Software Design Methodologies and Testing

(Subject Code: 410449)
(Class: BE Computer Engineering)

2012 Pattern

Objectives and outcomes

- Course Objectives
  - To understand and apply different design methods and techniques
  - To understand architectural design and modeling
  - To understand and apply testing techniques
  - To implement design and testing using current tools and techniques in distributed, concurrent and parallel
  - Environments
- Course Outcomes
  - To present a survey on design techniques for software system
  - To present a design and model using UML for a given software system
  - To present a design of test cases and implement automated testing for client server, distributed, mobile applications

Other Information

- Teaching Scheme Lectures:
  - 3 Hrs/Week
- Examination Scheme
  - In Semester Assessment: 30
  - End Semester Assessment : 70

UNIT-I

Concepts

Introduction to software Design,
Design Methods: Procedural and Structural Design methods, Object Oriented design method, Unified modeling language overview,
Static and Dynamic Modeling -Advance Use case,
Class, State, Sequence Diagrams

UNIT-I

Concepts

Contents

- Introduction to software Design,
- Design Methods: Procedural and Structural Design methods,
- Object Oriented design method, Unified modelling Language overview,
- Static and Dynamic Modelling -Advance Use case,
- Class, State, Sequence Diagrams

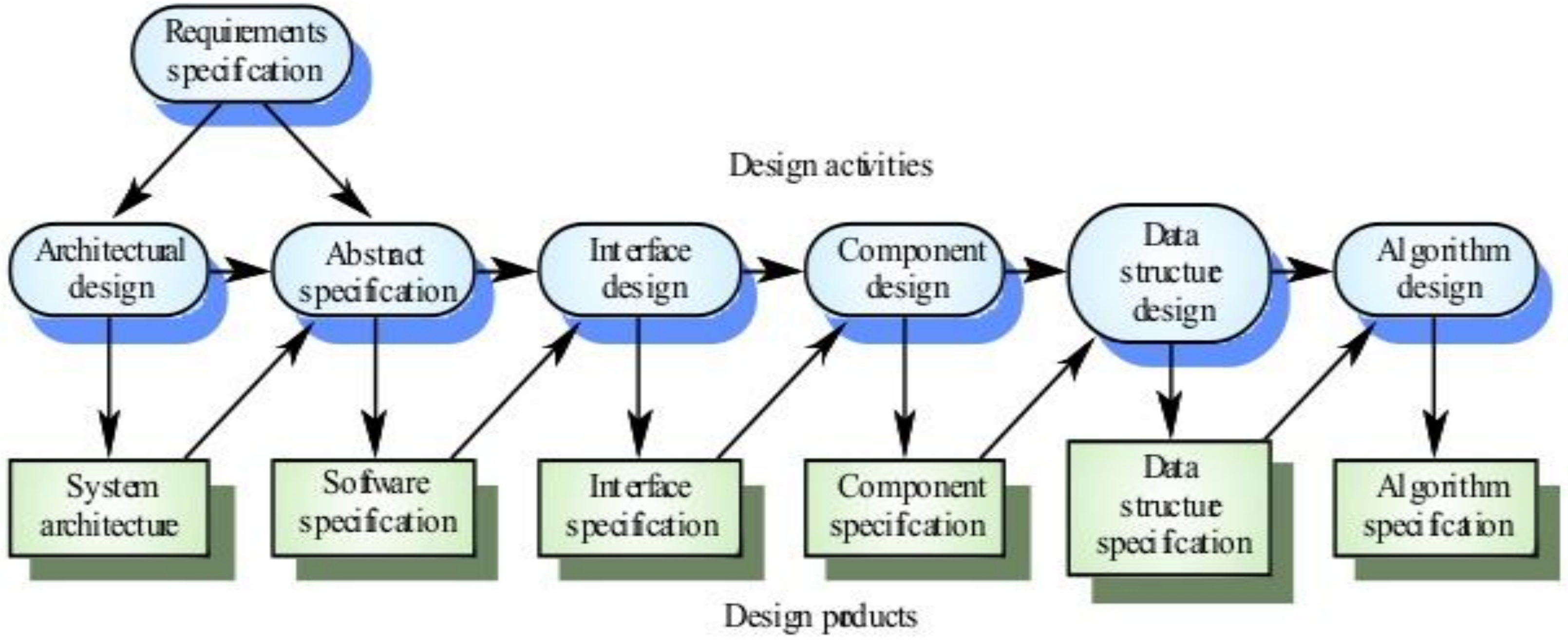
Introduction to Software Design (Cont…)

- Input of software design: Requirement analysis models and specification document
- Output of software design: Design models and design specification document
- Design-
  - translates the requirements into a completed design model for a software product.
  - provides the representations of software that can be assessed for quality.

Stages of Design

- Problem understanding
  - Look at the problem from different angles to discover the design requirements.
- Identify one or more solutions
  - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources.
- Describe solution abstractions
  - Use graphical, formal or other descriptive notations to describe the components of the design.
- Repeat process for each identified abstraction until the design is expressed in simple terms.

The software design process

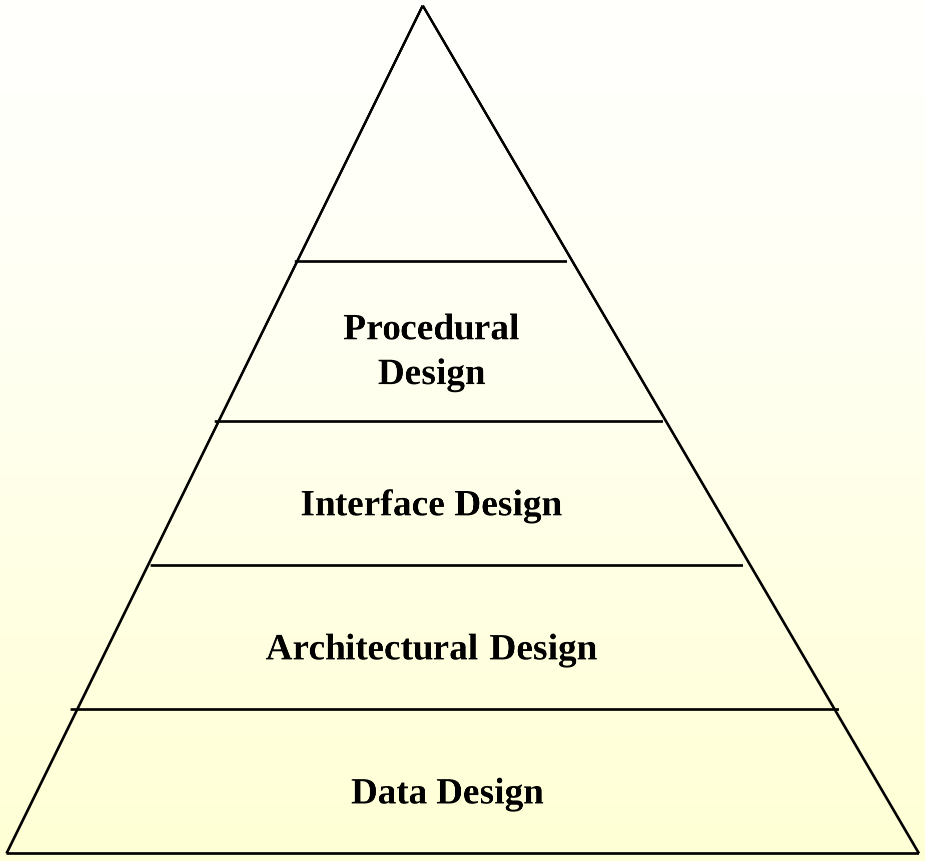


Software Design Process (Cont…)

- A number of design methods can be used to produce software design:
- Data design: transforms the information domain model into data structures.
- Architecture design: defines the relationship among major structural elements of the program.
- Interface design: describes how the software communicates with users.
- Procedure design: transforms structural elements of the program architecture into a procedural description of software components.

The Design Model

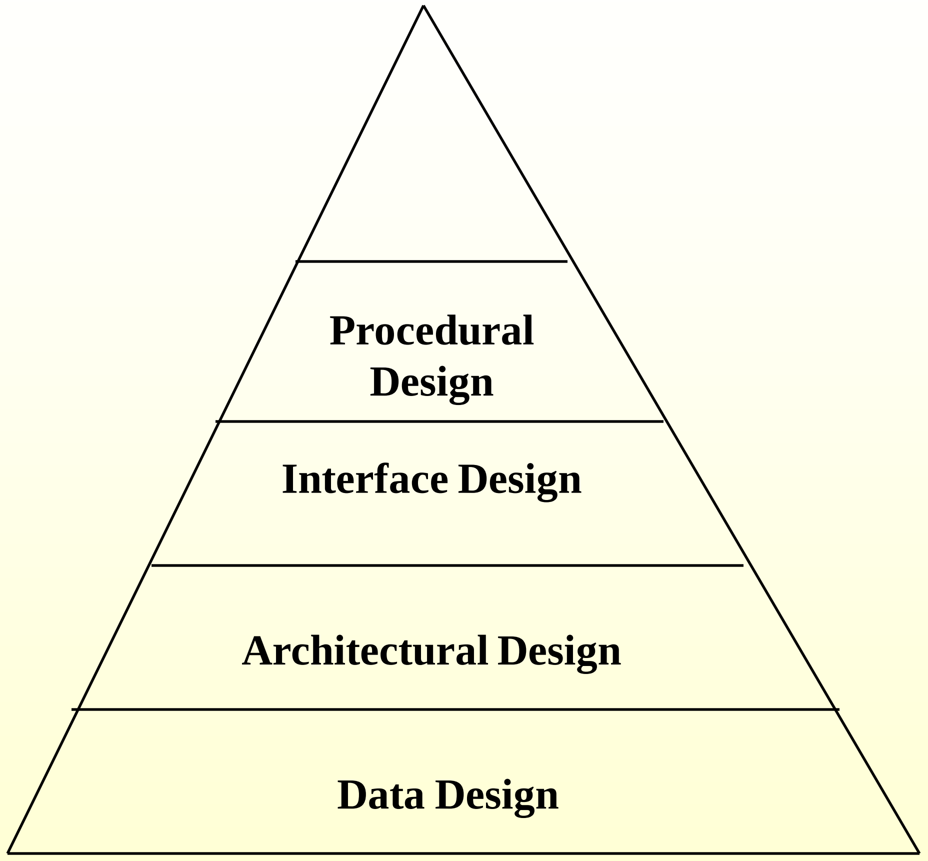
- Data Design
  - Transforms information domain model into data structures required to implement software
- Architectural Design
  - Defines relationship among the major structural elements of a program



The Design Model

The Design Model

- Interface Design
  - Describes how the software communicates with itself, to systems that interact with it and with humans.
- Procedural Design
  - Transforms structural elements of the architecture into a procedural description of software construction



The Design Model

software architecture

- A **software architecture** separates the overall structure of the system, in terms of components and their interconnections, from the internal details of the individual components.
- *programming-in-the-large*
- *programming-in-the-small*
- software architecture can be described at different levels
- The software quality attributes of a system should be considered when developing the software architecture

Features of Software Design

- Common features of software design methods:
- - A mechanism for translation of an analysis model into a design representation
- - A notation for representing functional components and their interfaces
- - Heuristics for refinement and partitioning
- - Guidelines for quality assessment

The Design Process

- Mc Glaughlin’s suggestions for good design:
  - Design must enable all requirements of the analysis model and implicit needs of the customer to be met
  - Design must be readable and an understandable guide for coders, testers and maintainers
  - The design should address the data, functional and behavioral domains of implementation

Design Guidelines

- A design should exhibit a hierarchical organization
- A design should be flexible
- A design should contain both data and procedural abstractions
- Modules should exhibit independent functional characteristics
- Interfaces should reduce complexity
- A design should be obtained from a repeatable method, driven by analysis

Design Methods: Procedural

- Procedural Design
- Transforms structural elements of the architecture into a procedural description of software construction
- A design methodology combines a systematic set of rules for creating a program design with diagramming tools needed to represent it.
- Procedural design is best used to model programs that have an obvious flow of data from input to output.
- It represents the architecture of a program as a set of interacting processes that pass data from one to another.

Design Methods: Procedural

- A **data flow diagram** (**DFD**) is a graphical representation of the "flow" of data through an information system, modelling its *process* aspects.
- A DFD is often used as a preliminary step to create an overview of the system, which can later be elaborated.

Data Flow Diagrams (DFD)

- DFDs describe the **flow of data** or information into and out of a system
  - ***what does the system do to the data?***
- A DFD is a graphic representation of the flow of data or information through a system

Object Oriented Design Methods

- Objectives:
- To explain how a software design may be represented as a set of interacting objects that manage their own state and operations
- To describe the activities in the object-oriented design process
- To introduce various models that can be used to describe an object-oriented design
- To show how the UML may be used to represent these models

Object-oriented development

- Object-oriented analysis, design and programming are related but distinct.
- OOA is concerned with developing an object model of the application domain.
- OOD is concerned with developing an object-oriented system model to implement requirements.
- OOP is concerned with realising an OOD using an OO programming language such as Java or C++.

Characteristics of OOD

- Objects are abstractions of real-world or system entities and manage themselves.
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services.
- Shared data areas are eliminated and Objects communicate by message passing.
- Objects may be distributed and may execute sequentially or in parallel.

Advantages of OOD

- Easier maintenance. Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects.

METHOD AND NOTATION

- A **software design notation** is a means of describing a software design either graphically or textually, or both. For example, class diagrams
- A **software design concept** is a fundamental idea that can be applied to designing a system. For example, information hiding is a software design concept.

Collaborative Object Modeling and Design Method, or COMET

- **A UML-BASED SOFTWARE MODELING AND DESIGN METHOD FOR SOFTWARE APPLICATIONS**
- uses the UML
- COMET is based on the design concepts of information hiding, classes, inheritance, and concurrent tasks
- COMET is an iterative use case driven and object-oriented software development method that addresses the requirements, analysis, and design modeling phases of the software development life cycle.

MULTIPLE VIEWS OF SOFTWARE ARCHITECTURE

- **Use case view**
    - **To develop s/w architecture**
- **Static view**
    - **Classes & relationships**
- **Dynamic interaction view**
    - **Objects & messages**
- **Dynamic state machine view**
    - **State machine**
- **Structural component view**
    - **components & interconnections**
- **Dynamic concurrent view**
    - Concurrent components executing on distributed nodes, and communicating by messages
- **Deployment view**
    - specific configuration of the distributed architecture with components assigned to hardware nodes

What is UML?

- Standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, business modeling and other non-software systems.
- The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.
- The UML is a very important part of developing object oriented software and the software development process.
- The UML uses mostly graphical notations to express the design of software projects.
- Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

Overview of UML Diagrams

Structural

: element of spec. irrespective of time

- Class
- Component
- Deployment
- Object
- *Composite structure*
- *Package*

Behavioral

: behavioral features of a system / business process

- Activity
- State machine
- Use case
- *Interaction*

Interaction

: emphasize object interaction

- Communication(collaberation)
- Sequence
- *Interaction overview*
- *Timing*

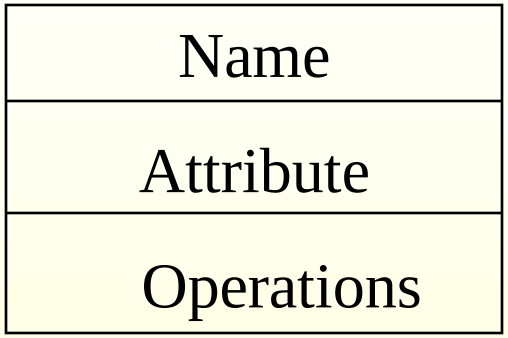
Structural Modeling

Classes

- Modeling a system involves identifying the things that are important to your particular view. These things form the vocabulary of the system you are modeling.
- In the UML, all the things are modeled as classes.
- A class is an abstraction of the things that are a part of your vocabulary. It is not a individual object, but represents a whole set of objects.
- Many programming languages directly support the concept of class.

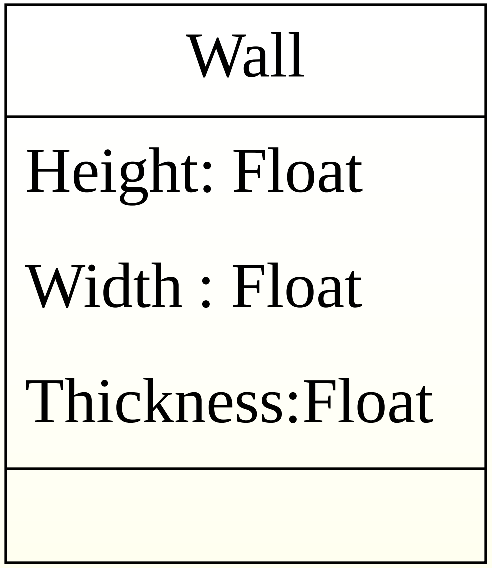
Classes

- The UML provides graphical representation of class.
-

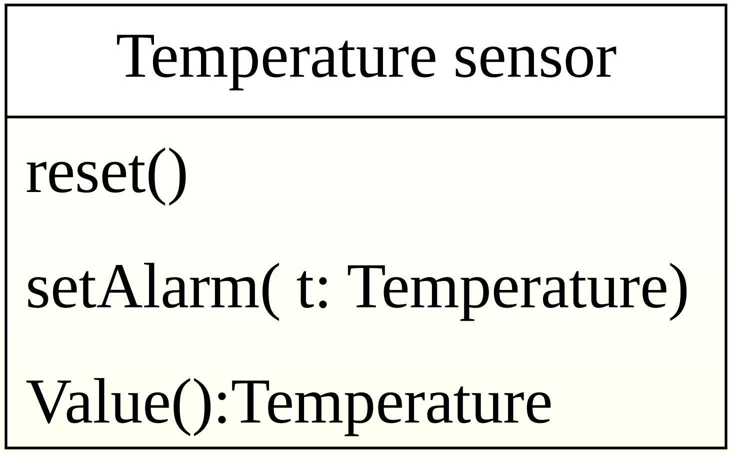


Terms and Concepts

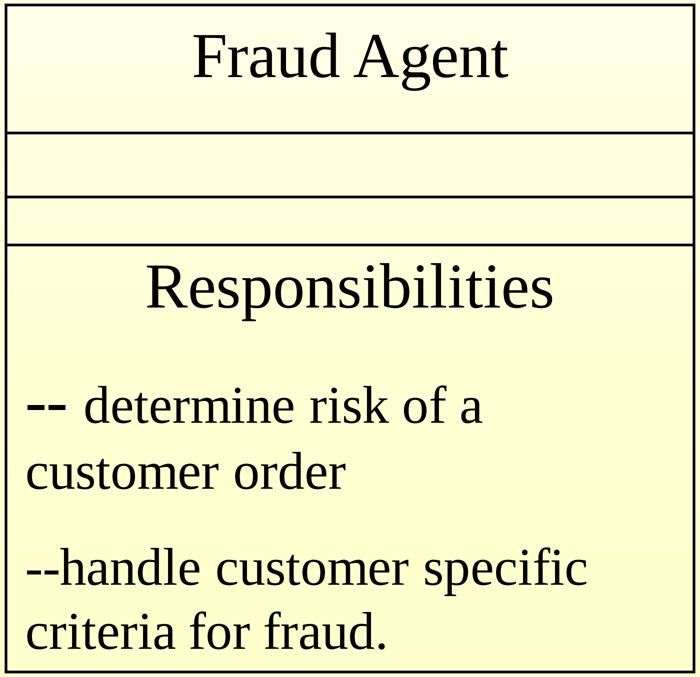
- A class is a description of a set of objects that share the same attributes, operation, relationships and semantics.
- Name: Every class must have a name that distinguishes it from other classes. It is called” Simple Name”
- Path Name: It is class name prefixed by the name of the package in which that class lives.
- Attributes: It is a named property of a class that describes a range of values that instance of the property may hold. Attribute represents some property of the thing you are modeling that is shared by all objects of that class.
- An attribute is an abstraction of the kind of data or state an object of the class might encompass.



Attributes and their classes



Operations and their signatures


Fraud Agent
Responsibilities
-- determine risk of a customer order
--handle customer specific criteria for fraud.

Responsibilities

Terms and Concepts

- Operations: It is the implementation of a service that can be requested from any object of the class to affect behavior. It is an abstraction of something you can do to an object and that is shared by all objects of that class.
- Responsibilities: It is a contract or an obligation (responsibility) of a class. All objects of the class have the same kind of state and behavior. These corresponding attributes and operations are just features by which these class’s responsibilities are carried out. When you model classes , a good starting point is to specify the responsibilities of the things in your vocabulary.
- Graphically , responsibilities can be drawn in a separate compartment at the bottom of the class icon.

Common Modeling Techniques

- To model the vocabulary:
  - Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case base analysis to help find these abstractions.
  - For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
  - Provide attributes and operations that are needed to carry out these responsibilities for each class.

Common Modeling Techniques

- Modeling the Distribution of Responsibilities in a System
  - Identify a set of classes that work together closely to carry out some behavior.
  - Identify a set of responsibilities for each of these classes.
  - Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
  - Consider the ways in which those classes collaborate with one another and redistribute their responsibilities accordingly so that no class with a collaboration does too much or too little.

Common Modeling Techniques

- Modeling Nonsoftware Things:
  - Model the thing you are abstracting as a class.
  - If you want to distinguish these things from the UML's defined building blocks, create new building block by using stereotypes to specify these new semantics and to give a distinctive visual sign.
  - If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well , so that you can further expand on its structure.

Common Modeling Techniques

- Modeling Primitive Types
  - Model the thing you are abstracting as a type or an enumeration , which is rendered using class notation with the appropriate stereotype.
  - If you need to specify the range of values associated with this type , use constraints.

Relationship

- In object oriented modeling, there are three kinds of relations that are especially important.
- Dependencies, which represent using relationships among classes
- Generalizations, which link generalized classes to their classes to their specialization.
- Associations, which represent structural relationship s provides a different way of combining your abstraction.

Terms and Concepts

- A relationship is a connection among things. In object oriented modeling, the three most important relationships are dependencies, generalizations and associations.
- Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.
- Dependency:
  - It is using relationship that state change in specification one thing may affect another thing that uses it.
  - Graphically, it is rendered as a dashed directed line, directed to the thing being dependant on.

- Generalization:
  - It is relationship between a general thing and more specific kind of that thing.
  - It is “ is-a-kind-of” relationship i.e. one is a kind of a more general thing.
  - It means that objects of child may be used anywhere the parent may appear . The child is substitutable for the parent. A child inherits properties of its parents . Child has attributes and operations in addition to those found in its parent.
  - An operation of a child that has the same signature as as an operation in a parent overrides the operation of parent. This is called as “Polymorphism”
  - Class may have 0,1 or more parents. A class that has no parents and 1 or more children is called “root class”. A class that has no children is called a “leaf class”.

Terms and Concepts

- Association :
  - It is a structural relationship that specifies that objects of one thing are connected to objects of another. You can navigate from an object of one class to an object of other class and vice versa.
  - Association which connects exactly two classes is called a "binary association" while associations which connects more than two classes are called "n-ary association."
  - An association rendered as a solid line connecting the same or different classes
  - Name: An association can have a name and you use that name to describe the nature of the relationship. Direction to the name can be provided by direction triangle that points in the direction you intend to read the name.

- Association :
  - Role: When a class participates in an association, it has a specific role that it plays in that relationship. It is just the face the class at the near end of the association presents to the class at other end of association.
  - Multiplicity: An association represents a structural relationship among objects. In many modeling situations, it is important for you to specify how many objects may be connected across an instance of an association. " How-many" is called the multiplicity of an association's role and is written as an expression that evaluates to a range of values or an explicit value.
  - You can show a multiplicity of exactly one(1), zero or one (0 .. 1), many (0..*) or one or more (1..*) or exact number.

- Association :
  - Aggregation: You will want to model a “whole/part” relationship, in which one class represents a larger thing, which consists of smaller things. This kind of relationship is called “ Aggregation”.
  - It represents a “has-a” relationship.
  - It is just a special kind of association and is specified by adorning a plain association with open diamond at the whole end.

Common Modeling Techniques

- Modeling simple Dependencies:
  - Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.
- Modeling Single Inheritance:
  - Given a set of classes , look for responsibilities, attributes and operations that are common to 2 or more classes.
  - Elevate these common responsibilities, attributes and operations to a more general class. If necessary create a new class to which you can assign these elements.
  - Specify that the more-specific classes inherit from the more general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.

Common Modeling Techniques

- Modeling Structural Relationships:
  - For each pair of classes, if you need to navigate from objects of another, specify an association between the two. The data driven view of association.
  - For each pair of of classes, if objects of one class need to interact with the objects of the other class other than as parameters to an operation , specify an association between the two.This is more behavioral view of association.
  - For each of these associations, specify a multiplicity, as well as role name.
  - If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole.

Common Mechanisms

- The UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language. i.e. Specifications, adornments, common divisions, extensibility mechanism.
- Notes are the important kind of adornment that stands alone.
- The UML’s extensibility mechanism permit you to extend the language in controlled ways which includes stereotypes, tagged values and constraints.

Terms and Concepts

- Note: It is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. Graphically, it is rendered as a rectangle with a dog-eared corner , together with a textual or graphical comment.
- A note that renders comment has no semantics impact.
- If your implementation allows, you can put live URL inside a note .
- Stereotypes: It is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to exiting one. But specific to your problem.
- It is rendered as a name enclosed by guillemets and place above the name of another element.Stereotype is a meta type.

- With stereotype , you can add new things to the UML.
- Tagged Values: It is an extension of the properties of a UML, allowing you to create new information in that element's specification. Graphically, it is rendered as a string enclosed by brackets and placed before name of another element.
- Tagged values can add new properties.You can define tags for existing elements in UML or you can apply them to individual stereotype.
- Tagged value is not same as a class attribute.
- Constraints: It is an extension of the semantics of a UML element, allowing you to add new rules or to modify existing ones.. Graphically, it is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships.

Common Modeling Techniques

- Modeling Comment:
  - Put your comments as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
  - Remember that you can hide or make visible the elements of your model as you see fit. This means that you don’t have to make your comments visible everywhere the elements to which it is attached are visible.
  - If your comment is lengthy or involves something richer than plaintext, consider putting your comment in an external document and linking embedding that documents in a note attached to your model.
  - As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, and they are of historic interest- discard the others.

Common Modeling Techniques

- Modeling New Building Blocks:
  - Make sure there is not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are there's already some standard stereotype that will do what you want.
  - If you are convinced there's no other way to express these semantics, identify the primitive thing in UML that's most like what you want to model and define new stereotype for that thing.
  - Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.
  - If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype.

Common Modeling Techniques

- Modeling New Properties:
  - Make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there is already some standard tagged value that will do what you want.
  - If you are convinced there's no other way to express these semantics , add this new property to an individual element or a stereotype. The rules of generalization apply- tagged values defines for one kind of element apply to its children.

Common Modeling Techniques

- Modeling New Semantics:
  - Make sure there is not already a way to express what you want by using basic UML. If you have common modeling problem , chances are that there's already some standard constraint that will do what you want.
  - If you are convinced there is no other way to express these semantics, write your as text in a constraint and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
  - If you need to specify your semantics more precisely and formally, write your new semantics using OCL.

Diagrams

- Diagrams are the means by which you view these building blocks . It is a graphical representation of a set of elements, most often rendered as a connected graph of vertices and arcs. You use diagrams to visualize your system from different perspectives. Because no complex system can be understood in its entirely from only one perspective , the UML defines a number of diagrams so that you can focus on different aspects of your system independently.
- In the context of software , there are five complementary views that are most important in visualizing, specifying, constructing and documenting a software architecture.:
  - 1) Use Case View 2)Design View 3) Process View
  - 4) Implementation View 5) Deployment View

Terms and Concepts

- A system is a collection of subsystems organized to accomplish a purpose and describe by a set of models, possibly from diff. View points.
- A subsystem is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained elements.
- A model is a semantically closed abstraction of a system, meaning that it represents a complete and self consistent simplification of reality, create in order to better understand the system.
- In modeling real systems, no matter what the problem domain, you will fine yourself creating the same kind of diagrams, because they represent common views into models .

- The static part of a system can be view by using one of the four following diagrams.
  - Class Diagram
  - Object Diagram
  - Component Diagram
  - Deployment Diagram

- Additional five diagrams can be used to view the dynamic parts of the system.
  - Use case Diagram
  - Sequence Diagram
  - Collaboration Diagram
  - State chart Diagram
  - Activity Diagram

Structural Diagrams

- **Class Diagram:** It shows classes , interfaces and collaborations and their relationships. It illustrate the static design view of a system.
- **Object Diagram:** It shows a set of objects and their relationships. It illustrates data structures, the static snapshots of instances of the things found in class diagram.
- **Component Diagram:** It shows set of components and relationships. It illustrates the static implementation view of the system.
- **Deployment Diagram:** It shows set of nodes and their relationship. It illustrates the static deployment view of an architecture.

Behavioral Diagram

- Use Case Diagram: It shows a set of use cases and actors and their relationship. It illustrates the static use case view of a system.
- Interaction Diagram: It is the collective name given to sequence diagrams and collaboration diagrams
- Sequence Diagram: It is an interaction diagram that emphasizes on the time ordering of messages. It shows a set of objects and the message sent and received by those objects. It illustrates the dynamic view of a system.
- Collaboration Diagram: It is an interaction diagram that emphasizes the structural organization of objects that send and receive messages.It shows set of objects ,links among those objects and messages sent and received by those objects. It also illustrates the dynamic view of a system.

Behavioral Diagram

- State Chart Diagram: It shows state machine, consisting of states, transitions, events and activities. It illustrates the dynamic view of a system. They are important in modeling the behavior of an interface, class or collaboration. It emphasize the event ordered behavior of an object, which is especially useful in modeling reactive systems.
- Activity Diagram: It shows the flow from activity to activity within system. It shows a set of activities, the sequential or branching flow from activity to activity and objects that act and acted upon. It illustrates the dynamic view of a system. They are important in modeling the function of a system. Activity diagrams emphasize the flow of control among objects.

Class Diagrams

- These are the most common diagrams found in modeling object –oriented systems. A class diagram shows a set of classes, interfaces and collaborations and their relationships.
- Class diagrams are used to model the static design view of a system. These are also the foundation for a couple of related diagrams: component, deployment
- These are not only important for visualizing, specifying and documenting structural models, but also for constructing executable systems through forward and reverse engineering.
- It shows classes , interfaces and collaborations and their relationships.
- Graphically, it is a collection of vertices and arcs.

- Common Properties: A class diagram is a special kind of diagrams and shares the same common properties as do all other diagrams- a name and graphical content that are a projection into a model.
- Class diagrams commonly contain the following things:
  - Classes
  - Interfaces
  - Collaborations
  - Dependency, Generalization and association relationships
- Class diagram is used model the static view of a system. This view primarily supports the functional requirement of a system- the services the system should provide to its end users.
- Class diagram can be use in one of the three ways.
  - To model the vocabulary
  - To model simple collaboration
  - To model a logical database schema

Common Modeling Techniques

- **Modeling Simple Collaborations:**
  - Identify the mechanism you would like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from interaction of a society of classes, interfaces and others.
  - For each mechanism identify classes, interfaces and other collaborations that participate in this collaboration. Identify the relationships among these things.
  - Use scenario to walk through these things.
  - Be sure to populate these elements with their contents.

- Modeling a Logical Database Schema:
  - Identify those classes in your model whose state must transcend the life time of their applications.
  - Create class diagram that contains these classes and mark them as persistent.
  - Expand the structural details of these classes.
  - Watch for common patterns that complicate physical database design.
  - Consider also the behavior of these classes by expanding operations that are important for data access and data integrity.
  - Where possible, use tools to help you transform your logical design into a physical design.

Forward and Reverse Engineering

- Forward Engineering: It is the process of transforming a model into code through mapping to an implementation language. It results in a loss of information.
- To forward Engg. A class diagram:
  - Identify the rules for mapping to your implementation language.
  - Depending on the semantics of the languages you choose, you may have constrain your use of certain UML features.
  - Use tagged values to specify your target language.
  - Use tools to forward engineer your models.

- Reverse Engineering: It is the process of transforming code into a model through a mapping from a specific implementation language.
- To reverse engineer a class diagram:
  - Identify the rules for mapping from your implementation language.
  - Using tool, point to the code you would like to reverse engineer.
  - Using your tool, create class diagram by querying the model.

Interfaces , Types and Roles

- An interface is a collection of operations that are used to specify a service of a class or a component. We use interface to visualize, specify, construct and document the seams within our system.
- Types and roles provide a mechanism for us to model the static and dynamic conformance of an abstraction to an interface in a specific context.
- An interface is a collection of operations that are used to specify a service of a class or a component.
- A type is a stereotype of a class used to specify a domain of objects, together with operation.
- A role is a behavior of an entity participating in a particular context.

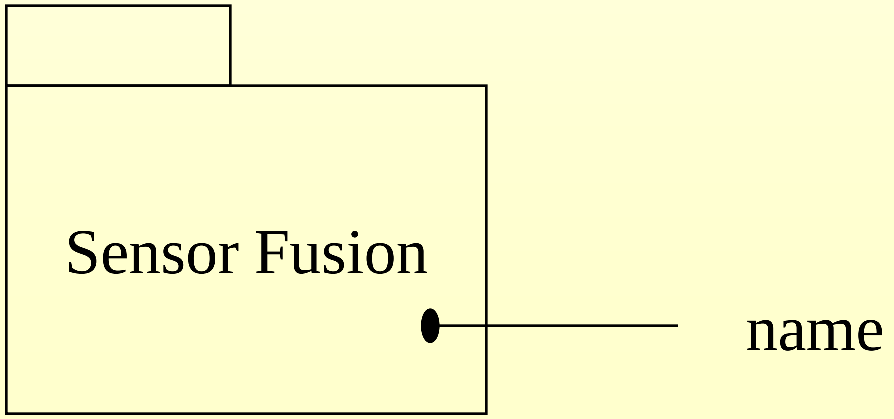
- Every interface must have a name that distinguishes it from other interfaces. A name is textual string. That name alone is known as a simple name; a path name is the interface prefixed by the name of the package in which that interface lives.
- Operations: An interface is a named collection of operations used to specify service of a class or component. An interface may have any number of operations. These may be adorned with visibility properties, concurrency properties, stereo type, tagged values and constraints.
- Relationship :An interface may participate in generalization, association and dependency relationships. It may participate in realization relationships. Realization is a semantic relationship between two classifiers in which one specifies a contract that another classifier guarantees to carry out.

Types and Roles

- A class may realize many interfaces. An instances of that class must therefore support all those interfaces, because an interface defines a contract, and any abstraction to that interface must carry out that contract. In a given context ,an instance may present only one or more of its interfaces as relevant. In that case, each interface represents a role that the object plays. A role names a behavior of an entity participating in a particular context.

Packages

- Visualizing, specifying, constructing and documenting large systems involves manipulating potentially large number of classes, interfaces, components, nodes, diagrams, and other element.
- In the UML, the package is a general purpose mechanism for organizing modeling elements into groups.
- We can use packages to arrange our modeling elements into larger chunks that can manipulate as a group.
- Graphically, a package is rendered as a tabbed folder.



- Names: A name is textual string. Name distinguish one package from other packages. That name alone is known as a simple name; a path name is the interface prefixed by the name of the package in which that package lives.
- Owned Element: A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams and even other packages.. Owing is a composite relationship, which means that elements is declared in the package . If the package is destroyed, the element is destroyed. Every element is uniquely owned by exactly one package.
- A package forms a namespace , which means that elements of the same kind of must be named uniquely within the context of its enclosing package.
- Different kinds of elements may have the same name. Elements of different kinds may have the same name within a package.

- Visibility:Typically, an element owned by a package is public, which means that is visible to the contents of any package that imports the elements enclosing package. Private elements can only be seen by children and private elements cannot be seen outside the package in which they are declared.
- Prefix + for public element
- Prefix – for protected or private element.
- Generalization: Two kinds of relationships between packages : import and access dependencies used to import into one package one elements export from another, and generalizations used to specify families of packages.

- Standard Elements: All of the UML's extensibility mechanisms apply to packages. We will use tagged values to add new package properties and stereotypes to specify new kinds of packages.
- Following are five standard stereotypes that apply to packages.

façade - Specifies a package that is only a view on some other package.

framework- Specifies a package mainly of patterns

stub- Specifies a package that serves as a proxy for the public contents of another package.

subsystem – Specifies a package representing an independent part of entire system being modeled.

system- Specifies a package representing the entire system being modeled.

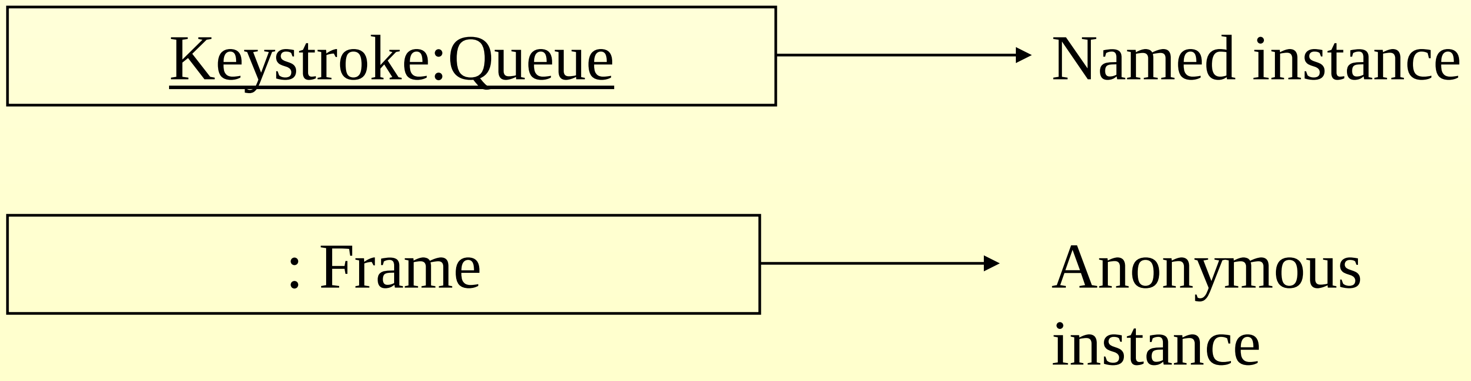
Common Modeling Techniques

- Modeling Groups of Elements:
  - Scan the modeling elements in particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
  - Surround each of these clumps in a package.
  - For each package, distinguish which elements should be accessible outside the package. Mark them public and all others protected or private.
  - Explicitly connect packages that build on others via import dependencies.
  - In the case of families of packages, connect specialized packages to their more general part via generalization.

- Modeling Architectural views:
  - Identify the set of architectural views that are significant in the problem Typically includes a design view, a process view, an implementation view, a deployment view and a use case view.
  - Place elements that are necessary and sufficient to visualize, specify, construct and document the semantics of each view into appropriate package.
  - As necessary, further group these elements into their own package.
  - There will typically dependencies across the elements in diff. Views.

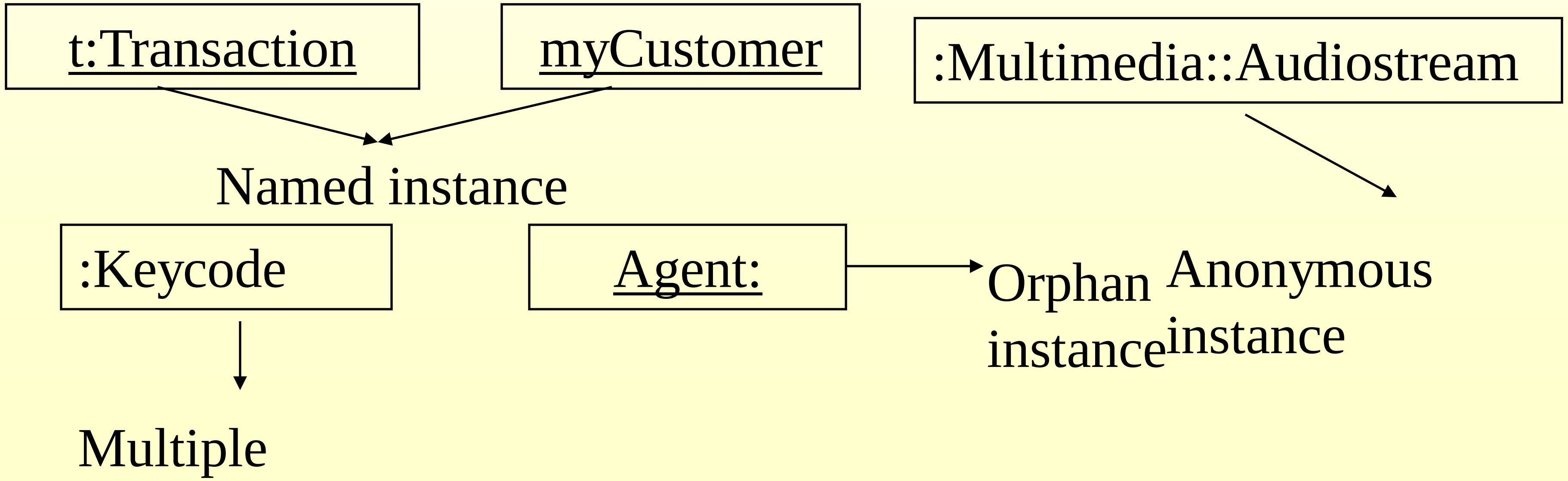
Instances

- An instance is a concrete manifestation of an abstraction to which set of operations may be applied and which may have a state that stores the effects of the operation
- Instance and objects are largely synonymous.
- Graphically instance is rendered by underlining its name.

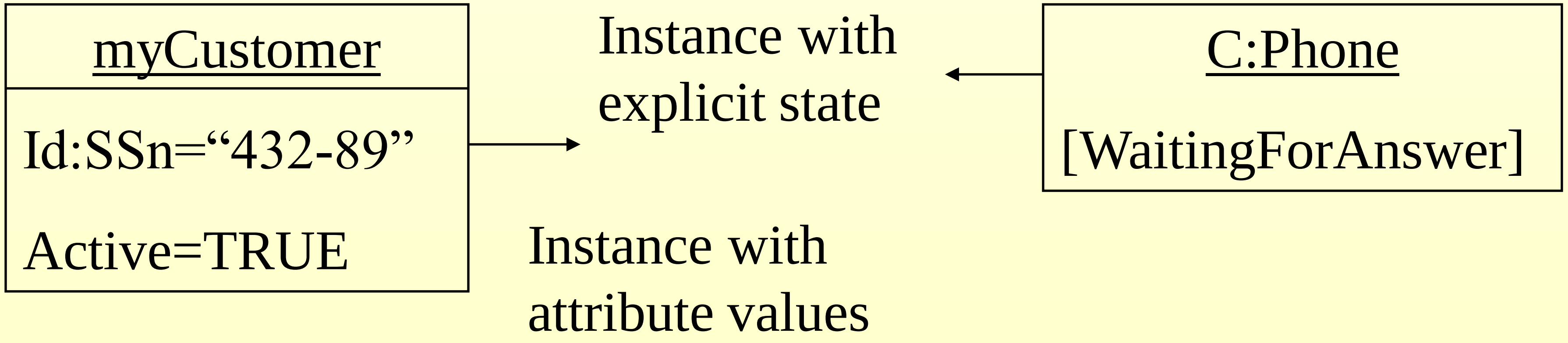


- Abstraction and Instances: Instances are almost always always tied to an abstraction.Most of the instances we model are of classes , but we can have instances of other things as components, associations etc.
- In UML, instances can be easily distinguishable from an abstraction. To, indicate instances we do underline to its name.
- We can use UML to model physical instances as well as things which are not so concrete. We can model indirect instances of abstract classes in order to show use of a prototypical instance of that abstract classes.
- The classifier of instance is usually classic.

- Names: Every instance must have a name that distinguishes it from other instances within its context. The name is textual string. An object lives within the context of an operation , a component or a node.
- The name alone is called “simple name”.
- The abstraction of the instance may be a simple name or it may be a path name which is the abstractions name prefixed by the name of the package in which abstraction lives.



- Operations: Not only an object takes up space in the real world , it also something you can do things to. The operations we can perform on an object are declared in the objects abstraction.
- State: An object also has state , which in this sense encompasses all the properties of the object plus the current values of each of these properties. These properties include attributes of the object , as well as all its aggregate parts.
- An object’s state is therefore dynamic.



- Standard Elements: We can not stereotype an instance directly , nor we can give it its own tagged values. An object's stereotype and tagged values derive from stereotype and tagged values of its associated abstraction.
- The UML define two standard stereotypes that apply to dependency relationship among objects and among classes.

instanceof	Specifies that client object is instance of supplier classifier.
instantiate	Specifies that client class creates instance of supplier class.

- The UML define two stereotypes related to objects that apply messages and transitions

become	specifies that the client is the same object as the supplier , but at later time and with possibly diff. Values, state, or roles.
copy	Specifies that client object is an exact but independent copy of the supplier

Common Modeling Techniques

- Modeling Concrete Instances:
  - Identify those instances necessary and sufficient to visualize, specify, construct, or document the problem.
  - Render these objects in the UML as instances. Where possible give each object a name.
  - Expose the stereotype, tagged values and attributes of each instance necessary and sufficient to model.
  - Render these instances and their relationships in an object diagram or other diagram appropriate to the kindof instances.

- Modeling Prototypical Instances:
  - Identify those Prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem.
  - Render these objects in the UML as instances. Where possible give each object a name.
  - Expose properties of each instance necessary and sufficient to model our problem.
  - Render these instances and their relationships in an interaction diagram or an activity diagram.

Object Diagrams

- Object diagram model the instances of things contained in class diagram. It shows a set of objects and their relationships at a point in time.
- Object diagrams are used to model the static design view or static process view of a system.
- Graphically, it is collection of vertices and arcs.
- An object diagram is a special kind of diagram and shares the same common properties as all other diagrams- as name, graphical contents that are a projection into a model.
- Object diagrams commonly contain
  - Objects
  - Links

It can have note and constraints. It may also contain packages or subsystems.

Common Modeling Techniques

- Modeling Object Structure:
  - Identify mechanism you would like to model. A mechanism represents some function or behavior of part of the system you are modeling that results from interaction of a society of classes , interfaces and others.
  - For each mechanism, identify the classes, interfaces and other elements that participate in this collaboration; identify the relationships among these things as well.
  - Consider one scenario that walks through this mechanism, Freeze it at a moment in time and render each object that participates in mechanism.
  - Expose the state and attribute values of each such object to understand the scenario.
  - Expose links among these objects, representing instances of associations among them.

Behavioral Modeling

Interactions

- An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose.
- We can model each interaction in two ways:
  - By emphasizing its time ordering of messages
  - By emphasizing its sequencing of messages in the context of some structural organization of object.
- Well structured interactions are like well structured algorithms-efficient, simple, adaptable and understandable.
- A message is a specification of a communication between objects that conveys information with the expectation that activity will proceed.

Terms & Concepts

- Context: We may find interaction whenever objects are linked to one another. We will find interactions in the collaboration of objects that exist in the context of our system or subsystem, in the context of an operation or in the context of class.
- Object & Roles: The objects that participate in an interaction are either concrete things or prototypical things.
- Links: A link is semantic connection among objects. Whenever there is link between two objects, one object can send a message to the other object. A link specifies path along which one object can dispatch a message to another

- We can adorn the appropriate end of the link with any of the following stereotypes.

  association: Specifies that the corresponding object is visible by association.

  self: Specifies that the corresponding object is visible because it is the dispatcher of the operation.

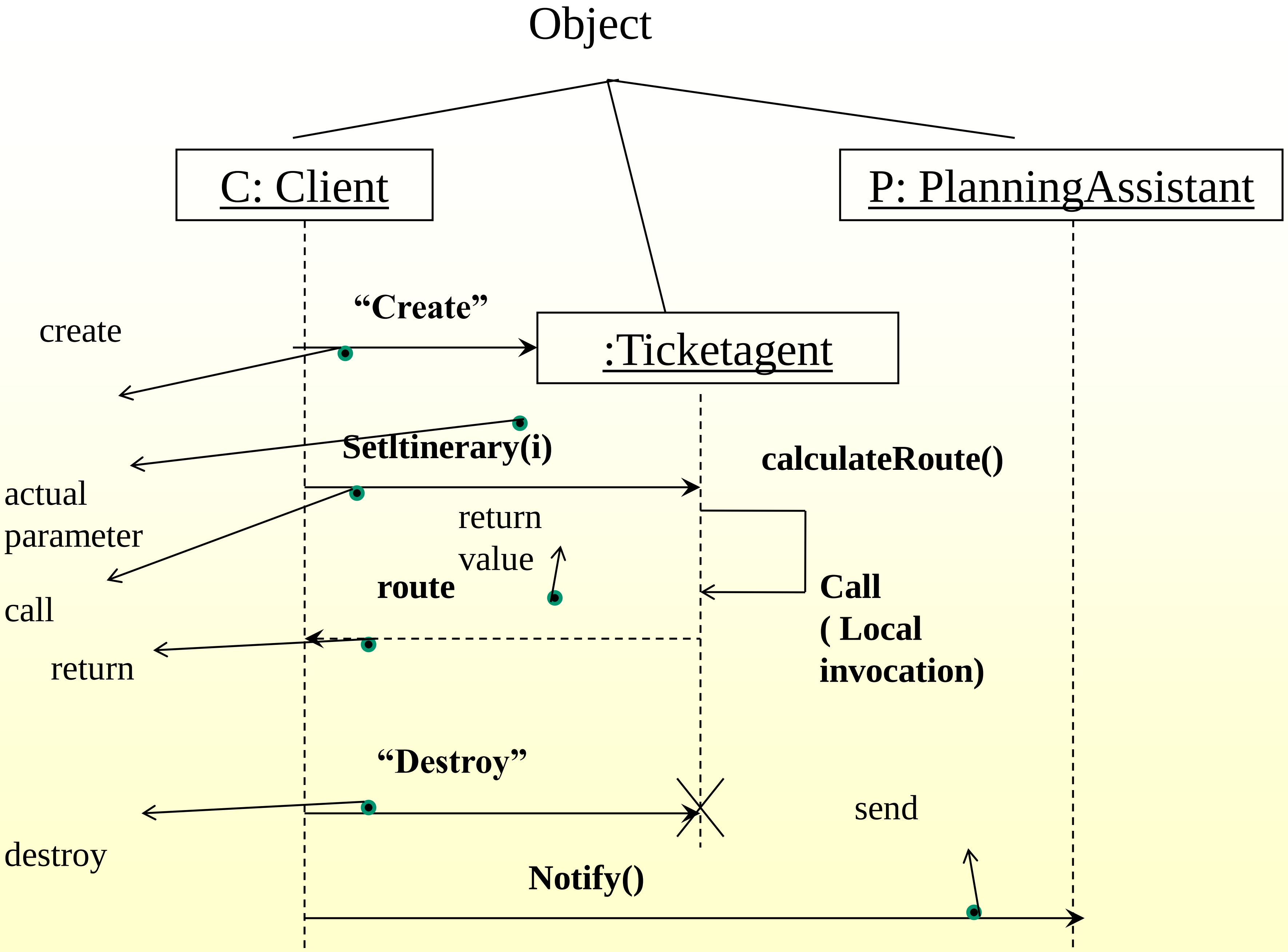
  global: Specifies that corresponding object is visible because it is in an enclosing scope.

  local : Specifies that the corresponding object is visible because it is in local scope.

  parameter: Specifies that the corresponding object is visible because it is a parameter.

- Message:
  - Object diagram models the state of society of objects at a given moment in time and are useful when want to visualize.
  - It is necessary to model the changing state of objects over a period of time.
  - Consider a motion picture set of objects , each frame representing a successive moment in time. Objects are passing messaging to other objects, sending events and invoking operations.
  - A message is the specification of a communication among objects that conveys information with the expectation that activity will ensure.
  - The receipt of a message instance may be considered an instance of an event.

- When you pass a message, the action that results is an executable statement that forms an abstraction of a computational procedure.
- Action may result in a change in state.
- In the UML, several kinds of actions can be modeled.
  - Call- Invokes operation on an object. Object can send message to itself, resulting local invocation of operation.
  - Return- return a value to caller.
  - Send- Sends signal to an object.
  - Create-Creates an object.
  - Destroy- Destroys an object.


Object
C: Client
P: PlanningAssistant
:Ticketagent
“Create”
create
SetItinerary(i)
calculateRoute()
actual parameter
return value
route
call
Call (Local invocation)
return
“Destroy”
send
destroy
Notify()

- Sequencing:
  - When object passes a message to another object , the receiving object might in turn send a message to another object, which might send a message to another object and so on.
  - This stream of message forms a sequence.
  - Any sequence have a beginning; start of every sequence is rooted in some process or thread.
  - Any sequence will continue as long as the process or thread that owns it lives.
  - A nonstop system, will continue to execute as long as the node it runs on is up.
  - Each process and thread in system defines distinct flow of control and within each flow, messages are ordered in sequence by time.

- To better visualize the sequence of a message , explicitly model the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator.

- Creation, Modification & Destruction:
  - Many times, the object participating in interaction exist for the entire duration of the interaction..
  - In some interactions, objects may be created and destroyed .
  - The same is true for links.
  - To specify if an object or link enters and /or leaves during an interaction, following constraint can be attached
    - **New-** specifies that the instance or link is created during execution of the enclosing interaction

    - **Destroyed-**specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
    - **transient -**specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution.

- Representation:
  - Objects and messages involved in an interaction in two ways
    - By emphasizing the time ordering of its messages- **sequence diagram**
    - By emphasizing the structural organization of the objects that send and receive the messages- **collaboration diagram**

- Common modeling techniques:
  - To model a flow of control
  - Set the context for the interaction, whether it is the system as a whole , a class or an individual operation.
  - Set the stage for interaction by identifying which object play a role. Set their initial properties, including their attribute values , state and role.
  - If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints.
  - In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and returns values to convey the necessary detail of interaction.
  - Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its sate and role

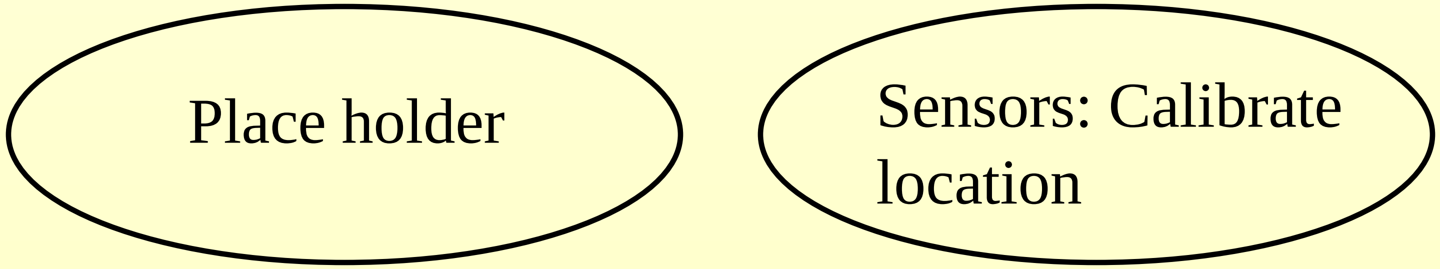
Use cases

- Every interesting system interacts with human or automated actors that use that system for some purpose.
- A use case specifies the behavior of a system or a part of a system and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.
- Use case are used to capture the intended behavior of the system developing, without having to specify how that behavior is implemented.

- Use cases provide a way for developers to come to a common understanding with the systems end users and domain experts.
- Use case can serve to help validate architecture of system and to verify system as it evolves during development.

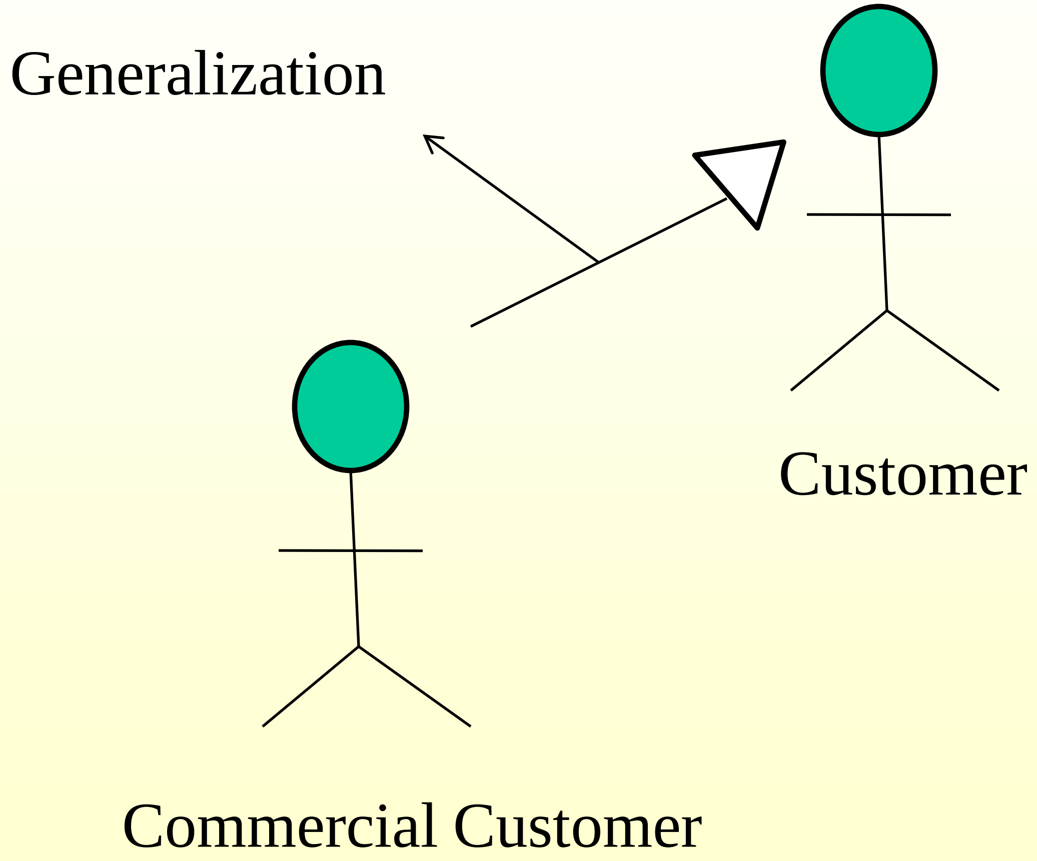
Terms and concepts

- A use case is a description of a set of sequences of action, including variants, that a system performs to yield an observable result of a value to an actor.
- Names: Every use case must have name that distinguishes it from other use cases. A name is textual string. A name alone is **simple name**; a path name the use case name prefixed by the name of the package in which that use case lives.



- Use cases and Actors:
  - An actor represents a coherent set of roles that users of use cases play when interacting with use cases.
  - Actor represents a role that a human , hardware device or even another system plays with a system.
  - An instance of an actor represents an individual interacting with the system in a specific way.
  - Actors are rendered as stick figures. Define general kind of actors and specialize them using generalization relationship is possible.
  - Actors are connected to use cases only by association

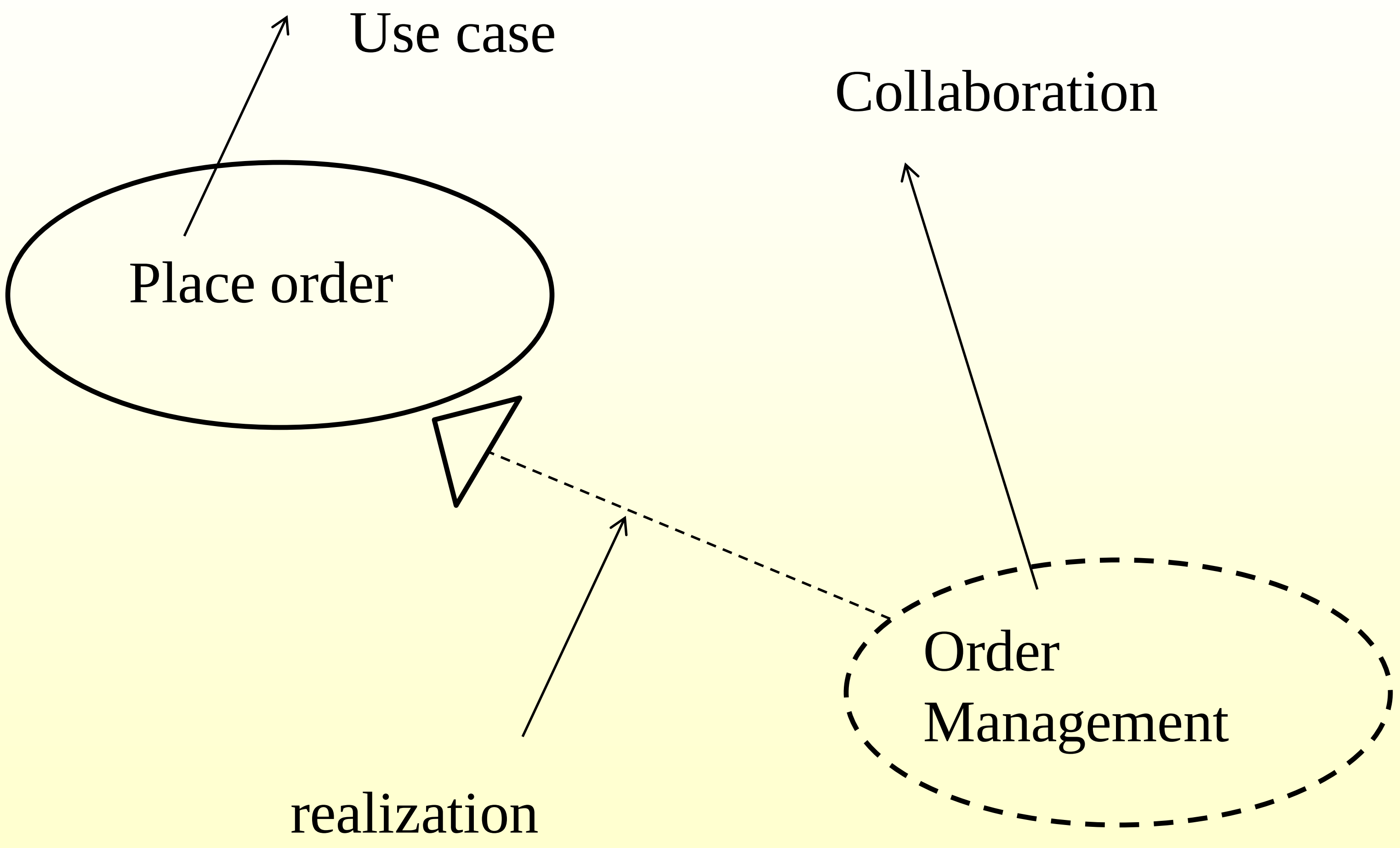
- An association indicates that the actor and use case communicate with one another, each one possibly sending and receiving messages.



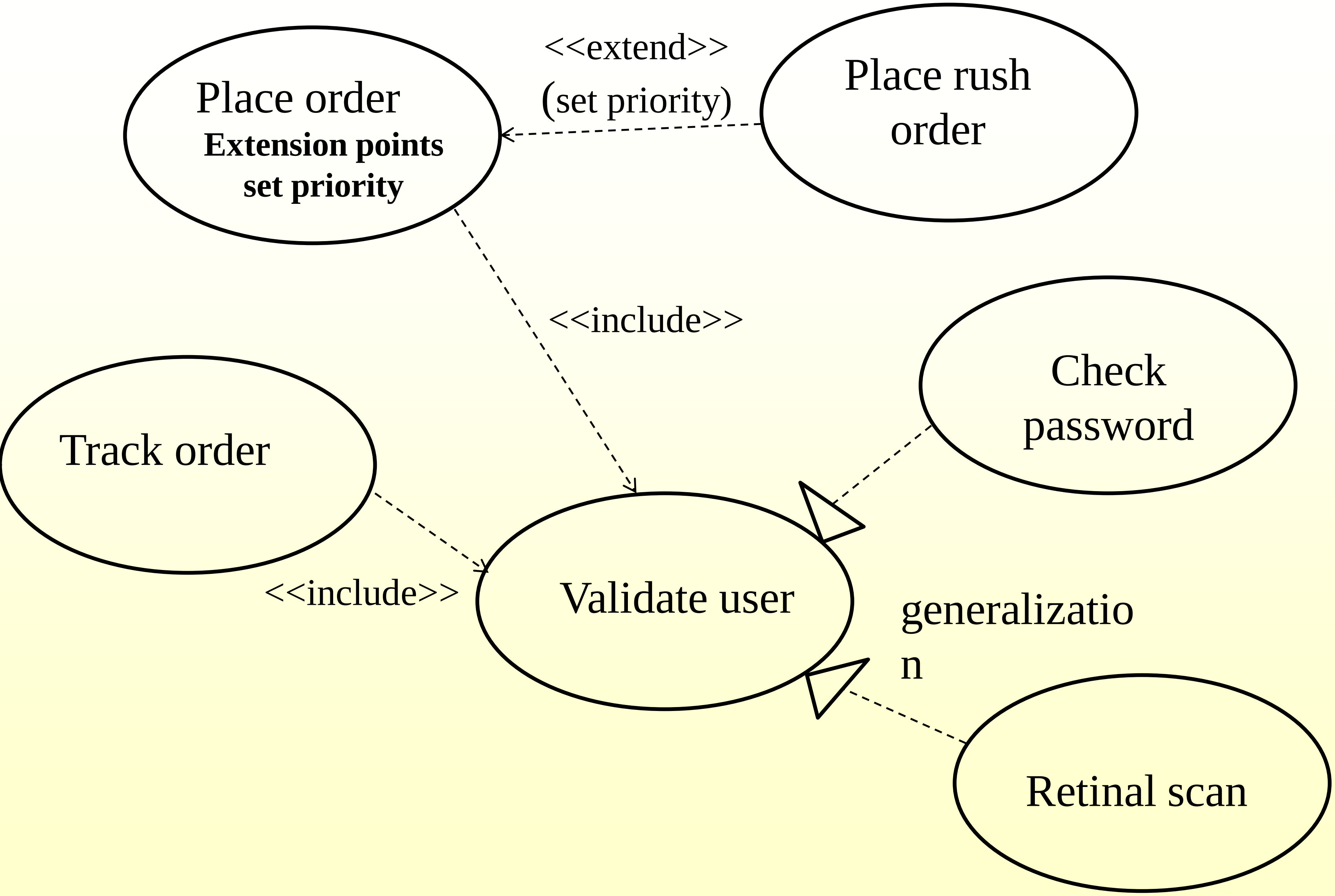
- Use case and Flow of events:
  - A use case describes what a system does but it does not specify how it does it.
  - The behavior of use case can be specified by describing a flow of events in text clearly enough for an outsider to understand it easily.
  - While writing flow of events, include how and when the use case starts and ends, when the use case interacts with actors and what objects are exchanged and the basic flow and alternative flows of the behavior.

- Use case and Scenarios:
  - It is necessary to use interaction diagrams to specify the flow of events graphically.
  - One sequence diagram to specify a use case’s main flow and variations of that diagram specify a use case’s exceptional flow.
- Use case and Collaboration:
  - A use case captures the intended behavior of the system we are developing, without having to specify how that behavior is implemented.
  - Use case are implemented by creating society of classes and other elements that work together to implement the behavior of use case.

- This society of elements including static and dynamic structure is modeled in the UML as a collaboration.



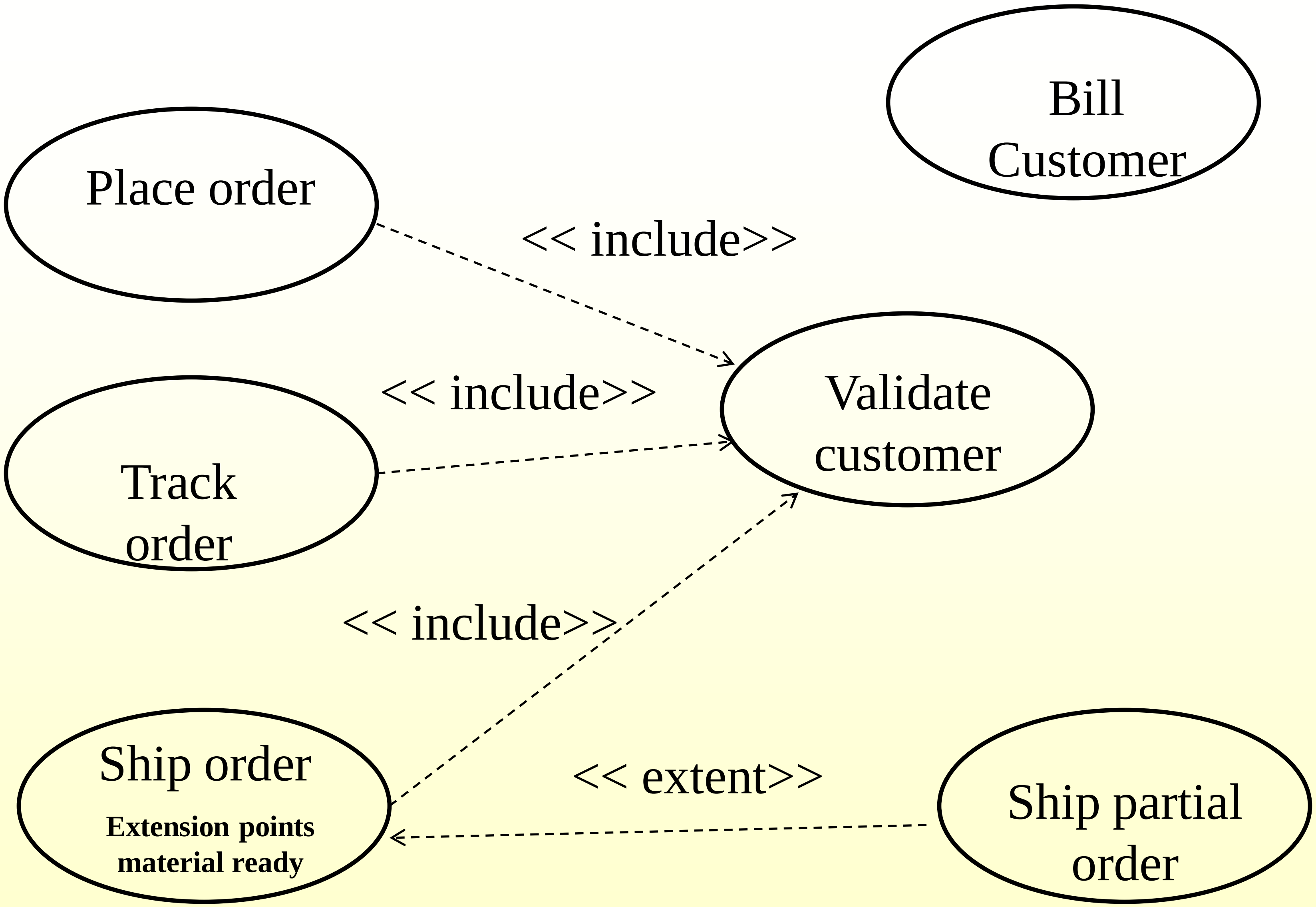
- Organizing Use Cases:
  - Use cases can be organized by grouping them in packages in the same manner as we organize classes.
  - Use cases can be also organized by specifying generalization , include and extend relationships among them
  - Apply these relationships in order to factor common behavior and in order to factor invariants.


Place order
Extension points
set priority
<<extend>>
(set priority)
Place rush order
<<include>>
Check password
Track order
<<include>>
Validate user
generalization
Retinal scan

Common Modeling Techniques

- Modeling the behavior of an element:
  - Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element’s function.
  - Organize actors by identifying general or more specialized roles.
  - For each actor consider the primary ways in which that actor interacts with the element. Consider also the change that state of the element or its environment or that involve a response to some event.

- Consider also the exceptional ways in which each actor interacts with element.
- Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior.


Bill Customer
Place order
<< include>>
Validate customer
<< include>>
Track order
<< include>>
Ship order
Extension points material ready
<< extent>>
Ship partial order

Use Case Diagrams

- The UML has 5 diagrams for modeling dynamic aspects of systems.
    - Use case diagrams
    - Activity diagrams
    - Statechart diagrams
    - Sequence diagrams
    - Collaboration diagrams
- Use case diagrams central to modeling the behavior of a system, a subsystem or a class.
- Each shows a set of use cases and actors and their relationship.

- Use case diagrams are applied to model the use case view of a system.
- Use case diagrams are important for visualizing, specifying and documenting the behavior of an element.
- They make systems, subsystems and classes approachable and understandable by presenting outside view of how those elements may be used in context.
- Use case diagrams are also important for testing executable systems through forward engineering and for comprehending executable systems through reverse engineering.

A use case diagram is a diagram that shows a set of use cases and actors and their relationships.

Common Properties

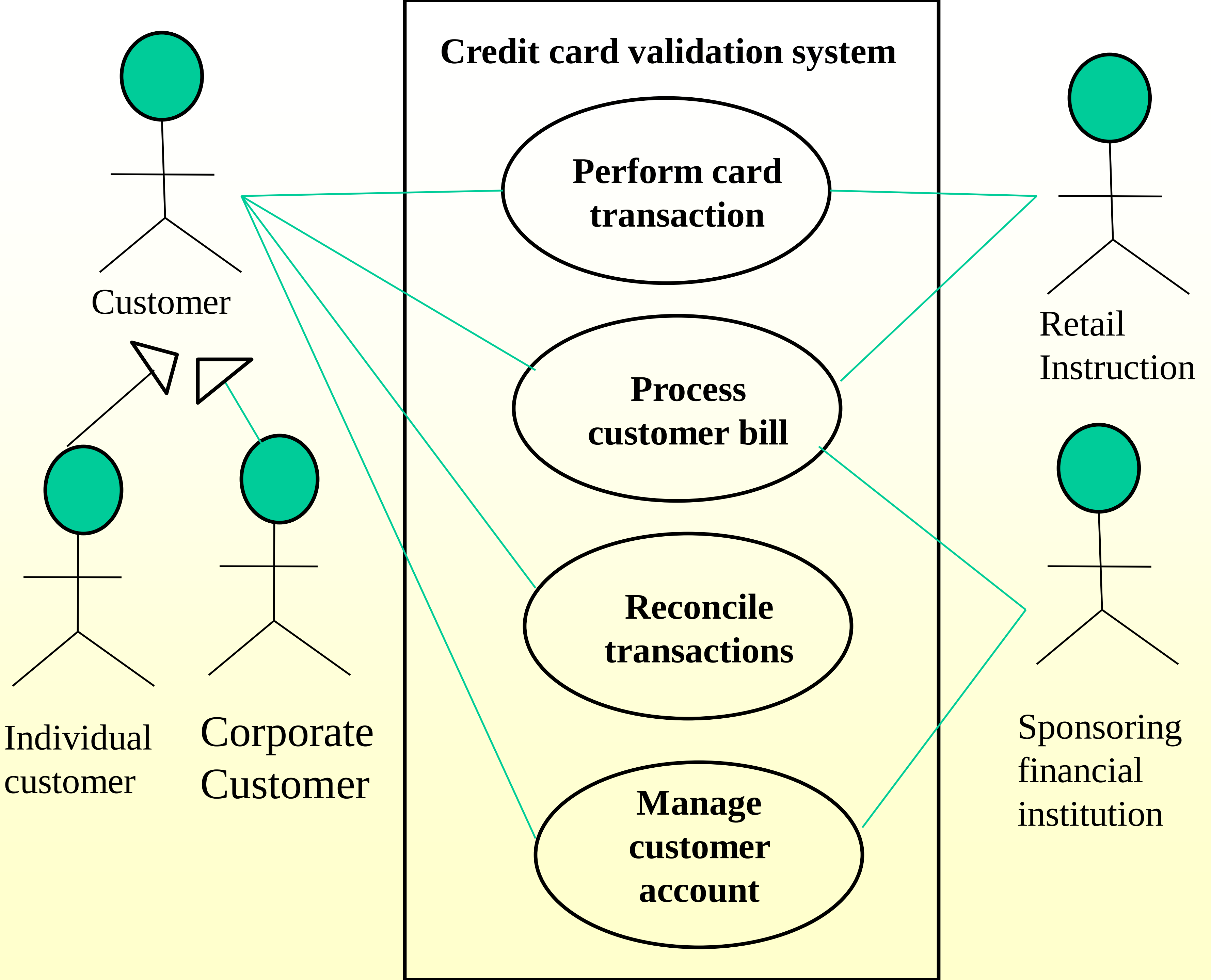
- A use case diagram is just a special kind of diagram and shares the same common properties as do all other diagrams- a name and graphical contents that are a projection into a model.
- Contents:
  - Use case diagrams commonly contain
    - Use cases
    - Actors
    - Dependency, generalization and association relationship
- Use case diagrams may contain notes and constraints

Common Properties

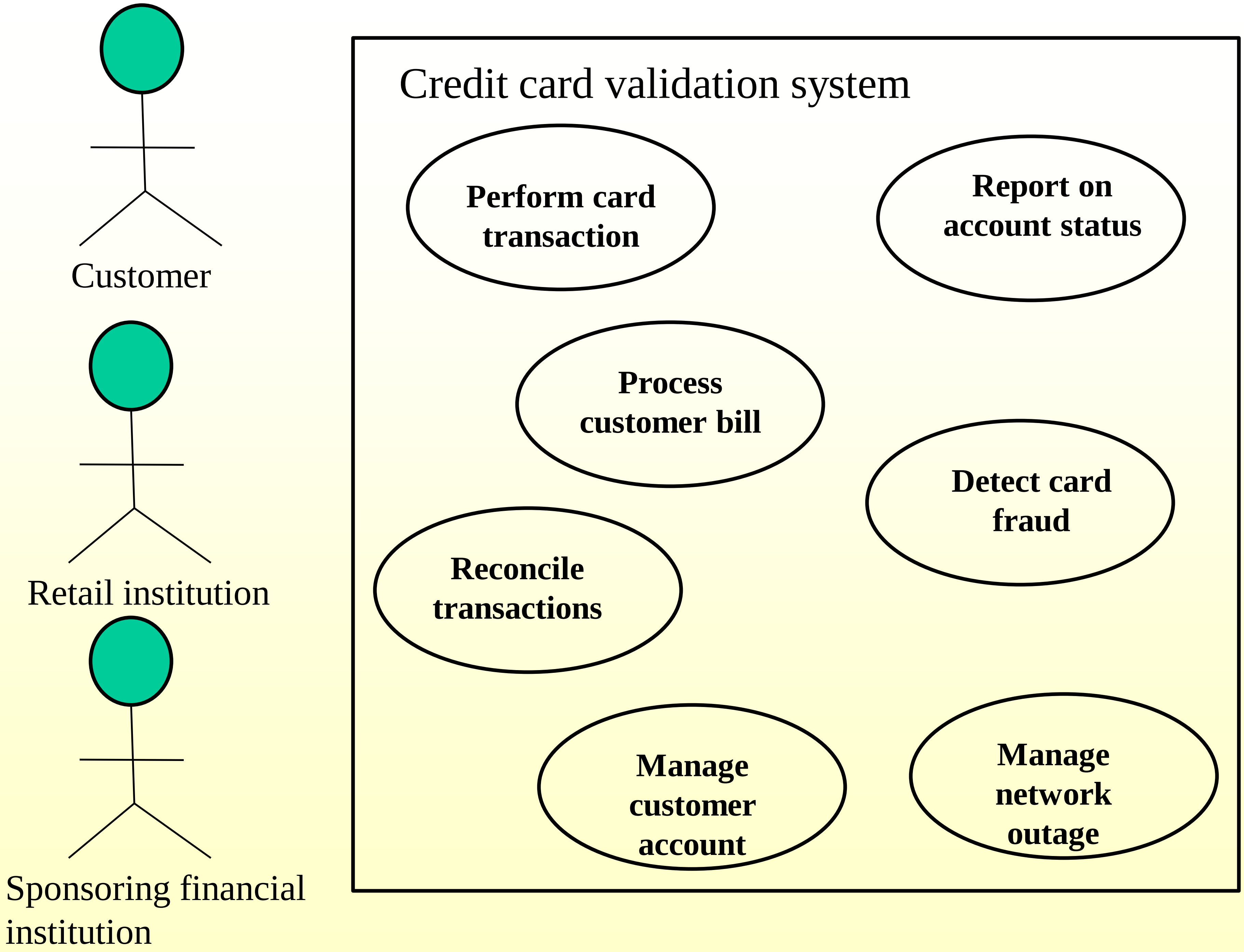
- Use case diagrams may also contain packages, which are used to group elements of model into larger chunks.
- Common Uses:
  - Use case diagrams can be applied to model the static use case view of a system
  - Use case diagrams can be applied in one of the two ways.
    - To model the context of a system
    - To model the requirements of a system

Common Modeling Techniques

- Modeling the context of a system:
  - Identify the actors that surrounds the system by considering which groups require help from these system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.
  - Organize actors that are similar to one another in a generalization hierarchy.
  - Where it aids understandability, provide stereotype for each such actor.
  - Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases.


Credit card validation system
Perform card transaction
Process customer bill
Reconcile transactions
Manage customer account
Customer
Individual customer
Corporate Customer
Retail Instruction
Sponsoring financial institution

- Modeling the requirements of a system:
  - Establish the context of the system by identifying the actors that surround it.
  - For each actor, consider the behavior that each expects or require the system to provide.
  - Name these common behaviors as use cases.
  - Factor common behavior into new cases that are used by others; factor variant behavior into new cases that extend more main line flows.
  - Model these use cases, actors and their relationships in a use case diagram.
  - Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system.


Customer
Retail institution
Sponsoring financial institution
Credit card validation system
Perform card transaction
Report on account status
Process customer bill
Detect card fraud
Reconcile transactions
Manage customer account
Manage network outage

Interaction diagrams

- **Sequence diagram** and **collaboration diagram** both are interaction diagrams.
- Interactions diagrams are used to model the dynamic aspects of a system.
- An interaction diagram shows an interaction , consisting of a set of objects and their relationships, including messages that may be dispatched among them.
- Sequence diagram emphasizes the time ordering of messages.
- Collaboration diagram emphasizes the structural organization of the objects that send and receive messages.

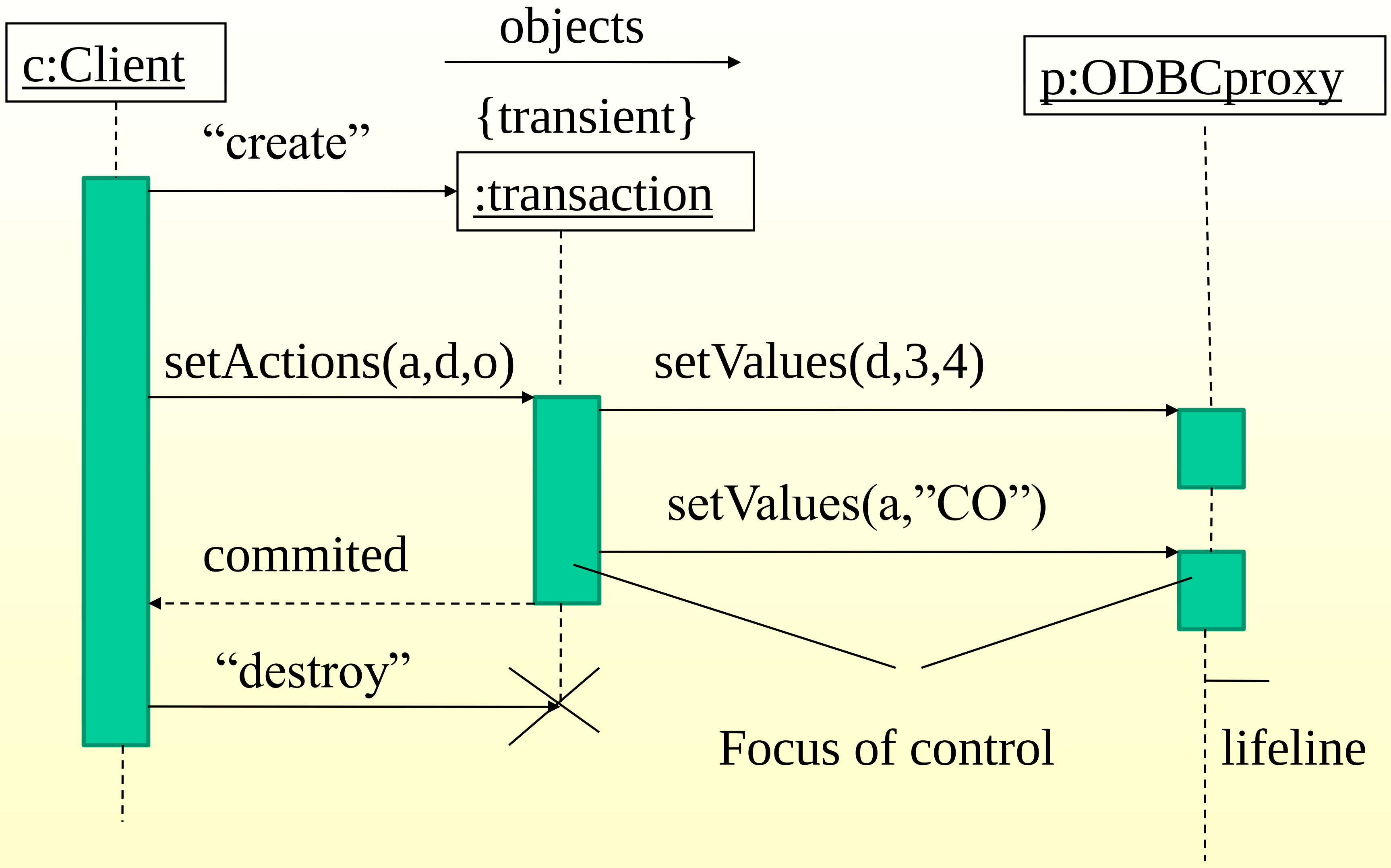
Terms and Concepts

- An interaction diagram shows an interaction , consisting of a set of objects and their relationships, including messages that may be dispatched among them.
- Sequence diagram emphasizes the time ordering of messages. Graphically , it is a table that shows objects arranged along the X-axis and messages ordered in increasing time.
- Collaboration diagram emphasizes the structural organization of the objects that send and receive messages. Graphically, it is a collection of vertices and arcs.

Common Properties

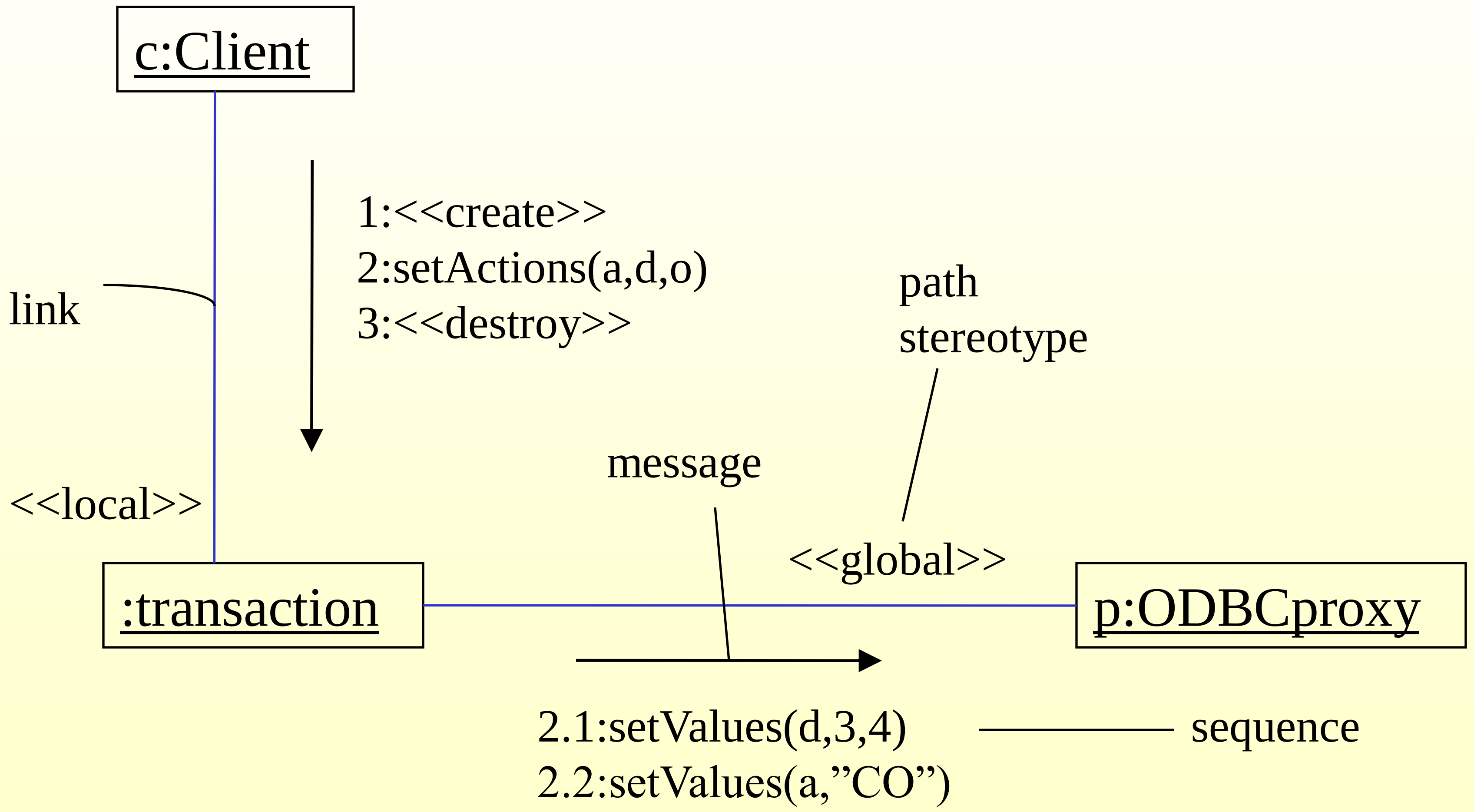
- An interaction diagram is just a special kind of diagram and shares the same common properties as do all other diagrams- a name and graphical contents that are a projection into a model.
- Contents:
  - Interaction diagram commonly contains
    - Objects
    - Links
    - Messages

- Sequence Diagram:
  - It emphasizes the time ordering of messages.



- Sequence diagram have two features that distinguish them form collaboration diagrams.
  - There is object lifeline. It is vertical dashed line that represents the existence of an object over a period of time.
  - There is the focus of control. It is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or though a subordinate procedure. The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion

- Collaboration diagram:
  - It emphasizes the organization of the objects that participates in an interaction.



- Collaboration Diagrams have two features that distinguish them from Sequence diagram:
  - There is path. To indicate how one object is linked with other object, a path stereotype can be attached to far end of link.
    - <<local>> -designated object is local to the sender
    - Need to render the path of the link explicitly for local, parameter, global and self path.
  - There is sequence number. To indicate the time order of a message, prefix the message with a number increasing monotonically for each new message in the flow of control. To show nesting decimal numbering can be used.

Activity Diagram

- It is one of the five diagrams in UML for modeling dynamic aspects of systems.
- An activity diagram is essentially a flow chart, showing flow of control from activity to activity.
- This involves modeling the sequential steps in a computational process.
- Also modeling of the flow of an object as it moves from state to state at different points in the flow of control.
- Activity diagrams may stand alone to visualize, specify, construct and document the dynamics or may be used to model the flow of control from object to object.

- Activity diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.

Terms and Concepts

- An activity diagram shows the flow from activity to activity.
- An activity is an outgoing nonatomic execution within state machine.
- Activities ultimately result in some action ,which is made up of executable atomic computations that result in a change in state of the system or the return of a value

- Actions encompass calling another operation, sending a signal, creating or destroying an object or some pure computations.
- Graphically, activity diagram is a collection of vertices and arcs.

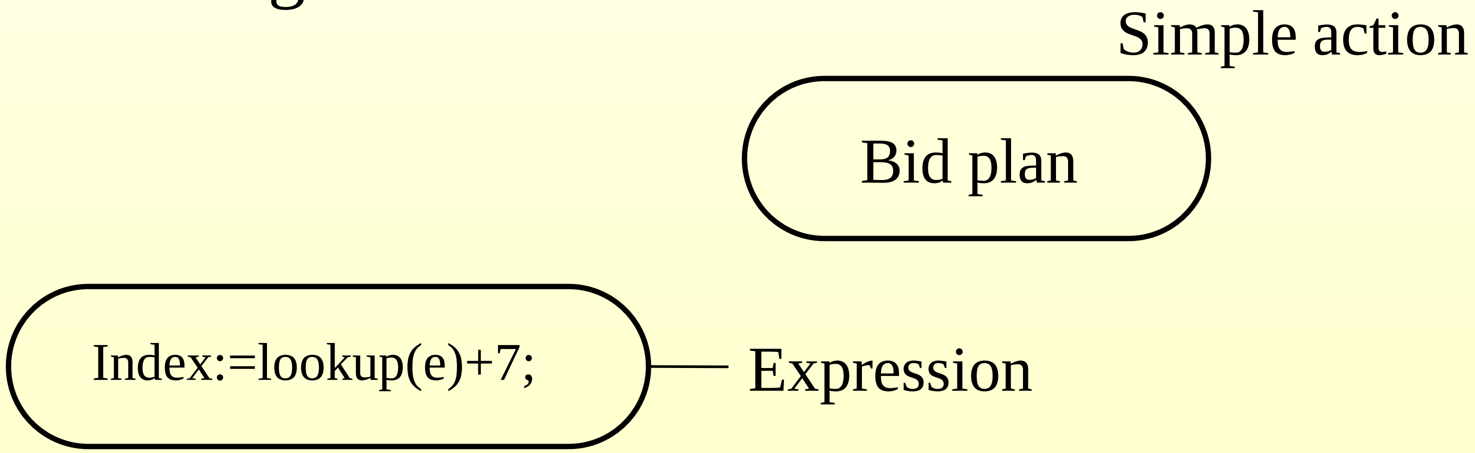
Common Properties:

- Activity diagram is just a special kind of diagrams and shares the same common properties like other diagrams- a name and graphical contents that are a projection in to model.

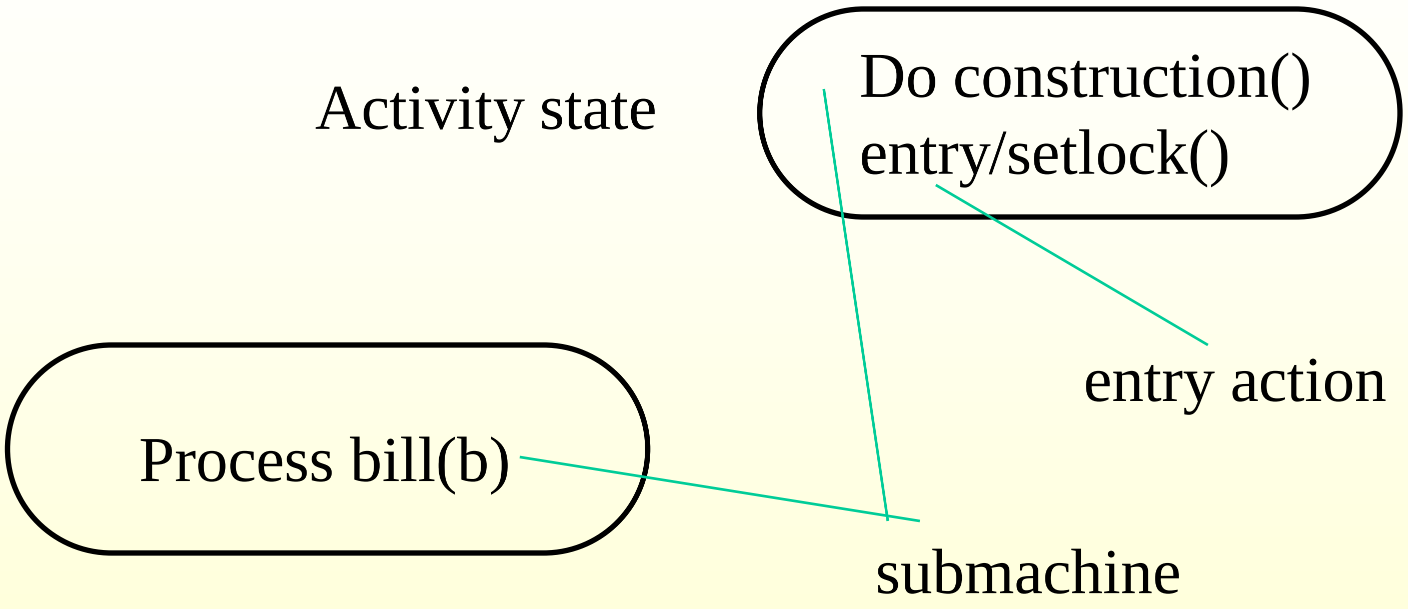
Contents:

- Activity diagrams commonly contain
  - Activity states and action states
  - Transitions
  - Objects
- **Action states and Activity states**
  - In the flow of control modeled by an activity diagram, things happen.
  - Some expressions can be evaluated that sets the value of an attribute or returns some value.
  - Alternatively, might cal an operation on an object and send a signal to object or create or destroy object.

- These executable atomic computations are called **actions states** because they are states of system, each representing the execution of an action
- Action states can't be decomposed.
- Actions are atomic means events may occur but the work of the action state is not interrupted.
- The work of an action state is generally considered to take insignificant execution time.

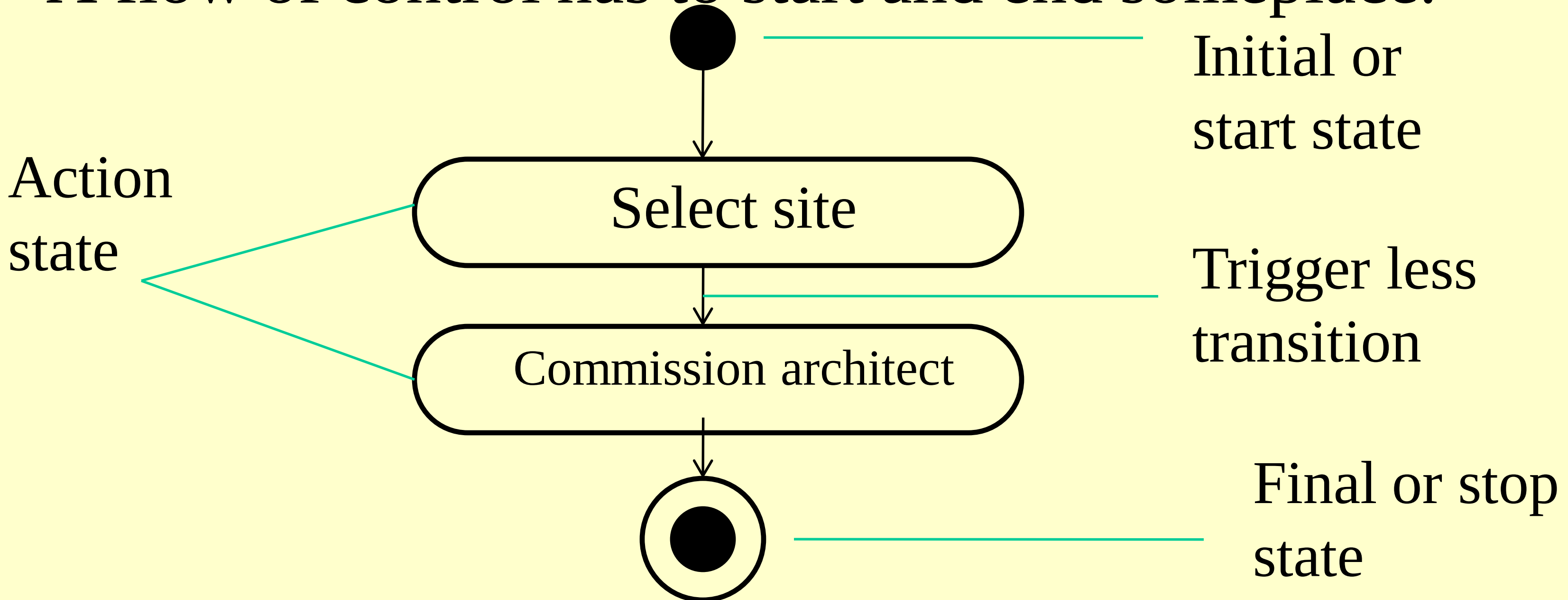


- Activity states can be further decomposed , their activity being represented by other activity diagram.
- Activity states are not atomic means they may be interrupted and are considered to take some duration to complete.
- Action state may be considered a special case of activity state.
- An action state is an activity state that cannot be further decomposed.
- Activity state can be a composite, whose flow of control is made up of other activity states.


Activity state
Do construction()
entry/setlock()
entry action
Process bill(b)
submachine

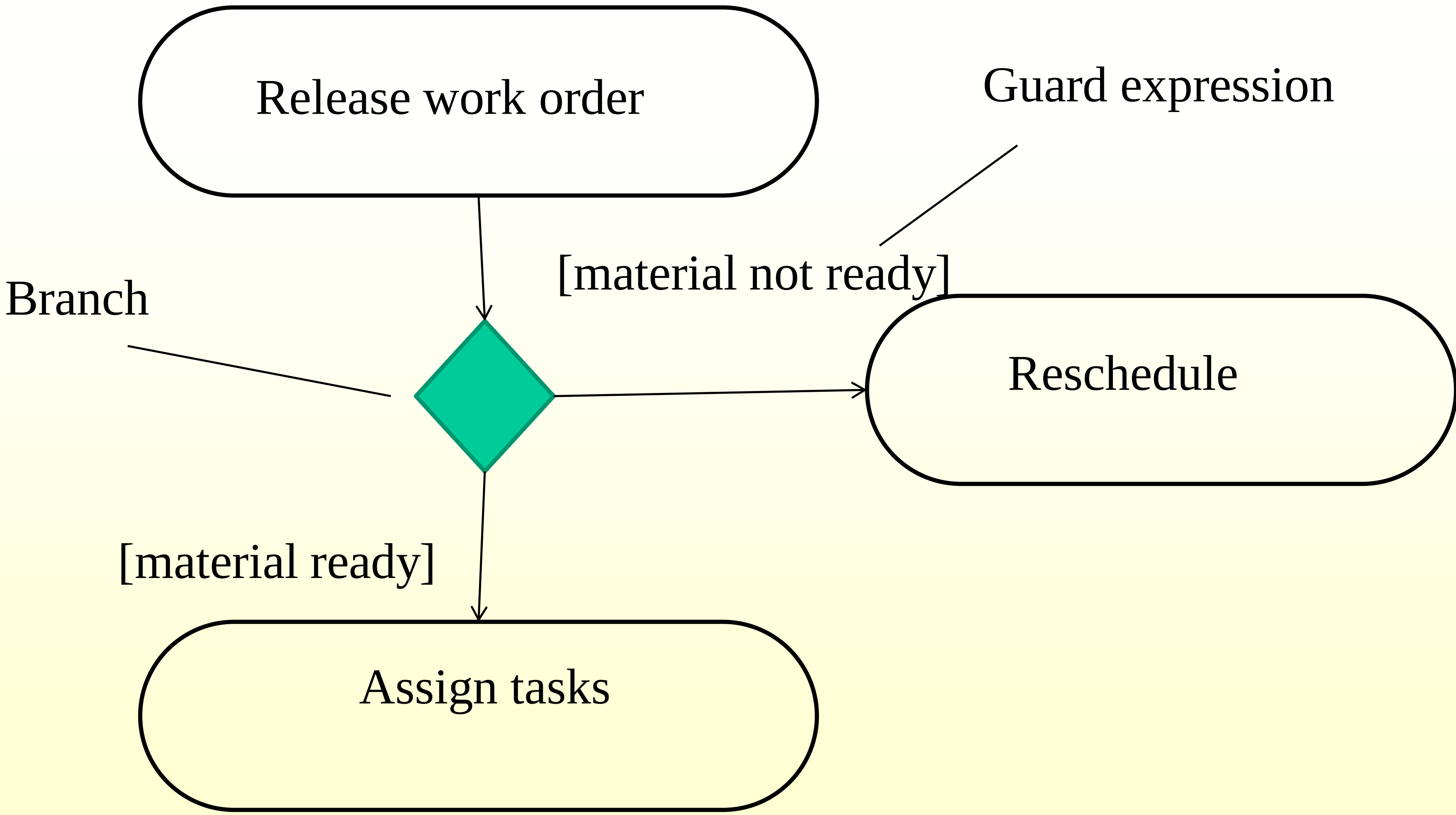
Transitions:

- When the action or activity of a state completes, flow of control passes immediately to the next action or activity state.
- This flow is represented by using transitions to show the path from one action or activity to next action or activity.
- A flow of control has to start and end someplace.



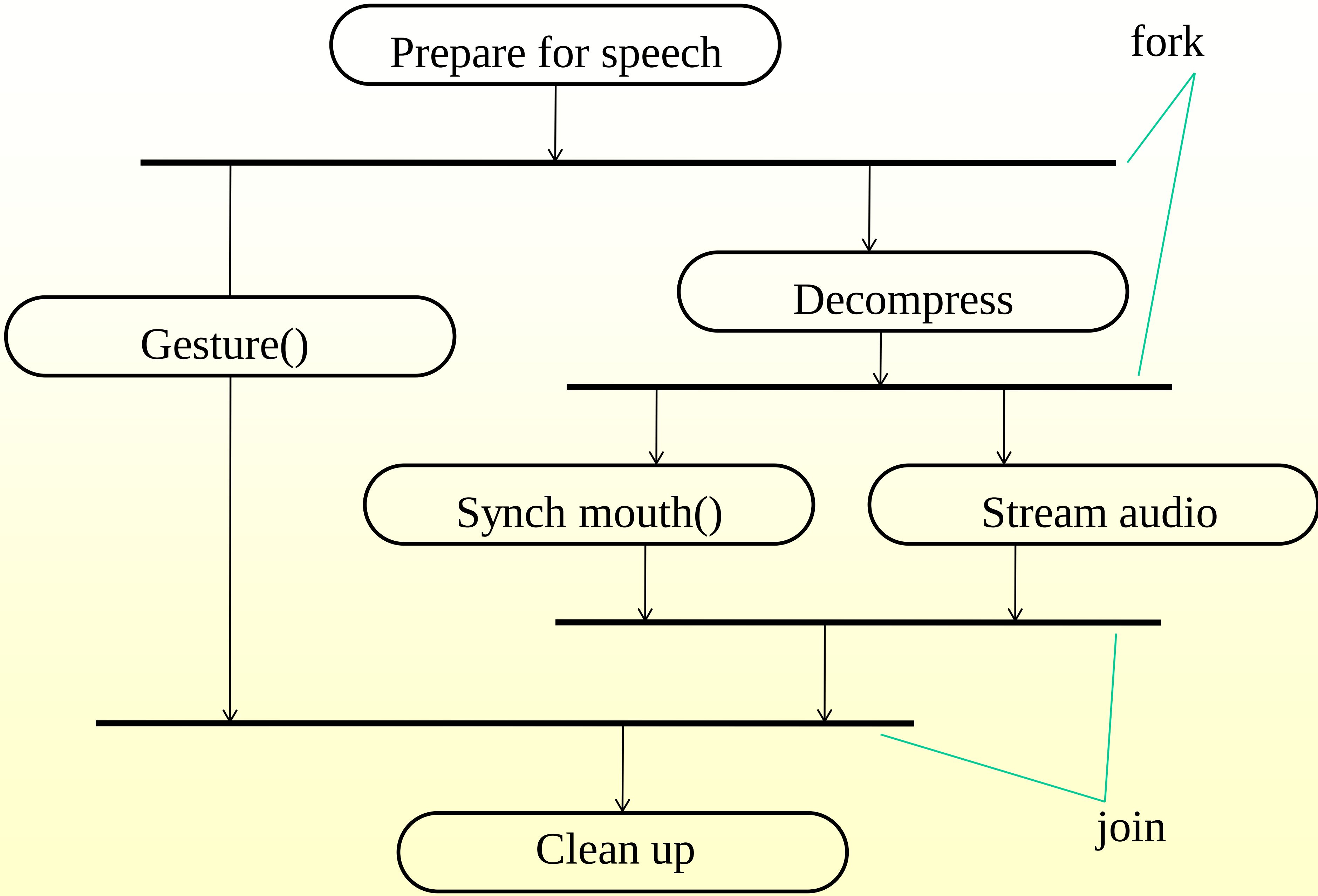
Branching:

- Sequential transition is common. As in flow chart, a branch can be included, which specifies alternate paths takes based on some Boolean expression.
- A branch is represented by diamond.
- A branch have one incoming transition and two or more outgoing ones. On each outgoing transition place a Boolean expression , which is evaluated only once on entering the branch.
- Across all these transitions, guards should not overlap.


Release work order
Guard expression
[material not ready]
Branch
Reschedule
[material ready]
Assign tasks

Forking and Joining:

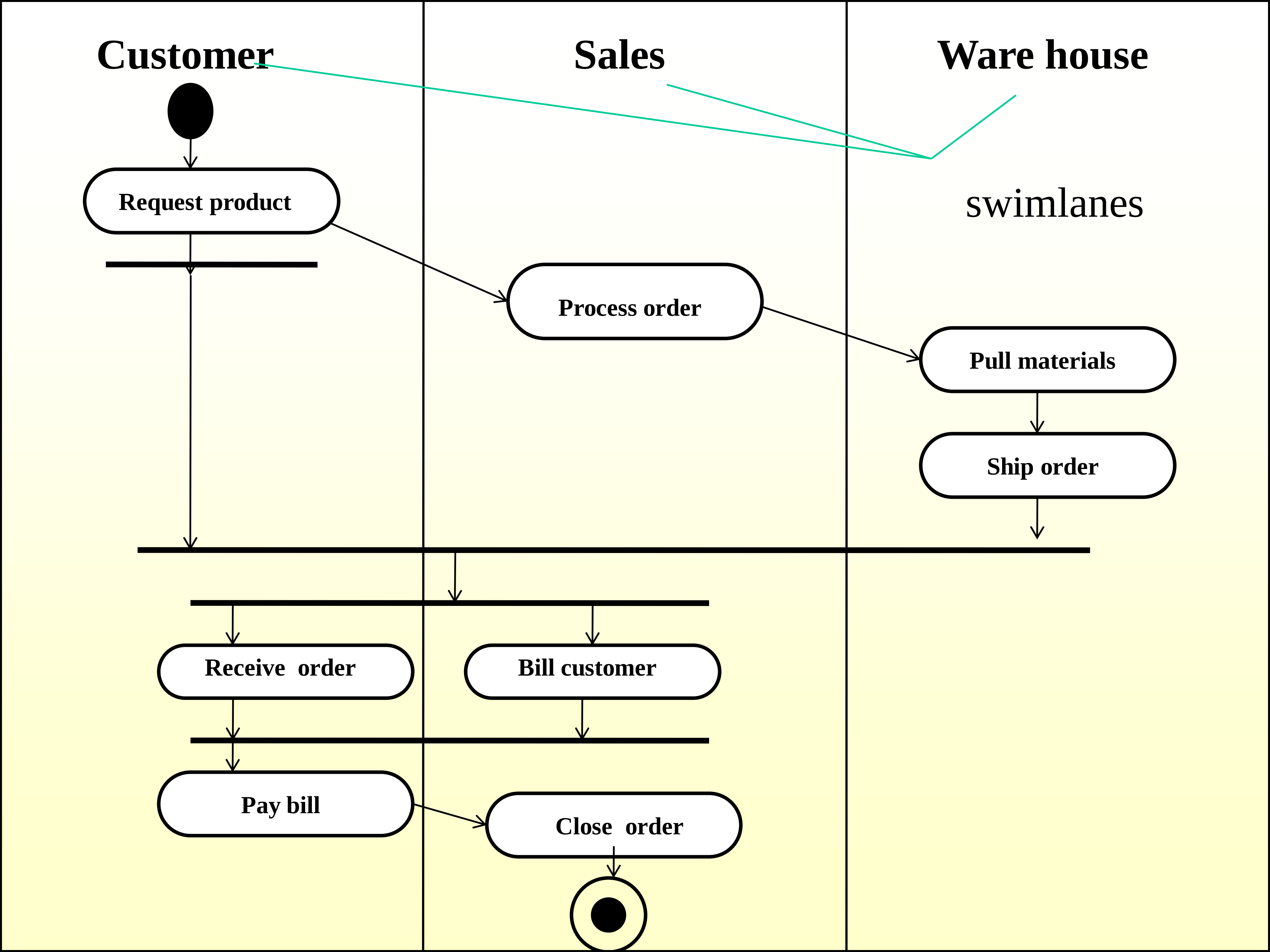
- Simple and branching sequential transitions are common.
- Sometimes while modeling flows of business process, concurrent flows can be encountered.
- In UML, a synchronization bar is used to specify the forking and joining of parallel flows of control.
- A synchronization bar is rendered as thick horizontal or vertical line.


Prepare for speech
fork
Decompress
Gesture()
Synch mouth()
Stream audio
join
Clean up

Swimlanes:

- When you are modeling workflows of business process to partition the activity states on an activity diagram into group, each group representing the business organization responsible for those activities swimlanes are useful.
- In the UML, each group is called swimlane because, visually each group is divided from its neighbor by vertical solid line.
- Swinlane specifies a locus of activities.
- Each swimlane has unique name within its diagram.
- It has no deep semantics.

- It represents high level responsibility for part of overall activity of an activity diagram.
- Each swimlane may eventually implemented by one or more classes.
- In an activity diagram partitioned into swinlanes, every activity belongs to exactly one swimlane but transitions may cross lanes.


Customer
Sales
Ware house
swimlanes
Request product
Process order
Pull materials
Ship order
Receive order
Bill customer
Pay bill
Close order

Advance Class Modeling

Advanced Object and Class Concepts

- Enumerations
- Multiplicity
- Scope
- Visibility
- Association Ends
  - Aggregation
  - Changability
  - Navigability
  - Visibility

Enumerations

- Datatype with a finite set of values
- Eg: Access Permissions of a file, different suits of the card game, different colours and ranks of the card game, colours of the rainbow etc..
- Do not use generalizations to capture the value of an enumerated attribute. Eg: dont introduce a generalization for a card, because most games dont differentiate the behaviour of suits, colours etc.
- You should introduce generalization only when there are significant attributes, operations or associations that do not apply to the super class.

Multiplicity

- It is a constraint on the cardinality of a set
- Used to specify for attributes also !!
- Eg: Person class as attributes :
  - Name : String [1]
  - Address : string [1..*]
  - PhoneNumber : string [*]
  - Birthdate : date [1]

Scope

- Scope indicates if a feature applies to a class or to an object
- Underline for an attribute distinguishes that attribute's scope is the class (static) and not object
- Such attributes are listed at the top of the attributes and operations of the class.
- Use an attribute with class scope to hold the extent of a class.
- They are used in OO Databases, but otherwise are discouraged, because they lead to an inferior model

Visibility

- Refers to the ability of a method to reference from another class.
- Possible values :- public, private, protected, package
- Methods of classes defined in the same package as the target class can access package features.
- UML denotes visibility with a prefix :
  - + precedes public
  - # precedes protected features
  - - precedes private
  - ~ precedes package features.

Visibility

- Issues to consider when choosing visibility
  - Comprehension : Understand all public features of the class to understand the capabilities of the class. Private, protected and package features can be ignored.
  - Extensibility : Many classes can depend on public methods, so it is highly disruptive to change their signature (change the parameters of public functions or remove a few public data members)
  - Context : The methods should be used within the context, otherwise incorrect results might be calculated.

Association Ends

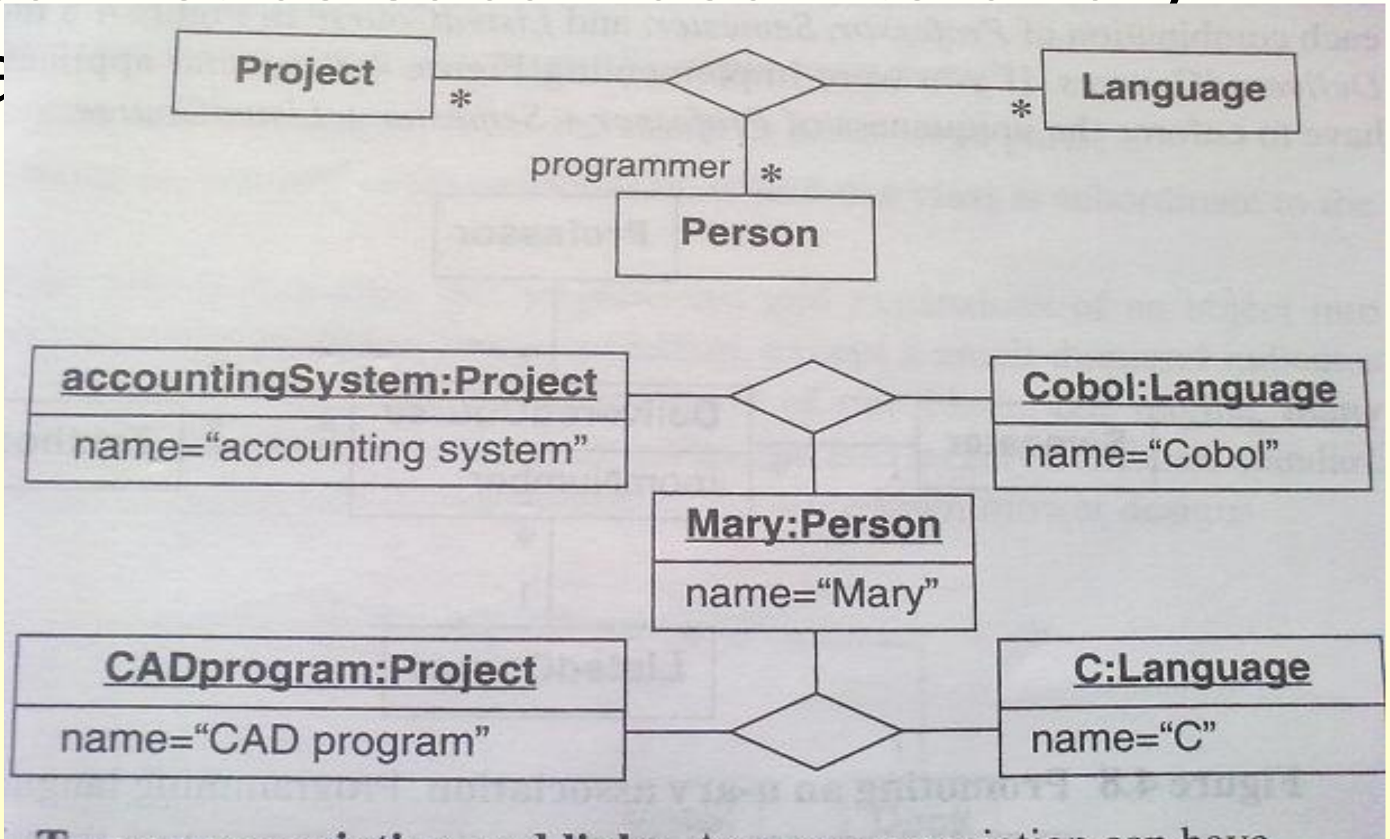
- Has
  - Association End name
  - Multiplicity
  - Ordering
  - Bags and sequences
  - Qualification
- Additional properties
  - Aggregation :
- Only binary associations can be an aggregation.
- Association end may be an aggregate or a constituent part.
- One association end must be an aggregate and the other end a constituent.

Association Ends

- Additional properties
  - Changeability : This property specifies the update status of an association end, which are either changeable or readonly
  - Navigability : Association can be traversed in any direction, but implementation may support one direction. UML shows navigability with an arrowhead attached to the target class.
  - Visibility : Association ends may be public, private, protected or packaged.

N-ary Associations

- n-ary association is an atomic unit and cannot be subdivided into binary a



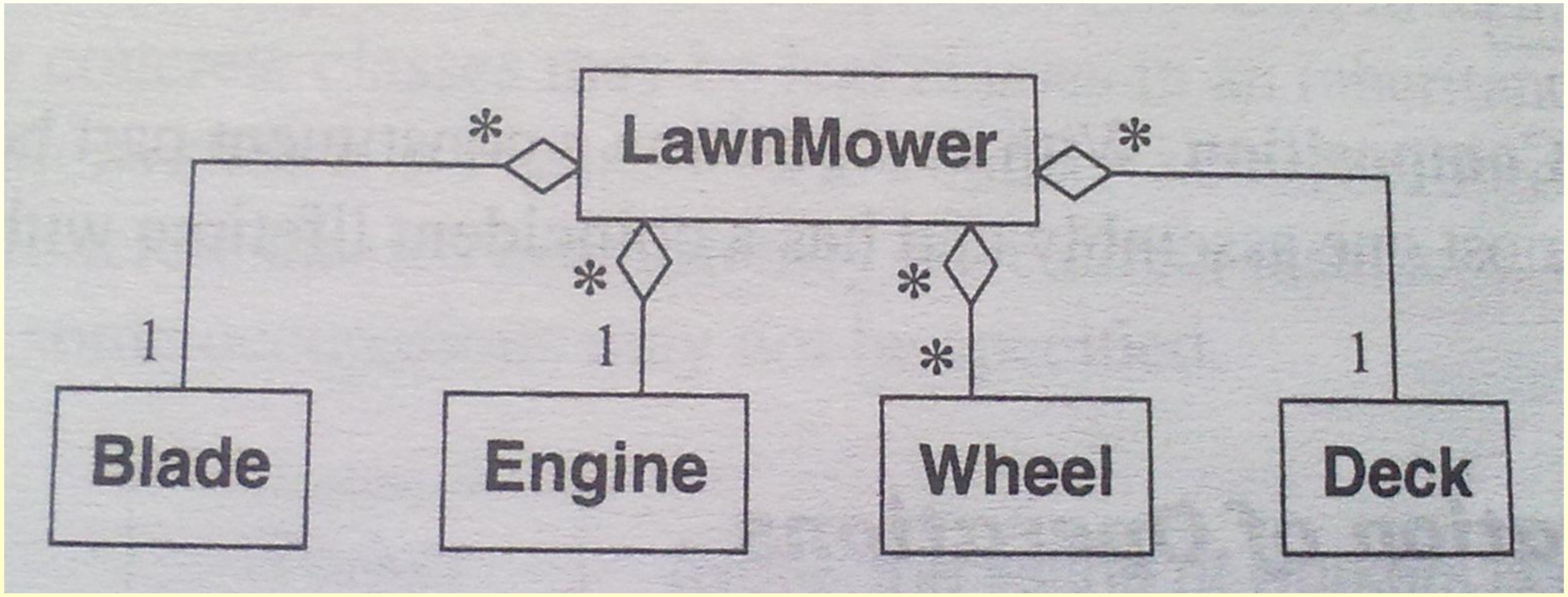
Ternary association and links. An n-ary association can have association end names, just like a binary association.

N-ary Associations

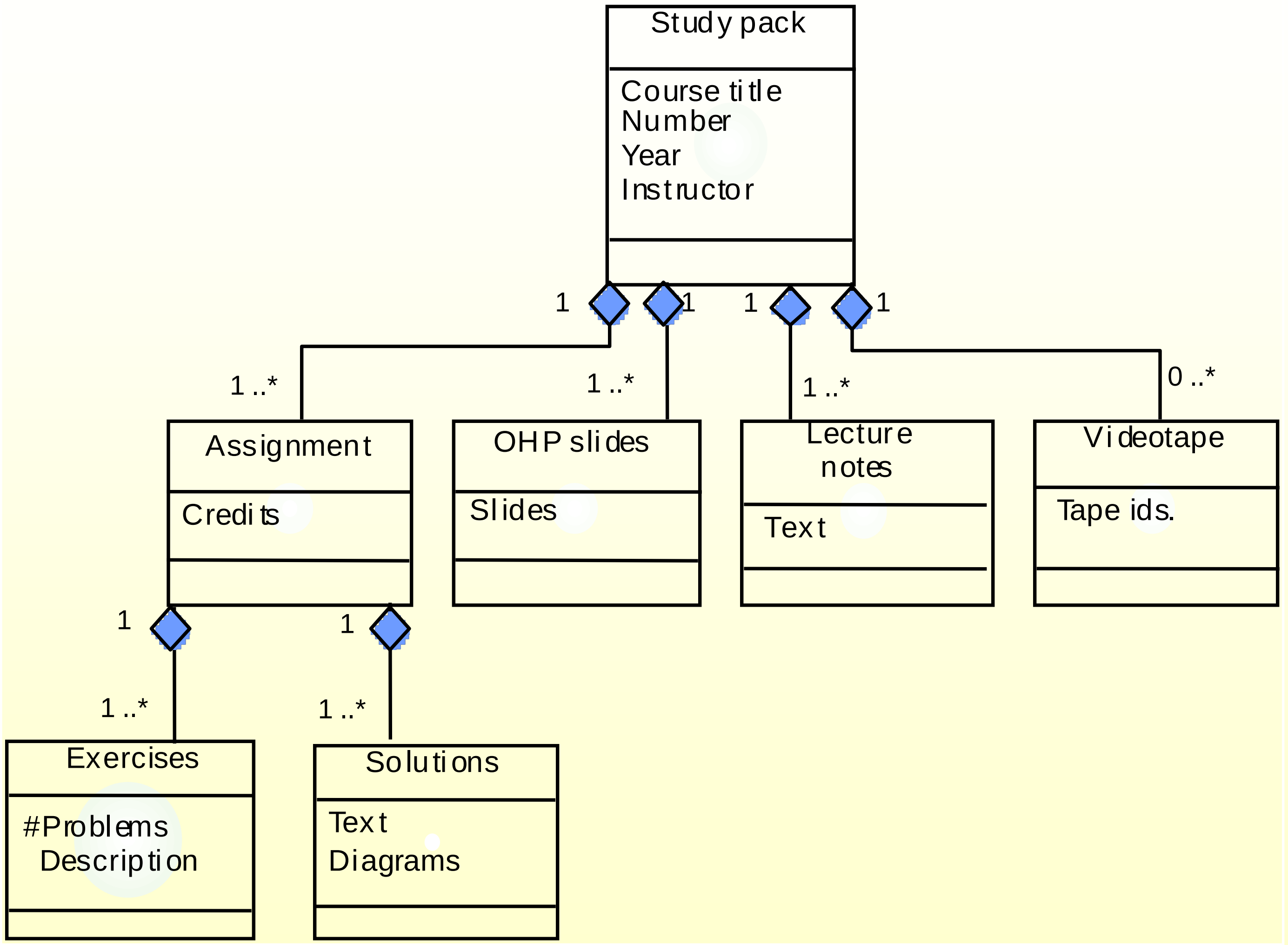
- n-ary association is an atomic unit and cannot
- be subdivided into binary associations
- Eg:- A professor teaches a course in a
- semester. Result is a delivered course in a particular room number. The delivered course has a text book.

Relationships: Aggregation

A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts



Relationships: Aggregation



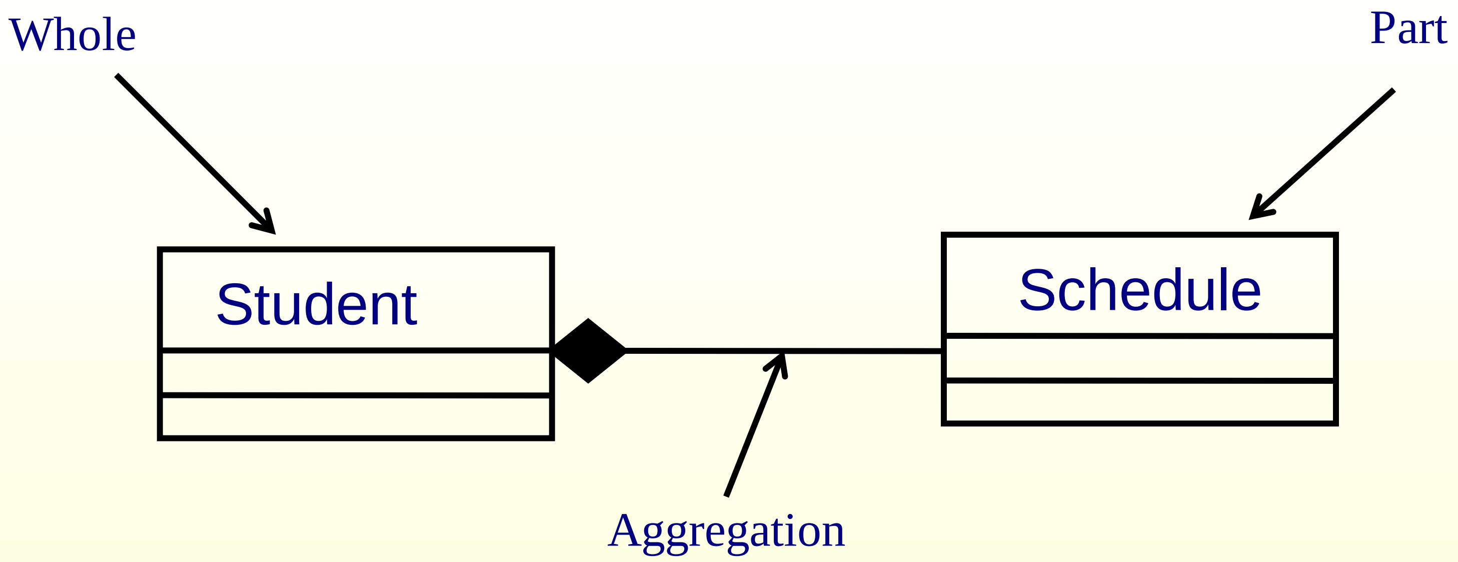
Aggregation Vs. Association

- Aggregation is a special form of association and not an independent concept
- If two objects are tightly coupled by a part-whole relationship, it is aggregation.
- If two objects are usually considered independent and, even though they may be linked, it is an association.
- Significant property of aggregation is transitivity and antisymmetric

Aggregation Vs. Composition

- Composition is a more restrictive form of aggregation.
- A form of aggregation with strong ownership and coincident lifetimes.
- The parts cannot survive the whole/aggregate
- It has two constraints :-
  - A constituent part can belong to atmost one assembly. Thus composition implies relationship of the parts by whole.
  - Deletion of an assembly triggers deletion of all the constituent objects via composition.
  - Denoted by a small dark diamond box next to the assemble class.

Aggregation Vs. Composition



- Write the composition for :- A company consists of divisions, which inturn consists of departments. A company is not a composition of its employees.

Propagation of Operations

- Propagation is the automatic application of an operation to a network of objects, when the operation is applied to some starting object.
- Eg:- Moving an aggregation, moves its parts
- The move operation is propagated to its parts which are in aggregation.
- Eg of Propagation : A person owns multiple documents, each document has many paragraphs which inturn has many

Propagation of Operations

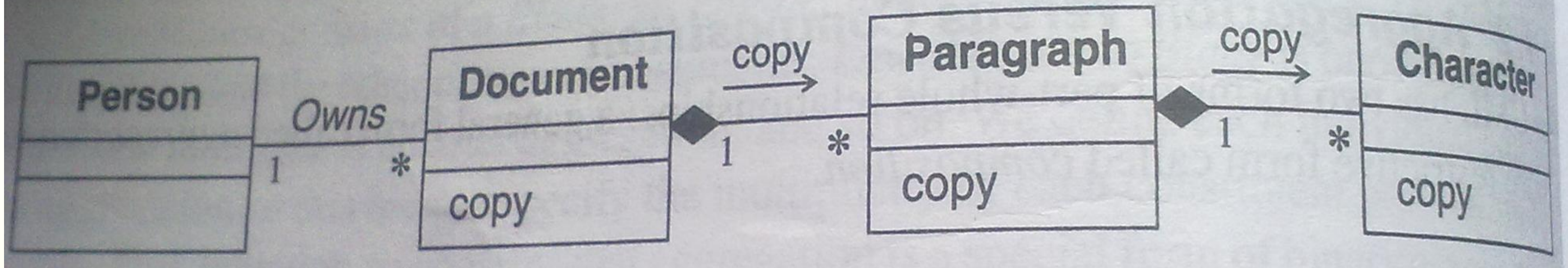


Figure 4.11 Propagation. You can propagate operations across aggregations and compositions.

- Here Copy operation propagates from documents to paragraphs to characters.
- The operation does not propagate in the reverse direction.
- Propagation is indicated with a small arrow in the direction of propagation and the operation name next to it.

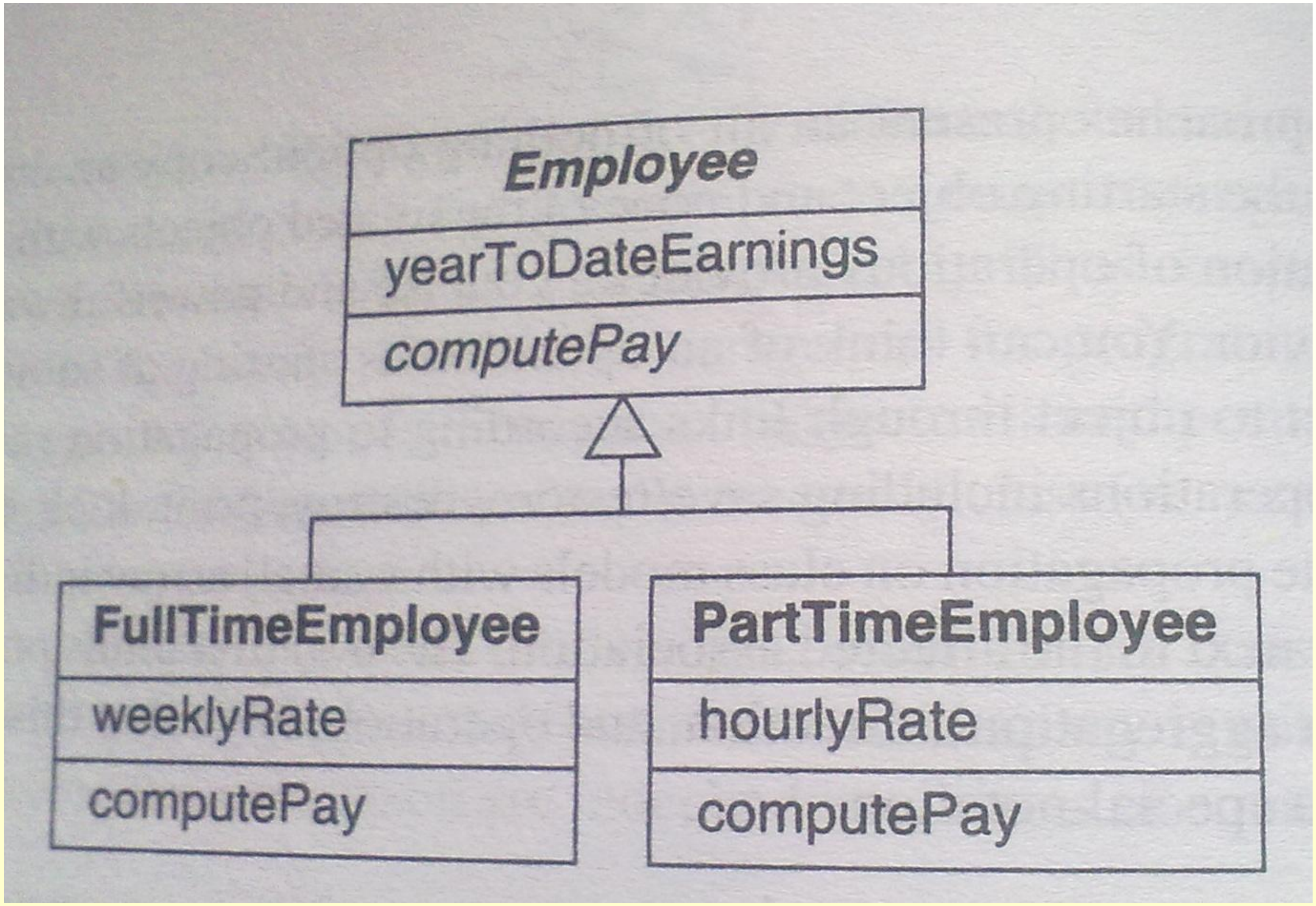
Abstract Classes

- An abstract class is a class that has no direct instances but whose descendant classes have direct instances.
- A concrete class is a class that is instantiable ,that is ,it can have direct instances.
- They can have derived classes, which can have instances of themselves. Such derived classes from **abstract base** classes are called **concrete classes**.
- Abstract class names are displayed in

Abstract Classes

- Abstract classes define methods that can be inherited by subclasses.
- Alternatively, they can define the signature of an operation and the derived classes have to implement the methods for the same. Such methods are called **Abstract Operations.**
- An Abstract operation defines the signature for an operation for which each concrete class must provide its own implementation.
- A Concrete class may have derived

Abstract Classes



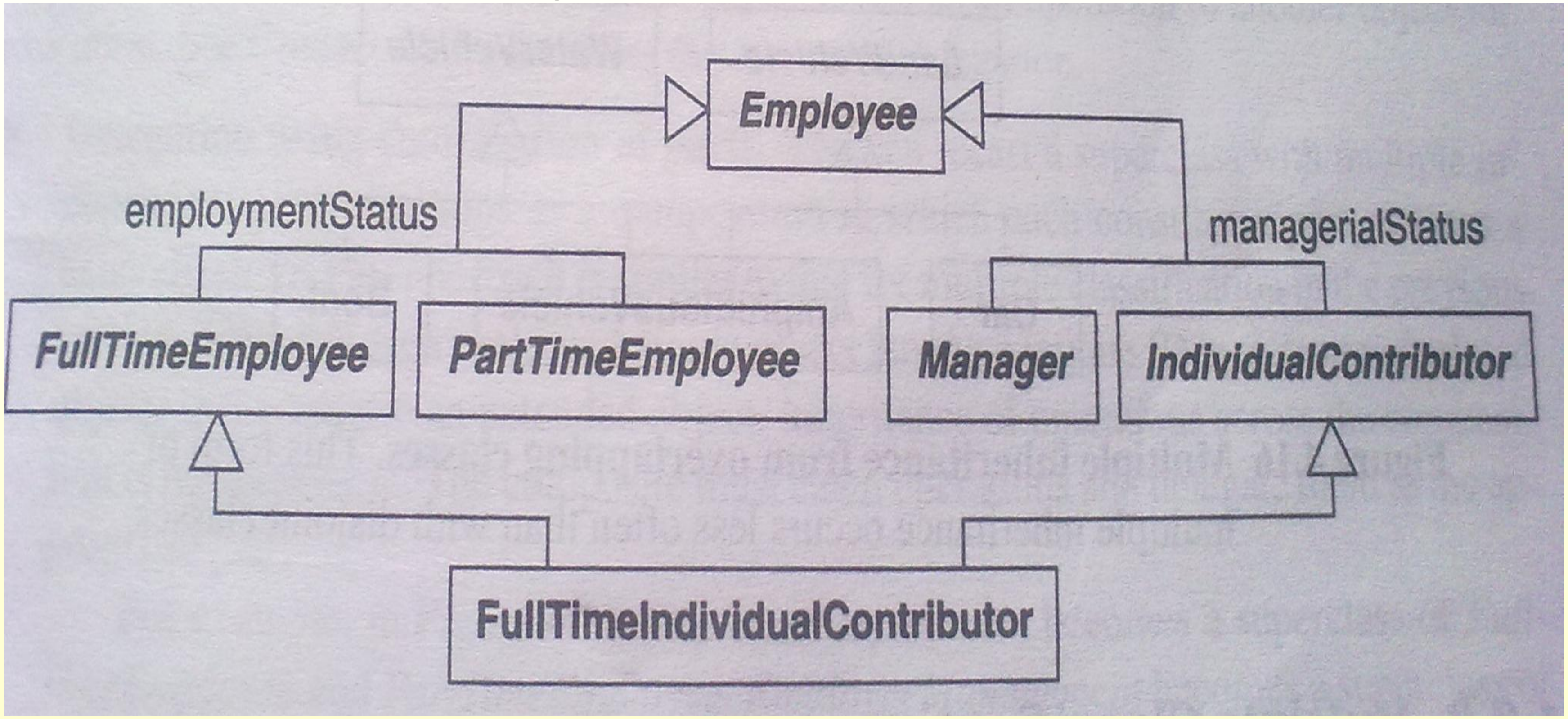
Multiple Inheritance

- Permits a class to have more than one base classes and to inherit features for all the parents.
- Advantage is that they have greater power in specifying classes and an increased opportunity for re-use.
- Disadvantage is loss of simplicity.

Kinds of Multiple Inheritance

- Multiple inheritance from disjoint classes.
- Eg(fig on next slide) :-
  - Each employee is either a fullTimeEmployee or PartTimeEmployee (disjoint) derived from Base class Employee.
  - Manager and IndividualContributor are also disjoint classes derived from classes Employee.
  - A FullTimeIndividualContributor is both FullTimeEmployee and IndividualContributor and derives both their features.
  - Class Employee independently specializes on

Multiple Inheritance from disjoint classes



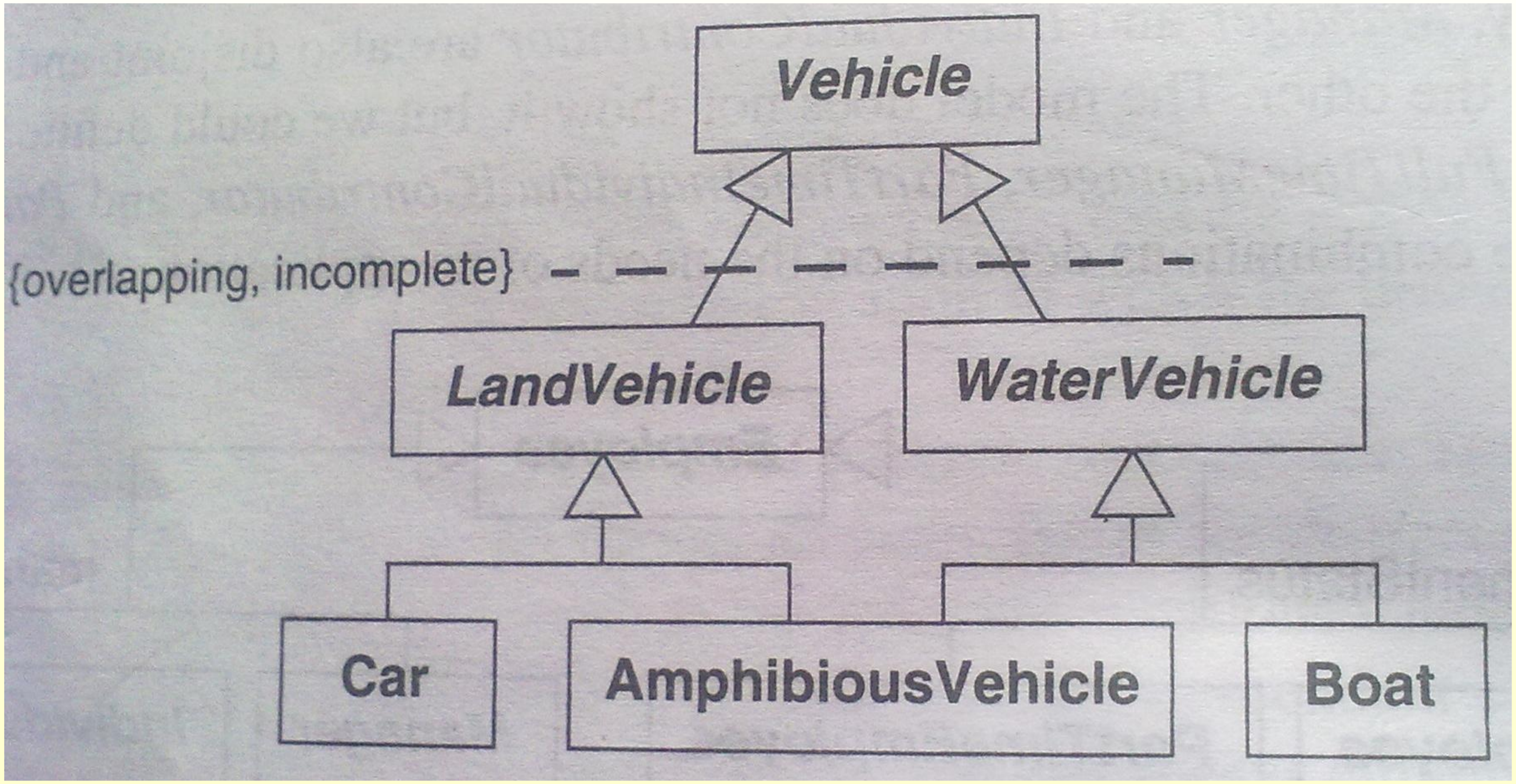
Kinds of Multiple Inheritance

- Attributes under the disjoint classes should be defined correctly otherwise they lead to ambiguity. Eg: FullTimeEmployee.name might give employee's name, but IndividualContributor.name might refer to his title.

Multiple Inheritance with Overlapping Subclasses

- Eg:- Amphibious vehicle is both a land vehicle and water vehicle, because it travels on both land and water.
- Eg:- An amphibian is a creature that is born as a fish, but then as it grows up, it moves to the land. A frog lives on land and water.

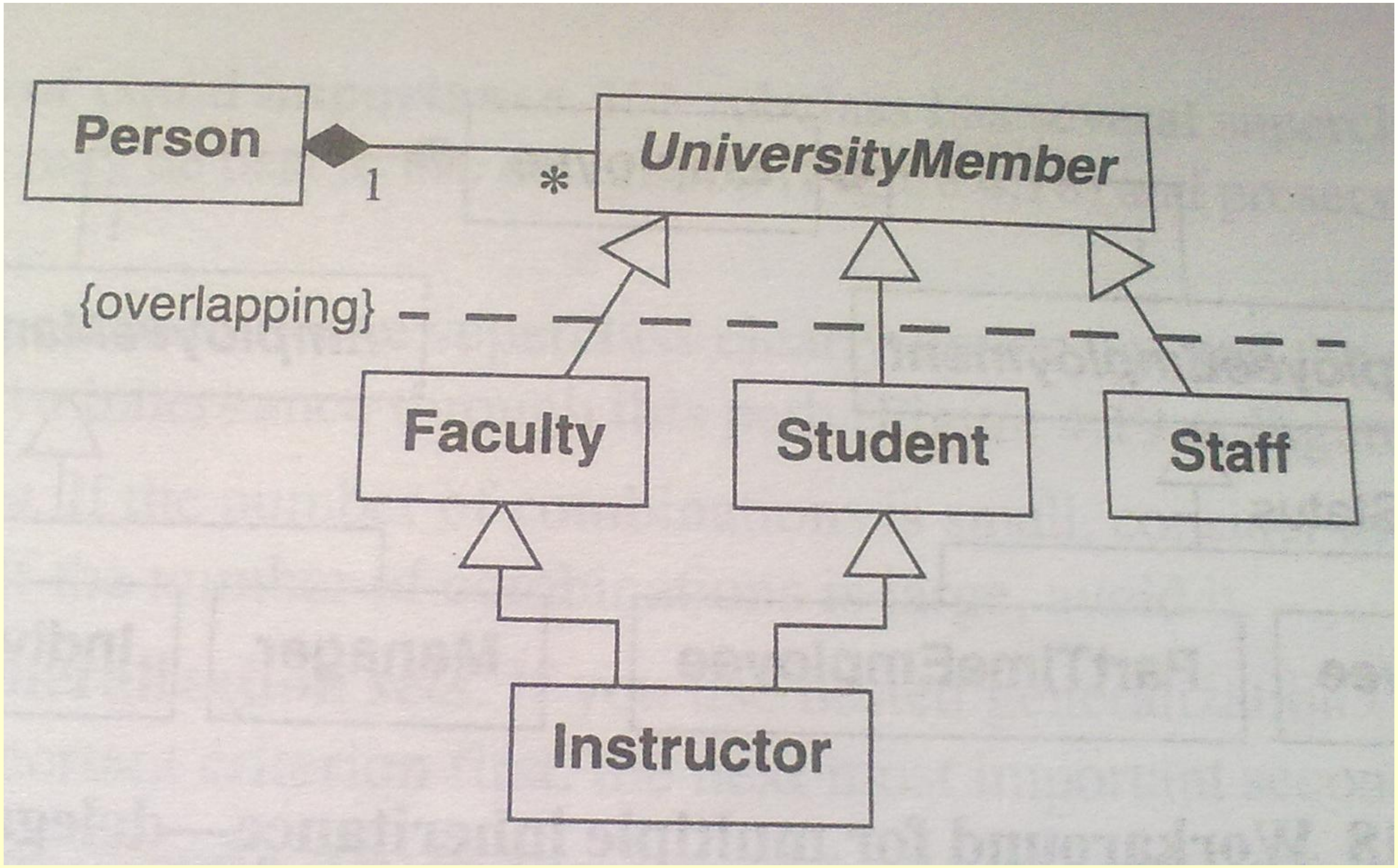
Multiple Inheritance with Overlapping Subclasses



Multiple Classification

- An Instance of the class is inherently an instance of all ancestors of the class.
- For Eg: Teaching faculty could also be an instructor, and a student could also be an instructor. Model in next slide.
- How about A Senior Professor in MIT ?
- There is no class to describe this combination.
- UML permits multiple classification, but most OO Languages handle it poorly.

Multiple Classification



Workarounds for Multiple Inheritance

- An Instance of the class is inherently an instance of all ancestors of the class.
- For Eg: Teaching faculty could also be an instructor, and a student could also be an instructor. Model in next slide.
- How about A Senior Professor in MIT ?
- There is no class to describe this combination.
- UML permits multiple classification, but most OO Languages handle it poorly.

Delegation using composition of parts

- Recast a superclass with generalizations as a composition in which each constituent part replaces a generalization.
- Inheritance of operations across the composition is not automatic.
- Fig next page, An operation sent to the Employee Object has to be redirected to the EmployeeEmployment or EmployeeManagement.

Delegation using composition of parts

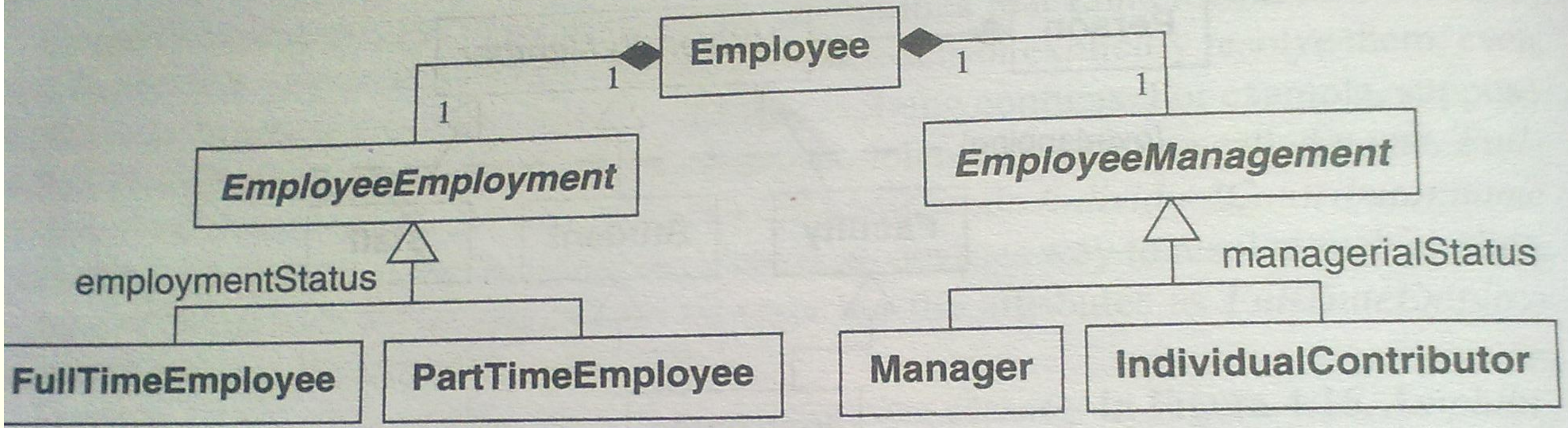
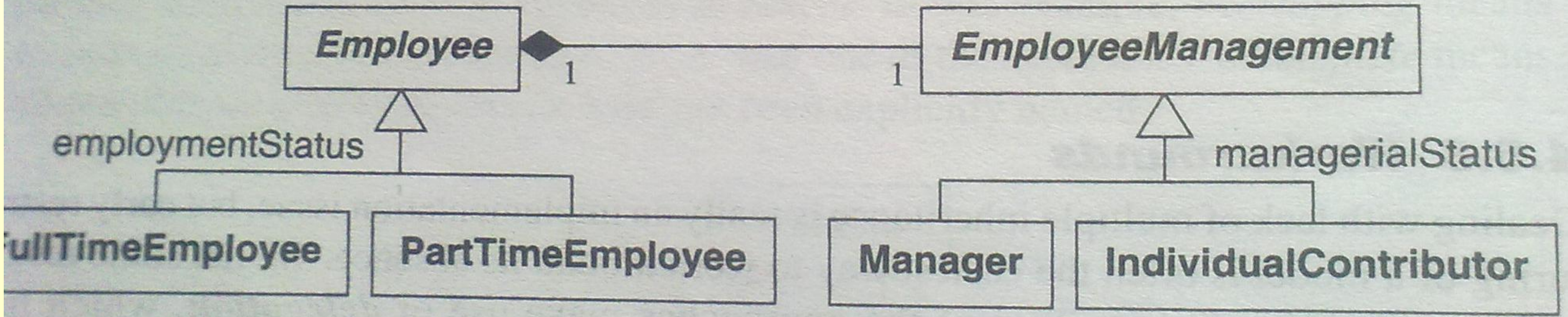


Figure 4.18 Workaround for multiple inheritance—delegation



igure 4.19 Workaround for multiple inheritance—inheritance and delegation

Inherit the most important class and delegate the rest

- If EmploymentStatus is important, inherit that and delegate the EmployeeManagement which specifies Manager and IndividualContributor.

Nested Generalizations

- Has 2 classes for manager and 2 for IndividualContributor.
- Preserves inheritance, but duplicates code violating spirits of OO Programming

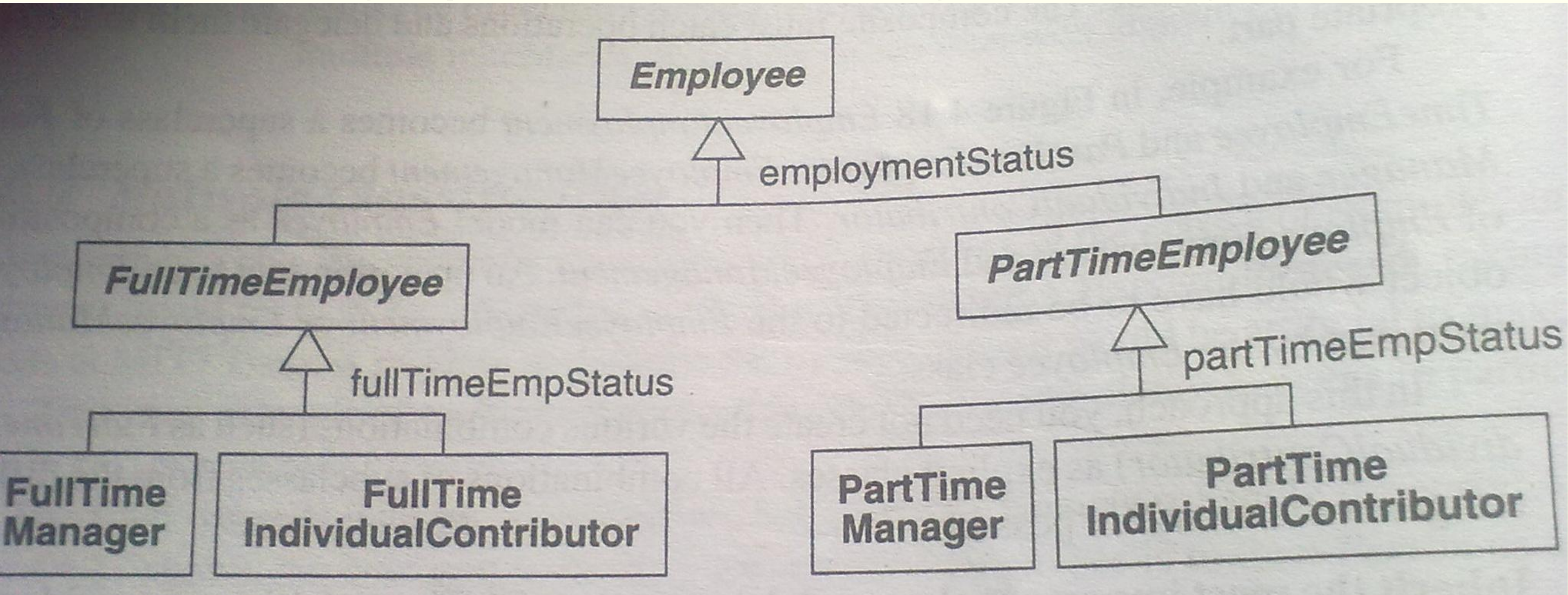


Figure 4.20 Workaround for multiple inheritance—nested generalization

When to use these workaround ?

- Super Classes of equal importance : When there are more than one super classes of equal importance, it is best to use **Delegation**
- Dominant Superclasses : If one superclass clearly dominates the others, preserve inheritance through this path.
- Few Subclasses :- consider nested generalizations.

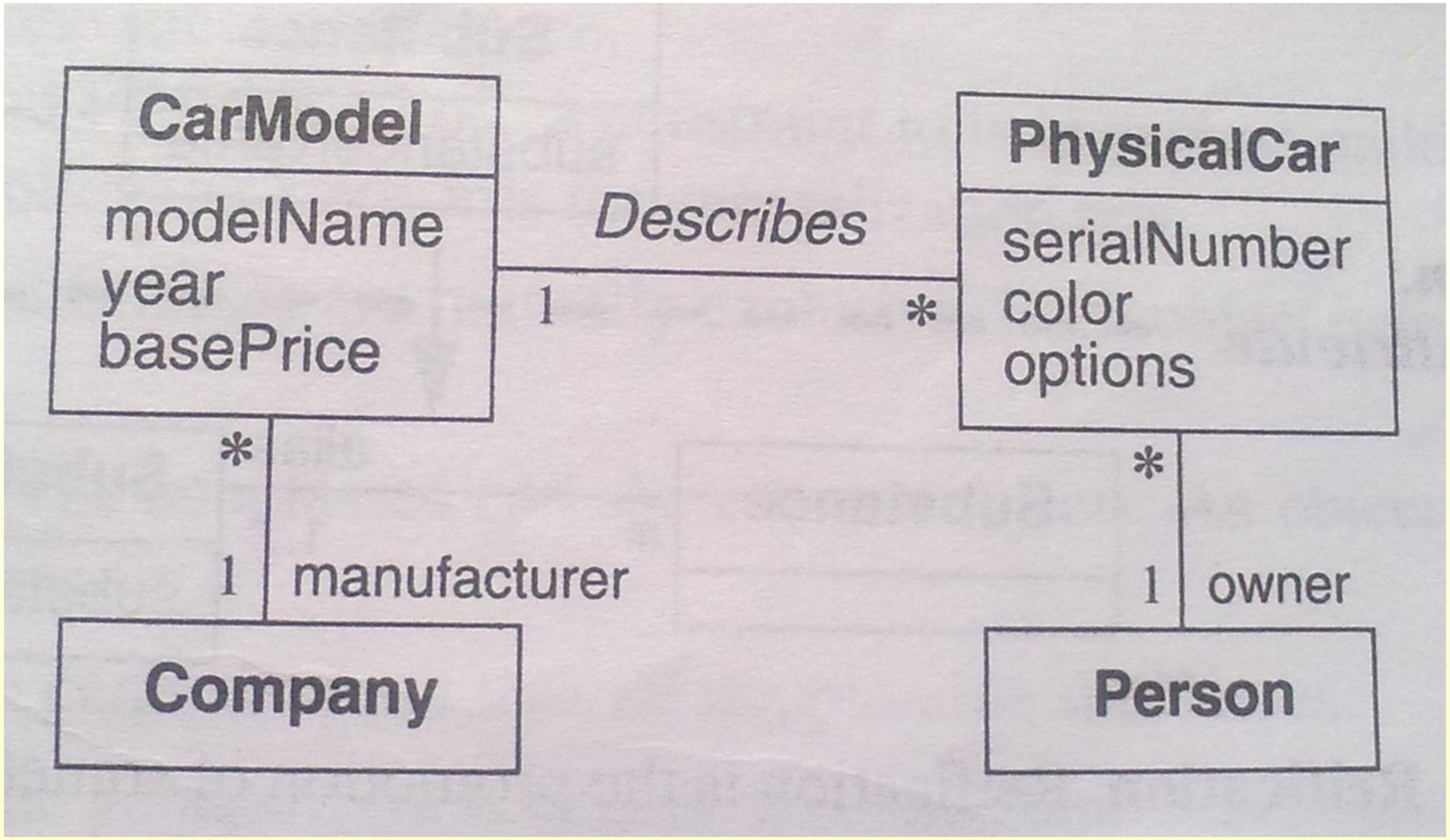
When to use these workaround ?

- Sequencing generalizations sets : Factor of the most important generalization first, then second and so on...
- Large quantities of code : if large quantities of code are generated, avoid nested generalizations.
- Identity : Only nested generalizations preserve strict identity.

Metadata

- Metadata is data that describes other data
- A Car model has : model name, price, make, company etc
- A Physical car has : color, reg no, owner etc..
- You can consider classes as objects, But Classes are meta-objects and not real world objects.
- Class descriptor objects have features and they inturn have their own classes which are called metaclasses

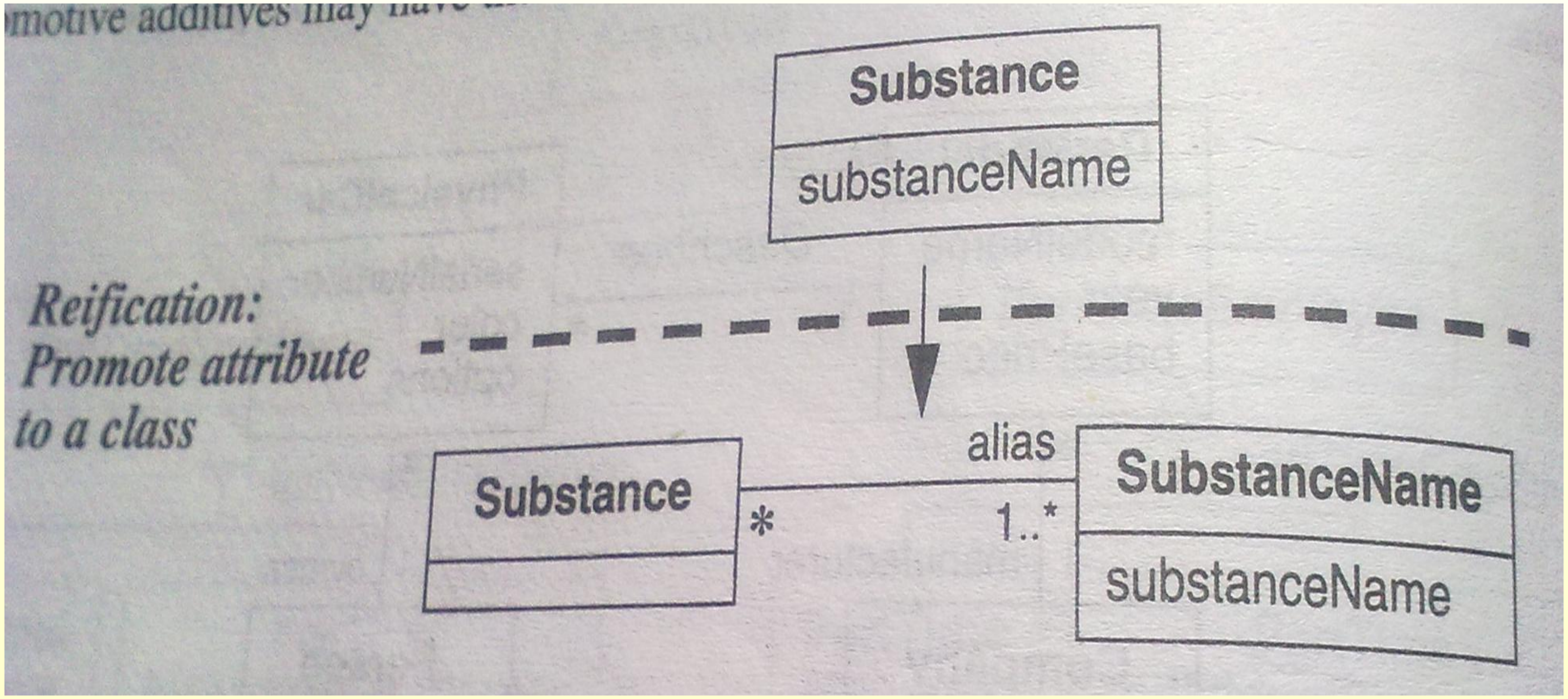
Metadata


CarModel
modelName
year
basePrice
Describes
1
*
PhysicalCar
serialNumber
color
options
*
1
manufacturer
Company
*
1
owner
Person

Reification

- It is the promotion of something that is not an object to an object.
- It is useful to promote attributes, methods, constraints and control information into objects so that you can describe and manipulate them as data.
- Helpful for meta applications because it helps you to shift the level of abstraction.
- Fig promotes the substanceName attribute to a class

- This captures a many to many



Reification

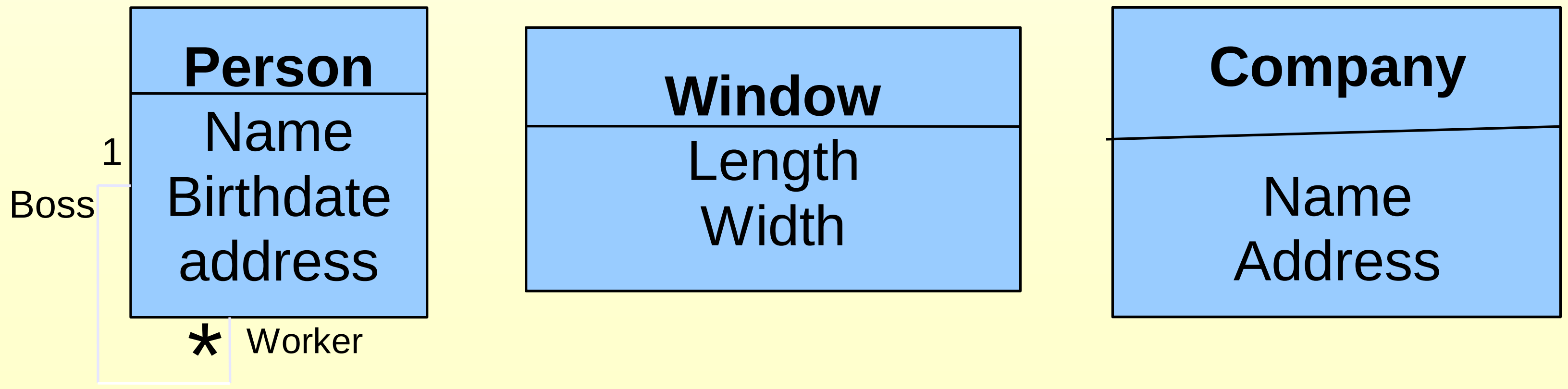
- Consider a database manager : Conventionally, a developer would write code to r/w data from files.
- It is better to reify the notion of data services and use a database manager. A database manager has abstract functionality that provides a general purpose solution to accessing data reliably and quickly for multiple users.
- Consider a state transition diagram : prepare a meta model and state transition model as data. A general purpose interpreter reads the contents of the meta model and executes the intent.

Constraints

- A Constraint restricts the values that the elements can assume.
- Elements means : Objects, Attributes, links, associations and generalization sets
- Constraints can be expresses using Object Constraint Language. (OCL)

Constraints on Objects

- Eg:
  - No employee's salary can exceed salary of his/her boss's salary.
  - No window can have its aspect ratio (length/width) less than 0.8 or greater than 1.5
  - Priority of a job may not increase over time.



Constraints on Generalization Sets

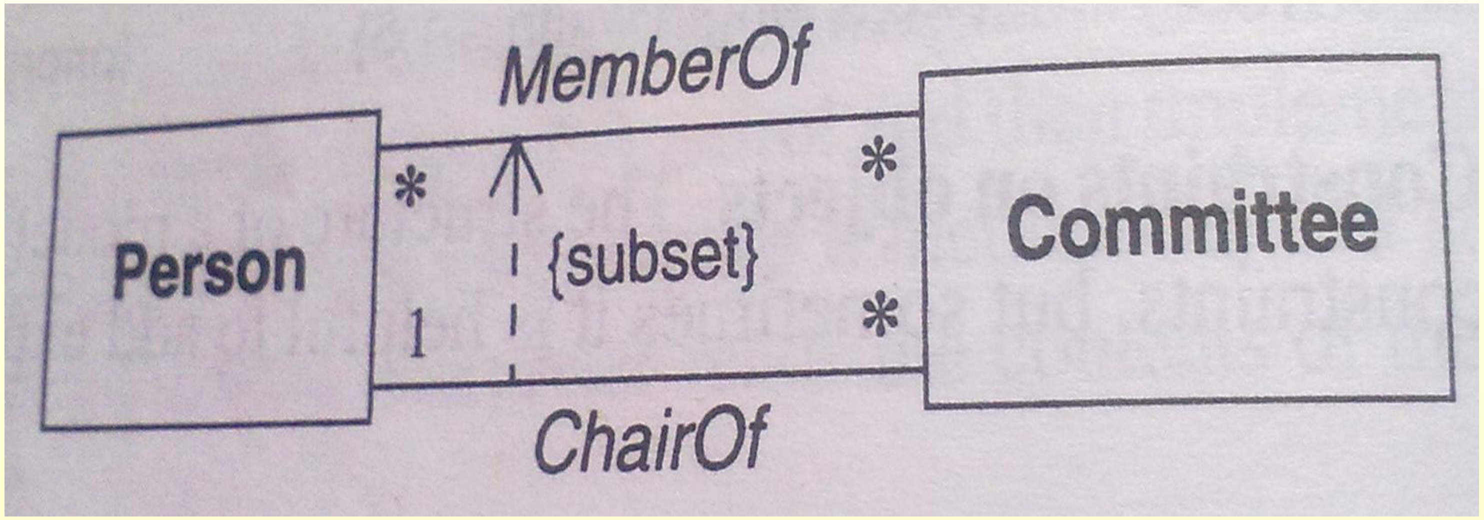
- Disjoint Classes
- Overlapping Classes
- Complete (Generalization lists all possible subclasses)
- Incomplete (Generalization may be missing some subclasses

Constraints on Links

- Multiplicity is a constraint on the cardinality of a set.
  - Multiplicity for association
  - Multiplicity for attributes
- Qualification is also a constraint on the association. A Qualifier signifies how many objects are there at an association end.
- Association class implies a constraint. It has a constraint that an ordinary class done not have !
  - It derives its identity from the instances of its related classes.
- Associations have the constraint {ordered}

Constraints on Links

- Associations have the constraint {ordered}
- Subset Constraint :- Fig shows how the Association ChairOf is a subset of the association MemberOf.

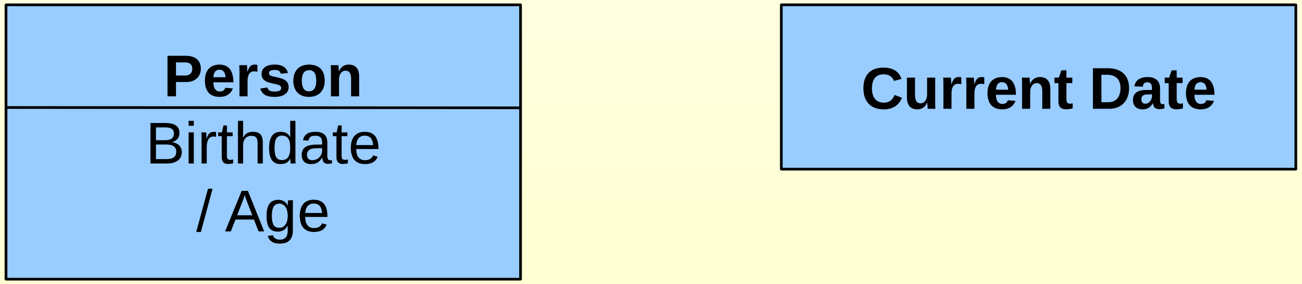


Constraints

- Constraints provide one criterion for measuring the quality of a class model.
- A Good class model captures many constraints thro its structure.
- A Constraint is placed with braces
- Dashed lines are used to connect constrained elements. A dashed arrow can connect a constrained element to the element on which it depends.

Derived Data

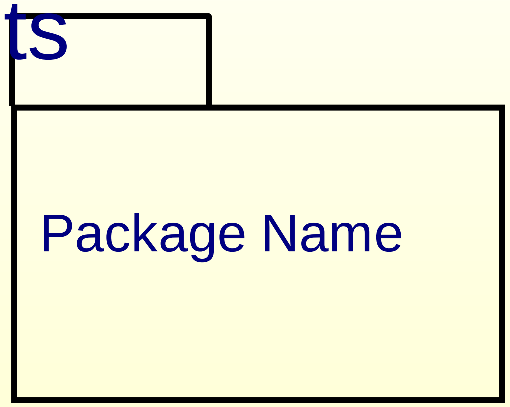
- Just like derived classes, attributes of a class and associations b/w classes can be derived
- A derived attribute is a function of one or more elements.
- Age can be derived from Birthdate and CurrentDate



What is a Package?

A package is a general purpose mechanism for organizing elements into groups

- A model element which can contain other model elements



OO Principle: Modularity

- Uses
  - Organize the model under development
  - A unit of configuration management

THE END