to be.
- ✓ Models permit us to **specify the structure or behavior** of a system.
- ✓ Models give us a **template** that guides us in constructing a system.
- ✓ Models document the **decisions** we have made.

Why Models are needed?

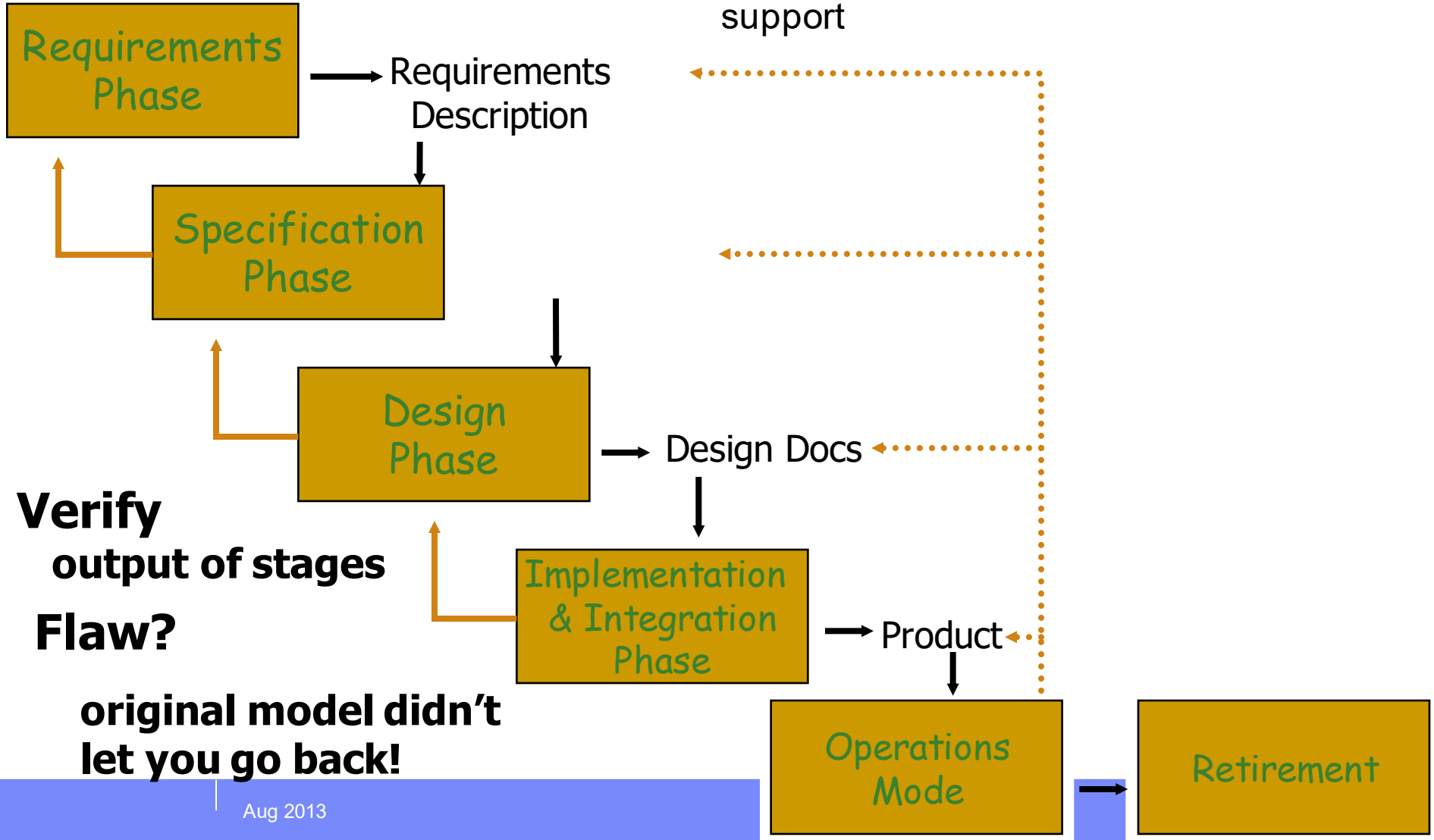
- Symptoms of inadequacy: the software crisis
  - scheduled time and cost exceeded
  - user expectations not met
  - poor quality

Build and Fix Model



Waterfall Model

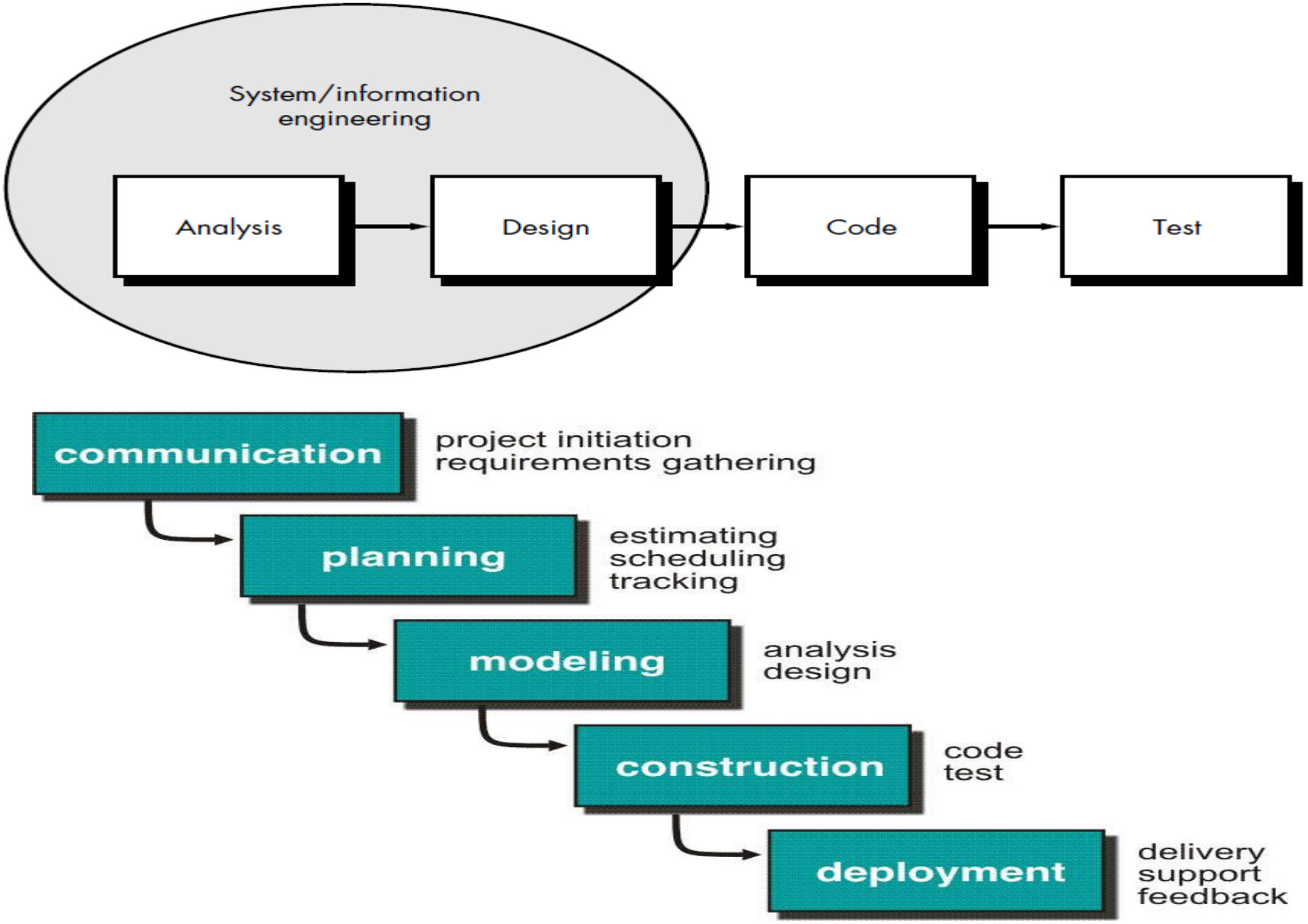
a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support



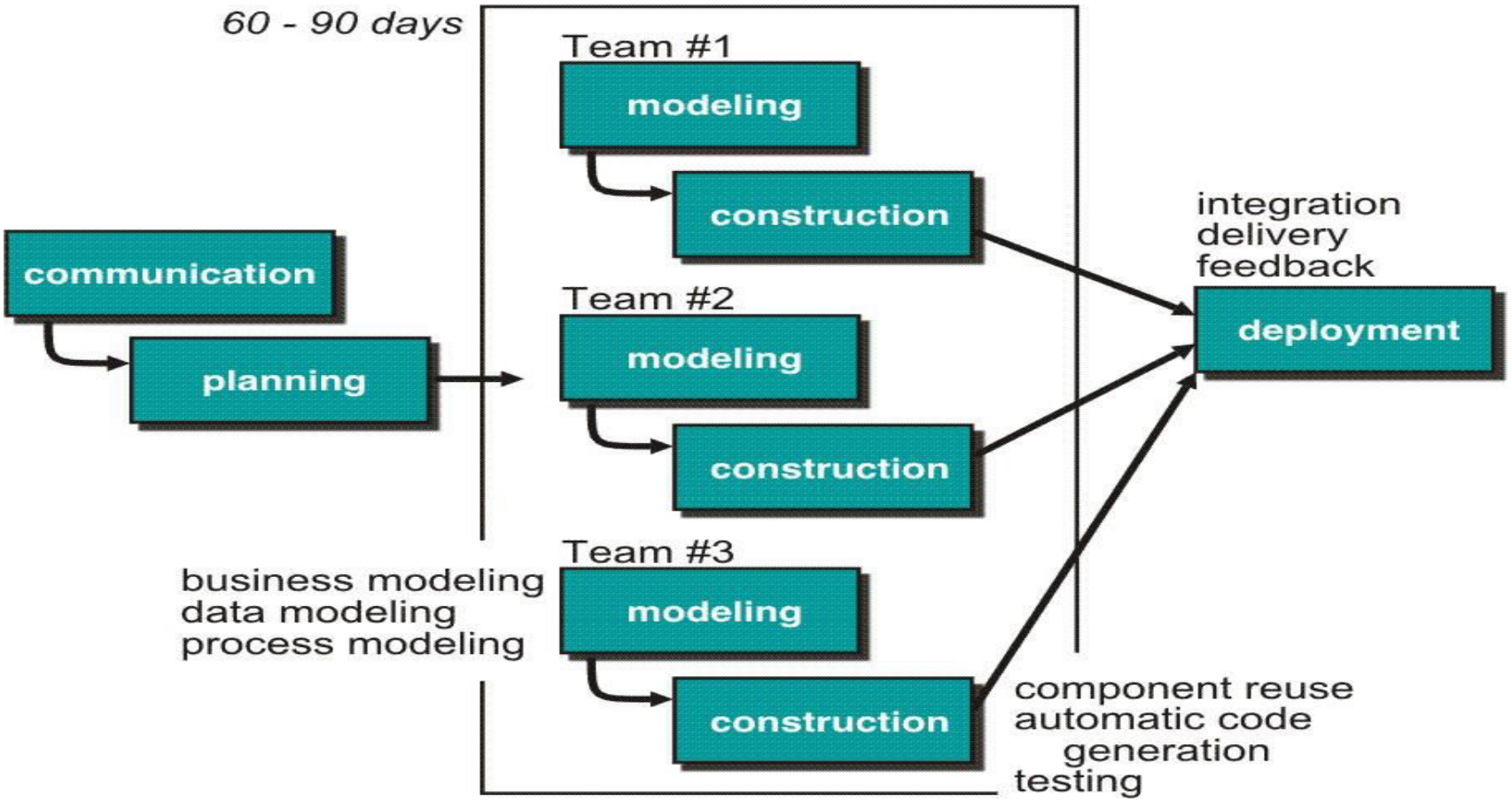
Verify
output of stages

Flaw?

original model didn't let you go back!

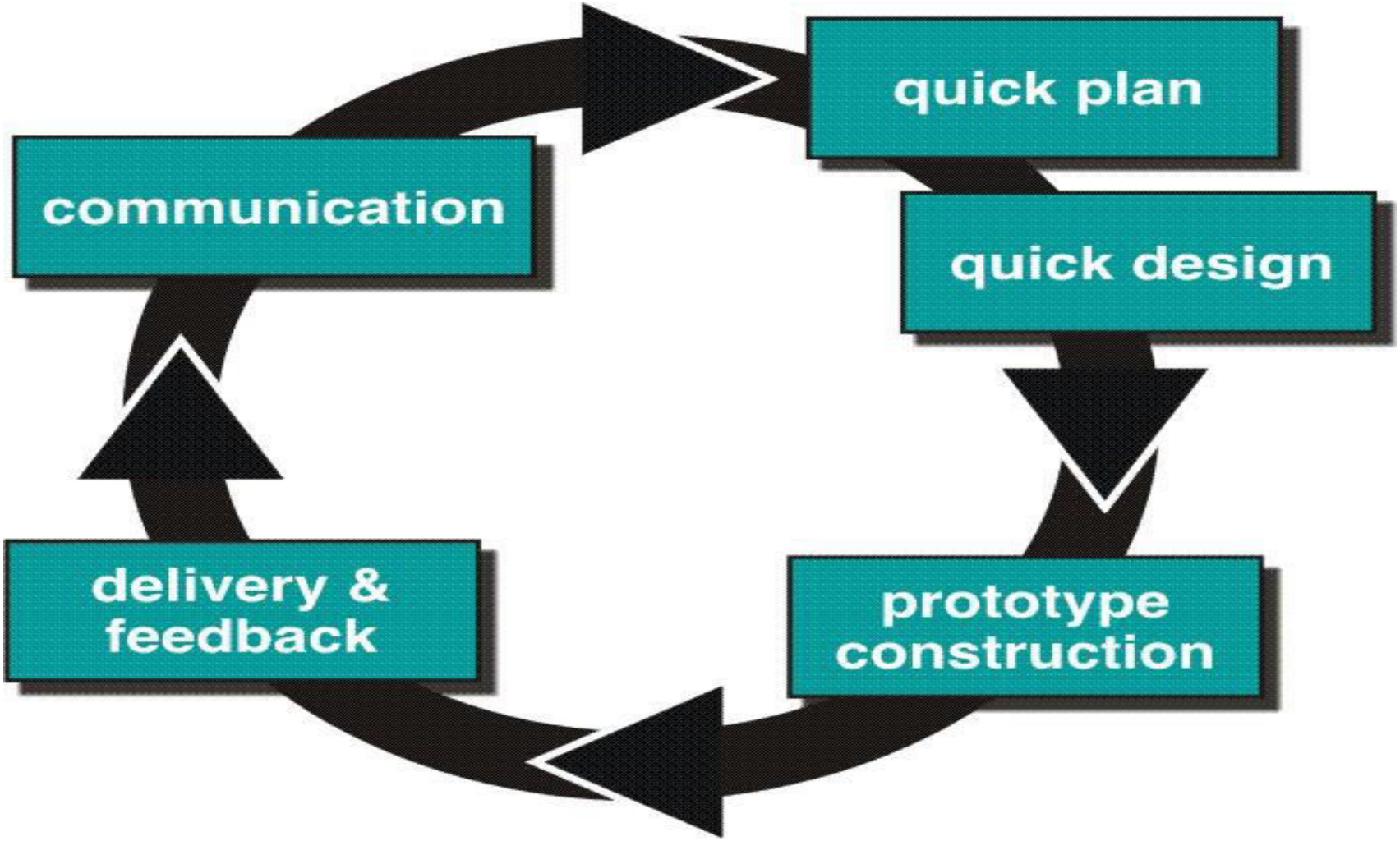

System/information engineering
Analysis
Design
Code
Test
communication
project initiation
requirements gathering
planning
estimating
scheduling
tracking
modeling
analysis
design
construction
code
test
deployment
delivery
support
feedback

Rapid Application Development (RAD) Model



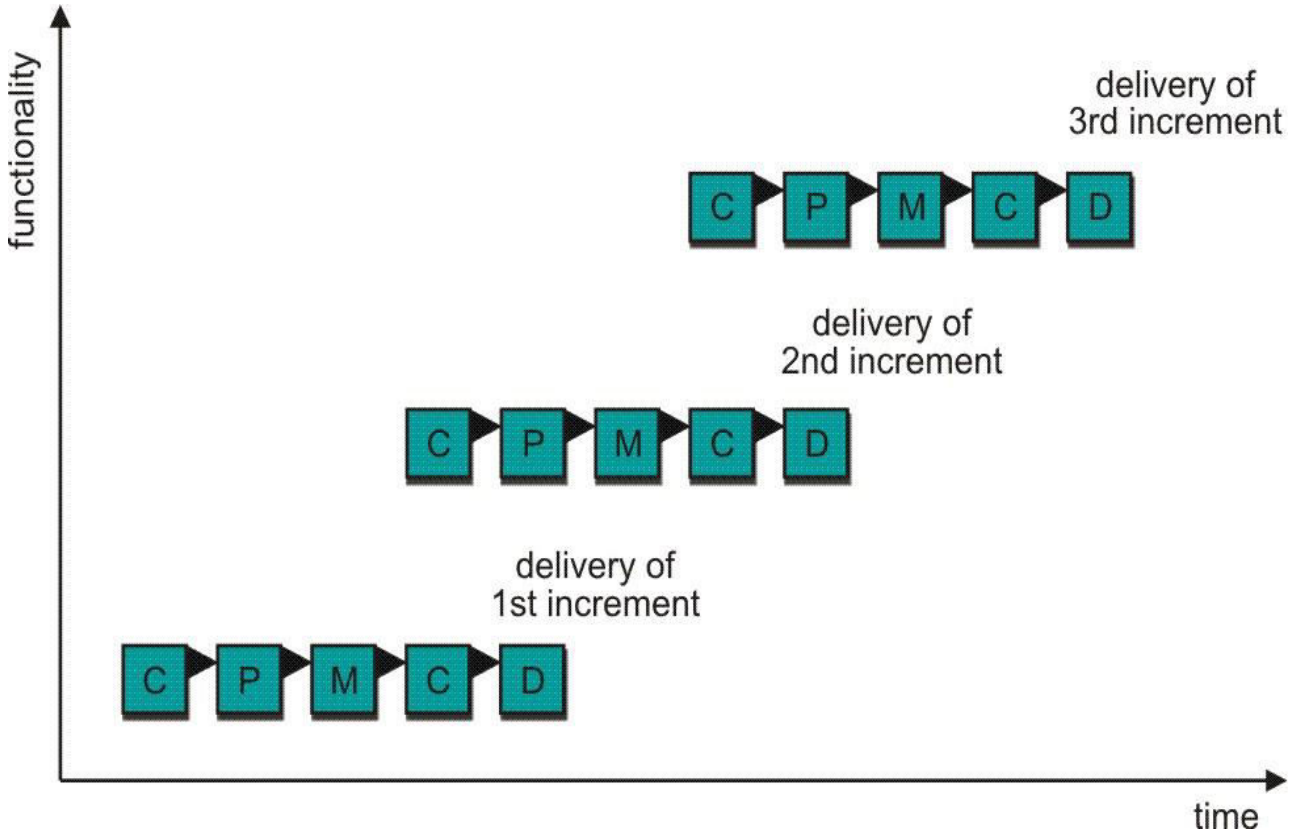
Makes heavy use of reusable software components with an extremely short development cycle

Prototype Model



Incremental Model

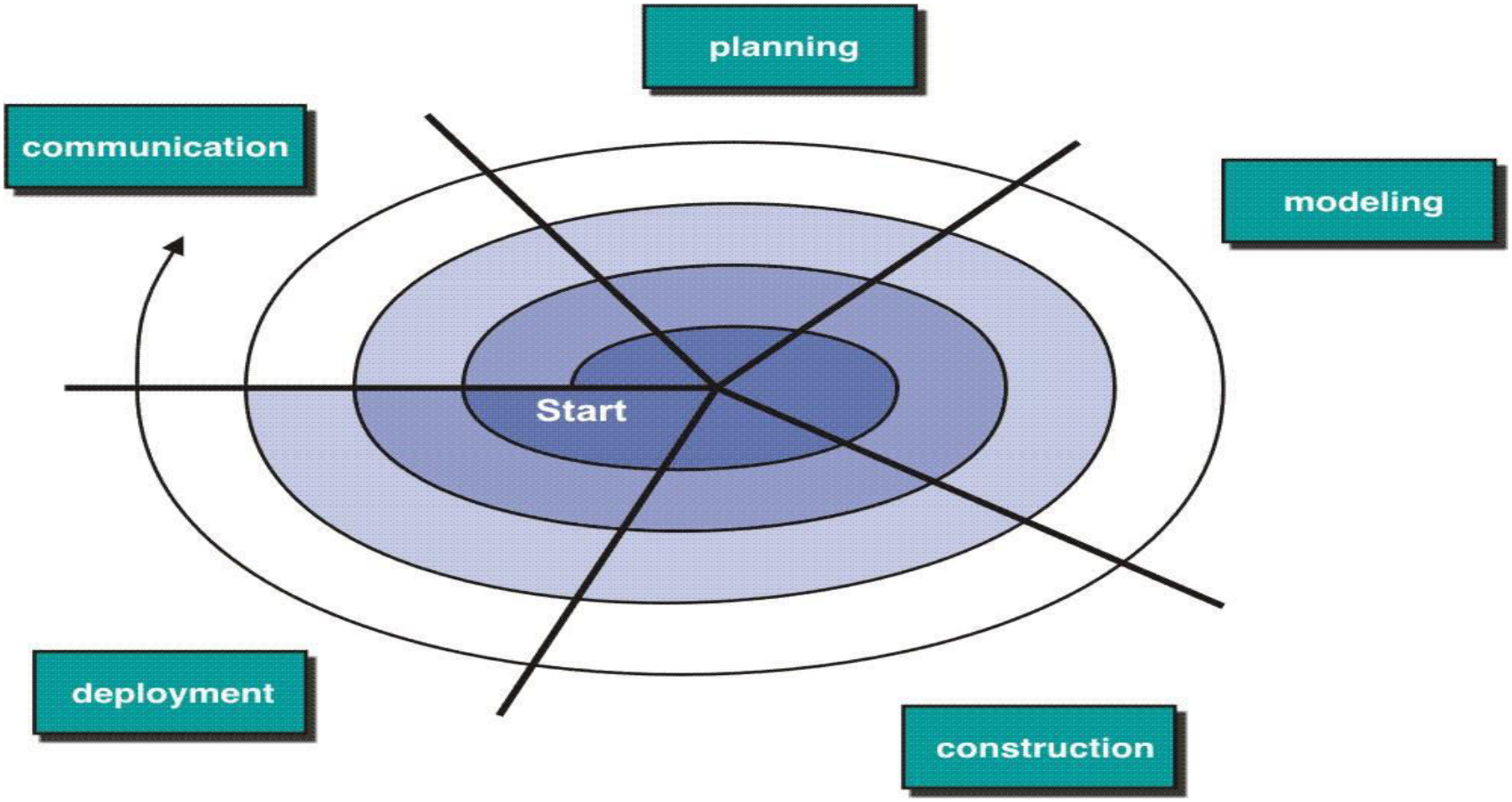
(combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping)



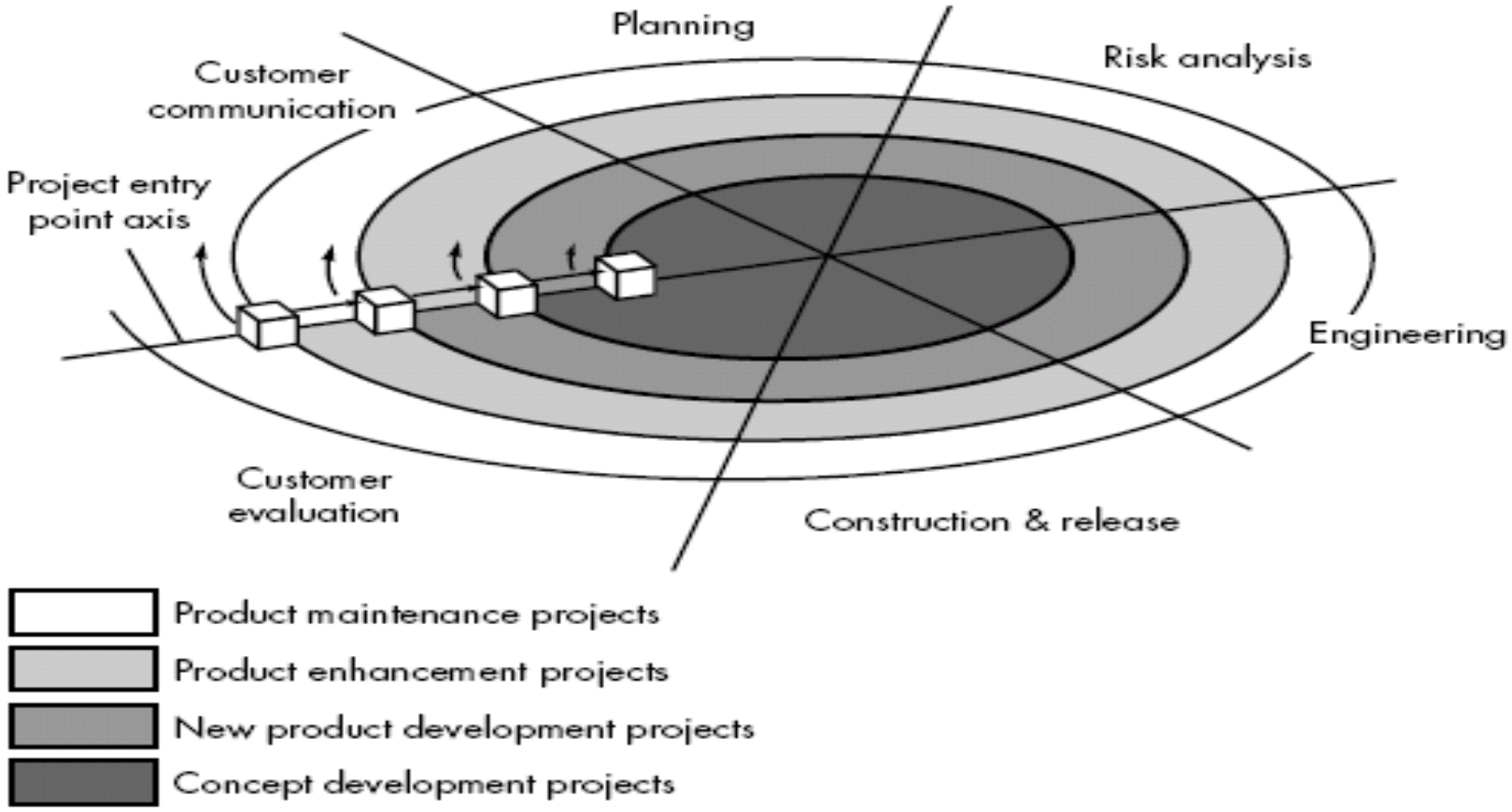
C- Communication
P - Planning
M – Modeling
C - Construction
D - Deployment

Delivers software in small but usable pieces, each piece builds on pieces already delivered

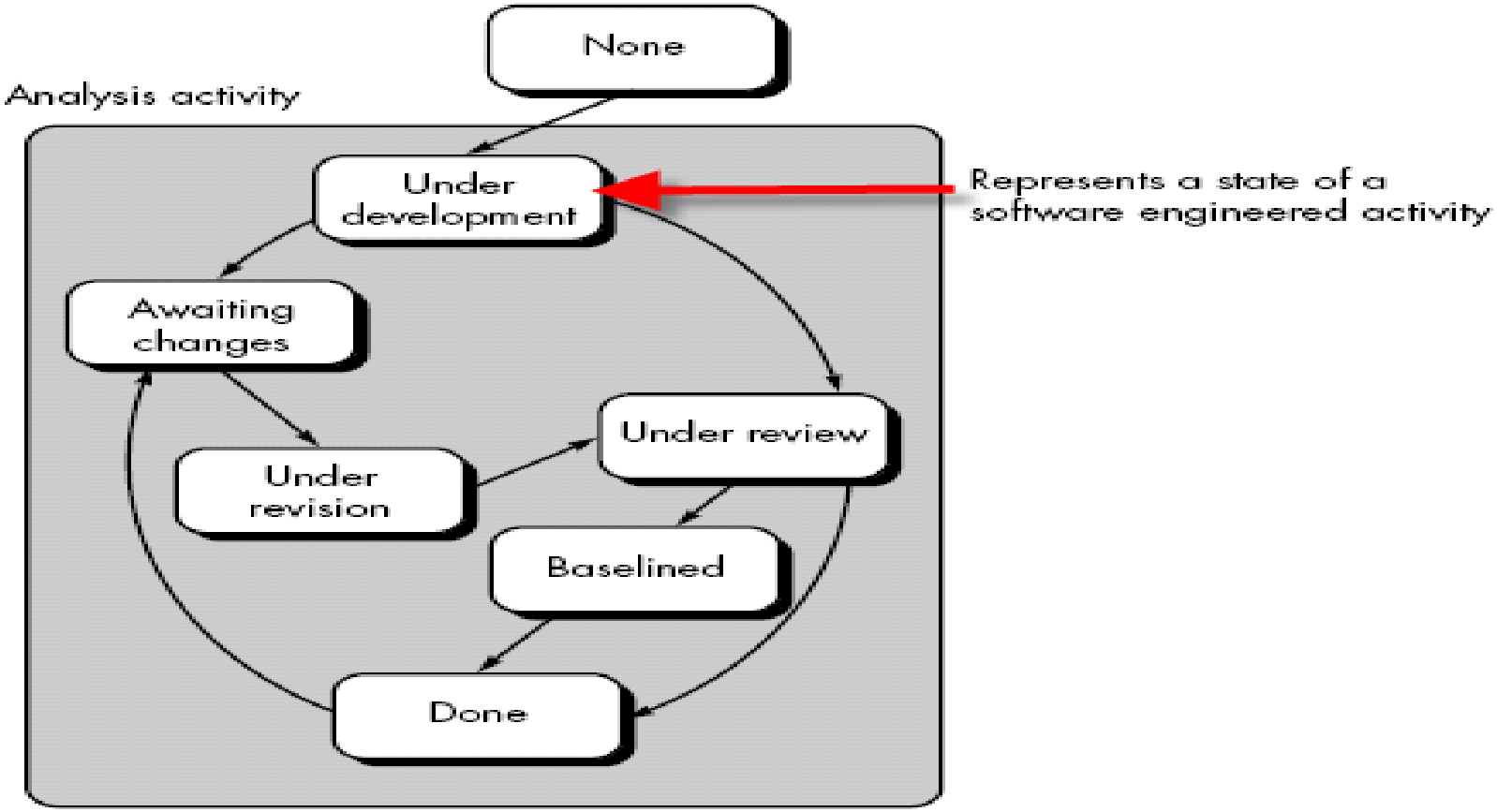
Spiral Model



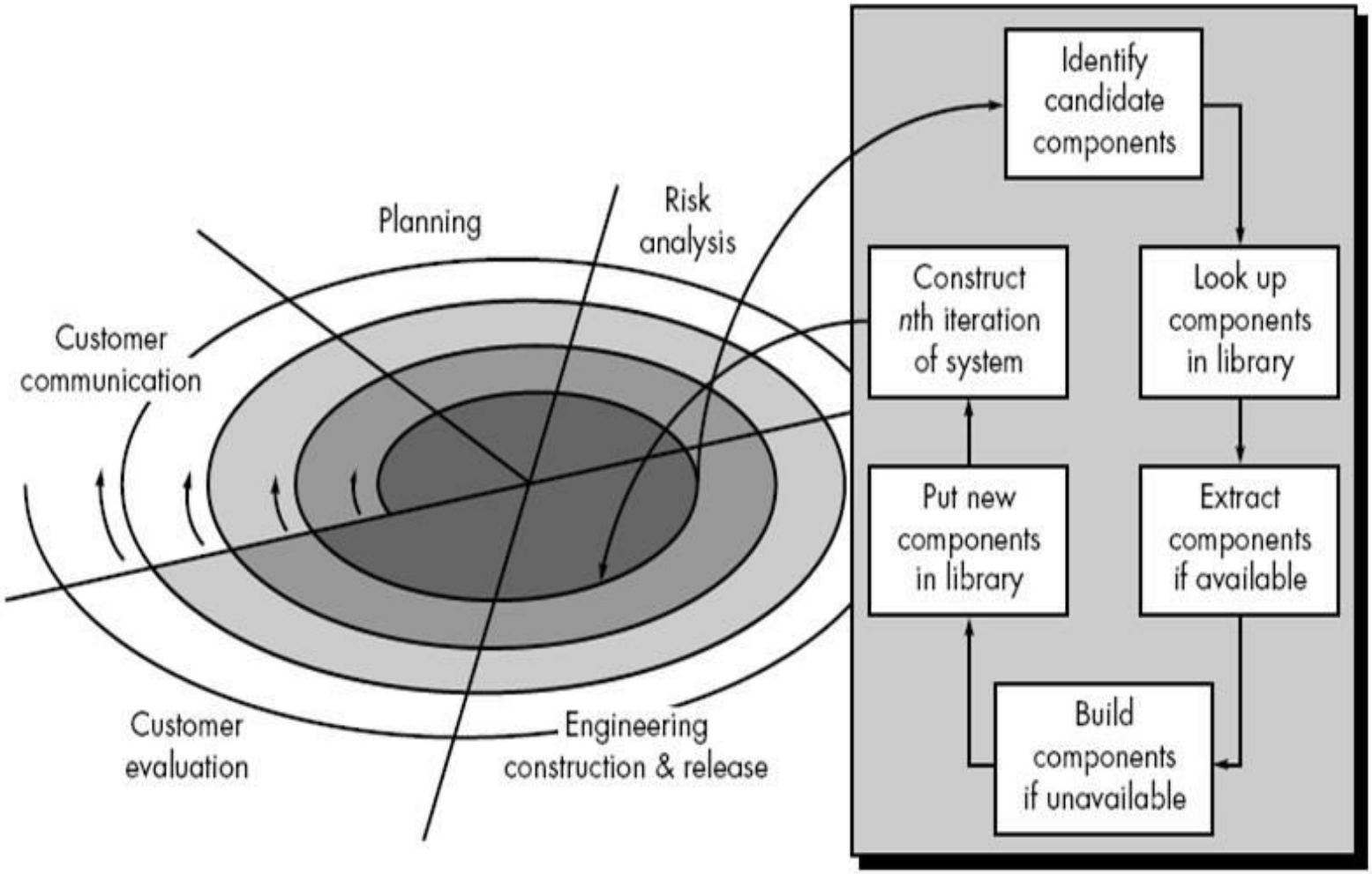
Spiral Model



Concurrent Development Model



CBD Model



The difference between procedural and object-oriented analysis….??

Object-Oriented Vs Traditional analysis

- Traditional procedural systems separate data and procedures, and model these separately
- Object orientation –views data and functions together; data abstraction is the basis
  - OO view more closely reflects the real world where humans are used to think in terms of things which possess both attributes and behaviors.
  - OO provides reuse possibility from the class hierarchy views of the system.
  - OO analysis centers on objects which combines data and methods.
  - Software extensibility is easy.

Object-orientation : Four important aspects

- Identity
- Classification
- Polymorphism or Encapsulation and
- Inheritance.

Contd..

- ***Identity***
  - data is quantized into discrete, distinguishable entities called “objects”.
  - Each object has its own inherent identity
- ***Classifications***
  - objects with the same data structure (attributes) and behavior (operations) are grouped into a class.
  - A class is an abstraction that describes properties important to an application and ignores the rest

- ***Inheritance***
  - is sharing of attributes and operations among classes based on a hierarchical relationship.
  - A superclass has general information that subclasses refine and elaborate
  - Subclasses need not repeat features of its superclass and add its own unique features
  - The ability to factor out common features of several classes into a superclass can greatly reduce repetition within designs and programs and is one of the main advantages of OO technology
- ***Polymorphism***
  - means that the same operation may behave differently for different classes.
  - Operation and method???

Importance of Objects

- The single most important principle of object oriented computing is that "objects are responsible for their own actions".

- If used properly, OOD improves the maintenance, reusability, and modifiability of the software.

The Basic Principles of Object Orientation

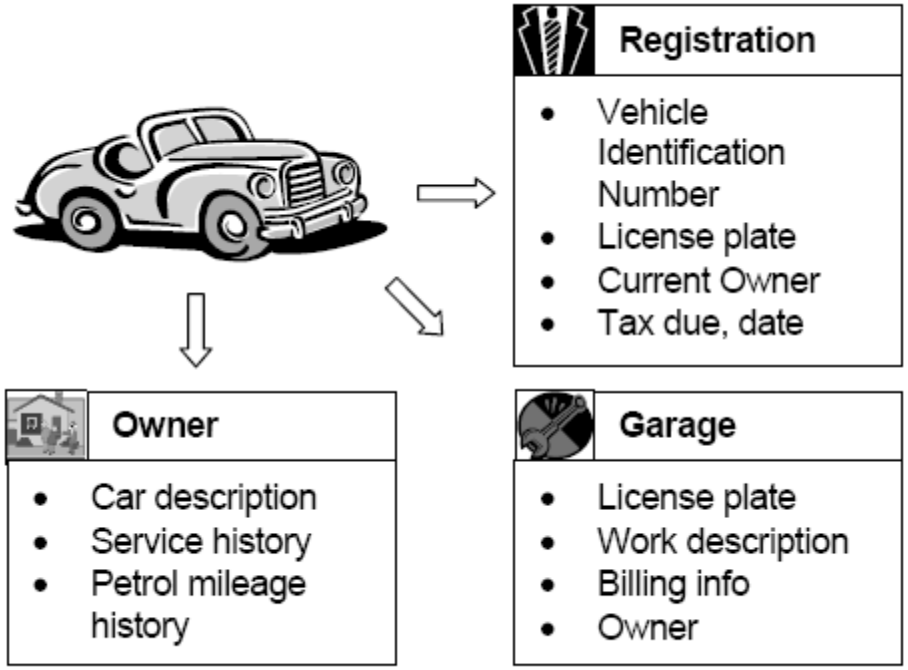
Four basic principles:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Abstraction

- An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

- Key concepts:
- Concentrating only on essential characteristics
  - Allows complexity to be more easily managed
- Abstraction is relative to the perspective of the viewer
  - Many different views of the same object are possible.


Registration
Vehicle Identification Number
License plate
Current Owner
Tax due, date
Owner
Car description
Service history
Petrol mileage history
Garage
License plate
Work description
Billing info
Owner

Encapsulation

- *Encapsulation* is the practice of including in an object everything it needs hidden from other objects. The internal state is usually not accessible by other objects.

Key concepts:

- Packaging structure and behavior together in one unit
  - Makes objects more independent
- Objects exhibit an *interface* through which others can interact with it
- Hides complexity from an object's clients

Modularity

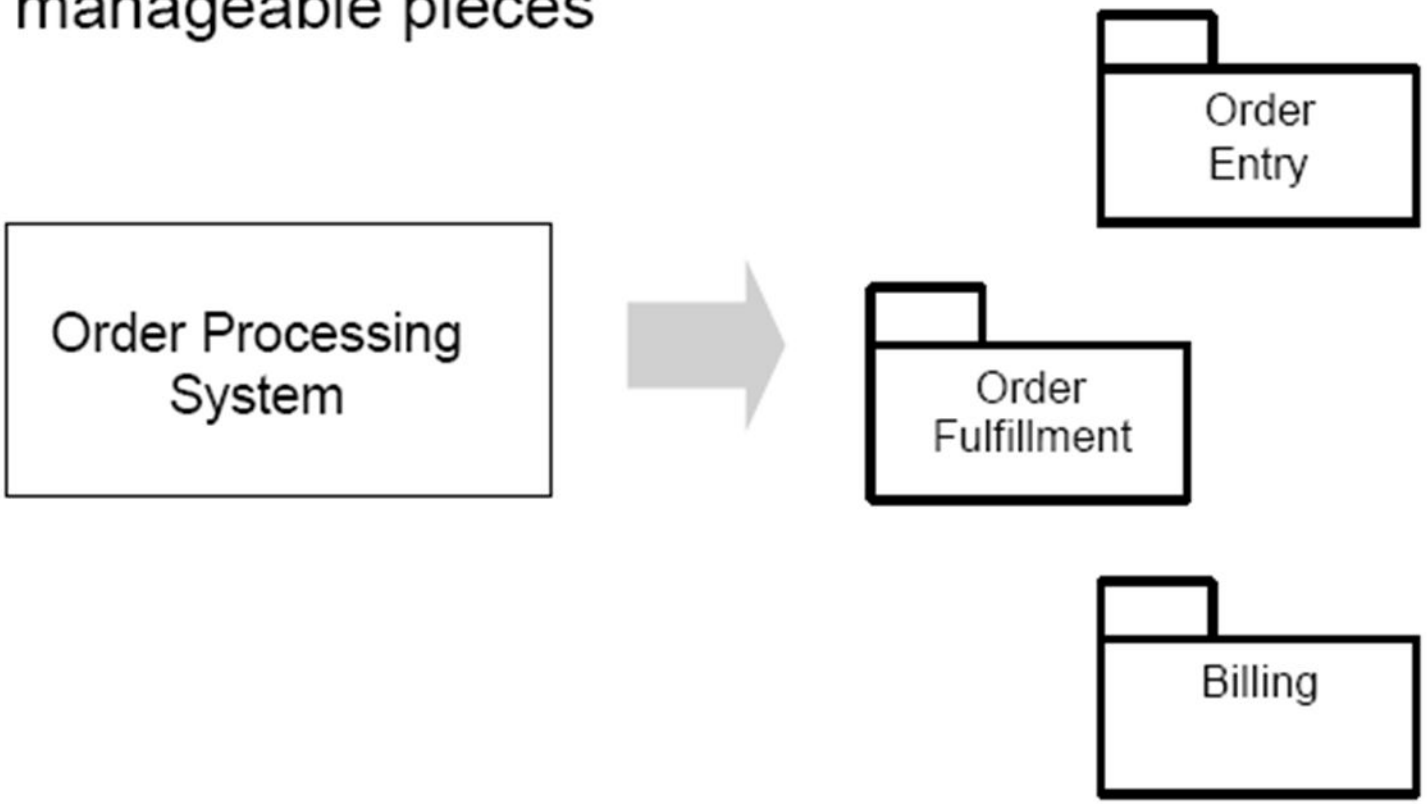
Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Module: a collection of related classes of objects

Key concepts:

- Modules are cohesive (performing a single type of tasks)
  - Makes modules more reusable
- Modules are loosely coupled (highly independent)
  - Makes modules more robust and maintainable

The breaking up of something complex into manageable pieces



Hierarchy

- Hierarchy is a ranking or ordering of abstractions

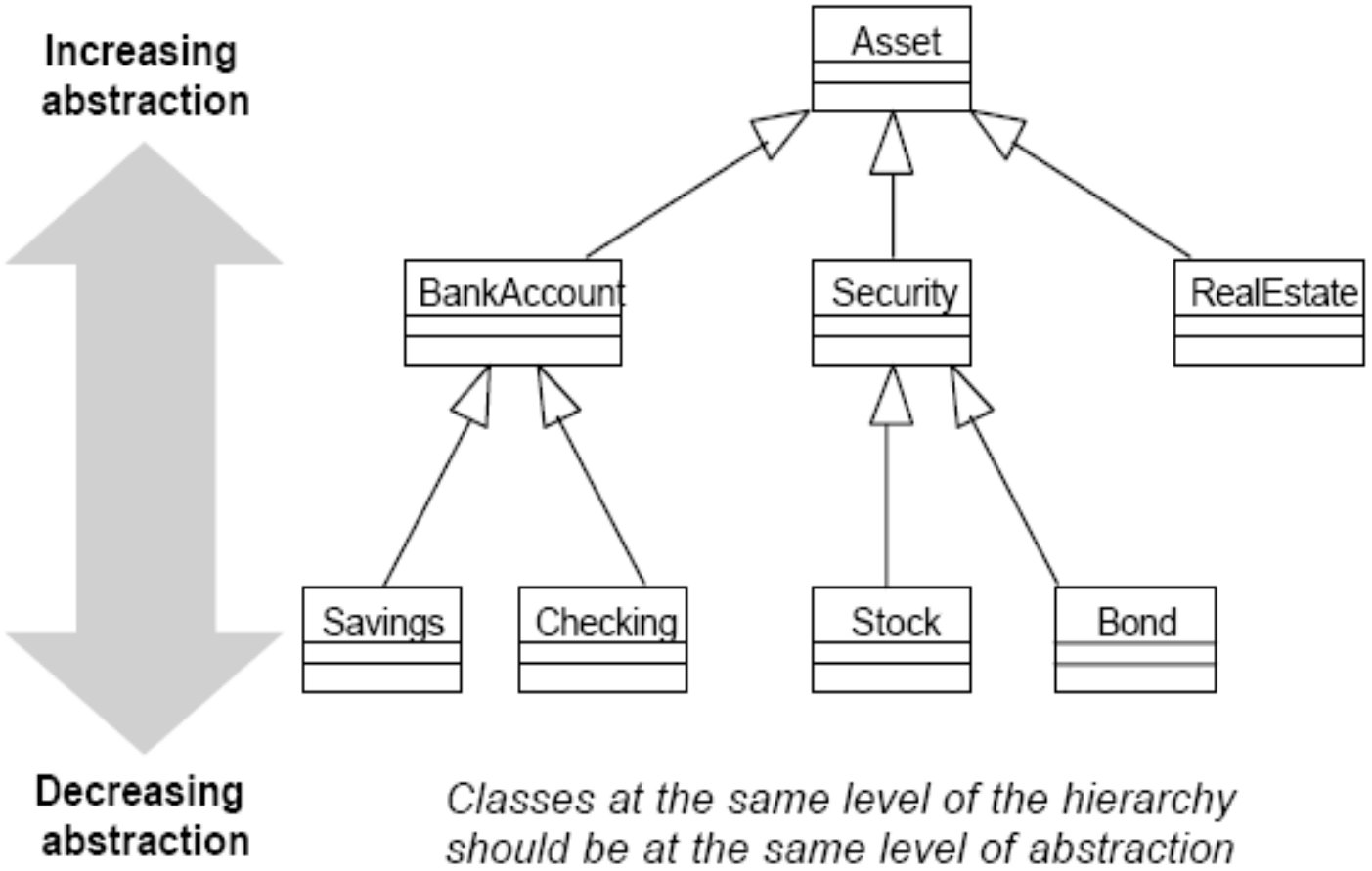
OR

- Organizing things into different levels of abstraction

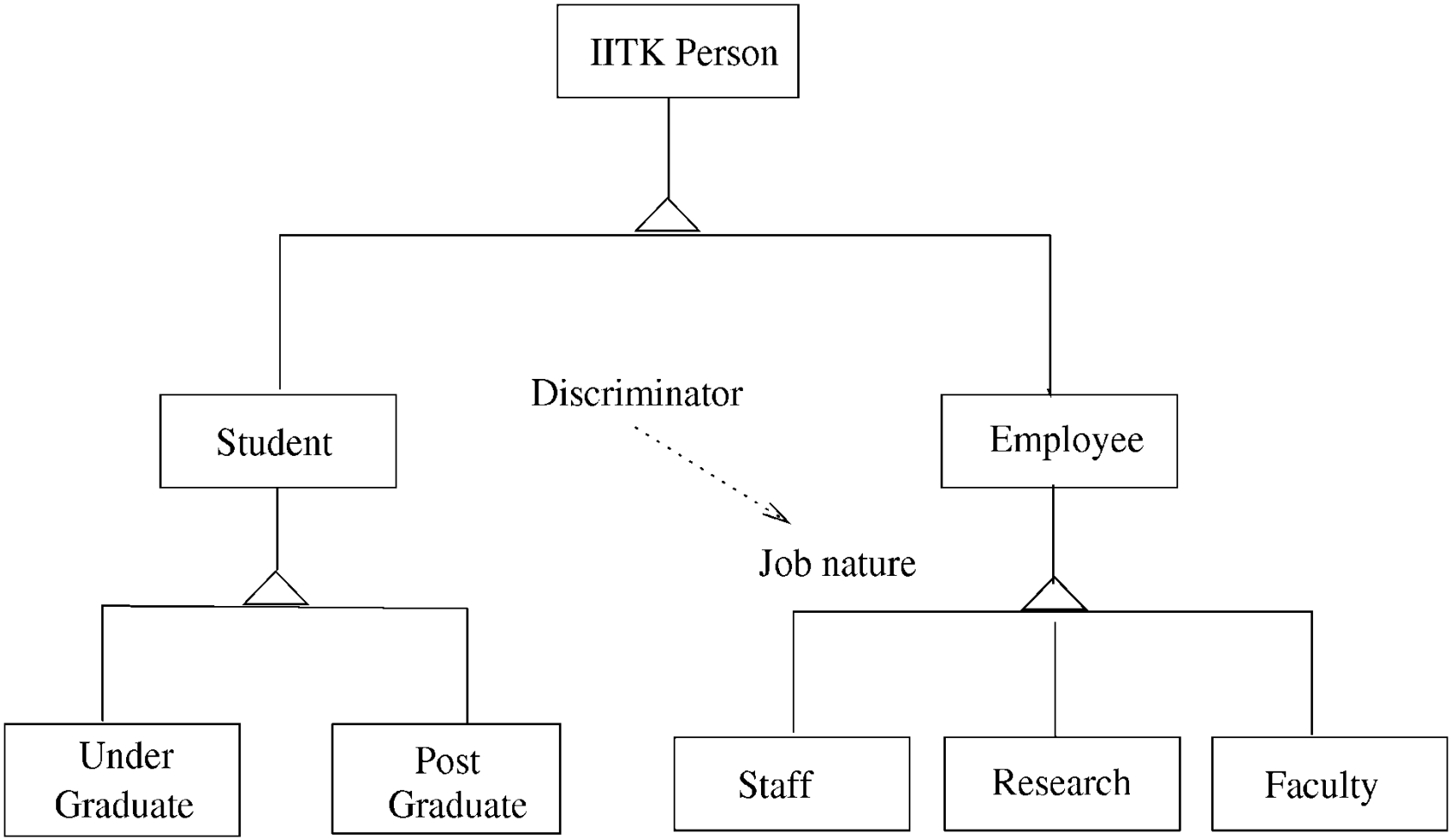
- Types of hierarchies:
- Class
- Aggregation
- Containment
- Inheritance
- Partition
- Specialization

Example: Hierarchy

- Levels of abstraction



Example: Class hierarchy



Basic Concepts of Object Orientation

- Object
- Class
- Attribute
- Operation
- Component
- Package

Object

- An object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain.

Or

- An "object" is anything to which a concept applies.
- Also, object is an instance of a class

An object can be

- A physical entity (car, fan, man, etc)
- A conceptual entity (A chemical process)
- A software entity (stack, queue, linked list)

Objects have the following qualities:

- Identity: Objects are distinguishable from one another
- Behavior: Objects can perform tasks
- State: Objects store information that can vary over time

Class

- A *class* is a blueprint that describes an object and defines attributes and operations for the object
- Classes use *abstraction* to make available only the elements essential to defining the object
- Classes use *encapsulation* to enforce an abstraction
- Simple terms: It is a collection of objects

Relations between Objects and Classes

- A class is an abstract definition of an object
  - It defines the structure and behavior of each object in the class
  - It serves as a template for creating the objects

- Objects are grouped into classes

Difference between Structures and Classes

Structures	Classes
Can define data members, properties	Can define data members, properties, and methods
Do not support inheritance	Extensible by inheritance
Default access type is public	Default access type is private
The convention is to use structure when the purpose is to group data	Use classes when we require data abstraction and, perhaps inheritance.

Attributes:

It describes the state (data) of an object

Operation:

Define its behavior

- What an object is capable of doing

Component:

- A non-trivial, nearly independent and replaceable part of a system that fulfills a clear function in the context of a well defined architecture.
- A component may be
  - A source code component
  - A run time component
  - An executable component

Package:

- A package is a general purpose mechanism for organizing elements into groups.
- A model element which can contain other model elements.

Strengths of Object Orientation:

- Facilitates architectural and code reuse
- Models more closely reflect the real world
  - More accurately describe corporate data and processes
  - Decomposed based on natural partitioning
  - Easier to understand and maintain
- Stability
  - A small change in requirements does not mean massive changes in the system under development

Analysis

- **Analysis** - investigation of the problem.
  - What does the system do?

- Requirements Analysis is discovering the requirements that a system must meet in order to be successful.

- Object Analysis is investigating the object in a domain to discover information important to meet the requirements.

Requirements Analysis

- One of the basic principles of good design is to defer decisions as long as possible.
- The more you know before you make a design decision, the more likely it will be that the decision is a good one.

- TFCL: ***Think First, Code Later!***

Design

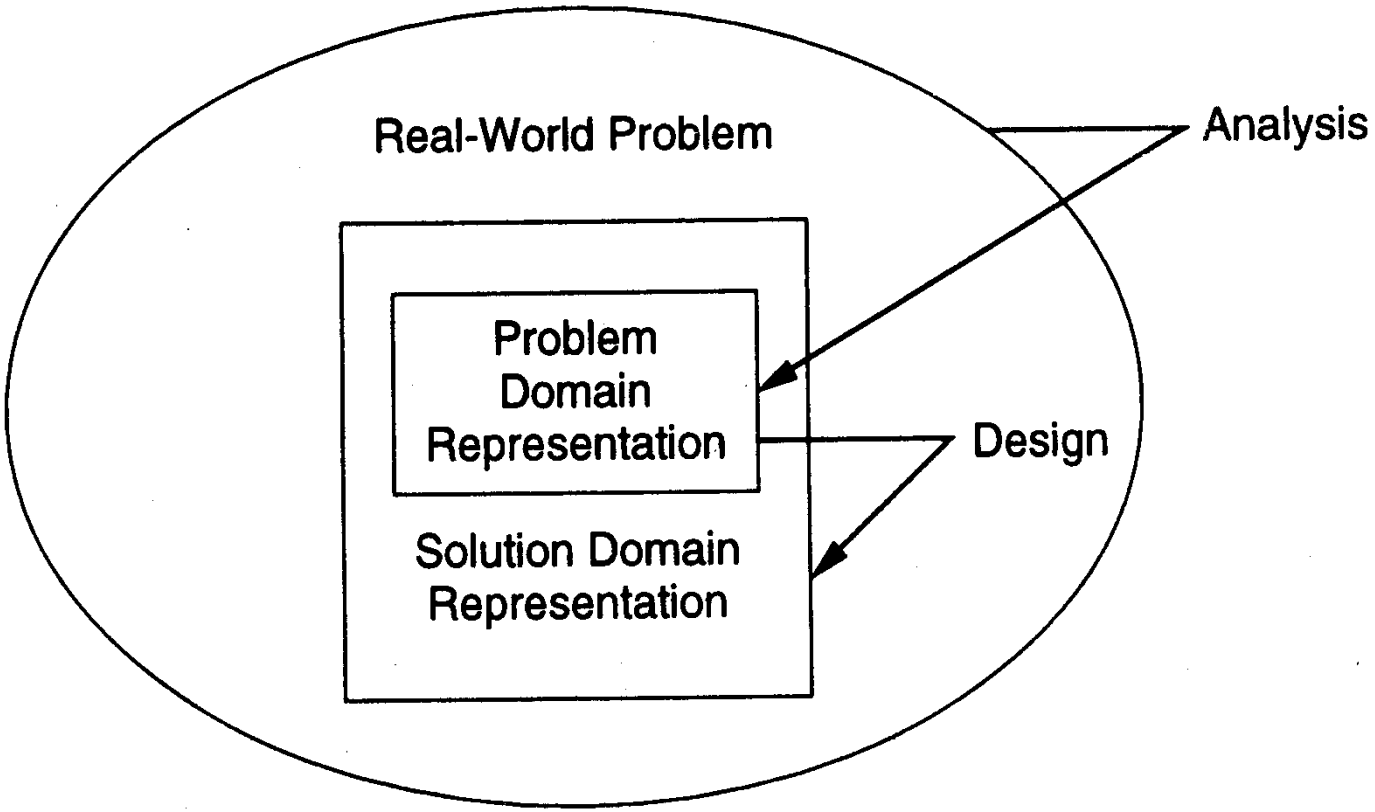
- Design emphasizes a conceptual solution that fulfills the requirements.
  - How a system is to be built?

- A design is not an implementation, although a good design can be implemented when it is complete.

- There are subsets of design, including architectural design, object design, and database design.

Analysis Vs Design

- **Analysis**
  - Focus on understanding the problem
  - (Generalized) Behavior
  - Separation of Concerns
  - System structure
  - Functional requirements
  - Some recognition for non-functional requirements
  - A small model
- **Design**
  - Focus on understanding the solution
  - Operations and Attributes
  - Performance, Efficiency…
  - Close to real code
  - Object lifecycles
  - Non-functional requirements in detail
  - A large model


Real-World Problem
Analysis
Problem
Domain
Representation
Design
Solution Domain
Representation

Goal of Analysis

- Understand the problem; try to build a visual model of what you are trying to do independent of implementation or technology concerns.

- Focus on translating the functional requirements into software concepts

Goal of Design

- **OOD enables a software engineer to**
  - Indicate the objects that are derived from each class
  - How these objects interrelate with one another

- **OOD depicts how**
  - Relationships among objects are achieved
  - Behavior is to be implemented
  - Communication among objects is to be implemented

OO analysis and design

- Essence of OO analysis –
  - consider a problem domain from the perspective of objects (real world things, concepts)

Ex: In a Library Management System, some of the concepts include *Book*, *Library*, and *Patron*.

- Essence of OO design –
  - define the solution as a collection of software objects (allocating responsibilities to objects)

Ex: In a Library Management System, a *Book* software object may have a *title* attribute and a *getChapter* method.

The End

Coupling

- Coupling is an inter-module concept, captures the strength of interconnection between modules
- More tightly coupled the modules, the more they depend on each other, more difficult to modify one
- Low coupling is desirable for making systems understandable and modifiable
- In OO, three types of coupling exists – interaction, component, and inheritance

Coupling…

- Interaction coupling occurs due to methods of a class invoking methods of other classes
  - Like calling of functions
  - Worst form if methods directly access internal parts of other methods
  - Still bad if methods directly manipulate variables of other classes
  - Passing information through temporary variables is also bad

Coupling…

- Least interaction coupling if methods communicate directly with parameters
  - With least number of parameters
  - With least amount of information being passed
  - With only data being passed
- I.e. methods should pass the least amount of data, with least number of parameters

Coupling…

- Component coupling – when a class A has variables of another class C
  - A has instance variables of C
  - A has some parameters of type C
  - A has a method with a local variable of type C
- When A is coupled with C, it is coupled with all subclasses of C as well
- Component coupling will generally imply the presence of interaction coupling also

Coupling…

- Inheritance coupling – two classes are coupled if one is a subclass of other
- Worst form – when subclass modifies a signature of a method or deletes a method
- Coupling is bad even when same signature but a changed implementation
- Least, when subclass only adds instance variables and methods but does not modify any

Cohesion

- Cohesion is an intra-module concept
- Focuses on why elements are together
  - Only elements tightly related should exist together in a module
  - This gives a module clear abstraction and makes it easier to understand
- Higher cohesion leads to lower coupling – many interacting elements are in the module
- Goal is to have higher cohesion in modules
- Three types of cohesion in OO – method, class, and inheritance

Cohesion…

- Method cohesion – why different code elements are together in a method (like cohesion in functional modules)
  - Highest form is if each method implements a clearly defined function with all elements contributing to implementing this function
  - Should be able to state what the module does by a simple statement

Cohesion…

- Class cohesion – why different attributes and methods are together in a class
  - A class should represent a single concept with all elements contributing towards it
  - Whenever multiple concepts encapsulated, cohesion is not as high
  - A symptom of multiple concepts – different groups of methods accessing different subsets of attributes

Cohesion…

- Inheritance cohesion – focuses on why classes are together in a hierarchy
  - Two reasons for subclassing
    - generalization-specialization
    - reuse
  - Cohesion is higher if the hierarchy is for providing generalization-specialization

Building Blocks of UML

What is UML?

- A standardized, general purpose modeling language
- Unified Modeling Language, which is mainly a collection of graphical notation that use to express the designs.
- The UML is language for visualizing, specifying, constructing and documenting the artifacts of software system

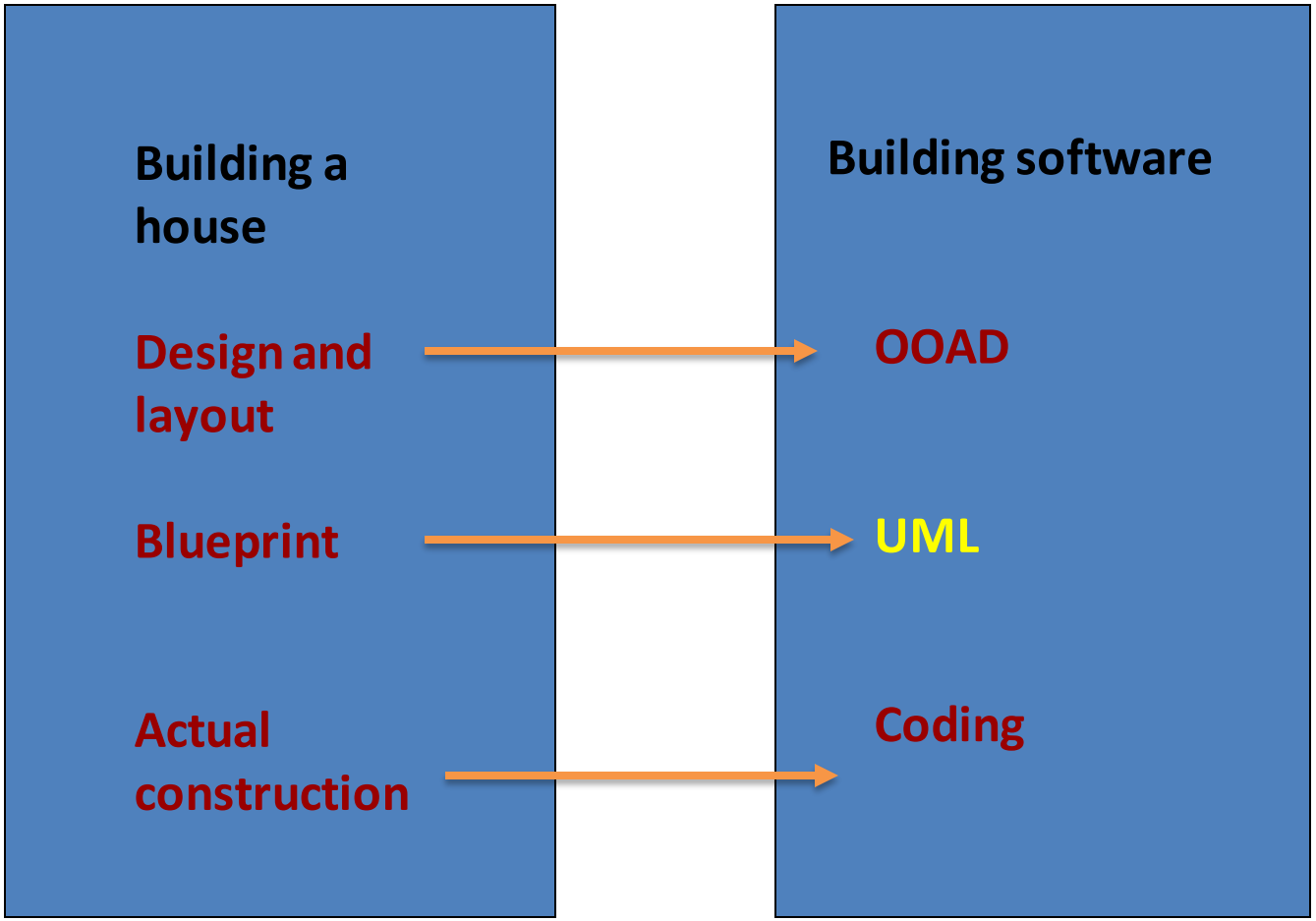
Why Modeling..?

- ✓ Models help us to **visualize** a system as it is or as we want it to be.
- ✓ Models permit us to **specify the structure or behavior** of a system.
- ✓ Models give us a **template** that guides us in constructing a system.
- ✓ Models document the **decisions** we have made.

Why Models are needed?

- Symptoms of inadequacy: the software crisis
  - scheduled time and cost exceeded
  - user expectations not met
  - poor quality
- Captures business processes
- Enhance communication and ensures the right communication
- Capability to capture the logical software architecture independent of the implementation language
- Manages the complexity
- Enables reuse of design

OOAD with UML

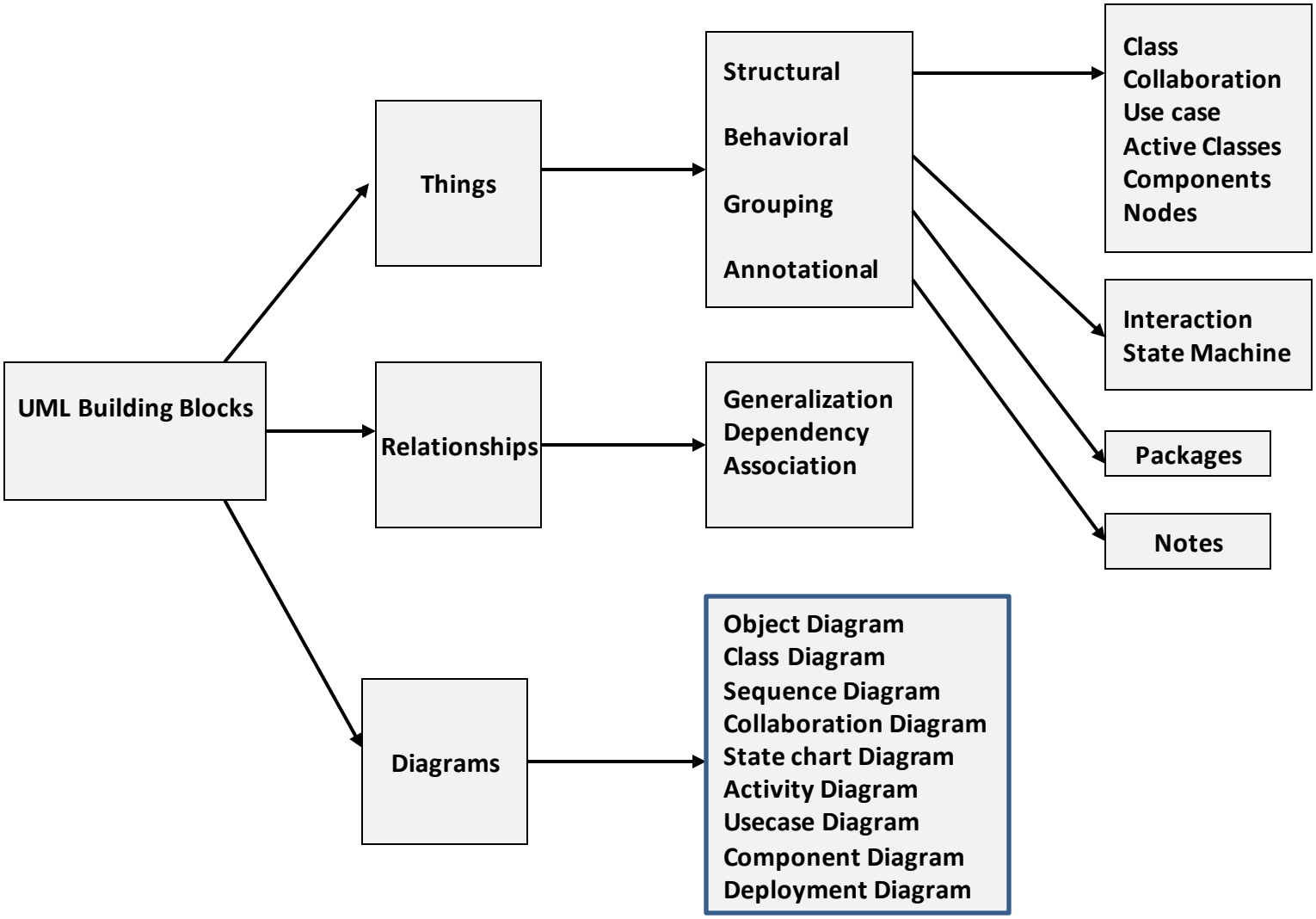


A conceptual model of the UML

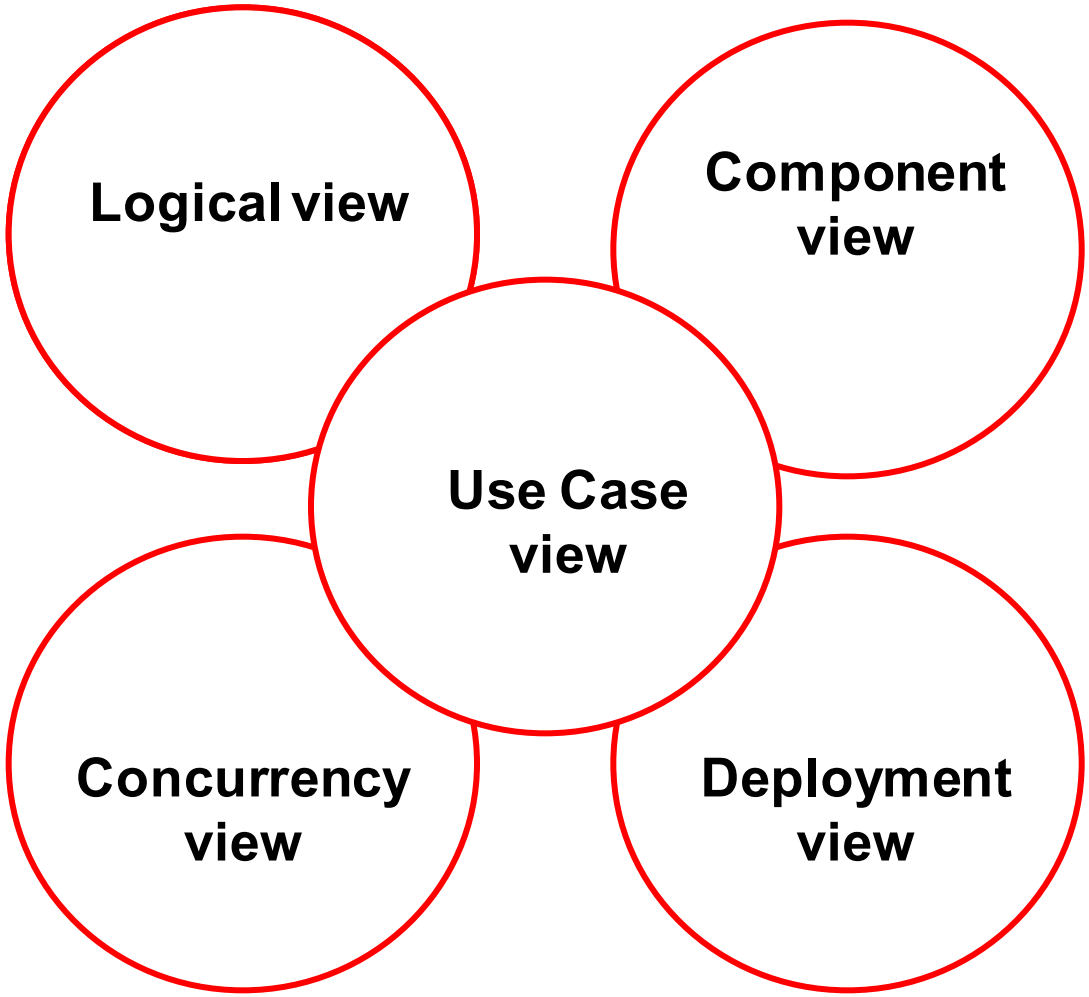
To understand the UML, you need to form a conceptual model of the language, and this requires learning three major elements:

- The UML's basic building blocks.
- The rules that dictate how these building blocks put together.
- Some mechanisms that apply throughout the UML.

Building Blocks of the UML:



Views in UML


Logical view
Component view
Use Case view
Concurrency view
Deployment view

Building Blocks of UML

- Things – abstractions that are first class citizen in a model
- Relationships - tie things together
- Diagrams - group interesting collections of things

Things in the UML

There are four kinds of things in the UML:

- ✓ Structural things
- ✓ Behavioral things
- ✓ Grouping things
- ✓ Annotational things

Structural Things

- Static part of a model, representing elements that are either conceptual or physical
  - Classes
  - Collaborations
  - Use cases
  - Active classes
  - Components
  - Nodes

- ***class***
  - a *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics
  - A class implements one or more interfaces.
- ***Collaboration***
  - defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior
  - Collaborations have structural, as well as behavioral dimensions
  - A given class may participate in several collaborations
  - therefore represent the implementation of patterns that make up a system

- ***use case*** - a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration.
- ***active class*** - a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements.
- ***component*** - a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations.
- ***node*** - a physical element that exists at run time and represents a computational resource, having at least some memory and, often, processing capability. A set of components may reside on a node and may also migrate from node to node.

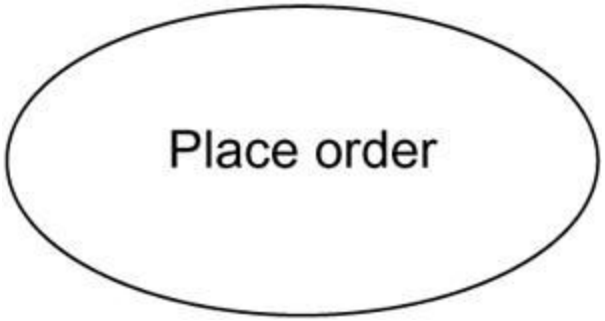


Figure 3: Use Cases

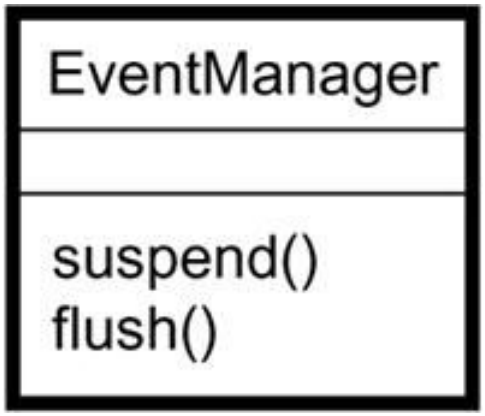


Figure 4: Active Classes

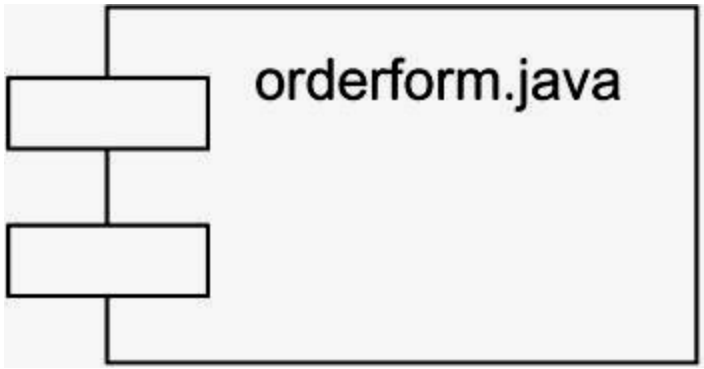


Figure 5: Components

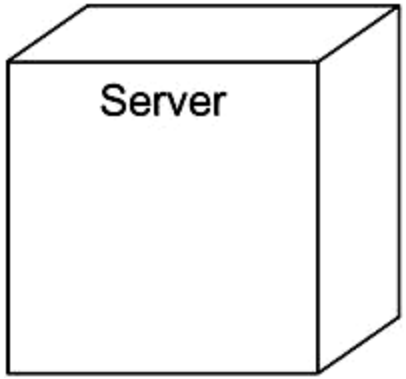


Figure 6: Nodes

Behavioral Things

- Behavioral things are the ***dynamic parts*** of UML models.
- These are the verbs of a model, representing behavior over time and space.
- **Two primary kinds** of behavioral things.
  - ***Interaction*** *(exchange of set of messages among a set of objects)(includes message, links, action sequence)*
  - ***State Machine*** *(specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events.)(includes states, transitions, events, activities)*

**** Shown in the figure 7 and 8***

Grouping Things

- Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed.
- There is one primary kind of grouping.
  - **Packages**(mechanism for organizing elements into groups) (a package is purely conceptual meaning that it exists only at development time)

    **** Shown in the figure 9***

Annotational Things

- Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model.
- There is one primary kind of annotational things.
  - **Note** (a symbol for rendering constraints and comments attached to an element or a collection of elements)

    **** Shown in the figure 10***

Figure 7: Messages

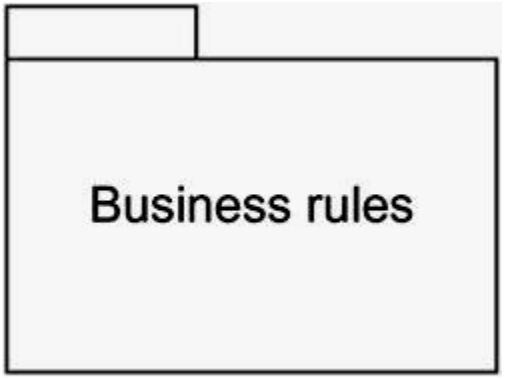


Figure 9: Packages

Figure 8: States

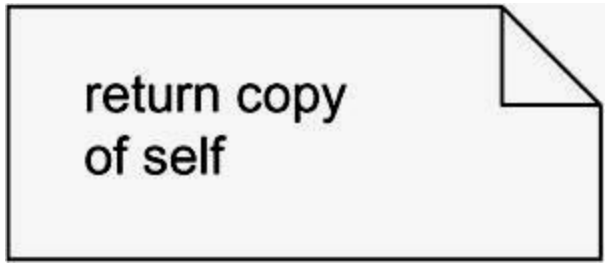


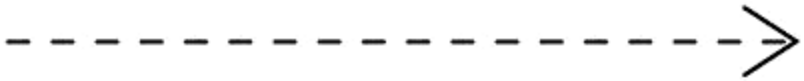
Figure 10: Notes

Relationships in the UML

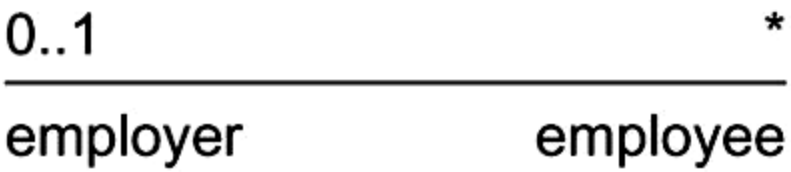
Relationships in the UML

- There are four kinds of relationships in the UML:
  1. Dependency *(a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing)*
  2. Association *(a structural relationship that describes a set of links, a link being a connection among objects)*
  3. Generalization *(a specialization / generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent))*

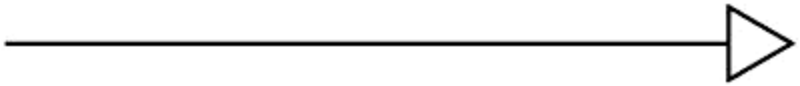
Relationships in the UML



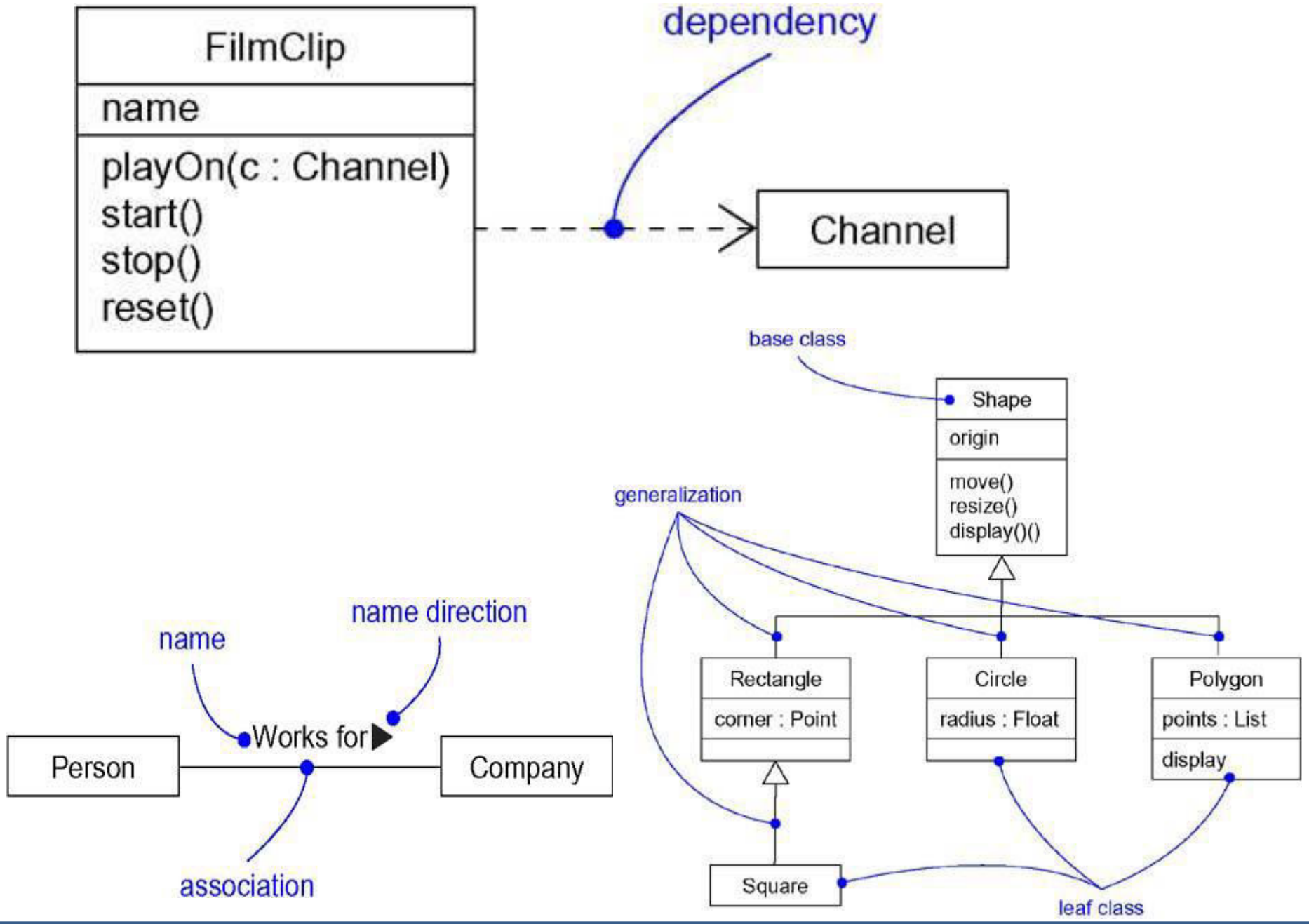
1. Dependencies: rendered as a dashed line, possibly directed, and occasionally including a label



2. Associations: rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names



3. Generalizations: rendered as a solid line with a hollow arrowhead pointing to the parent

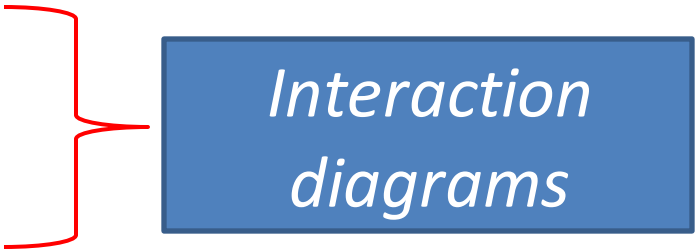

FilmClip
name
playOn(c : Channel)
start()
stop()
reset()
dependency
Channel
base class
Shape
origin
move()
resize()
display()()
generalization
Rectangle
corner : Point
Circle
radius : Float
Polygon
points : List
display
Square
leaf class
name direction
name
Works for
Person
Company
association

Diagrams in the UML

- graphical presentation of a set of elements, rendered as a connected graph of vertices (things) and arcs (relationships).
- a projection into a system
- The same element may appear in all diagrams, only a few diagrams (the most common case), or in no diagrams at all (a very rare case)

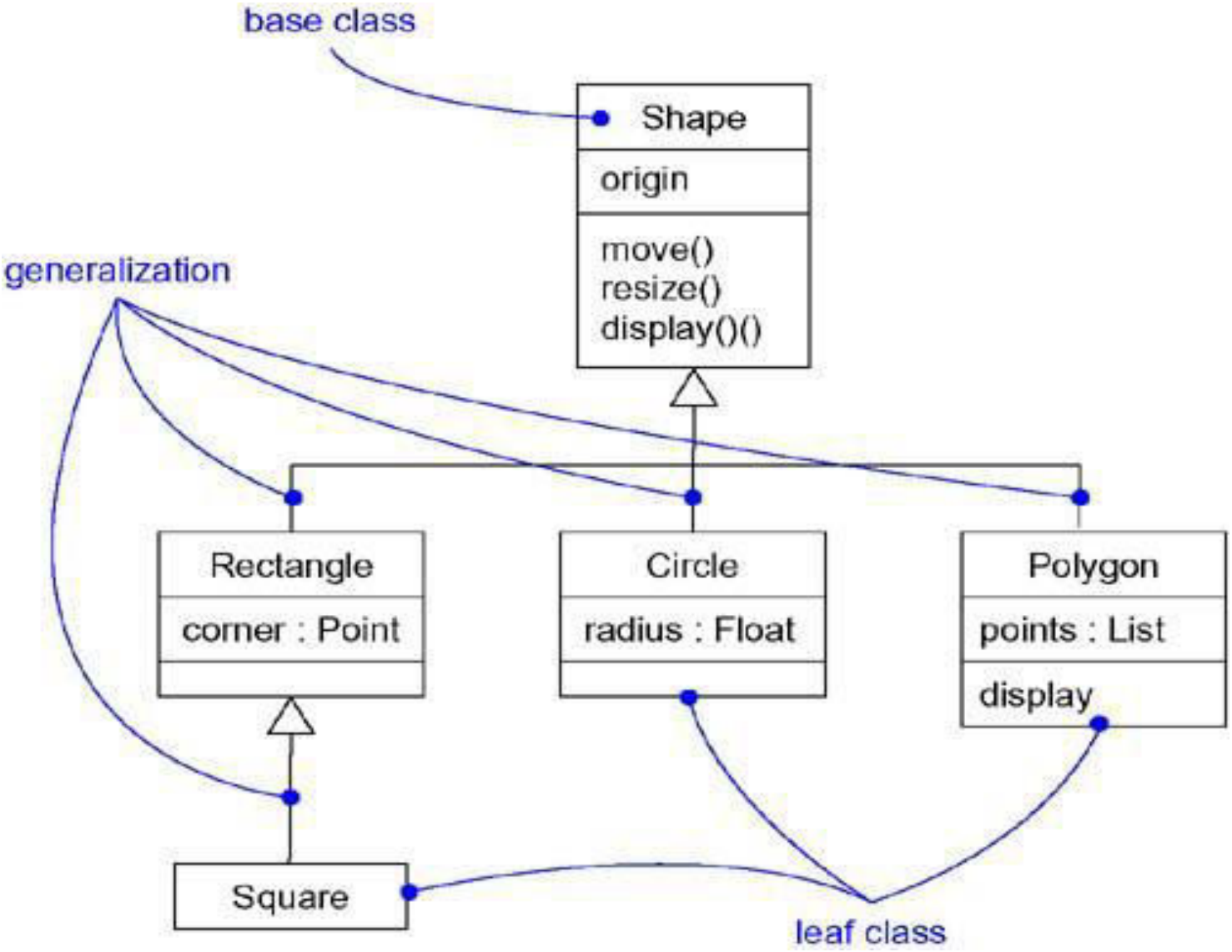
Diagrams in the UML

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram



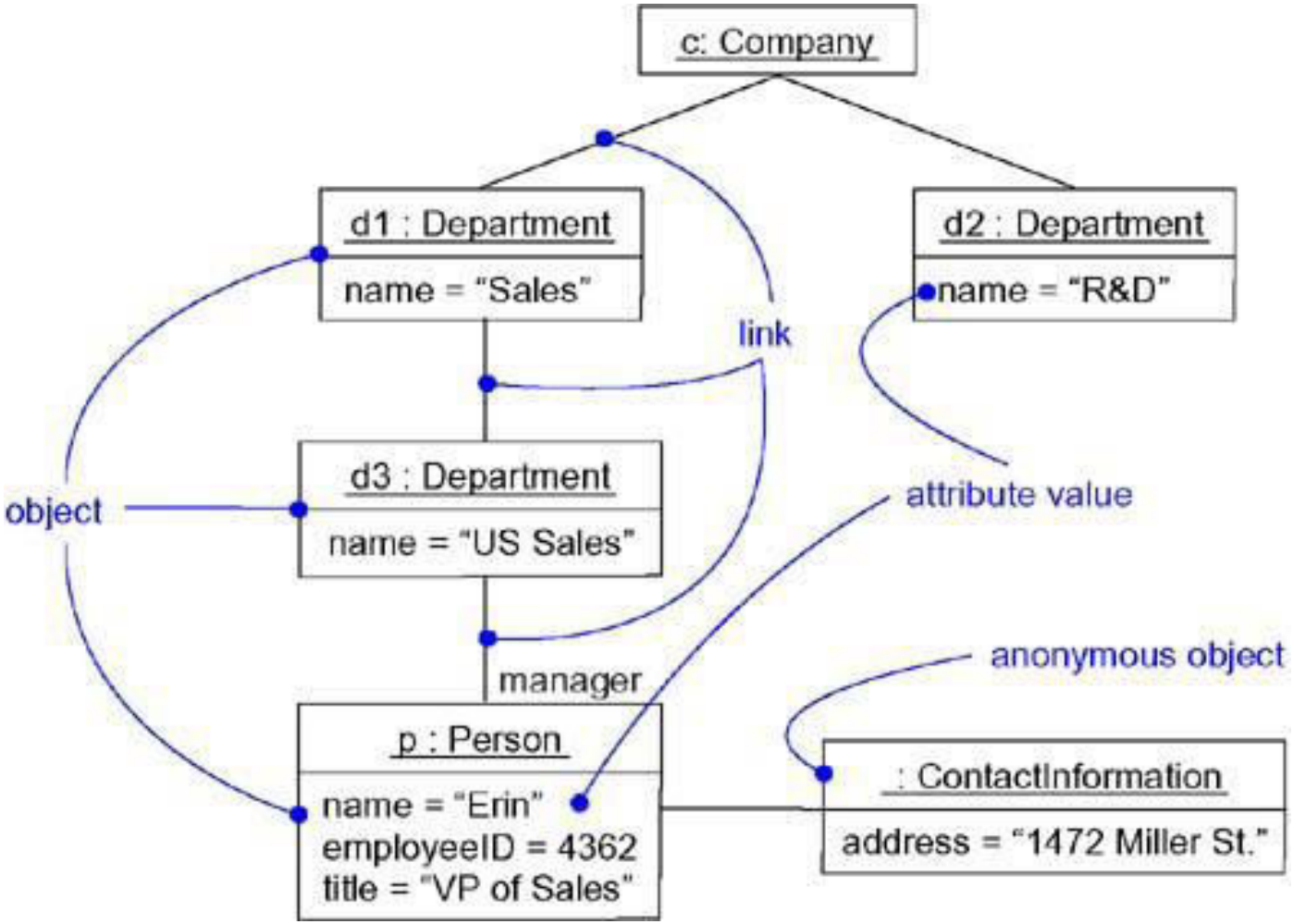
1. Class Diagram

- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- most common diagram found in modeling object-oriented systems.
- address the **static design** view of a system.
- Class diagrams that include active classes address the static process view of a system.


base class
Shape
origin
move()
resize()
display()()
generalization
Rectangle
corner : Point
Circle
radius : Float
Polygon
points : List
display
Square
leaf class

2. Object Diagram

- An object diagram shows a set of objects and their relationships.
- Object diagrams represent static snapshots of instances of the things found in class diagrams.
- address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

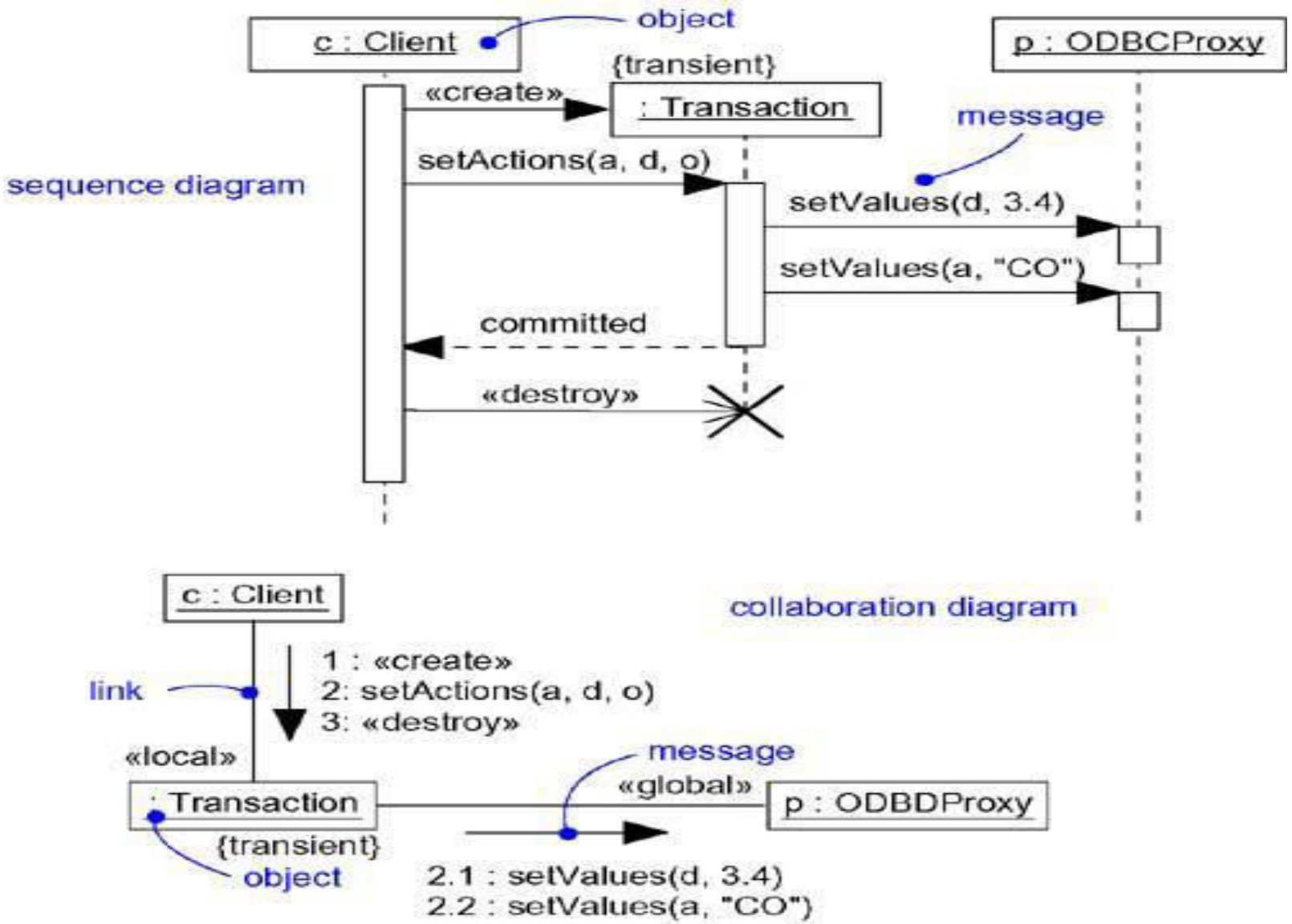

c: Company
d1 : Department
name = "Sales"
d2 : Department
name = "R&D"
link
object
d3 : Department
name = "US Sales"
attribute value
manager
anonymous object
p : Person
name = "Erin"
employeeID = 4362
title = "VP of Sales"
: ContactInformation
address = "1472 Miller St."

3. Use Case Diagram

- A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships.
- address the static use case view of a system.
- especially important in organizing and modeling the behaviors of a system.

Interaction Diagrams

- Both **sequence** diagrams and **collaboration** diagrams are kinds of interaction diagrams.
- Arc shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them.
- **Interaction diagrams address the dynamic view of a system**


c : Client
object
{transient}
«create»
: Transaction
p : ODBCProxy
message
sequence diagram
setActions(a, d, o)
setValues(d, 3.4)
setValues(a, "CO")
committed
«destroy»
c : Client
collaboration diagram
1 : «create»
2: setActions(a, d, o)
3: «destroy»
link
«local»
message
«global»
: Transaction
p : ODBDProxy
{transient}
object
2.1 : setValues(d, 3.4)
2.2 : setValues(a, "CO")

4. Sequence Diagram

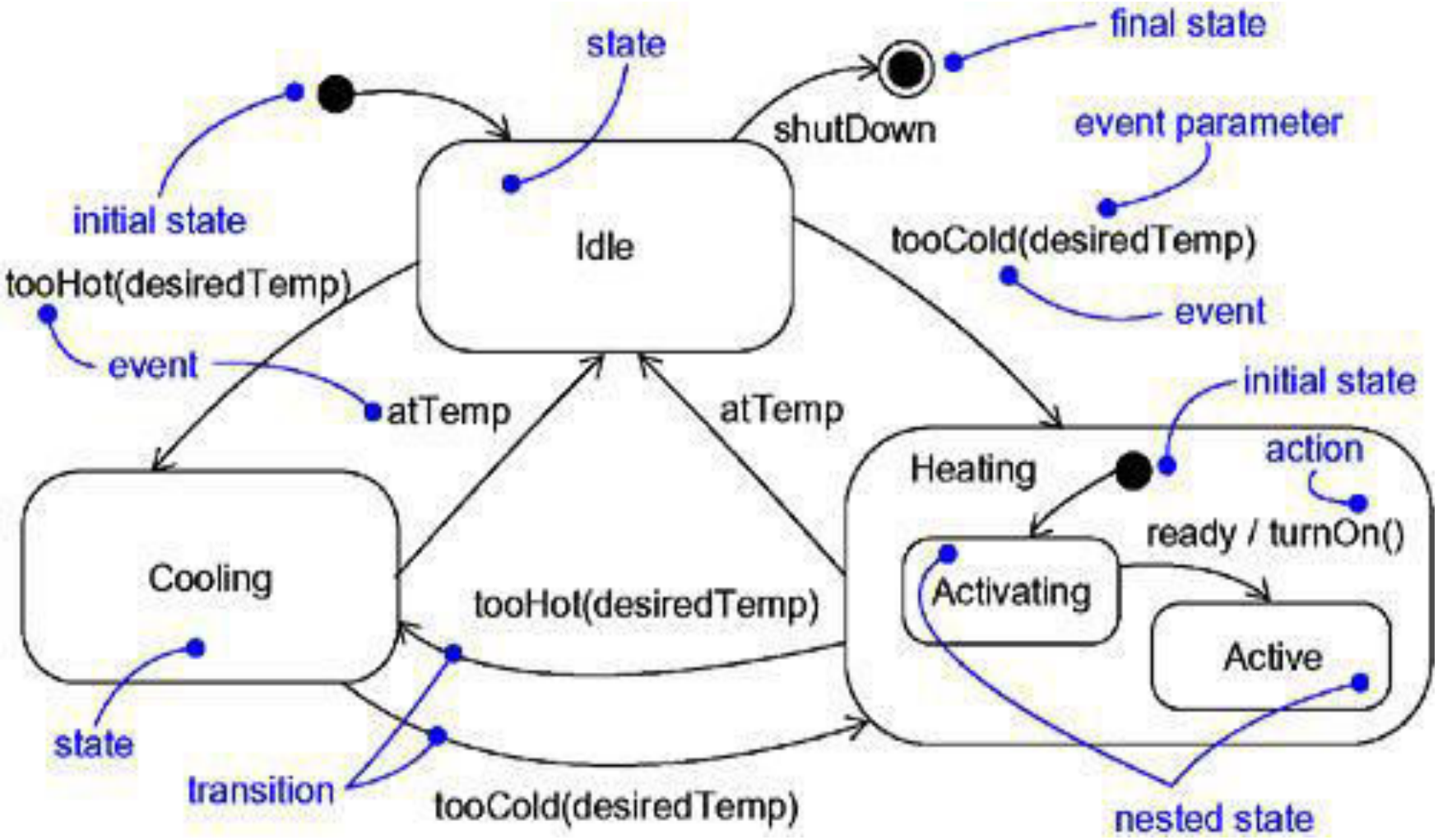
- A sequence diagram is an interaction diagram that emphasizes the **time-ordering** of messages.

5. Collaboration Diagram

- collaboration diagram is an interaction diagram that emphasizes the **structural organization** of the objects that send and receive messages.
- **Note:** Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

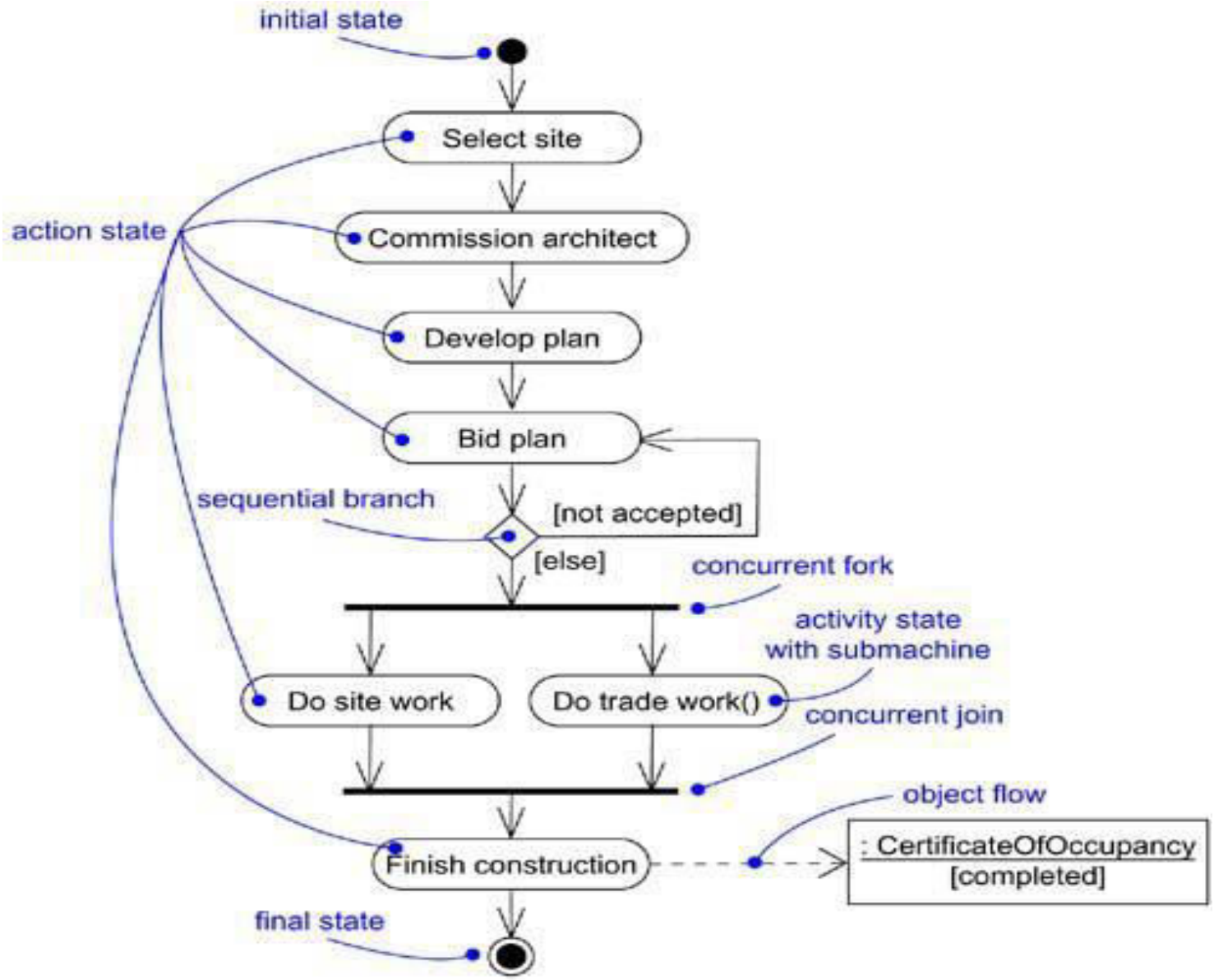
6. Statechart Diagram

- A **statechart diagram** shows a state machine, consisting of states, transitions, events and activities.
- Statechart diagrams address the dynamic view of a system.
- especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.


state
final state
shutDown
event parameter
initial state
Idle
tooCold(desiredTemp)
tooHot(desiredTemp)
event
event
atTemp
atTemp
initial state
Heating
action
ready / turnOn()
Cooling
Activating
tooHot(desiredTemp)
Active
state
transition
tooCold(desiredTemp)
nested state

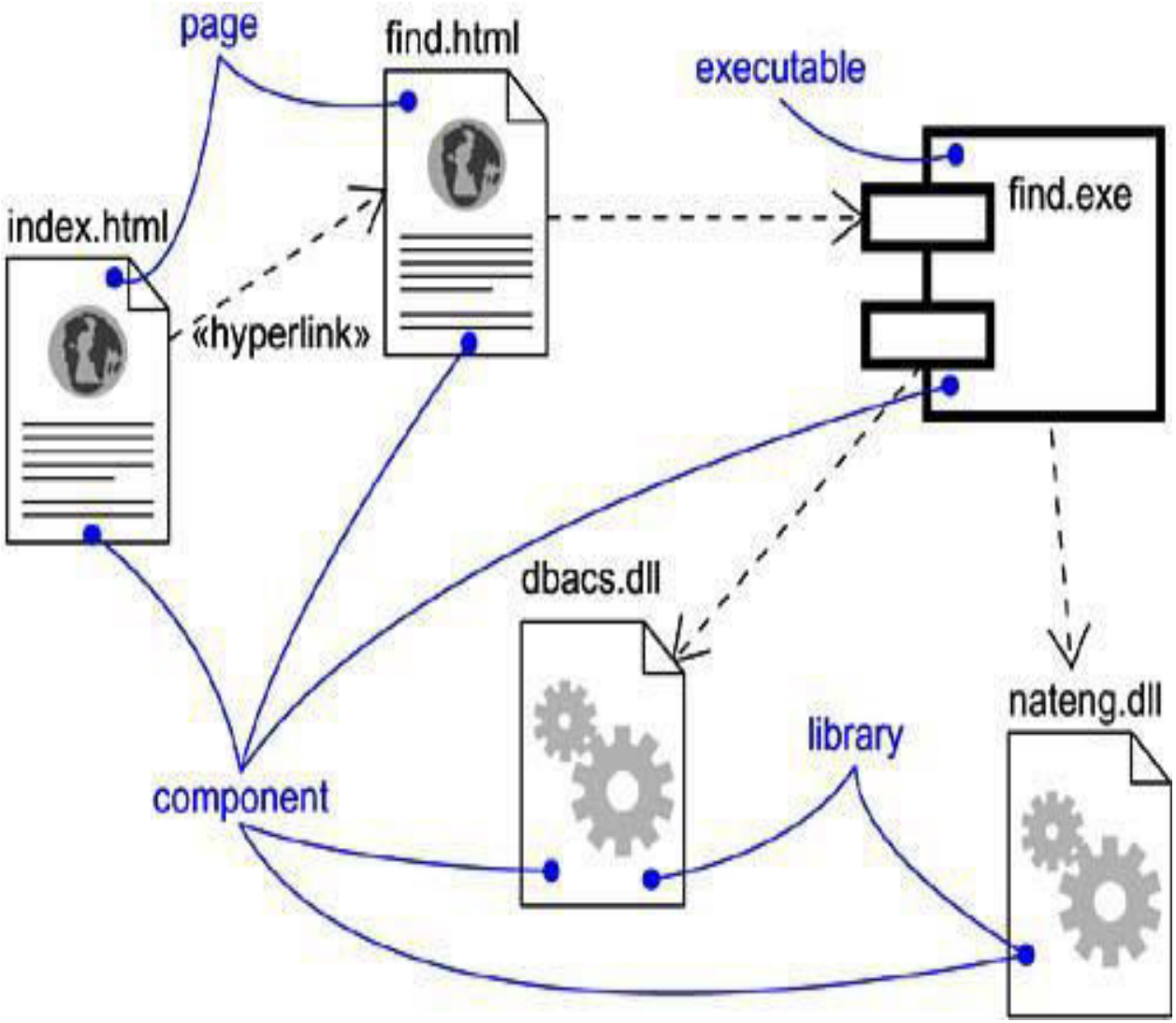
7. Activity Diagram

- An activity diagram is a special kind of a statechart diagram that shows the **flow from activity to activity** within a system.
- Activity diagrams address the dynamic view of a system.
- They are especially important in modeling the function of a system and emphasize the flow of control among objects


initial state
Select site
action state
Commission architect
Develop plan
Bid plan
sequential branch
[not accepted]
[else]
concurrent fork
activity state
with submachine
Do site work
Do trade work()
concurrent join
object flow
Finish construction
: CertificateOfOccupancy
[completed]
final state

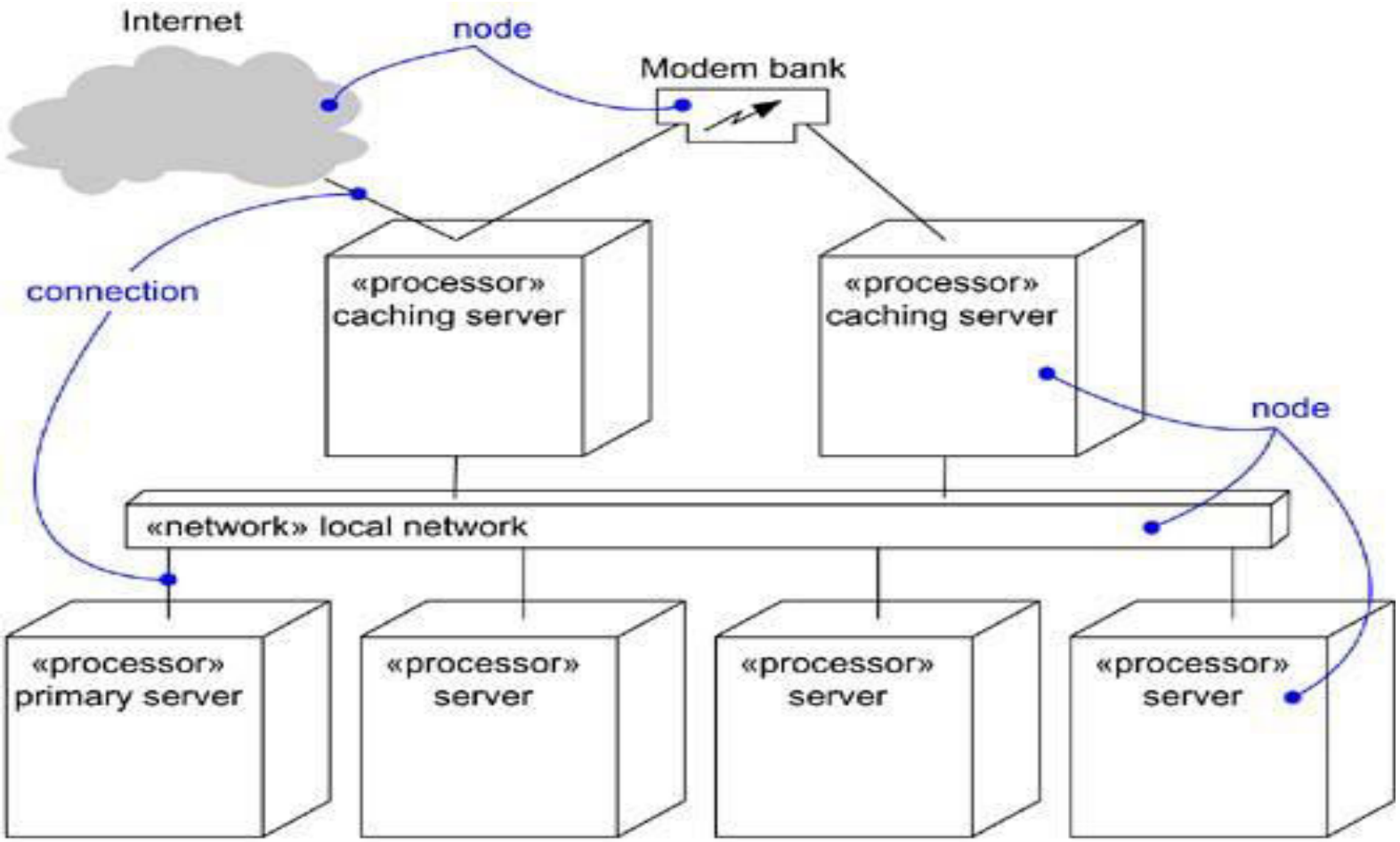
8. Component Diagram

- A component diagram shows the organizations and dependencies among a set of components.
- Component diagrams address the static implementation view of a system.
- They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.


page
find.html
executable
index.html
find.exe
«hyperlink»
dbacs.dll
nateng.dll
library
component

9. Deployment Diagram

- A deployment diagram shows the configuration of run-time processing nodes and the components that live on them.
- Deployment diagrams address the static deployment view of an architecture.
- They are related to component diagrams in that a node typically encloses one or more components.


Internet
node
Modem bank
connection
«processor»
caching server
«processor»
caching server
node
«network» local network
«processor»
primary server
«processor»
server
«processor»
server
«processor»
server

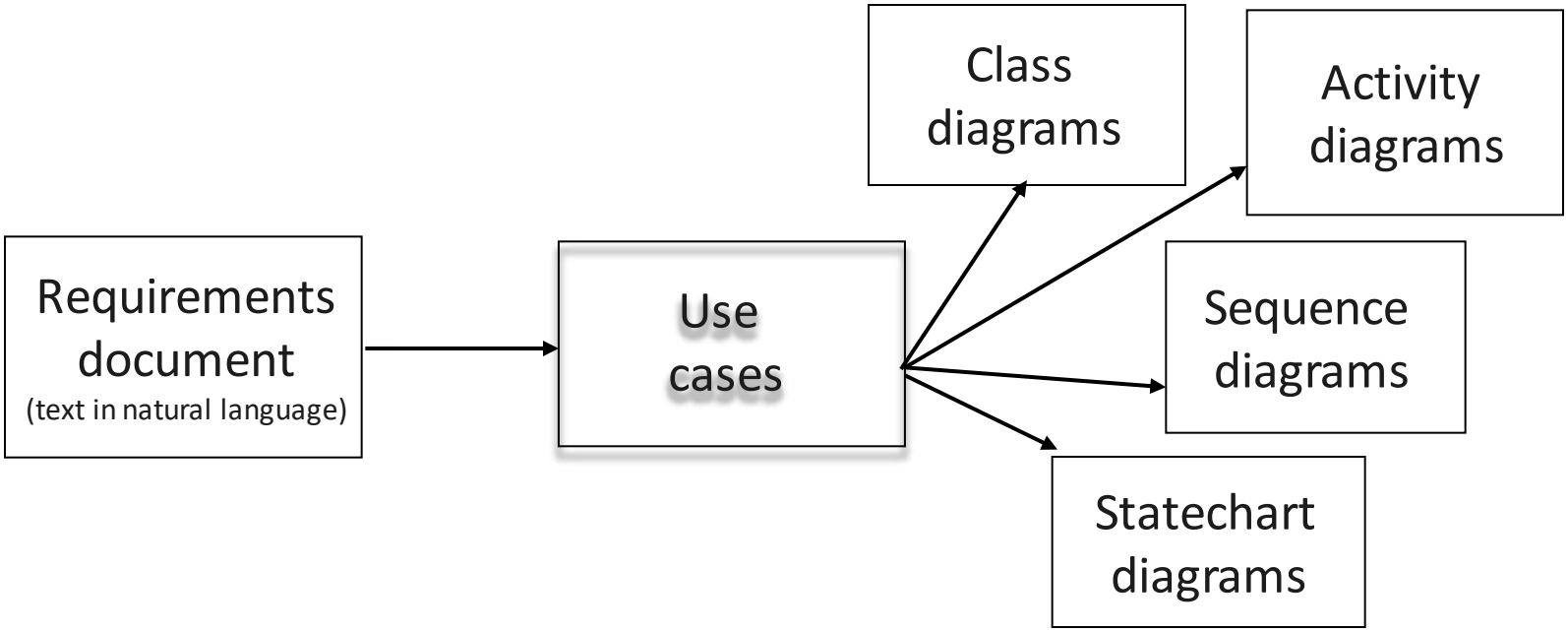
Use Case Diagram

Introduction

Use-cases are descriptions of the functionality of a system from a user perspective.

- Depict the behaviour of the system, as it appears to an outside user.
- Describe the functionality and users (actors) of the system.
- Show the relationships between the actors that use the system, the use cases (functionality) they use, and the relationship between different use cases.
- Document the scope of the system.
- Illustrate the developer's understanding of the user's requirements.

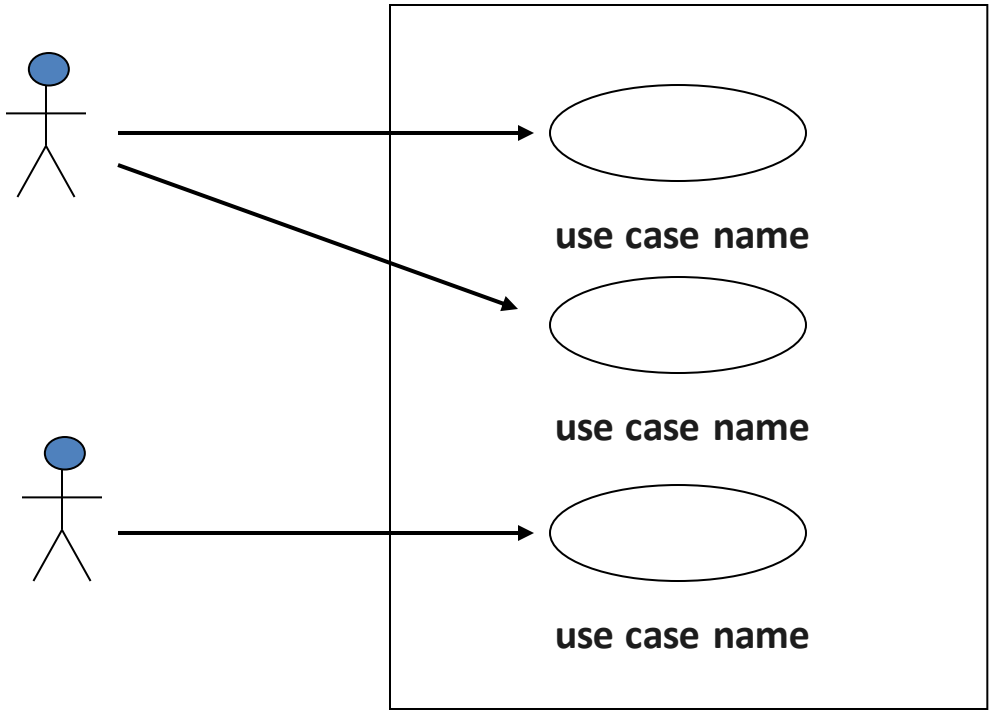
Use Case Diagram, purpose



- Use case models are developed at different levels of abstraction
  - system, system component, or a class.
- Use case modelling is an iterative and incremental process.
  - If user requirements change, the changes should be made in all the affected documents.

Use Case diagrams, basic UML notation

- Use Case: A Use Case is a description of a set of interactions between a user and the system.
- Components of use case diagram:
  - Actor
  - Use case
  - System boundary
  - Relationship

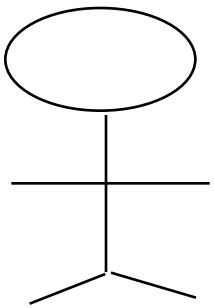


Use cases: Information captured

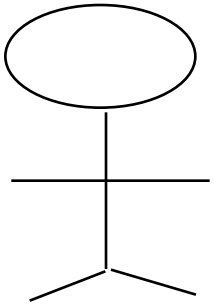
- Actors
- Relationships with other use cases
- Pre-conditions
- Details
- Post-conditions
- Exceptions
- Constraints
- Alternatives

ACTOR

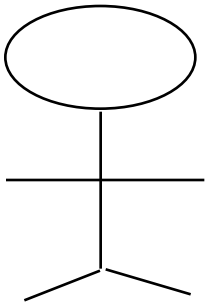
- An actor is some one or something that must interact with the system under development
- Actors can be human or automated systems.
- Actors are not part of the system.
- UML notation for actor is stickman, shown below.



Student



Faculty



Employee

ACTOR (contd..)

- It is role a user plays with respect to system.
- Actors carry out use cases and a single actor may perform more than one use cases.
- Actors are determined by observing the direct uses of the system

Primary and Secondary Actors

- Primary Actor
  - Acts on the system
  - Initiates an interaction with the system
  - Uses the system to fulfill his/her goal
  - Events Something we don't have control over
- Secondary Actor
  - Is acted on/invoked/used by the system
  - Helps the system to fulfills its goal
  - Something the system uses to get its job done

USE CASE

What is USE case?

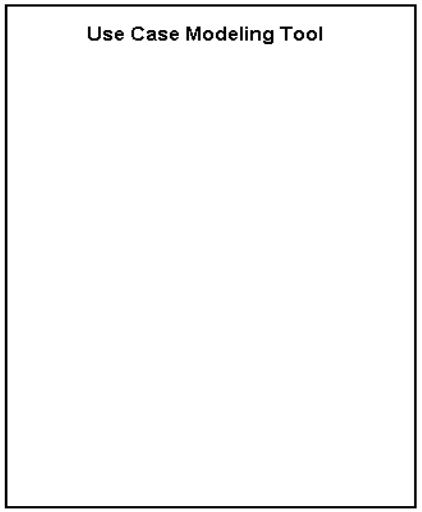
- A use case is a pattern of behavior, the system exhibits
- The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system
- The use cases are sequence of actions that the user takes on a system to get particular target
- USE CASE is dialogue between an actor and the system.
- Examples:

Contd..

- A use case typically represents a major piece of functionality that is complete from beginning to end.
- Most of the use cases are generated in initial phase, but may add some more after proceeding.
- A use case may be small or large. It captures a broad view of a primary functionality of the system in a manner that can be easily grasped by non technical user.

System Boundary

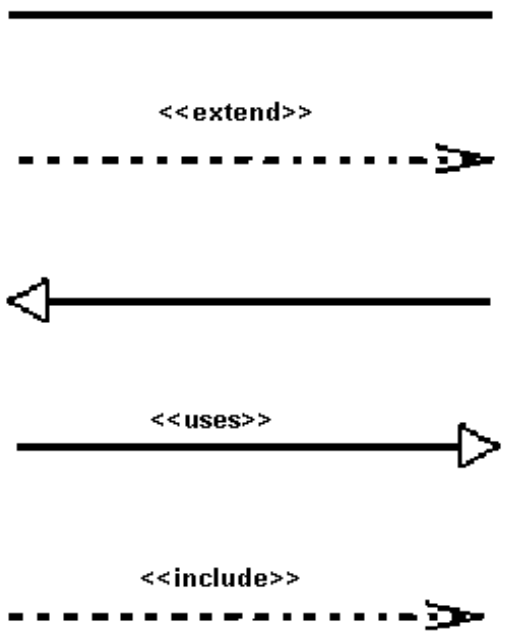
- It is shown as a rectangle.
- It helps to identify what is external versus internal, and what the responsibilities of the system are.
- The external environment is represented only by actors.



Relationship

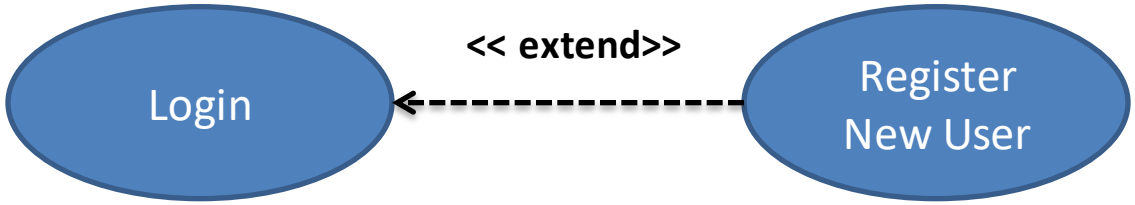
- Relationship is an association between use case and actor.
- There are several Use Case relationships:

  - Association
  - Extend
  - Generalization
  - Uses
  - Include



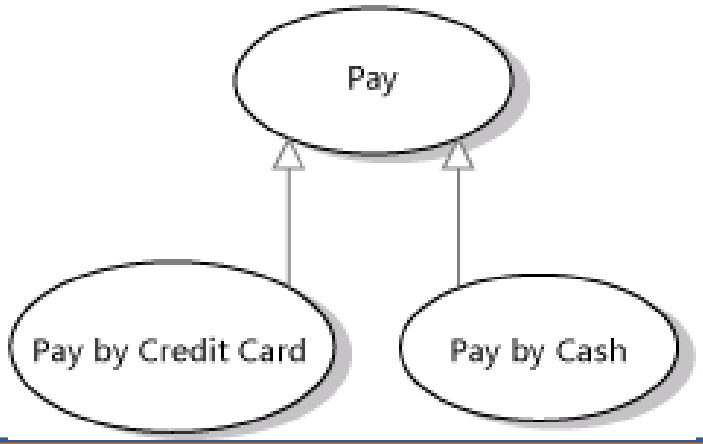
Extend Relationship

- The extended relationship is used to indicate that use case completely consists of the behavior of another use case at one or specific point
- The insertion of additional behavior into a base use case that does not know about it
- The base use case implicitly incorporates the behavior of another use case at certain points called extension points
- It is shown as a dotted line with an arrow point and labeled <<extend>>



Generalization

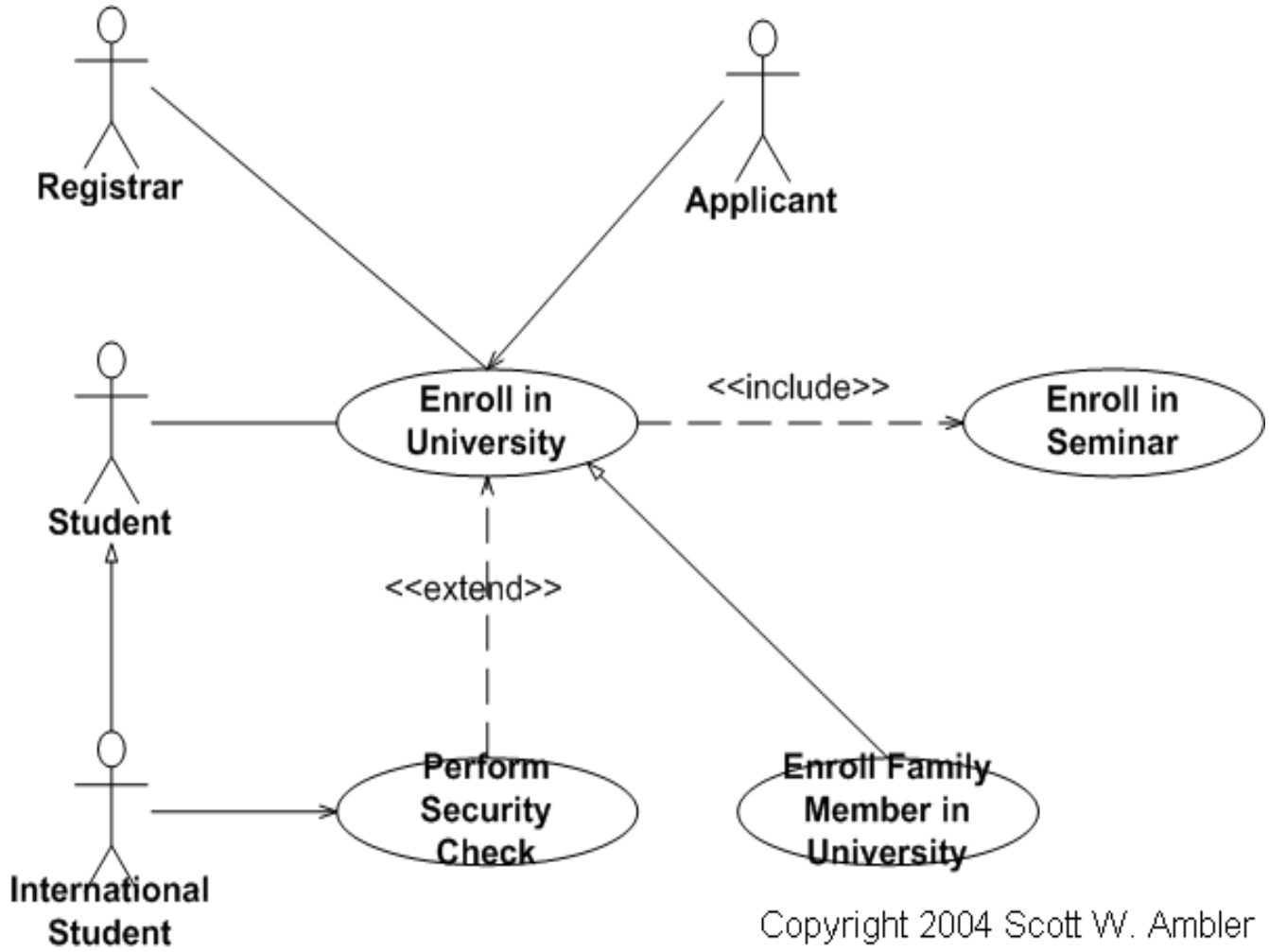
- Generalization is a relationship between a general use case and a more specific use case that inherits and extends features to it
- use cases that are specialized versions of other use cases
- It is shown as a solid line with a hollow arrow point



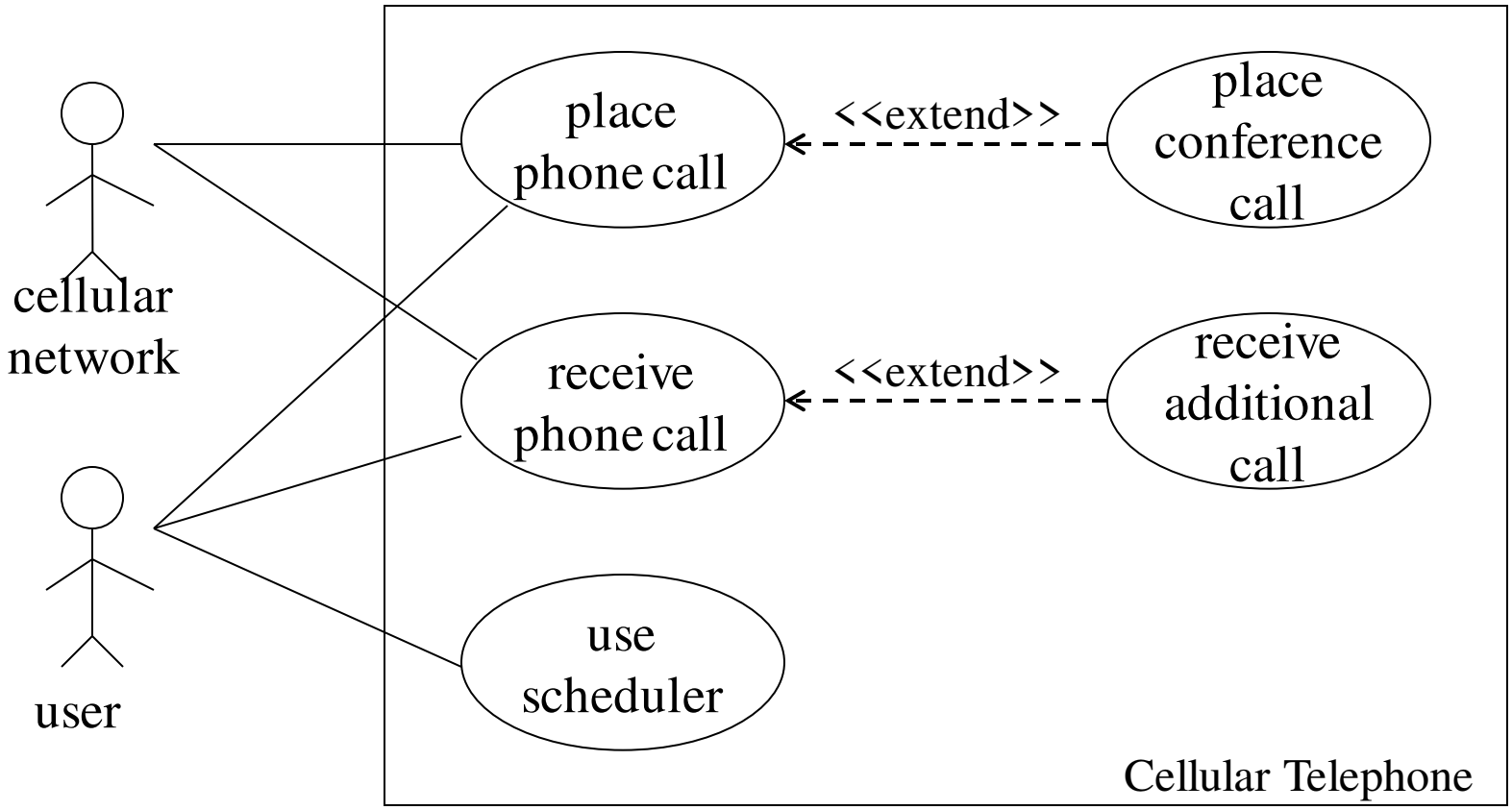
Include Relationship

- The insertion of additional behavior into a base use case that explicitly describes the insertion
- The base use case explicitly incorporates the behavior of another use case at a location specified in the base.
- They are shown as a dotted line with an open arrow and the key word <<include>>

Extend vs Include



Example


cellular network
user
place phone call
receive phone call
use scheduler
<<extend>>
<<extend>>
place conference call
receive additional call
Cellular Telephone

Use Case Description

Each use case may include all or part of the following:

- Title or Reference Name - meaningful name of the UC
- Author/Date - the author and creation date
- Modification/Date - last modification and its date
- Purpose - specifies the goal to be achieved
- Overview - short description of the processes
- Cross References - requirements references
- Actors - agents participating
- Pre Conditions - must be true to allow execution
- Post Conditions - will be set when completes normally
- Normal flow of events - regular flow of activities
- Alternative flow of events - other flow of activities
- Exceptional flow of events - unusual situations
- Implementation issues - foreseen implementation problems

Example- Money Withdraw

- Use Case: Withdraw Money
- Author: PKD
- Date: 11-09-2013
- Purpose: To withdraw some cash from user's bank account
- Overview: *The use case starts when the customer inserts his card into the system. The system requests the user PIN. The system validates the PIN. If the validation succeeded, the customer can choose the withdraw operation else alternative 1 – validation failure is executed. The customer enters the amount of cash to withdraw. The system checks the amount of cash in the user account, its credit limit. If the withdraw amount in the range between the current amount + credit limit the system dispense the cash and prints a withdraw receipt, else alternative 2 – amount exceeded is executed.*
- Cross References: R1.1, R1.2, R7

- Actors: Customer
- Pre Condition:
  - The ATM must be in a state ready to accept transactions
  - The ATM must have at least some cash on hand that it can dispense
  - The ATM must have enough paper to print a receipt for at least one transaction
- Post Condition:
  - The current amount of cash in the user account is the amount before the withdraw minus the withdraw amount
  - A receipt was printed on the withdraw amount
  - The withdraw transaction was audit in the System log file

Typical course of events

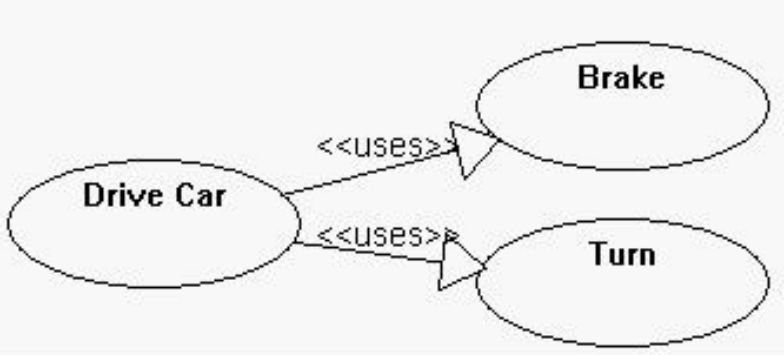
Actor Actions	System Actions
1. Begins when a Customer arrives at ATM	
2. Customer inserts a Credit card into ATM	3. System verifies the customer ID and status
5. Customer chooses "Withdraw" operation	4. System asks for an operation type
7. Customer enters the cash amount	6. System asks for the withdraw amount
	8. System checks if withdraw amount is legal
	9. System dispenses the cash
	10. System deduces the withdraw amount from account
	11. System prints a receipt
13. Customer takes the cash and the receipt	12. System ejects the cash card

- Alternative flow of events:
  - Step 3: Customer authorization failed. Display an error message, cancel the transaction and eject the card.
  - Step 8: Customer has insufficient funds in its account. Display an error message, and go to step 6.
  - Step 8: Customer exceeds its legal amount. Display an error message, and go to step 6.
- Exceptional flow of events:
  - Power failure in the process of the transaction before step 9, cancel the transaction and eject the card

- One method to identify use cases is actor-based:
  - Identify the actors related to a system or organization.
  - For each actor, identify the processes they initiate or participate in.
- A second method to identify use cases is event-based:
  - Identify the external events that a system must respond to.
  - Relate the events to actors and use cases.
- The following questions may be used to help identify the use cases for a system:
  - What are tasks of each actor ?
  - Will any actor create, store, change, remove, or read information in the system ?
  - What use cases will create, store, change, remove, or read this information ?
  - Will any actor need to inform the system about sudden, external changes ?
  - Does any actor need to be informed about certain occurrences in the system ?
  - Can all functional requirements be performed by the use cases ?

Uses Relationship

- When a use case uses another process, the relationship can be shown with the uses relationship
- This is shown as a solid line with a hollow arrow point and the <<uses>> keyword



Object & Class Diagram

- Objects:
  - a concept, abstraction, or thing with identity that has meaning for an application.
  - often appear as proper nouns or specific references in problem descriptions
  - All objects have identity and are distinguishable
  - the choice of objects depends on judgment and the nature of problem
- Classes:
  - Describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships and semantics.
  - Often appear as common nouns and noun phrases in problem description.
  - Objects in a class have the same attributes and forms of behavior
    - Difference in attribute values and specific relationship to other objects
  - The choice of classes depends on the nature and scope of an application and is matter of judgment
  - The interpretation of semantics depends on the purpose of each application and is a matter of judgment.

Class Diagram

- The class model captures the static view of an system
  - by characterizing the objects in the system
  - the relationship between the objects
  - and the attributes and operations for each class of objects

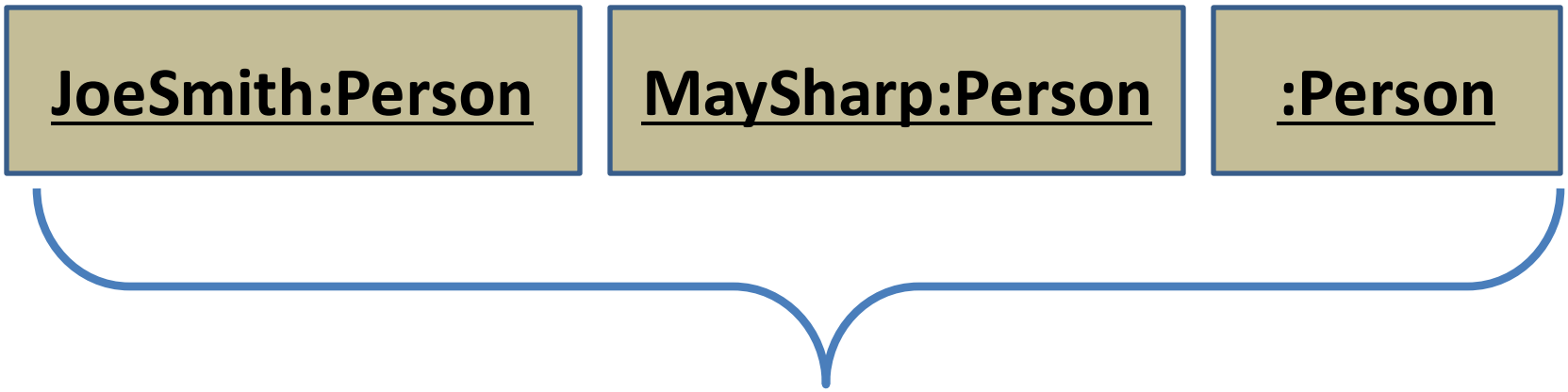
Class Diagram

- The class diagram is the main building block of object oriented modeling.
- used for
  - general conceptual modeling of the systematics of the application
  - detailed modeling translating the models into programming code.
- Class diagrams can also be used for data modeling

Class Diagram

- describes the attributes and operations of a class and also the constraints imposed on the system.
- widely used in the modeling of object oriented systems
  - can be ***mapped directly with object oriented languages***
  - ***widely used*** at the time of construction.

Object Diagram



Objects

An object diagram shows individual objects and their relationships

- Helpful for documenting test cases

A class diagram corresponds to an infinite set of object diagrams.

Class and Objects

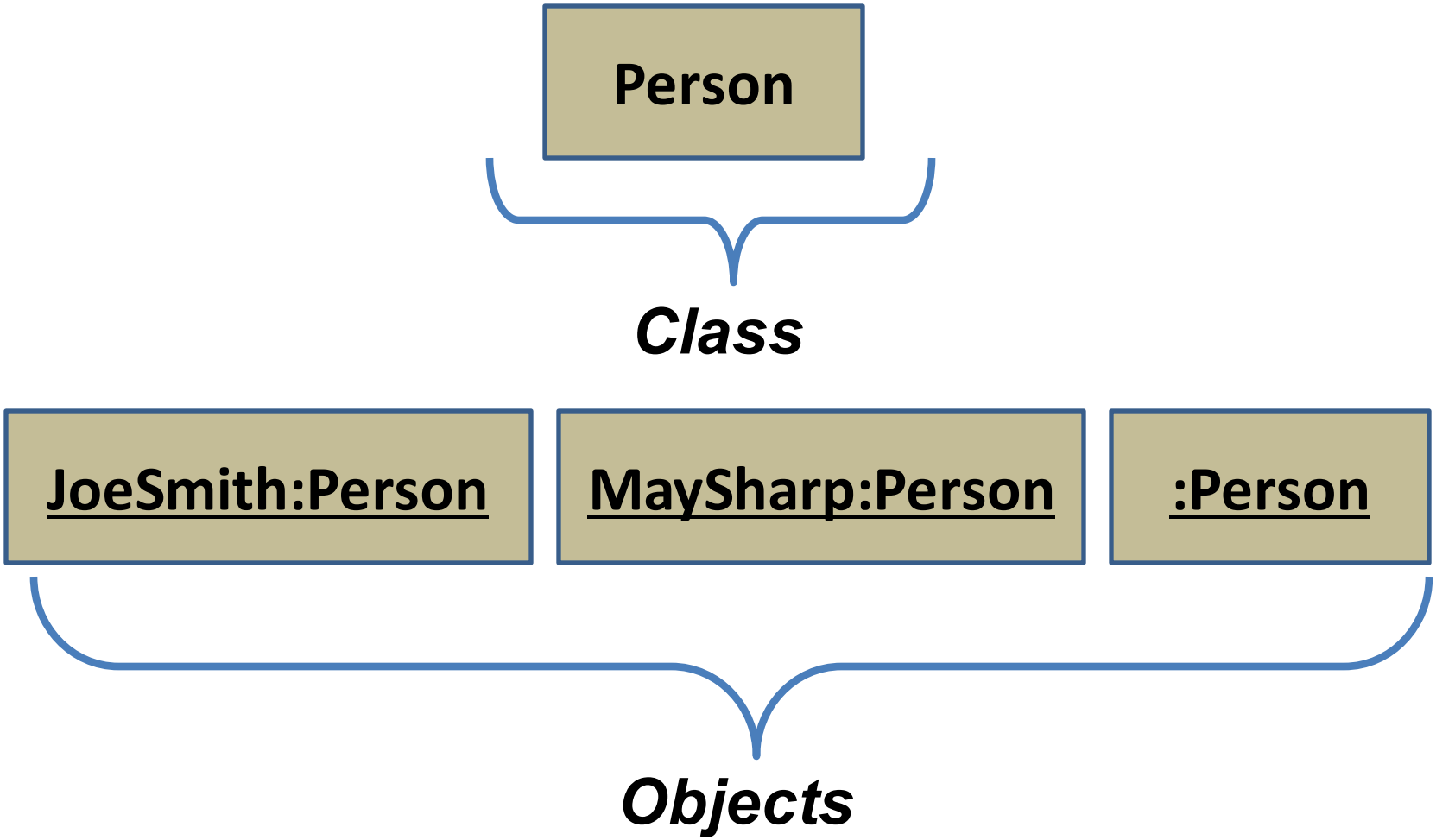
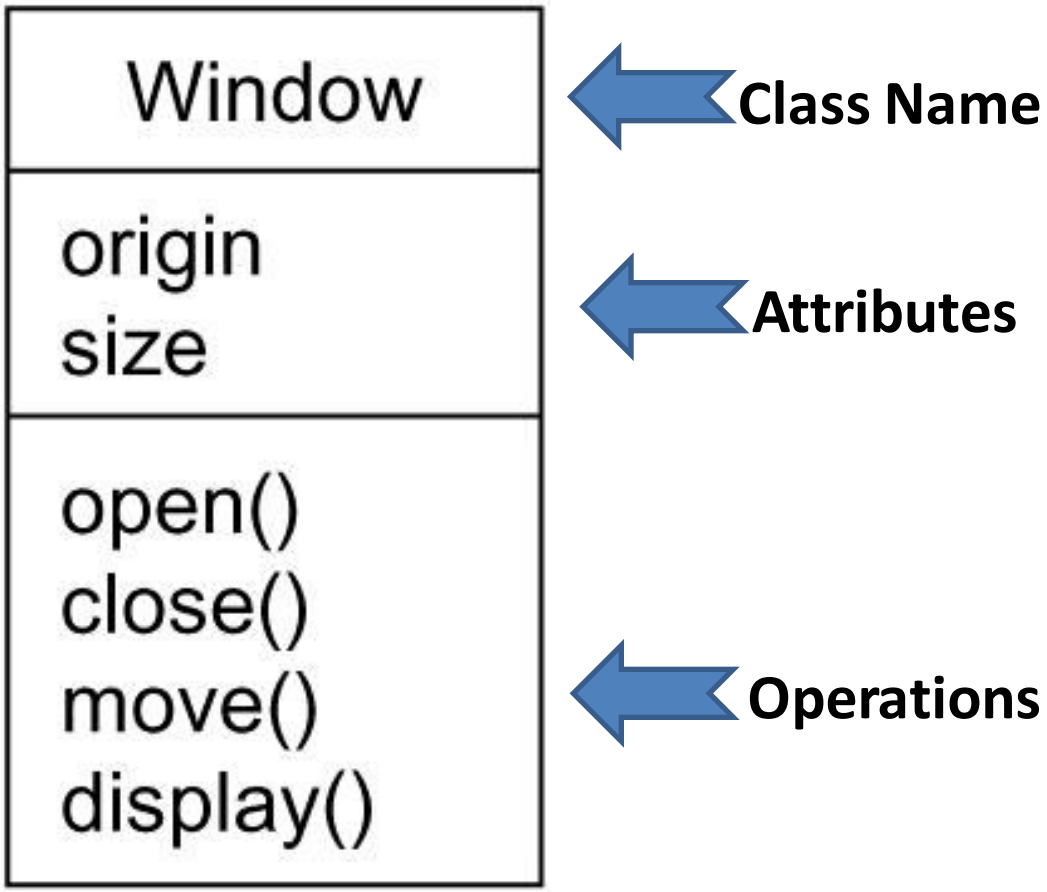


Fig: Objects and Classes are the focus of class modeling

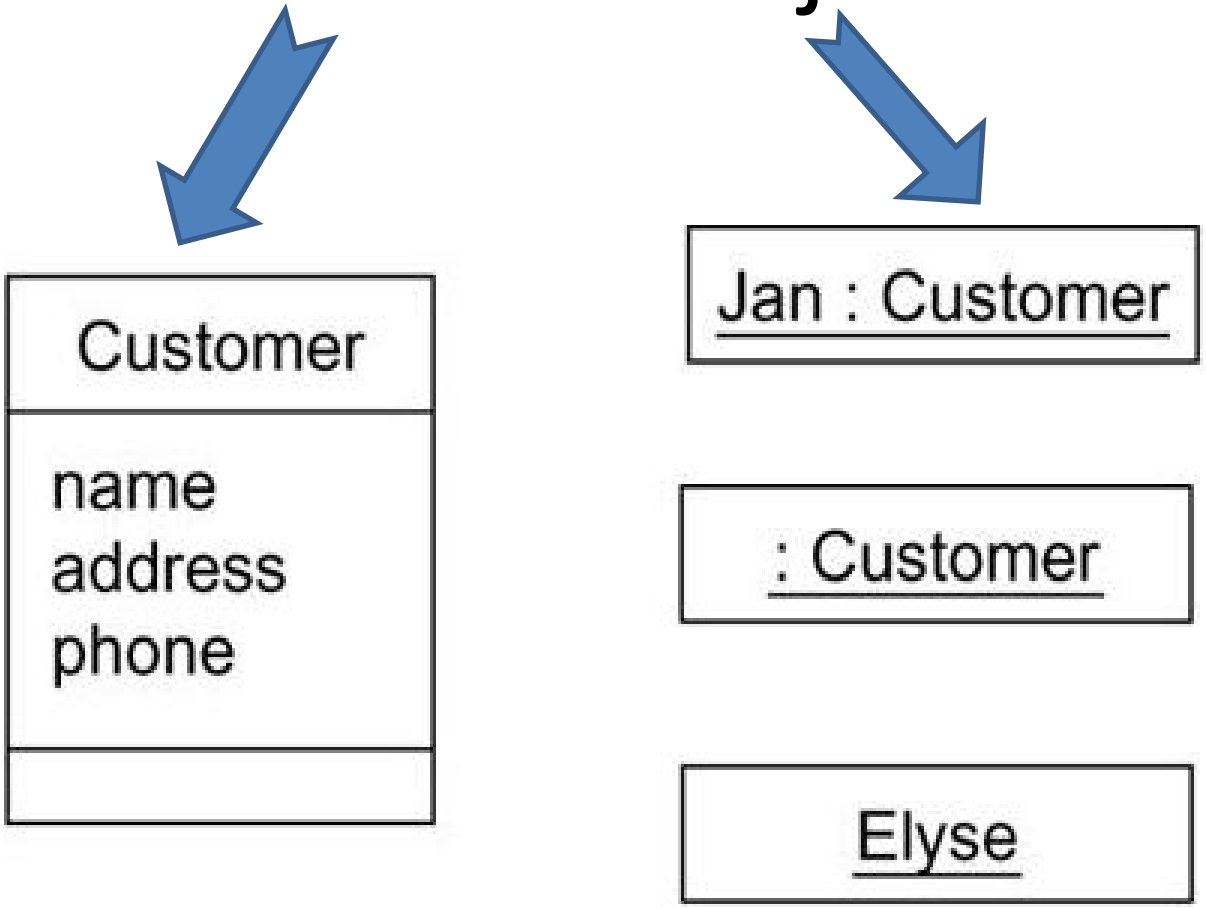
Class Diagram



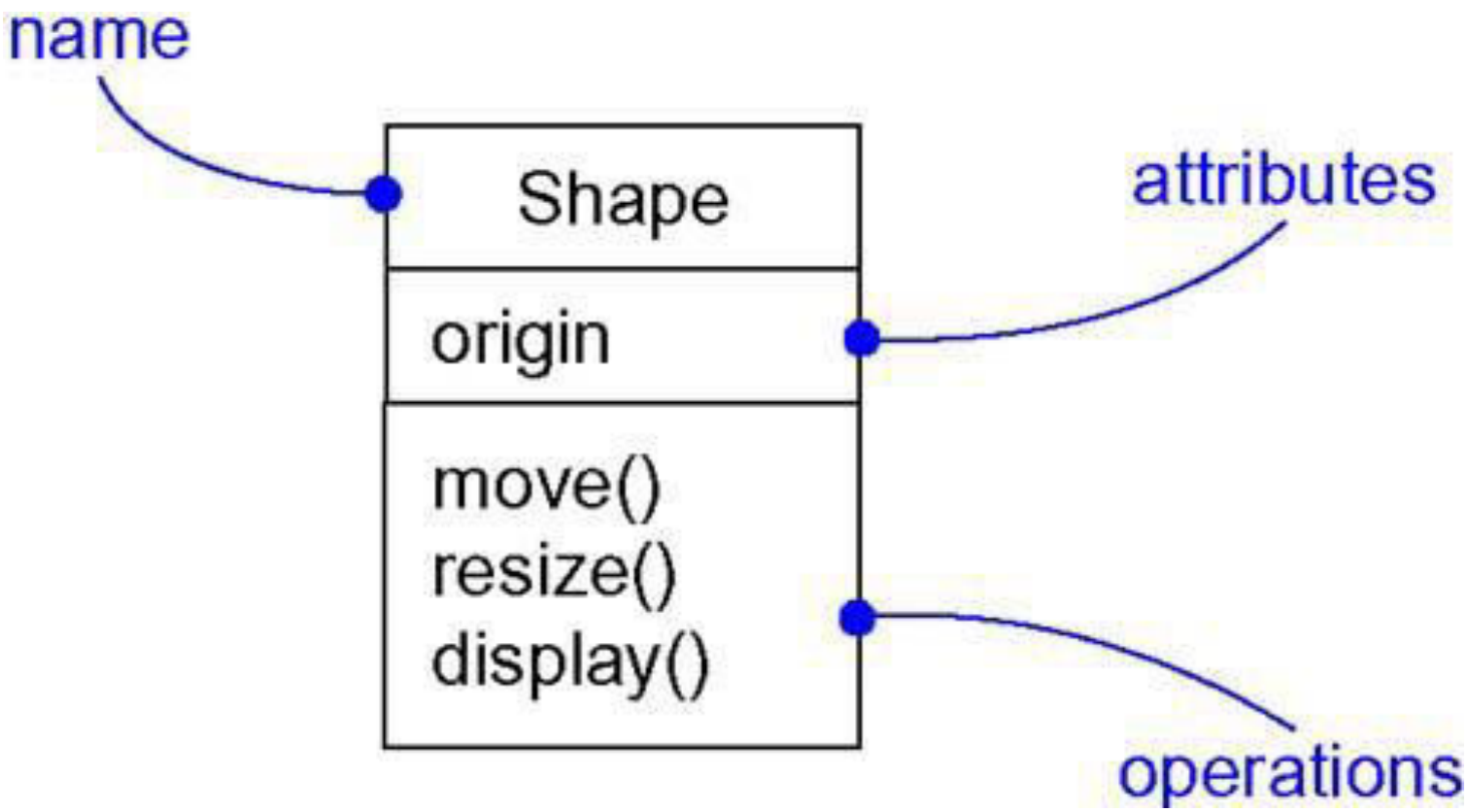
Class and Object

- First, there is the division of class and object.
  - A class is an abstraction;
  - an object is one concrete manifestation of that abstraction.
- In the UML, you can model classes as well as objects, as shown in following Figures

Class and Object



Class Diagram

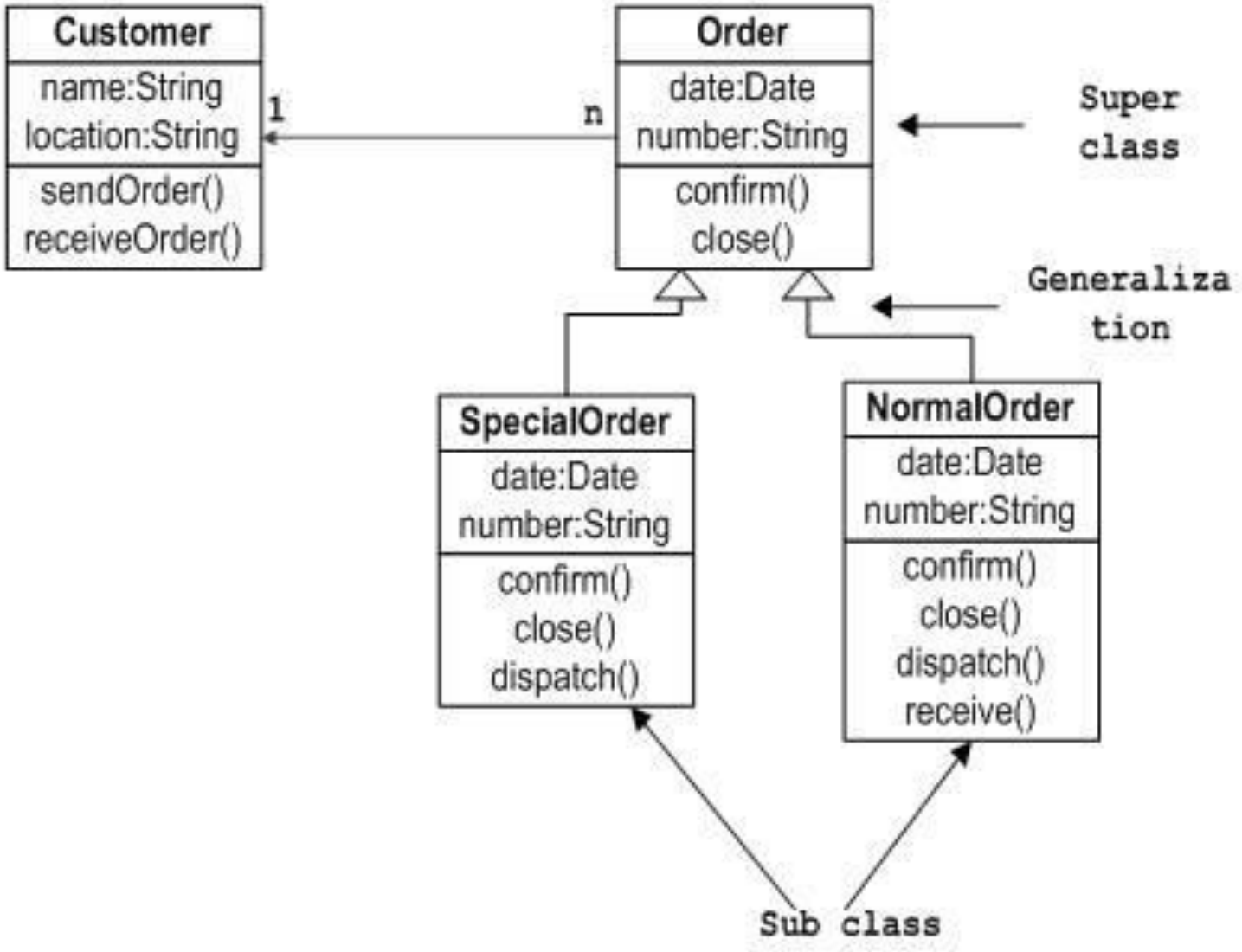


"Order System"

- First of all *Order* and *Customer* are identified as the two elements of the system and they have a *one to many* relationship because a customer can have multiple orders.
- We would keep *Order* class is an abstract class and it has two concrete classes (inheritance relationship) *SpecialOrder* and *NormalOrder*.
- The two inherited classes have all the properties as the *Order* class. In addition they have additional functions like *dispatch ()* and *receive ()*.

Class Diagram for Order System

Sample Class Diagram



Relationships among Classes

- **Links**
  - the basic relationship among objects
  - a tuple i.e. the list of objects
  - an instance of an association
- **Association**
  - represents a family of links with common structure and semantics.
  - Links of an association connect objects from same classes.
  - Describes a set of potential links
  - Binary associations (with two ends) are normally represented as a line.

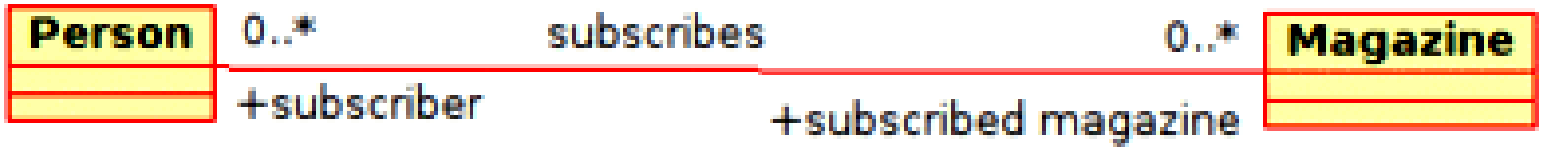
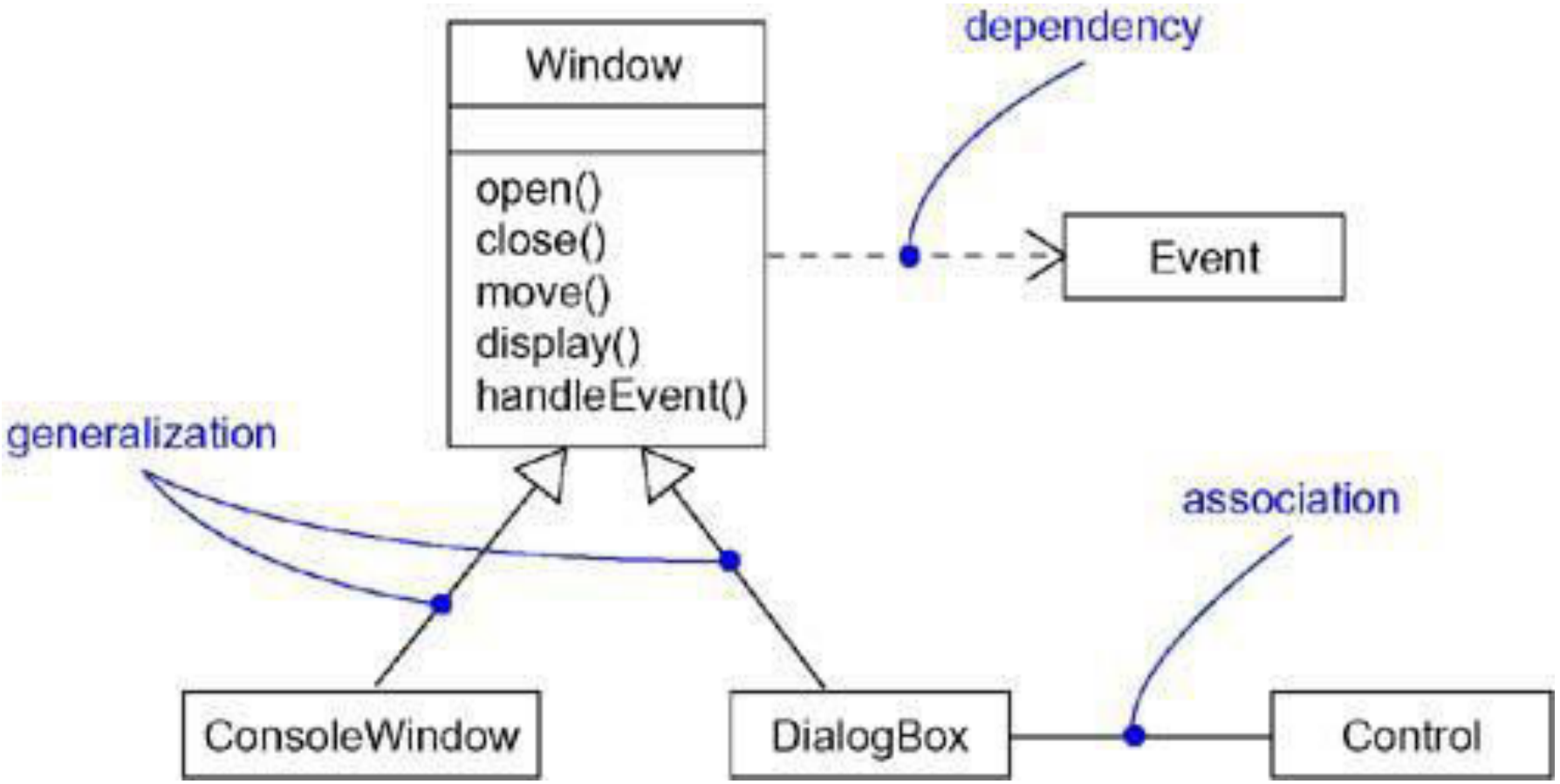


Fig: Class diagram example of association between two classes

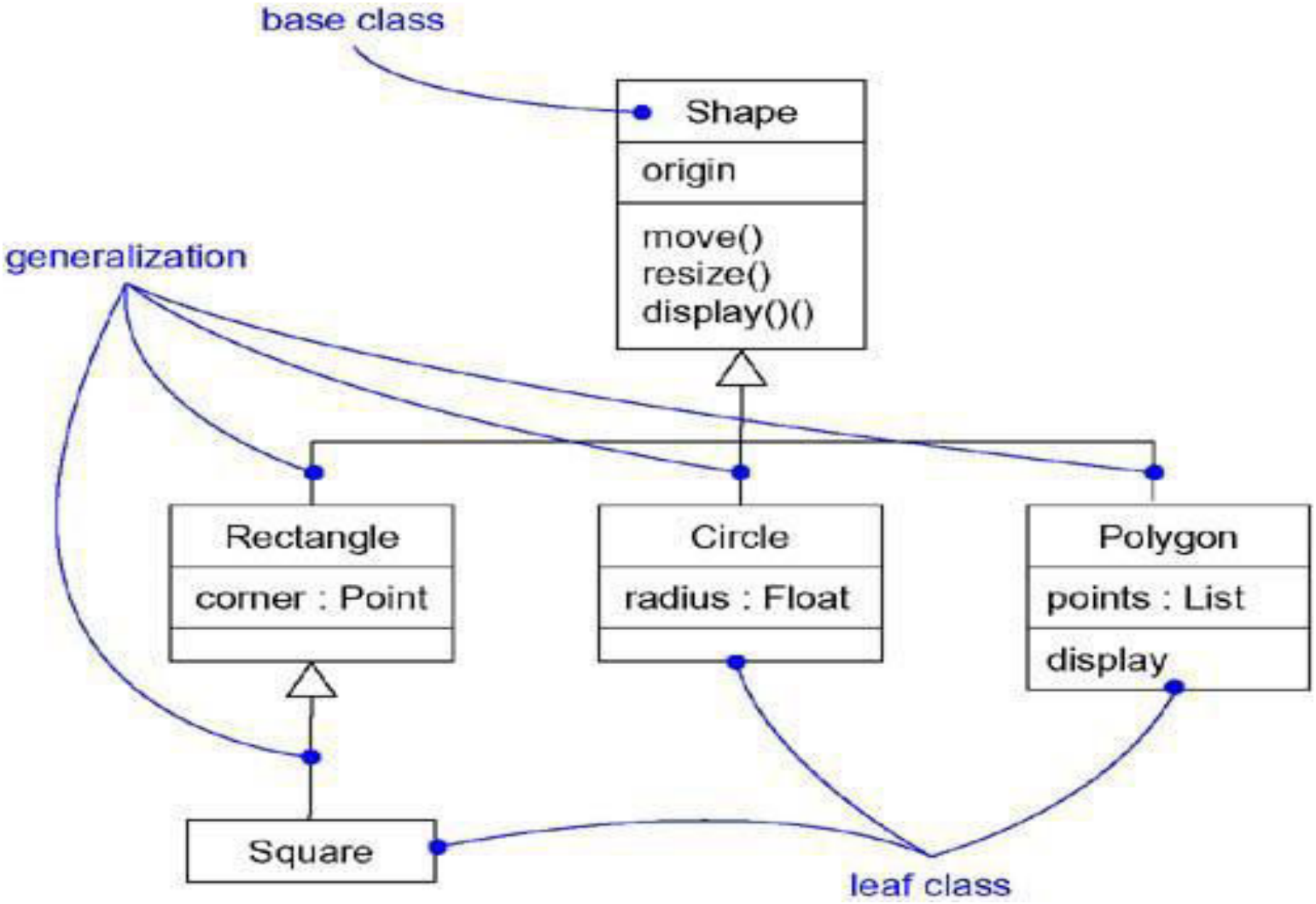
Notation

- Link
  - line between objects
  - If the link has name, it is underlined
- Association
  - Line between related classes
  - Association name is optional, if unambiguous
    - Ambiguity arise when a model has multiple associations among the same classes
  - Bidirectional
    - Can be traversed in either direction and equally meaningful

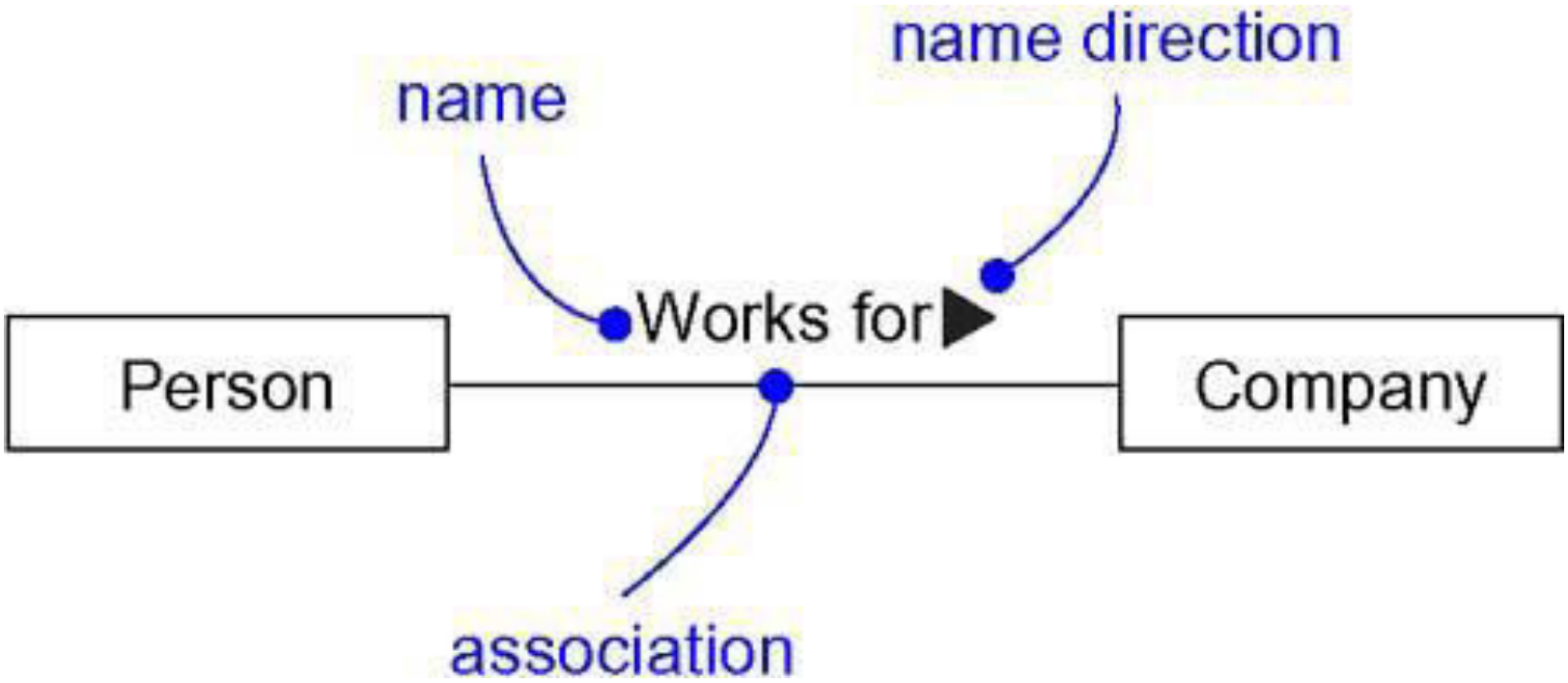
Class Diagram



Class Diagram: **Generalization**



Class Diagram: **Association Names**



Multiplicity

- Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.
- Multiplicity constrains the number of related objects.
- a subset of nonnegative integers.

Multiplicity

- UML specifies multiplicity with an interval, such as..

0..1 → No instances, or one instance (optional, may)

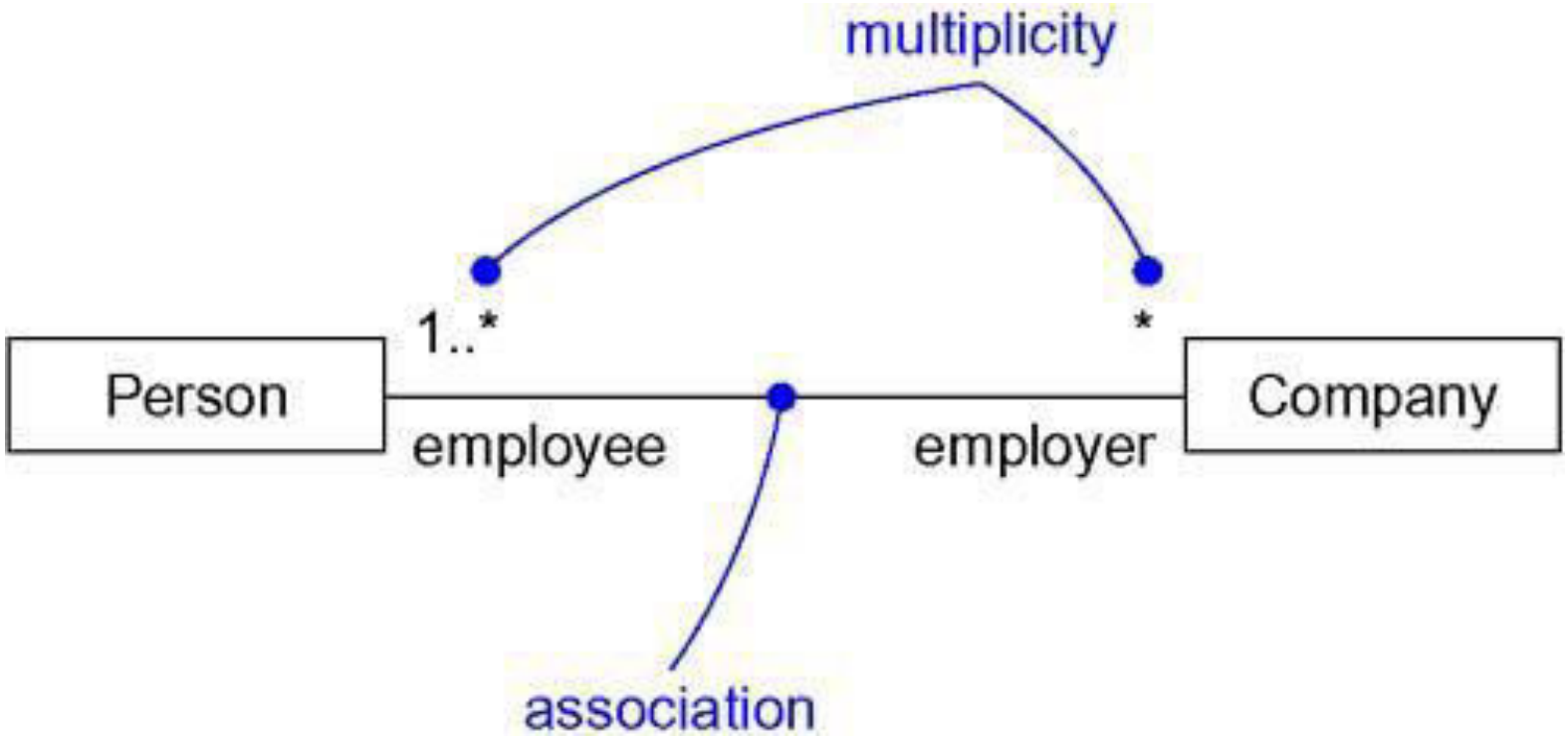
1 → Exactly one instance

0..* or * → Zero or more instances

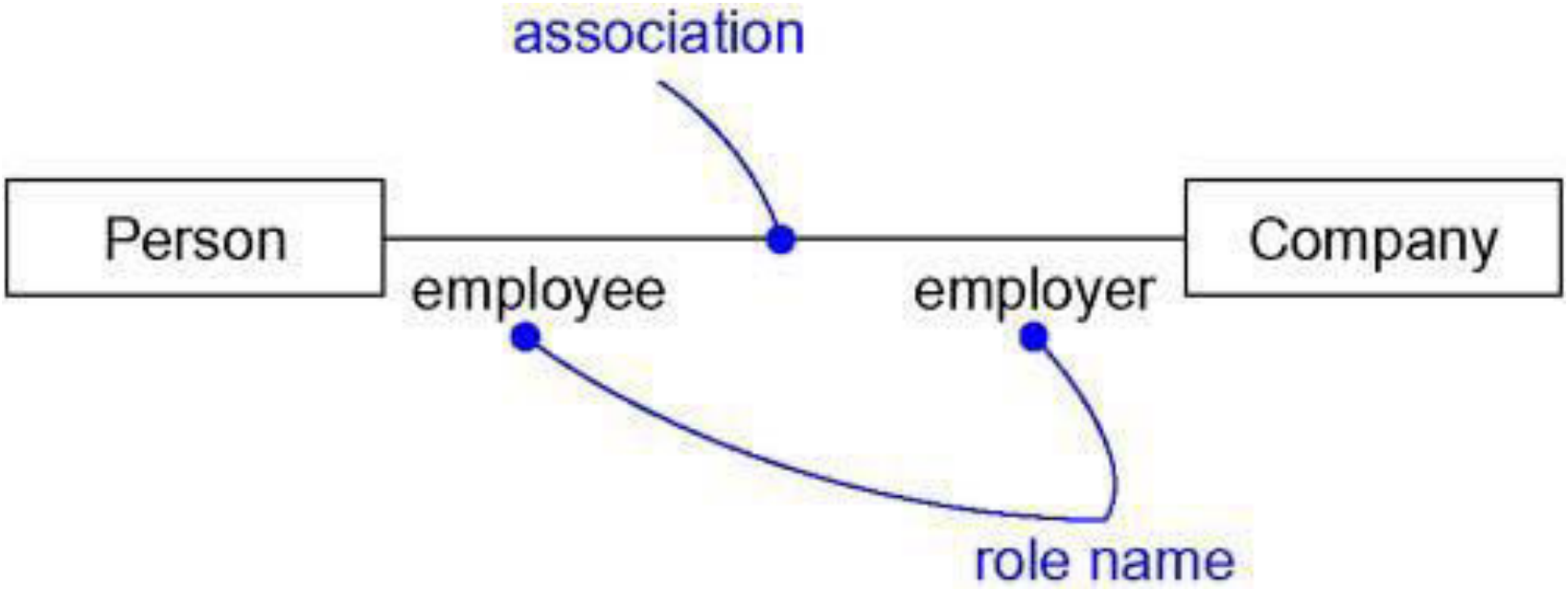
1..* → One or more instances (at least one)

3..5 → **three to five inclusive**

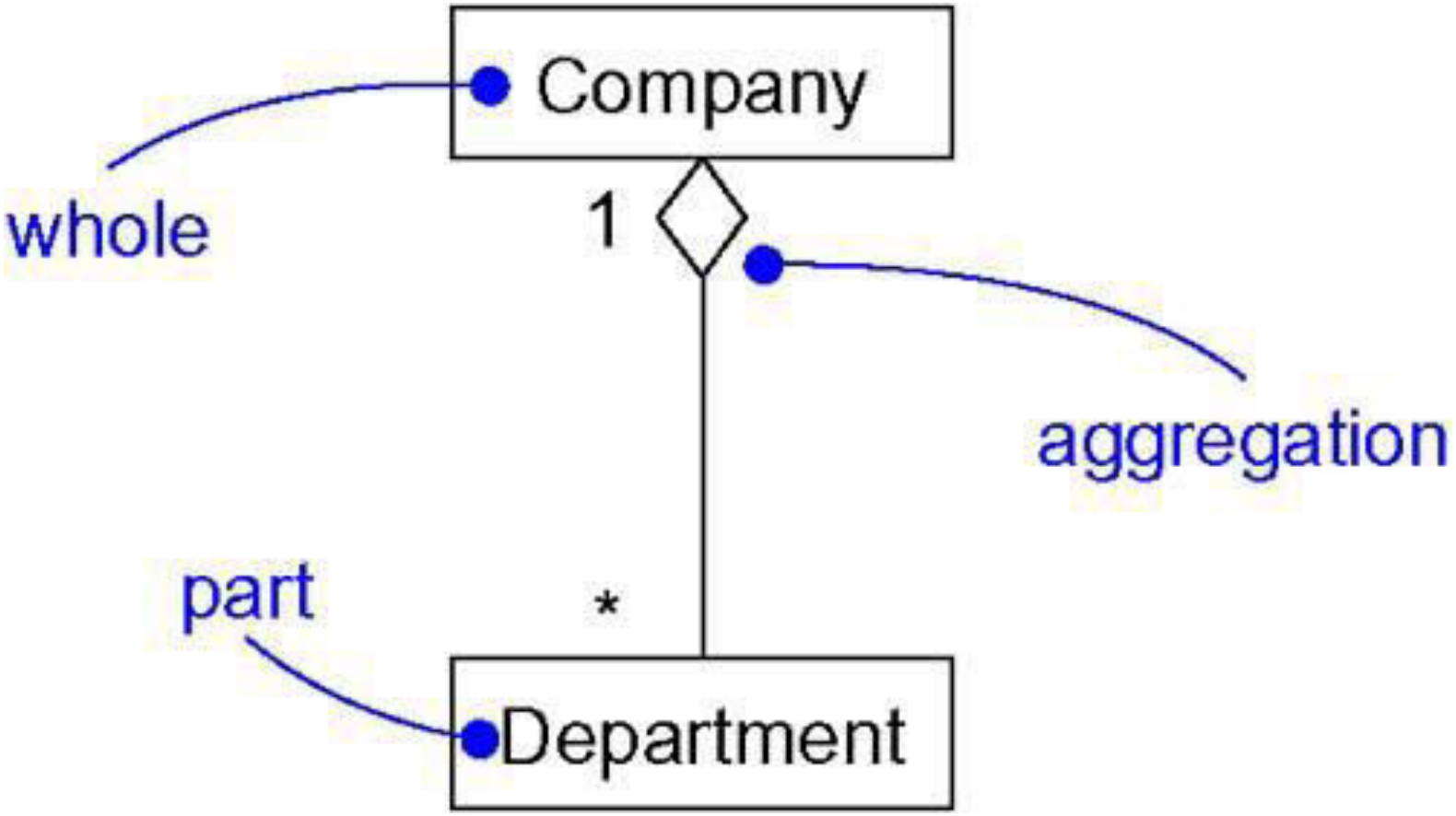
Class Diagram: **Multiplicity**



Class Diagram: **Roles**



Class Diagram: **Aggregation**



Visibility in UML

To specify the visibility of a class member (i.e., any attribute or method) these are the following notations that must be placed before the member's name:

"+" **Public**

"-" **Private**

"#" **Protected**

"/" **Derived** (can be combined with one of the others)

"~" **Static**

Association Types

- **Aggregation**
  - a variant of the "has a" association relationship;
  - aggregation is more specific than association
  - It is an association that represents a part-whole or part-of relationship.
  - can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong *life cycle dependency* on the container
  - if the container is destroyed, its contents are not.

CRC Card

http://www.math-cs.gordon.edu/courses/cps211/ATMExample/CRCCards.html

Introduction to CRC Cards

- Invented in 1989 by Kent Back and Ward Cunningham: Wilkinson 1995; Ambler 1995;
- An index card consists of heavy paper cut to a standard size, used for recording and storing small amounts of discrete data. It was invented by Carl Linnaeus, around 1760.
- CRC Card = Class Responsibility Collaborator Card
- a collection of standard index cards that have been divided into three sections,
  - A class represents a collection of similar objects,
  - a responsibility is something that a class knows or does, and
  - a collaborator is another class that a class interacts with to fulfill its responsibilities.

Format of CRC Cards

Class Name:

Responsibilities:	Collaborators:
(what class does or knows)	*(which classes help it perform each responsibility)*

What is CRC Card

- An effective way to analyze scenarios
- First was proposed by Beck & Cunningham as a tool for teaching OO programming.
- As a development tool that facilitates brainstorming & enhances communication among developers.
- The cards are arranged to show the flow of messages among instances of each class.
- As team members walk through the scenario,
  - they may assign new responsibilities to an existing class,
  - group certain responsibilities to form a new class or
  - divide responsibilities of one class into more fine grained ones
  - Perhaps distribute these responsibilities to a different class.

Contd..

- Responsibilities of a class (What information you wish to maintain about it)
  - Knowing responsibilities
    - instance of a class must be capable of knowing (the values of its attributes and its relationships)
    - Example: Student have names, addresses and phone no
  - Doing responsibilities
    - things that an instance of a class must be capable of doing
    - Example: Students enroll in seminars, drop seminars
- A class is able to change the values of the things it knows, but unable to change the values of what other classes know

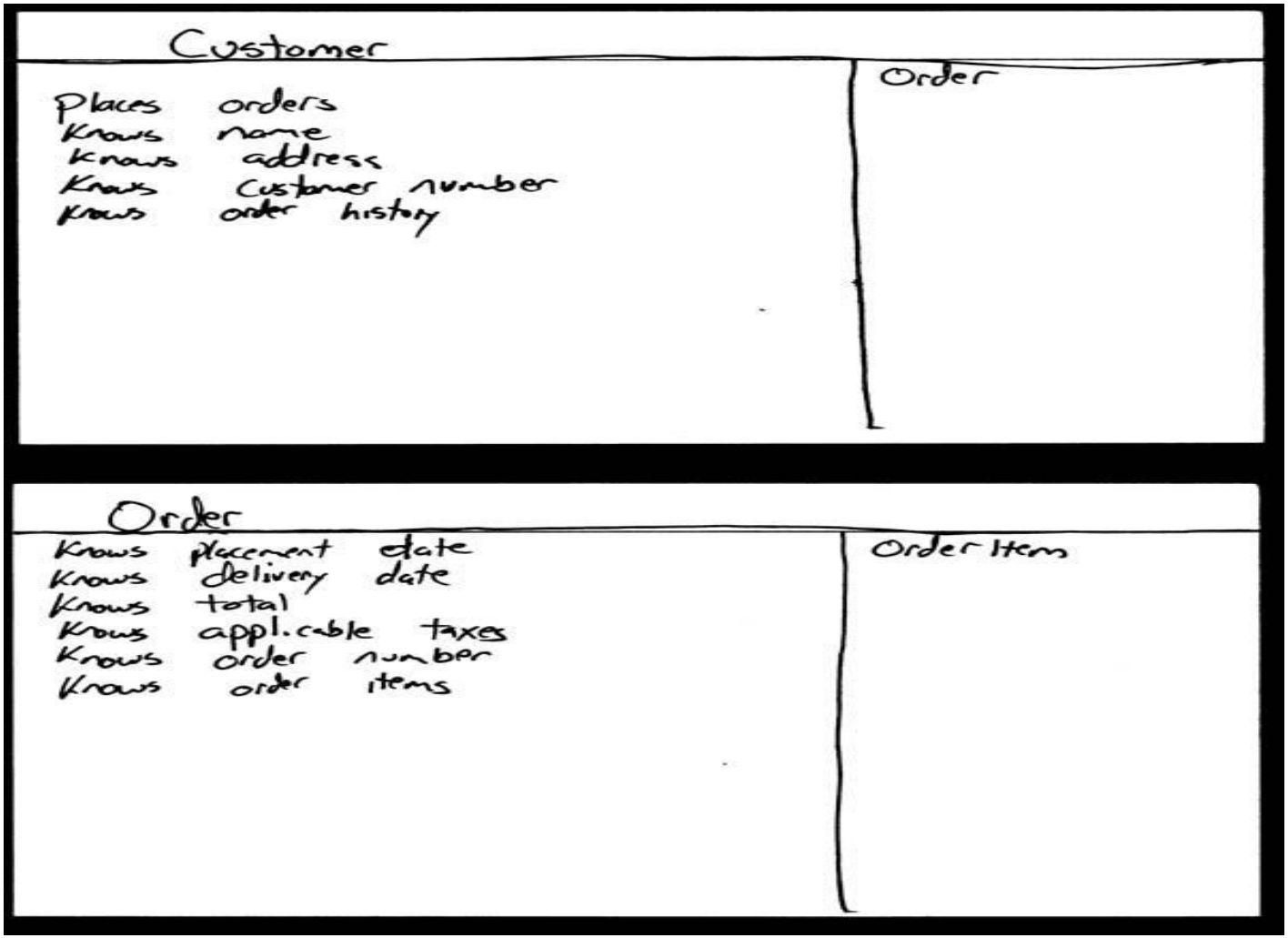
Contd..

- Sets of class forms a collaboration
  - Class does not have sufficient information to fulfill its responsibilities
  - A class must collaborate with other classes to get the job done
  - Example: Students enroll in seminars. To do this student needs to know if a spot is available and, if so, students then needs to be added to the seminar

- Allows the analyst to think in terms of an instance of a clients, servers, and contracts (request for information or a request to do something)
  - Client objects – sends a request to an instance of another class for an operation to be executed.
  - Server objects – receives the request from the client object.
  - Contracts objects – formalizes the interaction between the client and server objects.
  - Example:
    - the card *Student* requests an indication from the card *Seminar* whether a space is available, a request for information
    - *Student* then requests to be added to the *Seminar*, a request to do something
    - Another way: *Student* simply request *Seminar* to enroll himself into itself. Then have *Seminar* do the work of determining if a seat is available and, if so, then enrolling the student and, if not, then informing the student that he was not enrolled

Example…..

Hand-drawn CRC cards



Example…

student CRC card

Student	
Student number Name Address Phone number Enroll in a seminar Drop a seminar Request transcripts	Seminar

Example: Student CRC card

Professor	
Name Address Phone number Email address Salary Provide information Seminars instructing	Seminar

Transcript	
See the prototype Determine average mark	Student Seminar Professor Enrollment

Seminar	
Name Seminar number Fees Waiting list Enrolled students Instructor Add student Drop student	Student Professor

Enrollment	
Mark(s) received Average to date Final grade Student Seminar	Seminar

Student Schedule	
See the prototype	Seminar Professor Student Enrollment Room

Student	
Name Address Phone number Email address Student number Average mark received Validate identifying info Provide list of seminars taken	Enrollment

Room	
Building Room number Type (Lab, class, ...) Number of Seats Get building name Provide available time slots	Building

Building	
Building Name Rooms Provide name Provide list of available rooms for a given time period	Room

how do you create CRC models?

- Find classes
- Find responsibilities
- Define collaborators
  - To identify the collaborators of a class for each responsibility ask yourself "does the class have the ability to fulfill this responsibility?"
- Move the cards around
  - To improve everyone's understanding of the system, the cards should be placed on the table in an intelligent manner.
  - Two cards that collaborate with one another should be placed close together on the table, whereas two cards that don't collaborate should be placed far apart.
  - Furthermore, the more two cards collaborate, the closer they should be on the desk. By having cards that collaborate with one another close together, it's easier to understand the relationships between classes

FINDING CLASSES Method:

1. Read specification
2. Work through requirements, highlighting nouns and noun phrases to give candidate classes.
3. Work through candidates, deciding likely classes and rejecting unlikely.

READ SPECIFICATION

- If you don't have one, WRITE your own.
- The specification should:
  - describe the goals of the design
  - discuss the things the system should do
- i.e. desired responses to expected inputs

CANDIDATE CLASSES

- physical objects e.g. printer, switch
- cohesive entities e.g. file, window
- categories of classes (may become abstract superclasses)
- interfaces both to user and to other programs
- attribute values (NOT attributes) e.g. "circle has radius in real numbers" : circle and real are classes; radius is not.

FINDING CLASSES

- The process of identifying classes is iterative - **some will be missed**, **others will be amalgamated** Likely classes:
  - any physical objects
  - any 'cohesive abstractions' e.g. a file
  - any interfaces to outside world
  - any categories of class e.g. "there are various types of whatsits"
  - values of attributes of objects

CLASS MODELING

1. Steps to identify the Candidate CLASS from Given Requirements

- identify the objects and derive classes from
- identify attributes of classes
- identify relationships between the classes;
- write a data dictionary to support the class diagram;
- identify class responsibilities
- separate responsibilities into operations and attributes;

Noun Phrase Approach

- Nouns in the textual description are considered to be classes and verbs to be methods of the classes.
- All plurals are changed to singular, the nouns are listed and the list divided into
  - relevant classes,
  - fuzzy classes and
  - irrelevant class.

Identifying tentative classes

- Look for noun phrases and nouns in the use cases.
- Some classes are implicit or taken from general knowledge.
- All classes must make sense in the application domain.
- Carefully choose and define class names.

Selecting classes from the relevant and fuzzy classes.

- Redundant classes.
- Adjective classes.
- Attribute classes.
- Irrelevant classes.

Example: Bank ATM system: Identifying classes by using noun phrase approach:

Initial list of Noun phrases candidate classes

- Account
- Account Balance
- Amount
- Approval process
- ATM card
- ATM machine
- Bank
- Bank client
- Card
- Cash
- Check
- Checking
- Checking Account
- Client
- Client's Account
- Currency
- Dollar
- Envelope
- Four digits
- Fund
- Savings
- Savings Account
- Step
- System
- Transaction
- Transaction history
- Invalid PIN
- Message
- Money
- Password
- PIN
- Pin Code
- Record

- **The following irrelevant classes are removed from the above list:**
  - Envelope
  - Four Digits
  - Step
- **Reviewing the Redundant classes and Building a common vocabulary:**

Client, Bank client **à** Bank client
Account, Client's Account **à** Account
PIN, PIN code **à** PIN
Checking, Checking Account **à** Checking Account
Savings, Savings Account **à** Savings Account
Fund, Money **à** Fund
ATM card, card **à** ATM card

- **Reviewing the classes containing adjectives**

  In this example, we have no classes containing adjectives that we can eliminate.

- **Reviewing the Possible Attribute**

  Amount à A value not a class
  Account Balance à An attribute of the Account class
  Invalid PIN à It is only a value, not class
  Password à An attribute for Bank client class
  Transaction history à An attribute of transaction class
  PIN à An attribute of Bank client class

Reviewing the class purpose

The final candidate classes are:

- ATM machine class
- ATM card class
- Bank client
- Bank class
- Account class
- Checking Account class
- Savings Account class
- Transaction class

Example: Description of problem

R1 keep a complete list of all bikes and their details including bike number, type, size, make, model, daily charge rate, deposit; (this is already on the Wheels system);

R2 keep a record of all customers and their past hire transactions;

R3 work out automatically how much it will cost to hire a given bike for a given number of days;

R4 record the details of a hire transaction including the start date, estimated duration, customer and bike, in such a way that it is easy to find the relevant transaction details when a bike is returned;

R5 keep track of how many bikes a customer is hiring so that the customer gets one unified receipt not a separate one for each bike;

R6 cope with a customer who hires more than one bike, each for different amounts of time;

R7 work out automatically, on the return of a bike, how long it was hired for, how many days were originally paid for, how much extra is due;

R8 record the total amount due and how much has been paid;

R9 print a receipt for each customer;

R10 keep track of the state of each bike, e.g. whether it is in stock, hired out or being repaired;

R11 provide the means to record extra details about specialist bikes.

FIRST: Underline the nouns and noun phrases

R1 keep a complete list of all bikes and their details including bike number, type, size, make, model, daily charge rate, deposit; (this is already on the Wheels system);

R2 keep a record of all customers and their past hire transactions;

R3 work out automatically how much it will cost to hire a given bike for a given number of days;

R4 record the details of a hire transaction including the start date, estimated duration, customer and bike, in such a way that it is easy to find the relevant transaction details when a bike is returned;

R5 keep track of how many bikes a customer is hiring so that the customer gets one unified receipt not a separate one for each bike;

R6 cope with a customer who hires more than one bike, each for different amounts of time;

R7 work out automatically, on the return of a bike, how long it was hired for, how many days were originally paid for, how much extra is due;

R8 record the total amount due and how much has been paid;

R9 print a receipt for each customer;

R10 keep track of the state of each bike, e.g. whether it is in stock, hired out or being repaired;

R11 provide the means to record extra details about specialist bikes.

SECOND: List of *nouns*

- list of bikes , details of bikes: bike number, type, size, make, model, daily charge rate, deposit , Wheels system , record of customers , past hire transactions , bike, number of days, details of a hire transaction: start date, estimated duration, customer, receipt, different amounts of time, return of a bike, total amount due, state of each bike, extra details about specialist bikes

THIRD: Remove *attributes nouns*

- list of bikes , details of bikes: bike number, type, size, make, model, daily charge rate, deposit , Wheels system , record of customers , past hire transactions , bike, number of days, details of a hire transaction: start date, estimated duration, customer, receipt, different amounts of time, return of a bike, total amount due, state of each bike,extra details about specialist bikes

FOURTH: Remove *redundancy/duplicates*

- list of bikes , details of bikes, Wheels system , record of customers , past hire transactions , bike, details of a hire transaction, customer, receipt, return of a bike, specialist bikes

FIFTH: **Remove *vague* nouns**

- list of bikes , Wheels system , record of customers , hire transactions , bike, customer, receipt, return of a bike, specialist bikes

SIXTH: **Remove nouns tied up with physical inputs and outputs**

- list of bikes , Wheels system , record of customers , hire transactions , bike, customer, receipt, specialist bikes

SEVENTH: **Remove nouns that represent the whole system**

- Wheels system, hire transactions, bike, customer, specialist bikes

LIST OF CANDIDATE CLASSES: hire transactions, bike, customer, specialist bikes

Exercise 1

- A school has a principal, many students, and many teachers. Each of these persons has a name, birthdate, and may borrow and return books. Teachers and the principal are both paid a salary; the principal evaluates the teachers. A school board supervises multiple schools and can hire and fire the principal for each school. A school has many playgrounds and rooms. A playground has many swings. Each room has many chairs and doors. Rooms include restrooms, classrooms, and the cafeteria. Each classroom has many computers and desks. Each desk has many rulers.

Exercise 2

A drive has multiple discs; a hard drive contains many discs and a floppy drive contains one disc. *A disc is divided into* tracks which are in turn subdivided into sectors. A file system may use multiple discs and a disc may be partitioned across file systems. Similarly a disc may contain many files and a file may be partitioned across many discs. A file system consists of many files. Each file has an owner, permissions for reading and writing, date last modified, size, and checksum. Operations that apply to files include create, copy, delete, rename, compress, uncompress, and compare. Files may be data files or directory files. A directory hierarchically organizes groups of presumably related files; directories may be recursively nested to an arbitrary depth. Each file within a directory can be uniquely identified by its file name. A file may correspond to many directory–file name pairs such as through UNIX links. A data file may be an ASCII file or binary file.

Class Diagram Case Study 1

- An automobile is composed of a variety of parts. An automobile has one engine, one exhaust system, many wheels, many brakes, many brake lights, many doors, and one battery. An automobile may have 3, 4, or 5 wheels depending on whether the frame has 3 or 4 wheels and the optional spare tire. Similarly a car may have 2 or 4 doors. The exhaust system can be further divided into smaller components such as a muffler and tailpipe. A brake is associated with a brake light that indicates when the brake is being applied.

Class Diagram Case Study 2

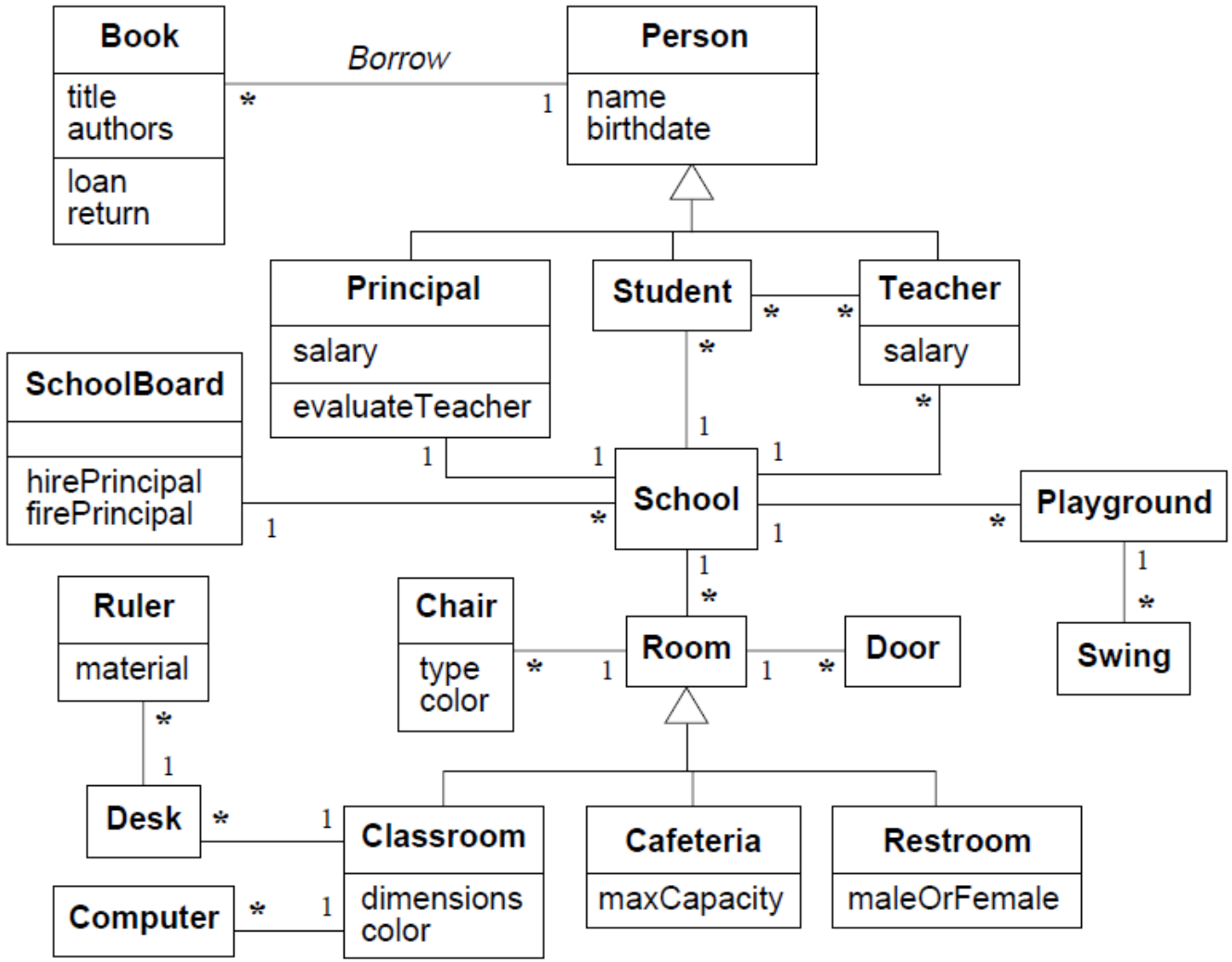
- A gas heating system is composed of furnace, humidification, and ventilation subsystems.
- The furnace subsystem can be further decomposed into a gas furnace, gas control, furnace thermostat, and many room thermostats. The room thermostats can be individually identified via the room number qualifier.
- The humidification subsystem includes a humidifier and humidity sensor.
- The ventilation subsystem has a blower, blower control, and many hot air vents. The blower in turn has a blower motor subcomponent.

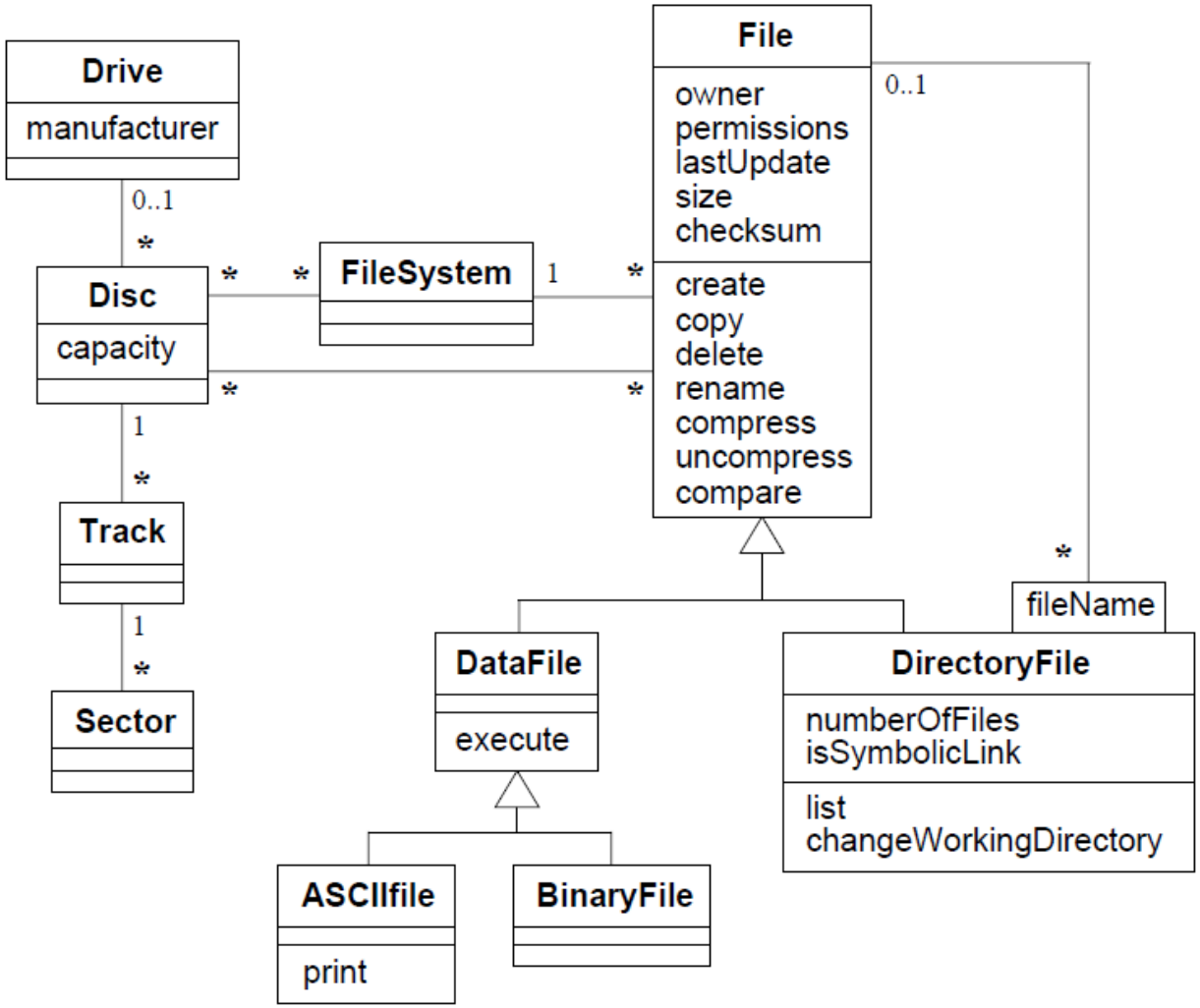
Class Diagram Case Study 3

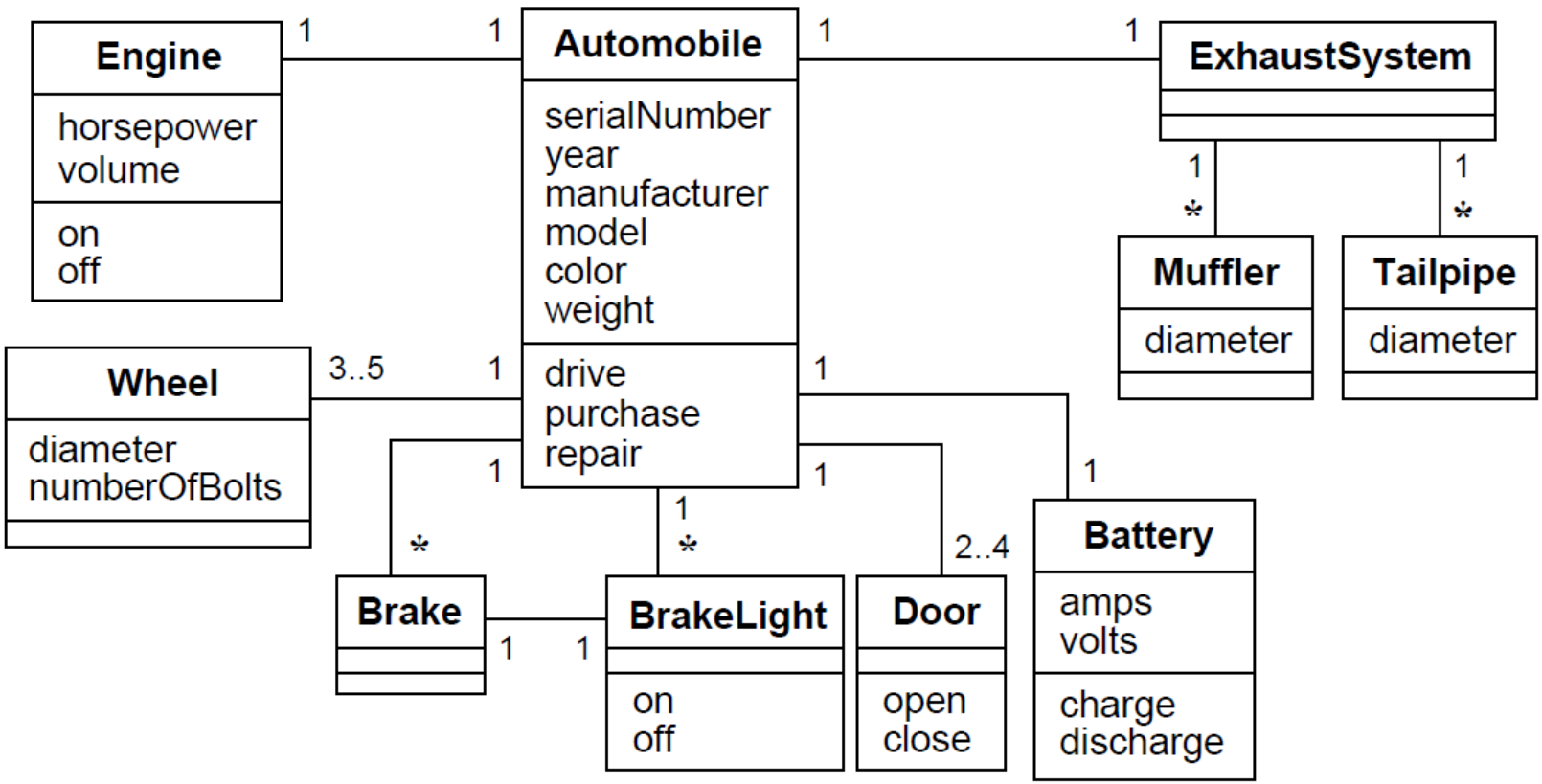
- A castle has many rooms, corridors, stairs, towers, dungeons, and floors. Each tower and dungeon also has a floor. The castle is built from multiple stones each of which has dimensions, weight, color, and a composition material. The castle may be surrounded by a moat. Each lord lives in a castle; a castle may be without a lord if he has been captured in battle or killed. Each lady lives in a castle with her lord. A castle may be haunted by multiple ghosts, some of which have hostile intentions. Each room is connected to a corridor and one corridor connects another corridor.

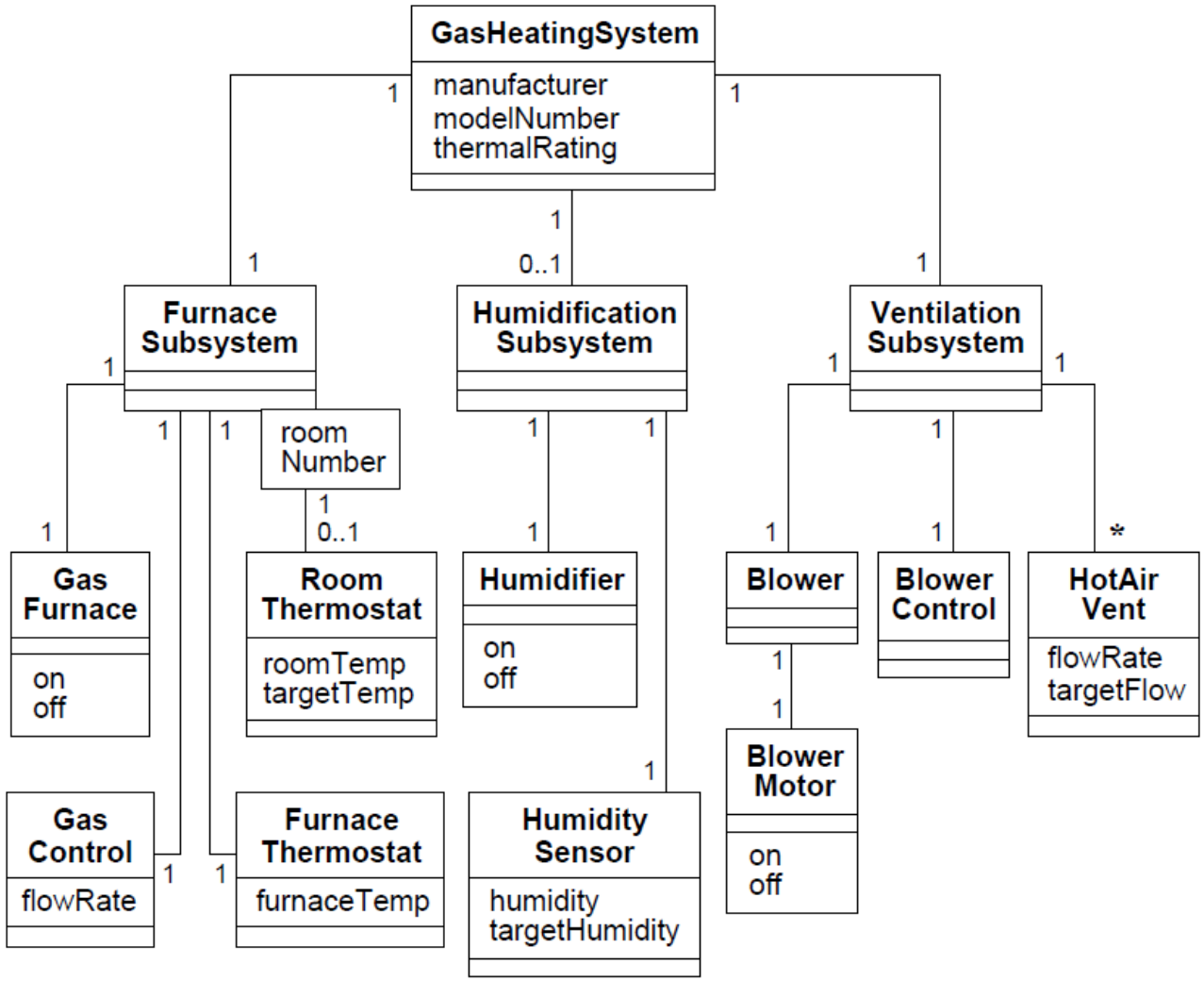
Prepare a class diagram

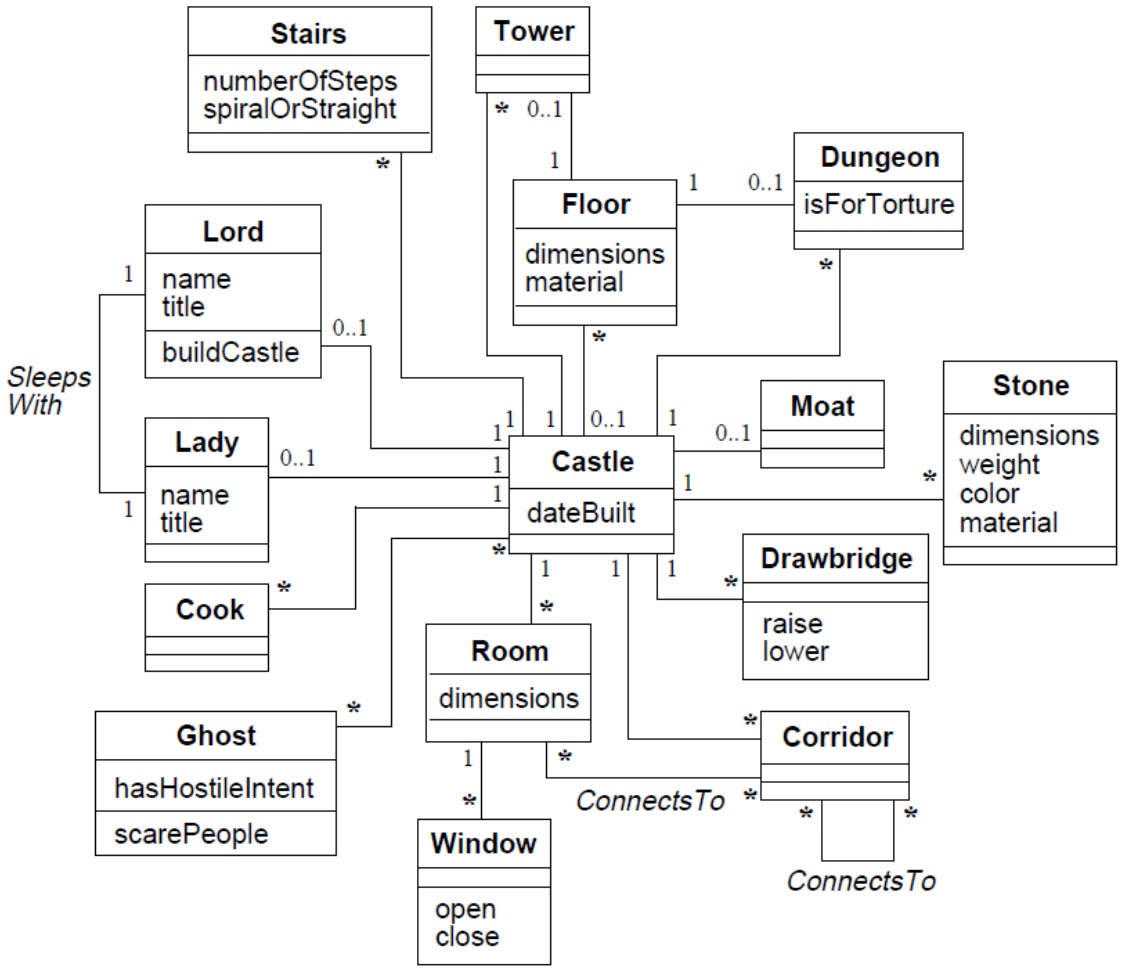
Consider the air transportation system. Many flights land and depart from city's airport. Some of the big cities may have more than one airports. Every flight belongs to specific airline. The planes may have many flights to different airports. Each plane is identified with serial number and model. There are specific pilots for each airline and they fly many flights. Each flight is identified by flight number and date on which flight is scheduled. The passenger reserves a seat for a flight. The seat is identified by a location.

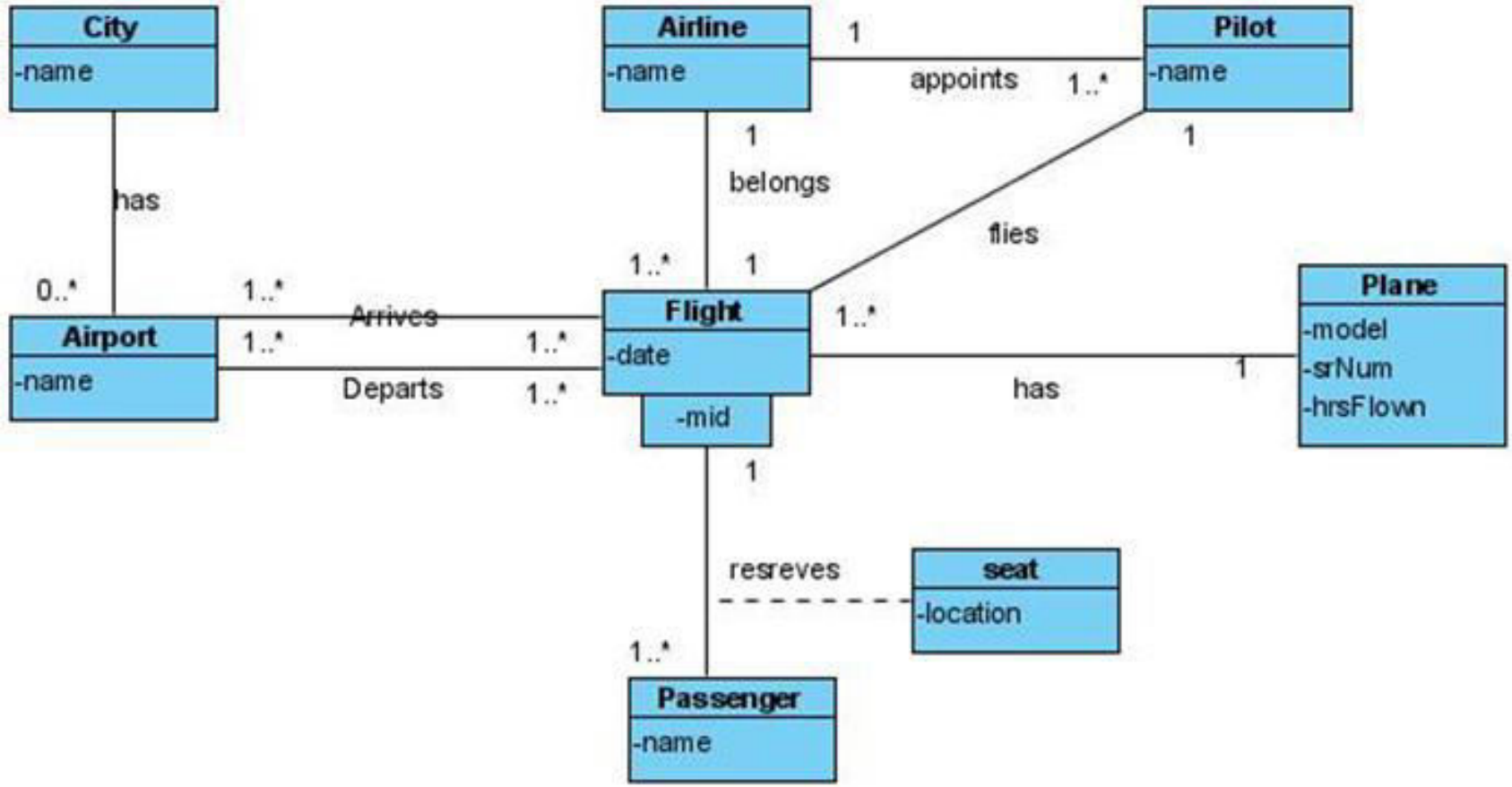

Book
title
authors
loan
return
Borrow
*
1
Person
name
birthdate
Principal
salary
evaluateTeacher
Student
Teacher
salary
SchoolBoard
hirePrincipal
firePrincipal
School
Playground
Swing
Ruler
material
Chair
type
color
Room
Door
Desk
Classroom
dimensions
color
Computer
Cafeteria
maxCapacity
Restroom
maleOrFemale


Drive
manufacturer
0..1
*
Disc
capacity
*
*
FileSystem
1
*
File
owner
permissions
lastUpdate
size
checksum
create
copy
delete
rename
compress
uncompress
compare
0..1
*
*
1
*
Track
1
*
Sector
DataFile
execute
fileName
DirectoryFile
numberOfFiles
isSymbolicLink
list
changeWorkingDirectory
ASCIIfile
print
BinaryFile


Engine
horsepower
volume
on
off
Automobile
serialNumber
year
manufacturer
model
color
weight
drive
purchase
repair
ExhaustSystem
Muffler
diameter
Tailpipe
diameter
Wheel
diameter
numberOfBolts
Brake
BrakeLight
on
off
Door
open
close
Battery
amps
volts
charge
discharge
1
*
3..5
2..4

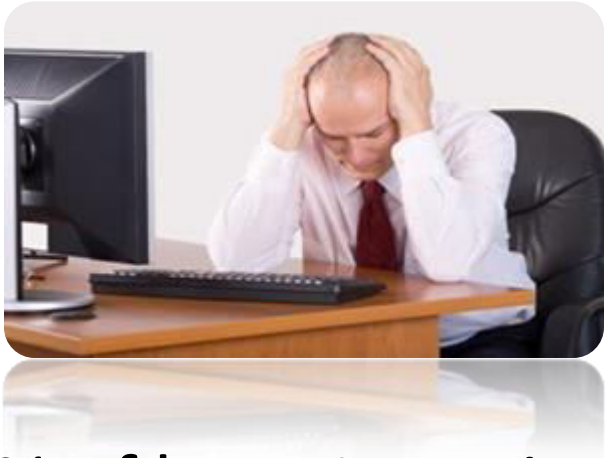

GasHeatingSystem
manufacturer
modelNumber
thermalRating
Furnace Subsystem
Humidification Subsystem
Ventilation Subsystem
room Number
Gas Furnace
on
off
Room Thermostat
roomTemp
targetTemp
Humidifier
on
off
Blower
Blower Control
HotAir Vent
flowRate
targetFlow
Blower Motor
on
off
Gas Control
flowRate
Furnace Thermostat
furnaceTemp
Humidity Sensor
humidity
targetHumidity
1
0..1
*


Stairs
numberOfSteps
spiralOrStraight
Tower
Dungeon
isForTorture
Floor
dimensions
material
Lord
name
title
buildCastle
Sleeps
With
Lady
name
title
Castle
dateBuilt
Moat
Stone
dimensions
weight
color
material
Drawbridge
raise
lower
Cook
Room
dimensions
Ghost
hasHostileIntent
scarePeople
Corridor
ConnectsTo
Window
open
close
ConnectsTo


City
-name
has
0..*
1..*
Airport
-name
Arrives
1..*
1..*
Departs
1..*
Airline
-name
1
appoints
1..*
Pilot
-name
1
belongs
1
1..*
1
flies
1..*
Flight
-date
-mid
1..*
has
1
Plane
-model
-srNum
-hrsFlown
1
resreves
seat
-location
1..*
Passenger
-name

An Introduction to the Rational Unified Process

68% of all Software Projects fail - ZDNET



- McKinsey – 17% of large IT Projects fail miserably
- Geneca - Large IT Projects run 45% over budget, 7% over time, delivering 56% less value
- 75% Project participants lack confidence in their project

Software Projects fail from around 5 to 47 factors – Alexandria University

- Organizational Structure
- Badly Defined Requirements
- Unrealistic or Unarticulated goals
- Inability to handle project complexity
- Sloppy development practices
- Inaccurate estimates

Project failures can be controlled

- Delivery dates impact project delivery
- Projects estimations can be made as close as possible
- Risks can be re-assessed, controlled and managed
- Staff can be awarded for long work hours

What is RUP ?

A software engineering process based on best practices in modern software development

- A disciplined approach to assigning and managing tasks and responsibilities in a development organization
- Focus on high quality software that meets the needs of its end users within a predictable schedule and budget

A process framework that can be tailored to specific organization or project needs

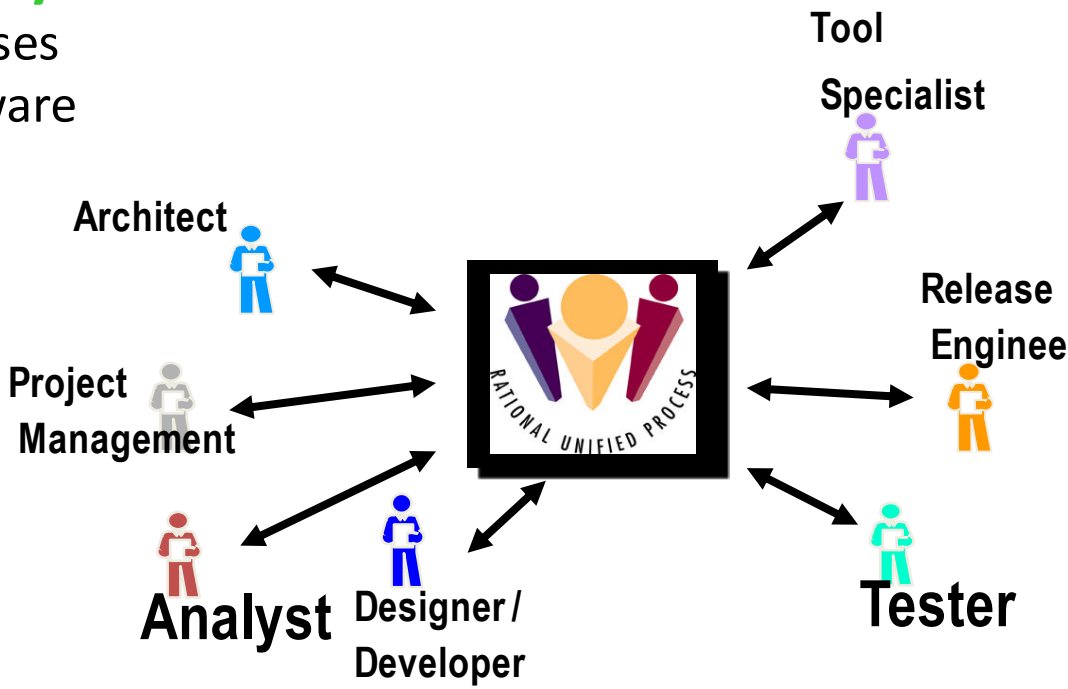
RUP is a methodology for delivering projects in a maximum performance manner

RUP uses an integration of approaches & initiatives

- **Team-Unifying Approach**

  The RUP unifies a software team by providing a common view of the development process and a shared vision of a common goal

- **Increased Team Productivity**
  - knowledge base of all processes
  - view of how to develop software
  - modeling language
  - Rational provides many tools



Key Aspects of RUP

Risk-driven process

- Risk management integrated into the development process
- Iterations are planned based on high priority risks

Use-Case driven development

- Use cases express requirements on the system's functionality and model the business as context for the system
- Use cases are defined for the intended system and are used as the basis of the entire development process

Architecture-centric design

- Architecture is the primary artifact to conceptualize, construct, manage, and evolve the system
- Consists of multiple, coordinated views (or models) of the architecture

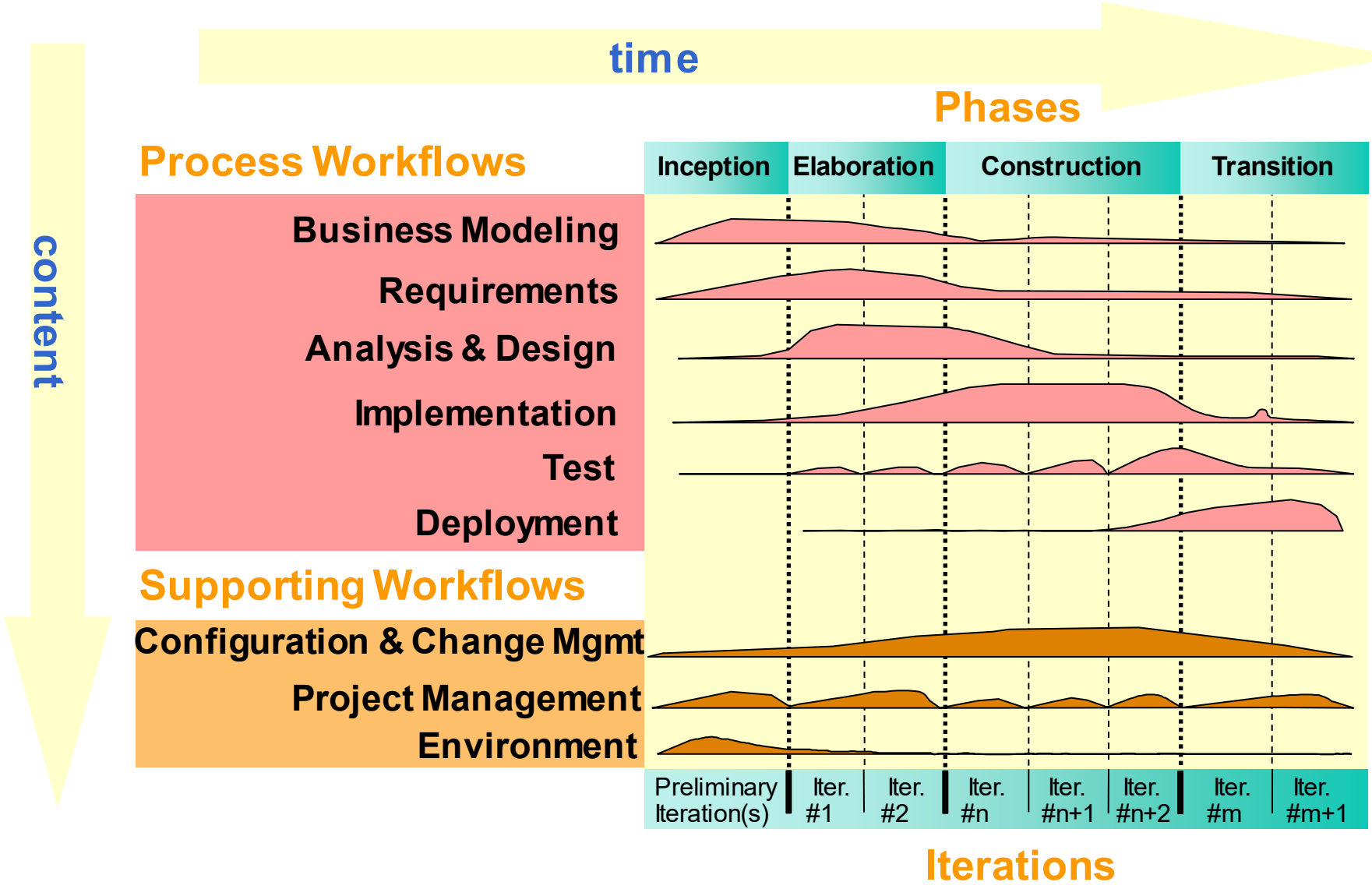
The Rational Unified Process

- RUP is a method of managing OO Software Development
- It can be viewed as a Software Development Framework which is extensible and features:
  - Iterative Development
  - Requirements Management
  - Component-Based Architectural Vision
  - Visual Modeling of Systems
  - Quality Management

The Development Phases

- Inception Phase
- Elaboration Phase
- Construction Phase
- Transition Phase

Rational Unified Process (RUP)



Inception Phase

- Goal is obtaining buy-in from all interested parties
- Initial requirements capture
- Cost Benefit Analysis
- Initial Risk Analysis
- Project scope definition
- Development of a disposable prototype
- Initial Use Case Model (10% - 20% complete)
- First pass at a Domain Model

- Understand what to build.
  - A vision document:
  - Optional business model
  - An initial project glossary
- Identify key system functionality.
  - A initial use-case model (10% -20%) complete.
- Determine at least one possible solution.
  - One or several prototypes.
- Understand the costs, schedule, and risks associated with the project.
  - An initial risk assessment.
  - Business case
- Decide what process to follow and what tools to use.
  - A project plan

Elaboration Phase

- Requirements Analysis and Capture(deeper)
  - Use Case Analysis
    - Use Case (80% written and reviewed by end of phase)
    - Use Case Model (80% done)
    - Scenarios
      - Sequence and Collaboration Diagrams
      - Class, Activity, Component, State Diagrams
  - Glossary (so users and developers can speak common vocabulary)
  - Domain Model
    - to understand the problem: the system's requirements as they exist within the context of the problem domain
  - Risk Assessment Plan revised
  - Architecture Document

Construction Phase

- Focus is on implementation of the design:
  - cumulative increase in functionality
  - greater depth of implementation
  - greater stability begins to appear
  - implement all details, not only those of central architectural value
  - analysis continues, but design and coding predominate

Transition Phase

- The transition phase consists of the transfer of the system to the user community
- It includes manufacturing, shipping, installation, training, technical support and maintenance
- Development team begins to shrink
- Control is moved to maintenance team
- Alpha, Beta, and final releases
- Software updates
- Integration with existing systems (legacy, existing versions, etc.)

Dynamic Elements Phases and Milestones

Major Milestones

Lifecycle Objectives

Lifecycle Architecture

Initial Operational Capability

Product Release

time

Inception	Elaboration	Construction	Transition
Define scope of project	Plan project, specify features, baseline architecture	Build product	Transition product to end user community