Object Oriented Analysis and Design (OOAD)

Abdulkareem Alali

Based on Larman's Applying UML and Patterns Book, 3d

Introduction to the Book, a crash course about the course!

Software Engineering is A Layered Technology

The bedrock that supports software engineering is **a Quality focus**.

The Methods

Methods

Requirements, Analysis and Design

- **Requirements** (Elicitation of requirements)
- **Analysis** or **Requirement Analysis** (an investigation of the requirements rather than a solution)
- **Design** emphasizes **a conceptual solution that fulfills the requirements**, rather than its implementation
- Do the right thing (Analysis),
  And do the thing right (Design)

Methods—Object Oriented Analysis and Design (OOAD)

With **OOA,** the goal is to find and describe the objects, or concepts, in the **problem domain**
(an investigation of the domain objects)

With **OOD,** the goal is to define objects and how they work together to fulfill the requirements

The Book

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, Craig Larman

1. UML vs. Thinking in Objects
2. OOD Principles and Patterns
3. Case Studies
4. Use Cases
5. Iterative Development, Agile Modeling, Agile UP (Unified Process)

Tools and Methods

1. Applying UML, THINK OO!

UML is just a standard diagramming notation, Unified Modeling Language

Knowing UML helps you communicate with others in creating software,

but the real work in this course is learning

OOA/D, not how to draw diagrams!

Tools 1. Applying UML, THINK OO!

UML is a standard diagramming notation

The UML is not OOAD approach or a method, but a useful tool

Use of UML, like use of an OO programming language does not guarantee a good result

So Coding in OO, doesn't mean your code is OO, diagraming in UML doesn't mean your diagram is a OOD

3. Case Studies

The text illustrates the material with on-going case studies, a **large one** and a **small one**

The large one allows for realistic situations to be encountered and handled

4. Use Cases, Requirements Analysis, Methods

OOAD is preceded by the elicitation of analysis of requirements

TFCL: Think First, Code Later!
A **principle** of a good design is to defer decisions as long as possible.
The more you know the more likely a decision is a good one!

Writing Use Cases is not a specifically Object Oriented practice.
However it is a best practice for elaborating and understanding requirements.

TFCL!!

Software engineering came to slow us down, don't be extremely slow as a Waterfall, be Agillic! A. A.

Being Agile means working in a lightweight, highly responsive way so that you deliver your product or services in the way the customer wants and at that time the customer needs them. There are rules, albeit not many rigid rules.

Other Necessary Skills, Methods

Requirements Analysis,

Object-Oriented Analysis,

and Object-Oriented Design is not a complete toolkit for a software developer

+

There are many other skills necessary in Software development, including programming.

This course only covers a subset of the necessary skills.

OOAD, Example, Methods

OOA finds and describe objects—or concepts—in problem domain

- E.g., *Flight information system*, concepts may include **Plane**, **Flight**, and **Pilot** with tailNumber attribute

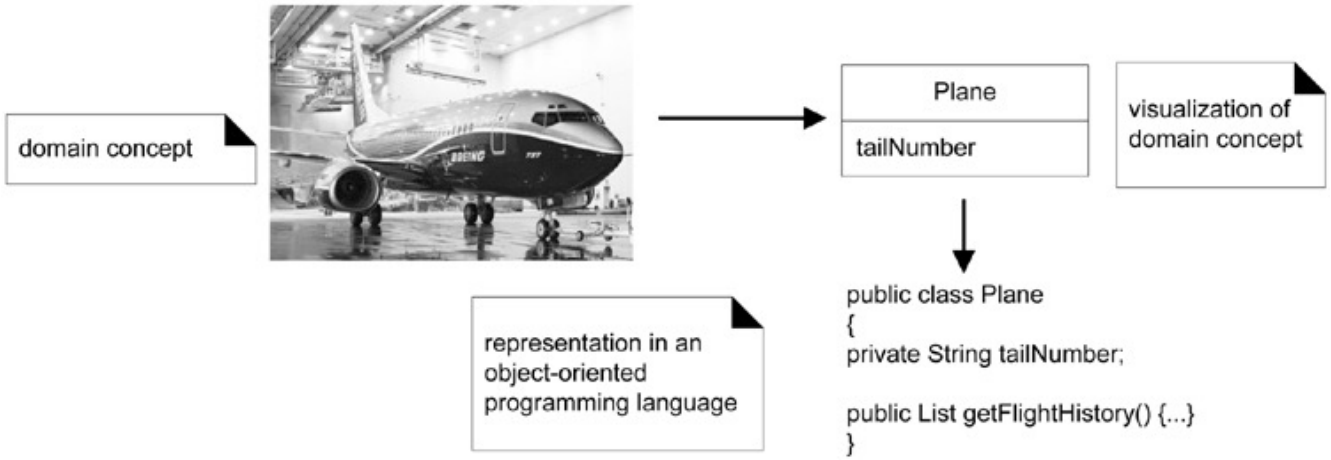
OOD defines software objects, their collaboration to fulfill the requirements

- E.g., a **Plane** software object may have a tailNumber attribute and a ***getFlightHistory*** method

Good skills in **OOAD** are essential for the creation of
well-designed, robust, and maintainable software

.. signs of a quality!

Example: Flight Information System



This course is about Modeling!

The Most Important Learning Goal?

Modeling

Many skills are required for good OOAD the most fundamental, breaking the problem down to objects and the ability to skillfully assign responsibilities to software objects

It is the one activity that must be performed either while

drawing a UML diagram or programming

The Most Important Learning Goal?

Modeling, why?

It strongly influences the robustness, maintainability, and re-usability of software components

Even if there is no time for modeling and a "rush to code" the assignment of responsibilities is inevitable

OO Programming (OOP) is Not Enough

Knowing an OO language is only the first step

A programming language is only a tool

A tool must be skillfully used or the resulting product will not be good

Owning a hammer doesn't make one a carpenter, let alone an architect

For OOAD knowing how to “think in objects” is critical!

5. The UP (Unified Process)

A standardized and agile approach to analysis and design

Helps to ensure that all necessary tasks are understood and completed in software development

Text, and the course, will focus on the Unified Process developed at Rational Software (IBM) by Ivar Jacobsen, Grady Booch, Jim Rumbaugh, and others.

2. OOD Principles and Patterns

The most important skill in Object-Oriented Analysis and Design is assigning responsibilities to objects

That determines how objects interact and what classes should perform what operations

Responsibility-Driven Design

2. OOD Principles and Patterns

In software development there are problems that occur repeatedly

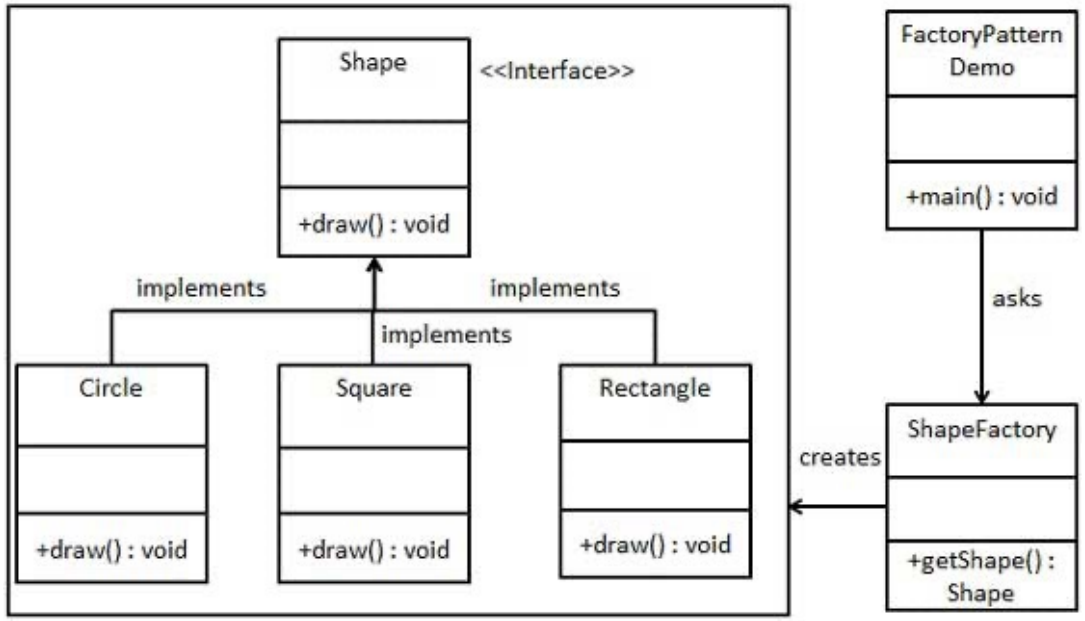
- E.g. one object needs to know when something has changed in another object, a listener?

Although each occurrence of the problem is different, the best solution for each occurrence is similar

Design Principles and Patterns are tried-and-true solutions to design problems that can be customized to different situations

By learning how to apply patterns and principles, skillful use of the fundamental object design idioms is achieved.

Design Patterns: Example .

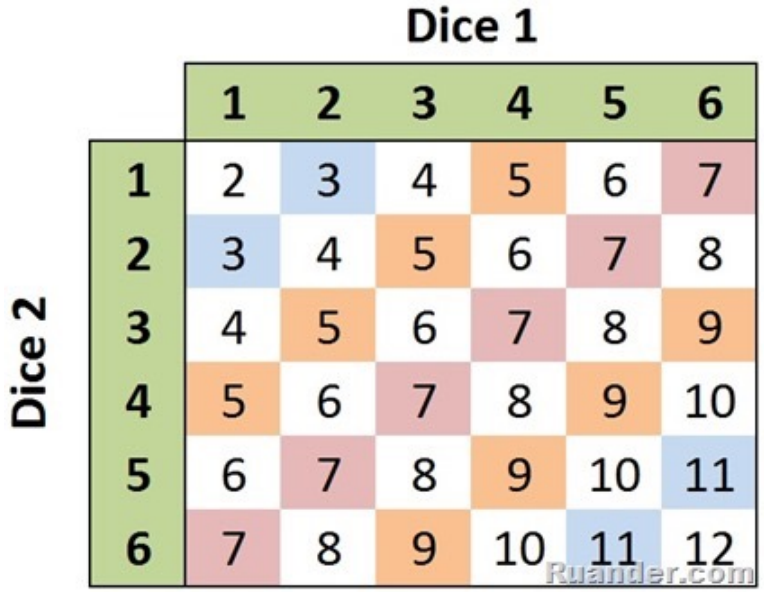


Factory Pattern is used when creating an object is not simple, you need a specialist, a factory

A Dice Game, Applying Methods

Developing and Modeling a Dice Game

A “dice game” in which software simulates a player rolling two dice. If the total is seven, they win; otherwise, they lose.



Dice Game Example, a Use Case

Requirements analysis may include **stories** or **scenarios** of how people use the application; these can be written as **use cases**

Use cases are not an object-oriented artifact—they are simply written stories. However, they are a popular tool in requirements analysis

Play a Dice Game (A use case):

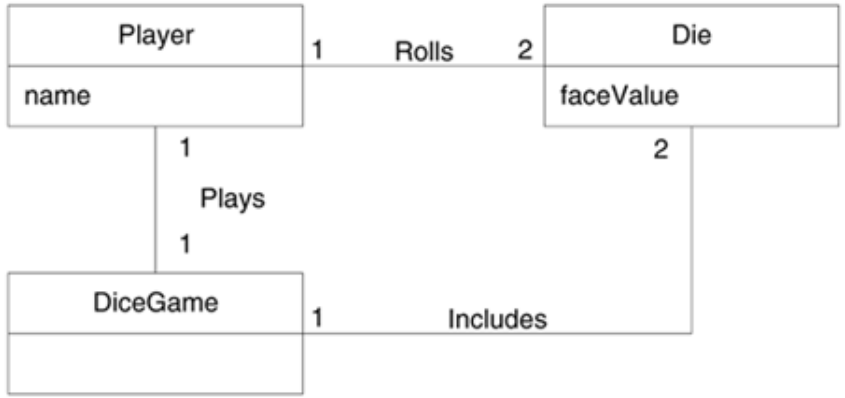
Player requests to roll the **dice**, **System** presents results

If the dice face value totals seven, player wins; otherwise, player loses.

Analysis—Domain Model (OOA)

Noteworthy domain concepts or objects.

Player, **Die**, and **DiceGame**, with their associations and attributes



A **domain model** is not a description of software objects; it is a visualization of the concepts or mental models of a real-world domain. Thus, it has also been called a conceptual object model

Design —Interaction Diagrams (OOD)

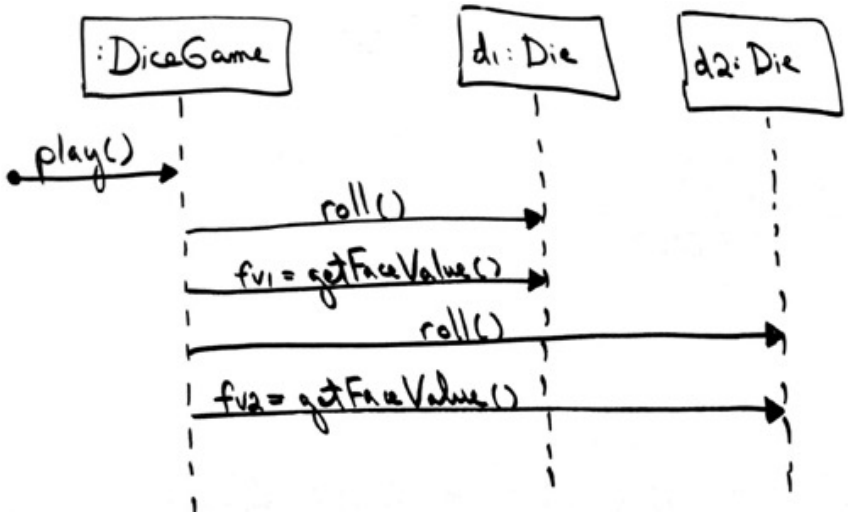
OOD defines software objects—their responsibilities and collaborations

Collaborations use UML sequence diagrams, flow of messages between software objects, and invocation of methods

A player rolls the dice, in the OOD the **DiceGame** object "rolls" the dice (sends messages to Die objects)

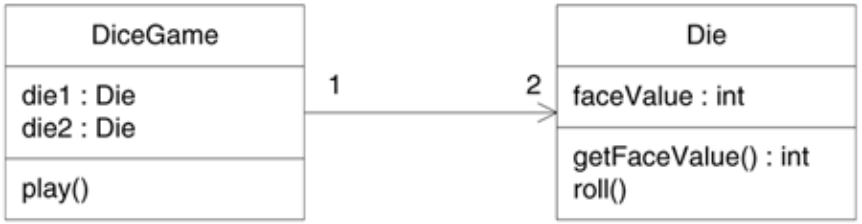
OOD is inspired by real-world domains but they are not direct models or simulations of the real world

A sketching on a whiteboard



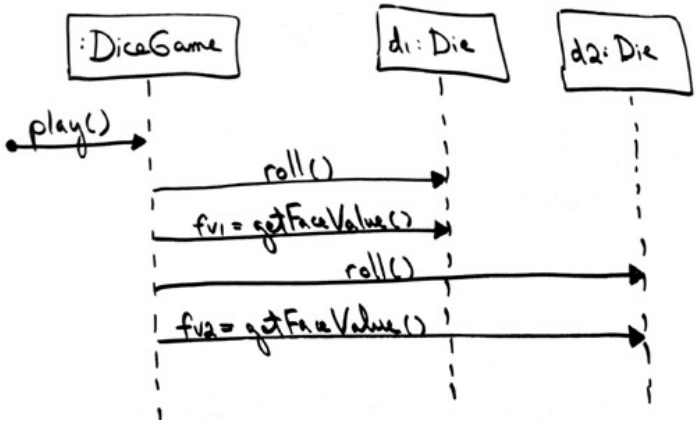
Design—Class Diagram (OOD)

Interaction sequence diagrams vs. A static view class diagram



Illustrates the attributes and methods of the classes

A sequence diagram leads to partial design class diagram



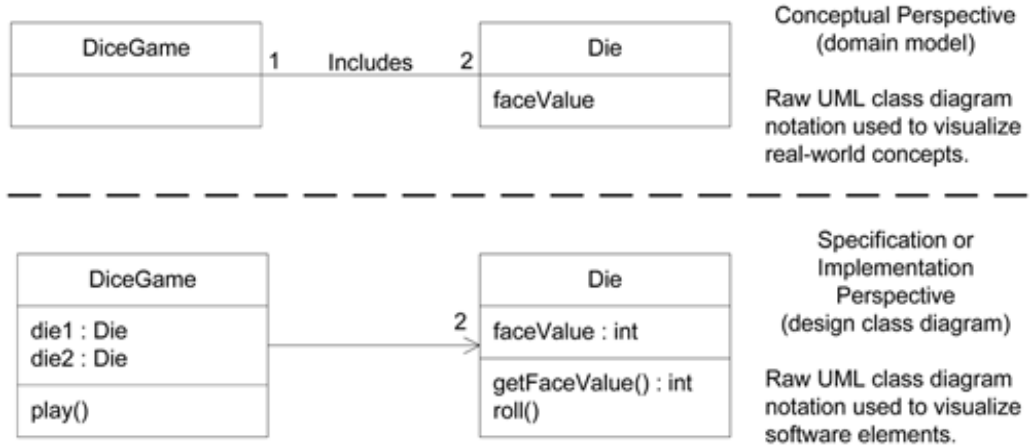
A play message is sent to a **DiceGame** object,

- **DiceGame** class requires a play method,
- while class **Die** requires a ***roll***,
- and ***getFaceValue*** method

Domain Model vs. OOD Model

A contrast. Domain model with real-world classes, class diagram has software classes

Although design class diagram is not the same as the domain model, some class names and content are similar



OOAD supports a lower representational gap (**LRG principle**):

- Software components vs. domain mental models
- OOD design minimizes differences with the domain logic it represents
- Improves comprehension!

UML under Unified Process (UP) Model

Agile modeling emphasizes UML as sketch.

The Meaning of UML "Class" in Different Perspectives:

- Conceptual class— real-world concept or thing
- Software class— a class representing a specification or implementation perspective of a software component
- Implementation class— a class implemented in a specific OO language such as Java or C++.

OOAD History

Larry Constatine

- 60's
- Researched good software design
- **Coupling**, **cohesion**
- One of the first to suggest design before implementation

O.J. Dahl, K. Nygaard

- Mid 60's
- **Classes**
- Simula programming language

OOAD History

Alan Kay, Adele Goldberg

- ~1970
- PARC, Palo Alto Research Center
- Smalltalk language
- Smalltalk personal computer
- **Kay coined the terms object-oriented programming** and personal computing
- Steve Jobs visited (1979), saw the Smalltalk computer and this became the vision for the Lisa and Macintosh

Edsger Dijkstra

- Research software correctness
- Early work proposed layers of abstraction with **separation**, **encapsulation**

OOAD History

Barbara Liskov

- 70's
- Theory and implementation of ADT's
- CLU programming language - hidden internal data representation
- Liskov Substitution Principle -- SOLID

Jean Ichbian, et. al.

- 80's
- Green programming language -> Ada

Bjarne Stroustrup

- Late 80's, 90's
- Martin Richard's BCPL -> B -> C -> C++

OOAD History

Bertrand Meyer

- "Melded best ideas of CS with best ideas of OO"
- Eiffel programming language
- Programming by contract

Grady Booch, Ivar Jacobson, James Rumbach

- 90's
- There were a number of graphical modeling languages with many similarities, many minor differences, they "unified" various models into UML, "unified modeling language"
  - 1991, The OMT method by Rumbaugh et al.
  - 1991, The Booch method
  - 1992, Jacobson, Objectory method, promoted OOA/D, use cases
  - Others
- The Three Amigos

Summary

OOAD is the research of requirement analysis and collaboration between objects via assigning responsibilities

UML is a modeling tool, enhance documentation and communications

Use cases are not an object-oriented artifact

Unified Modeling Language, UML 2.x

Abdulkareem Alali

ACK Jonathan Maletic, Grady Booch, IBM Rational Software, Martin Fowler, …

UML Part I

- Introduction to UML
- Overview and Background

Unified Modeling Language

UML is a general-purpose notation and tool that is used to

- visualize,
- specify,
- construct, and
- document

the artifacts of a software systems

History of the UML

Current version = 2.5.1
(Dec. 2017)

UML 2.0
(2004)

UML 1.5
(March, '03)

UML 1.1
(Sept. '97)

UML 1.0
(Jan. '97)

UML 0.9 and UML 0.91
(June '96) (Oct. '96)

Unified Method 0.8
(OOPSLA '95)

OOSE

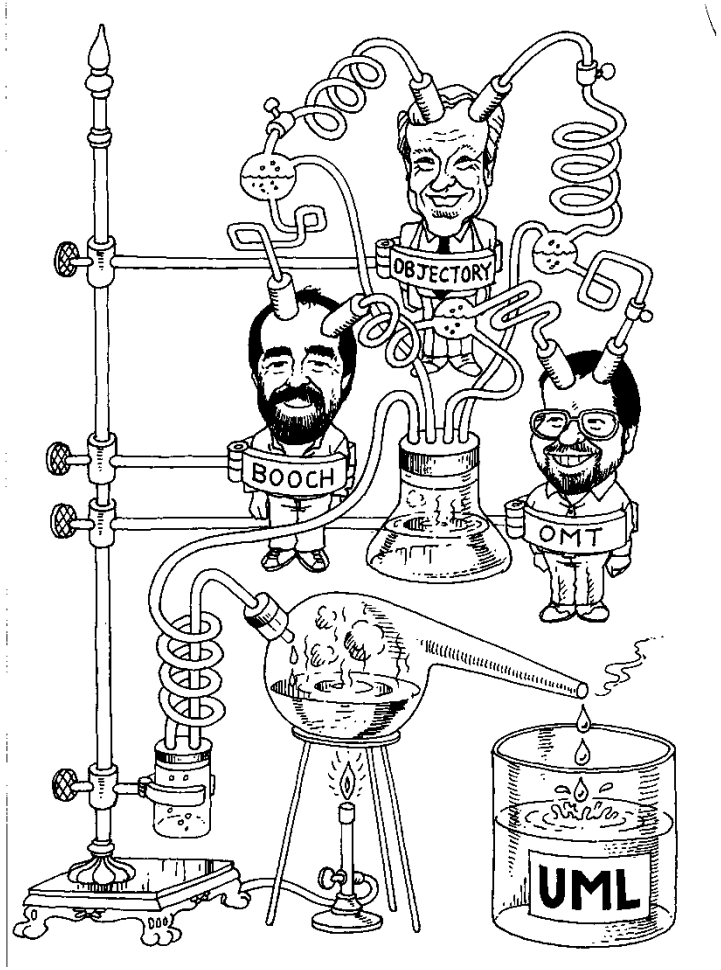
Other Methods

Booch '91

Booch '93

OMT - 1

OMT - 2



Types of UML Diagrams

Structural Diagrams

The static aspects of the software system

- Class, Object, Component, Deployment, …

Behavioral Diagrams

The dynamic aspects of the software system

- Use-case, Interaction, State Chart, Activity, …

Structural Diagrams

Class Diagram

Set of classes and their relationships.

Describes interface to the class (set of operations describing services)

Object Diagram

Set of objects (class instances) and their relationships

Structural Diagrams

Component Diagram

Logical groupings of elements and their relationships

Deployment Diagram

Set of computational resources (nodes) that host each component

Behavioral Diagram

Use Case Diagram

High-level behaviors of the system, goals, actors

Sequence Diagram

Focus on time ordering of messages

Behavioral Diagram

Collaboration Diagram

Focus on structural organization of objects and messages

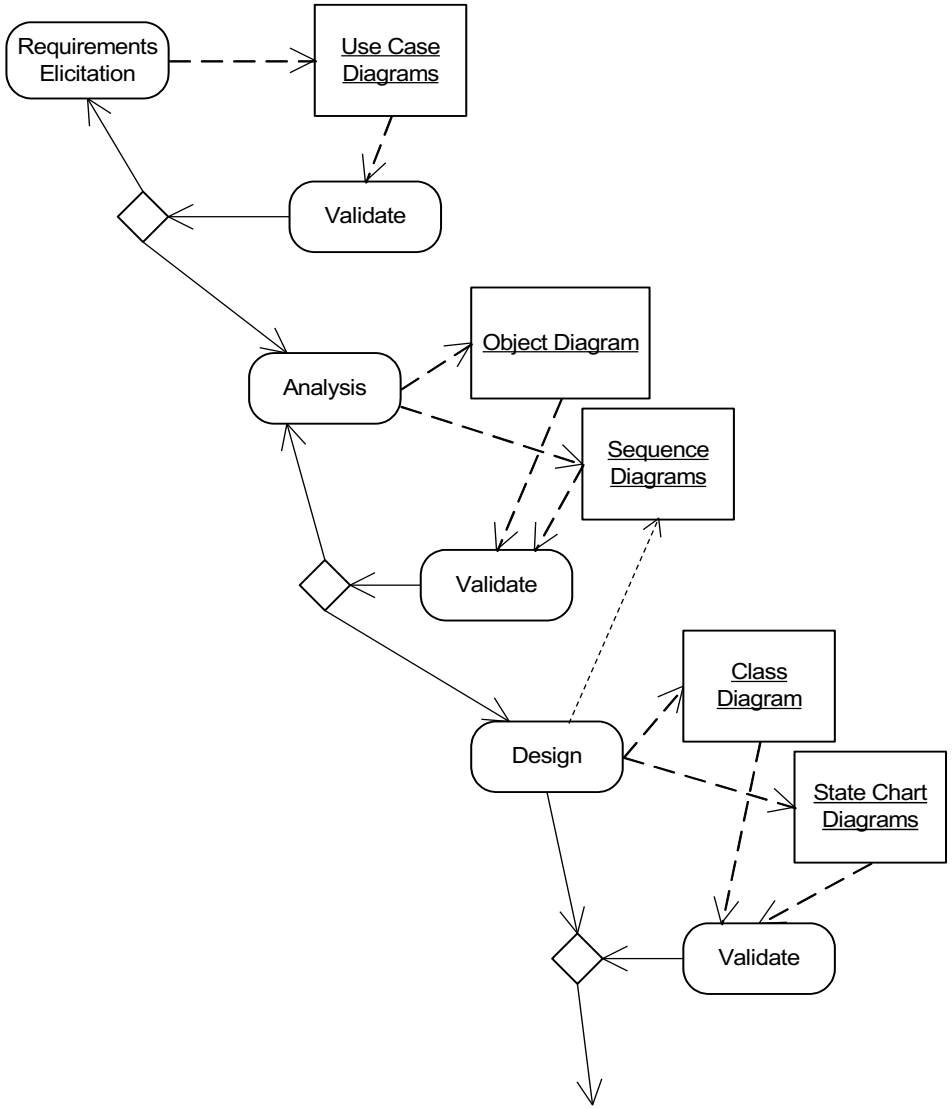
State Chart Diagram

Event driven state changes of system

Activity Diagram

Flow of control between activities

UML Driven Process Model



UML Part II

Modeling Requirements

- Use Cases
- Scenarios

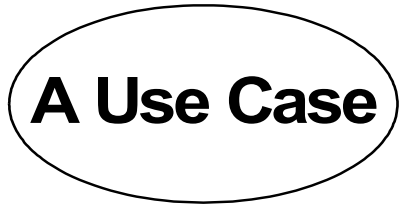
Use Case Diagram

A set of sequences

Each sequence represents the interactions of Actors outside the system with the system itself

Use cases represent the functional requirements of the system (non-functional requirements must be given elsewhere)

Use Case



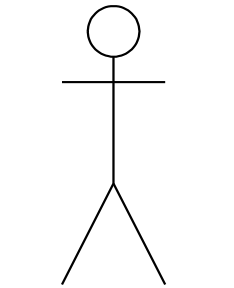
Each use case has a descriptive name

Describes what a system does but not how it does it

Use case names must be unique within a given package

Examples: Withdraw Money, Process Loan

Actor



An Actor

Actors have a name

An actor is a set of roles that users of use cases play when interacting with the system

They are external entities

They may be external as a System or DB

Examples: Customer, Loan officer, Bank

What is a Use Case

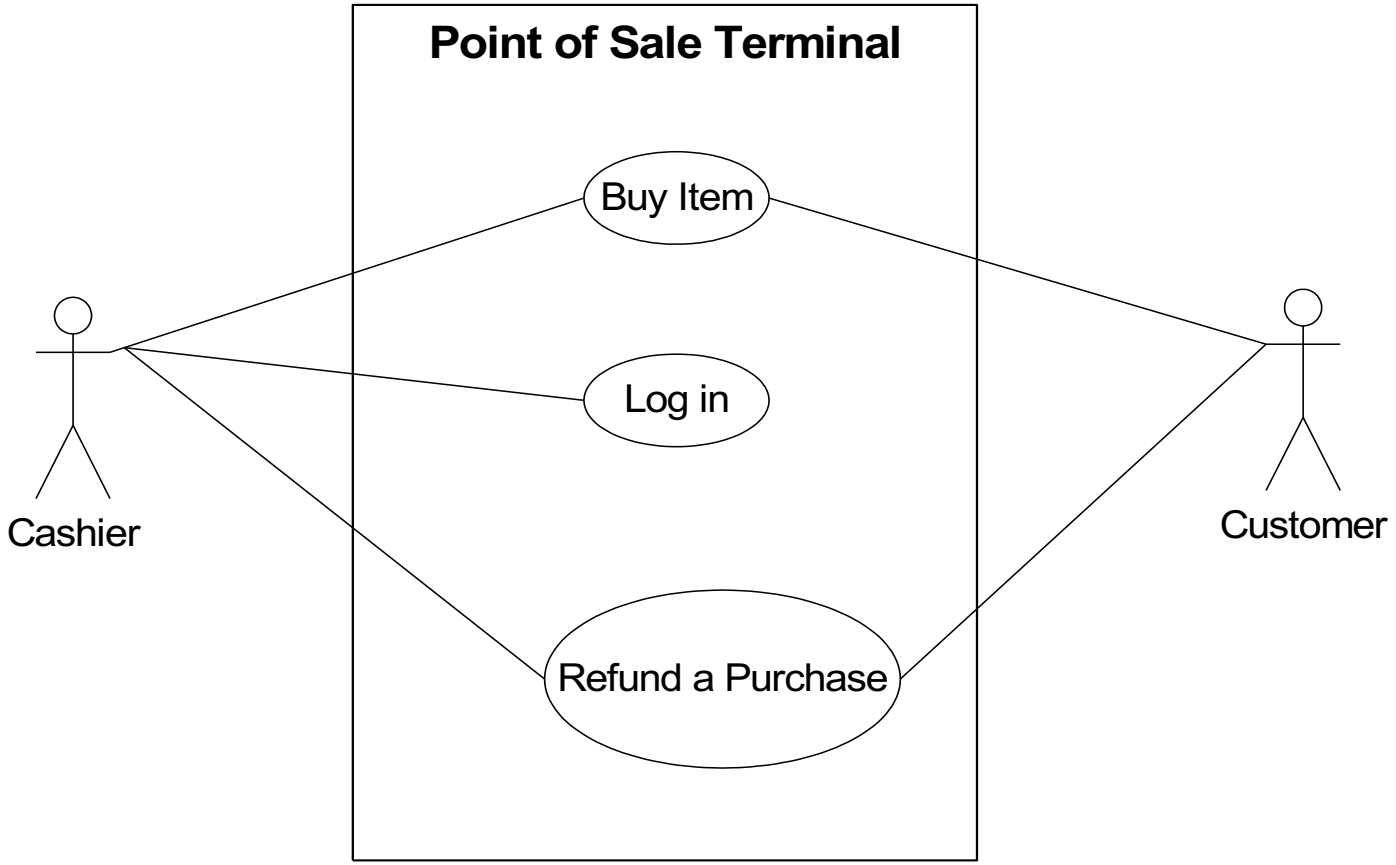
Use case captures some user-visible functionality

Granularity of functionality depends on the level of detail in your model

Each use case achieves a discrete goal for the user

Use Cases are generated through requirements elicitation

Use Case Diagram, POS Example



Use Case Elicitation—
Goals vs. Interaction

Goals – a thing the user wants to achieve

- Format a document
- Ensure consistent formatting of two documents

Interaction – things the user does to achieve the goal

- Define a style
- Change a style
- Copy a style from one doc to the next

Developing Use Cases

Understand what the system must do, capture the goals

Understand how the user must interact to achieve the goals, capture user interactions

Identify sequences of user interactions

Start with **goals** and refine into **interactions**

Refining Use Cases

Separate internal and external issues

Describe flow of events in text, clearly enough for customer to understand

- Main flow of events
- Exceptional flow of events

Show common behaviors with **includes**

Describe extensions and exceptions with **extends**

Use Case —Buy Item

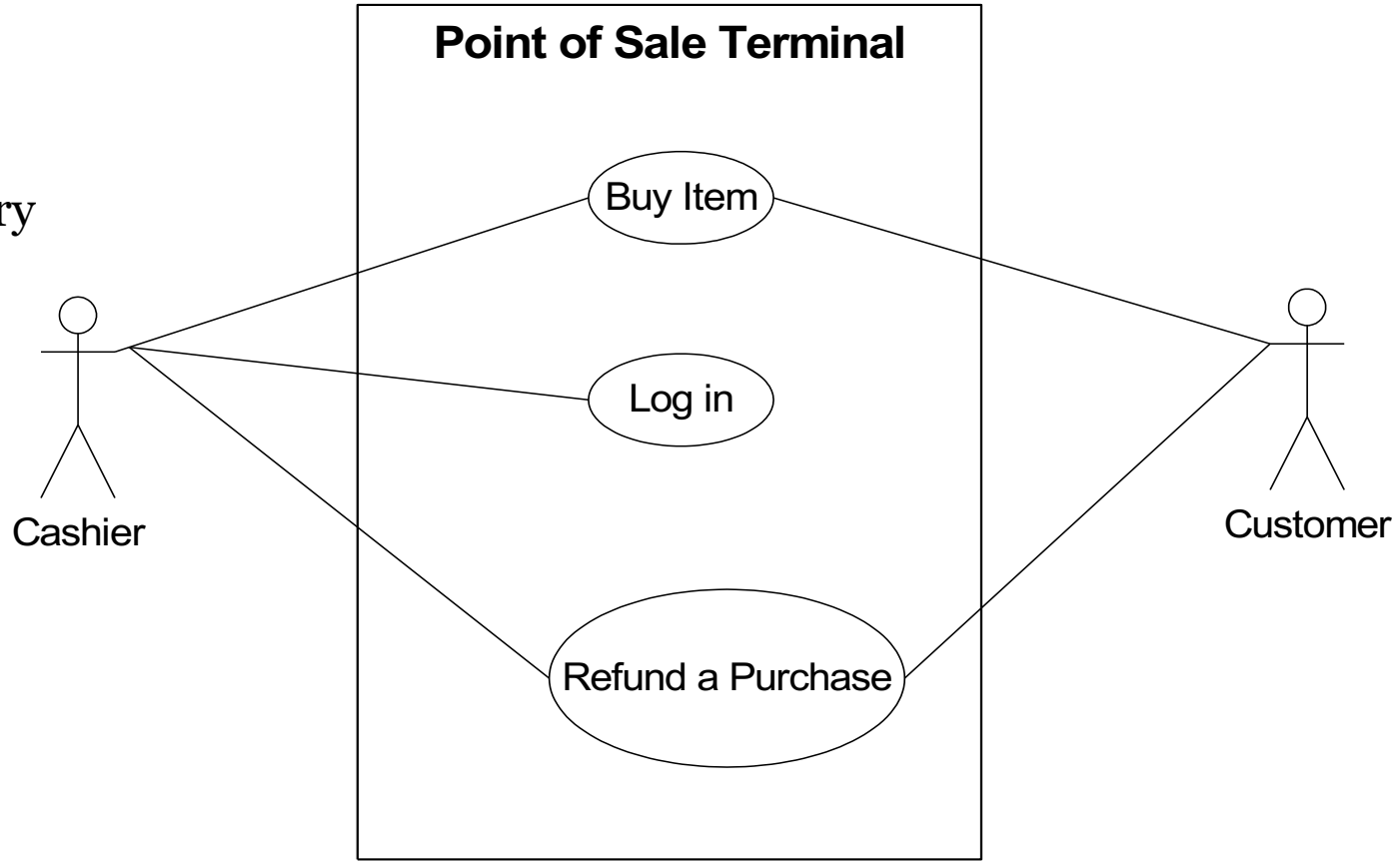
Actors: Customer (initiator), Cashier

Type: Primary

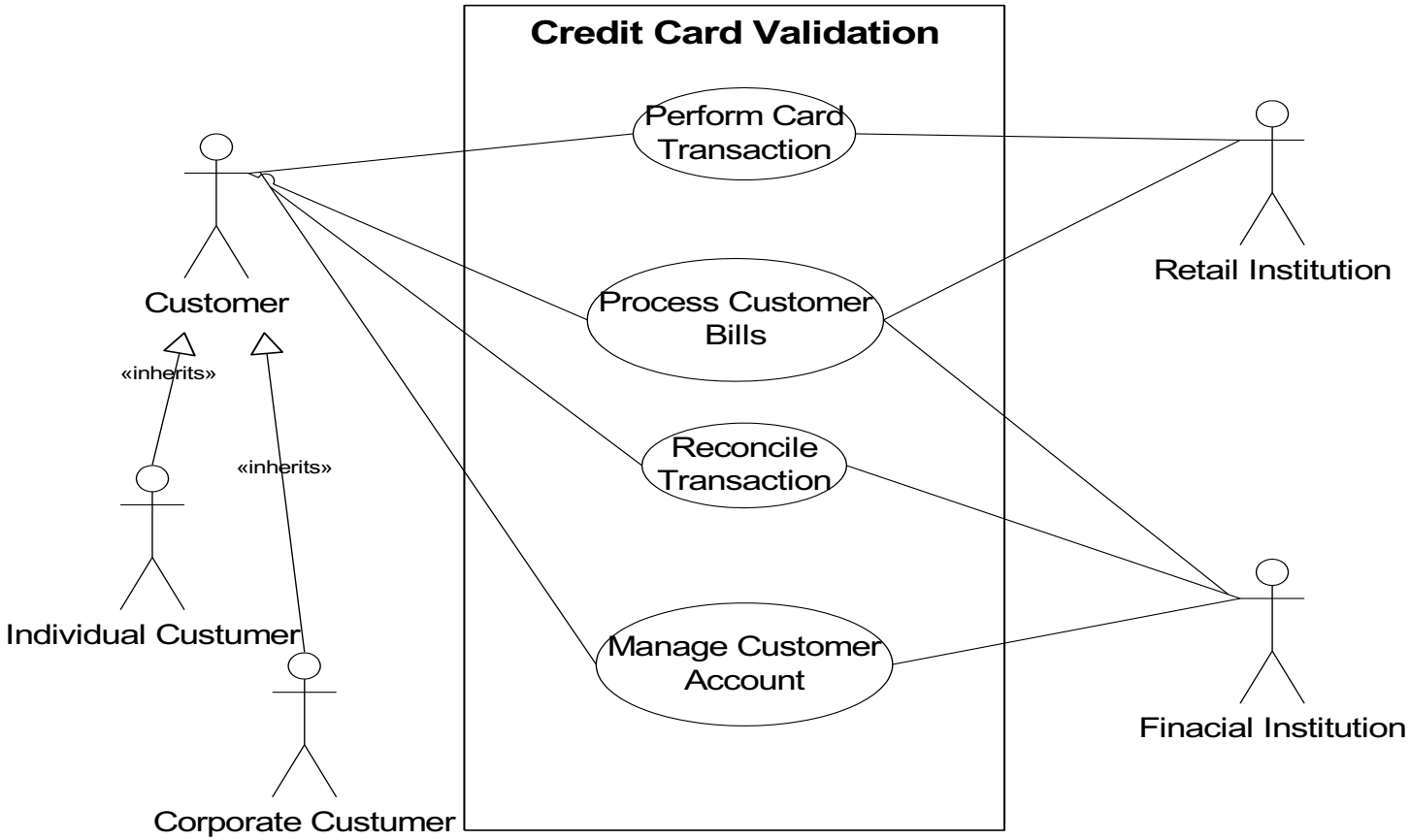
Description: The costumer arrives at the checkout with items to purchase. Cashier records purchases and collects payment. Customer leaves with items

POS Use Cases, in a Diagram

Use cases has a system boundary



Credit Card Validation System



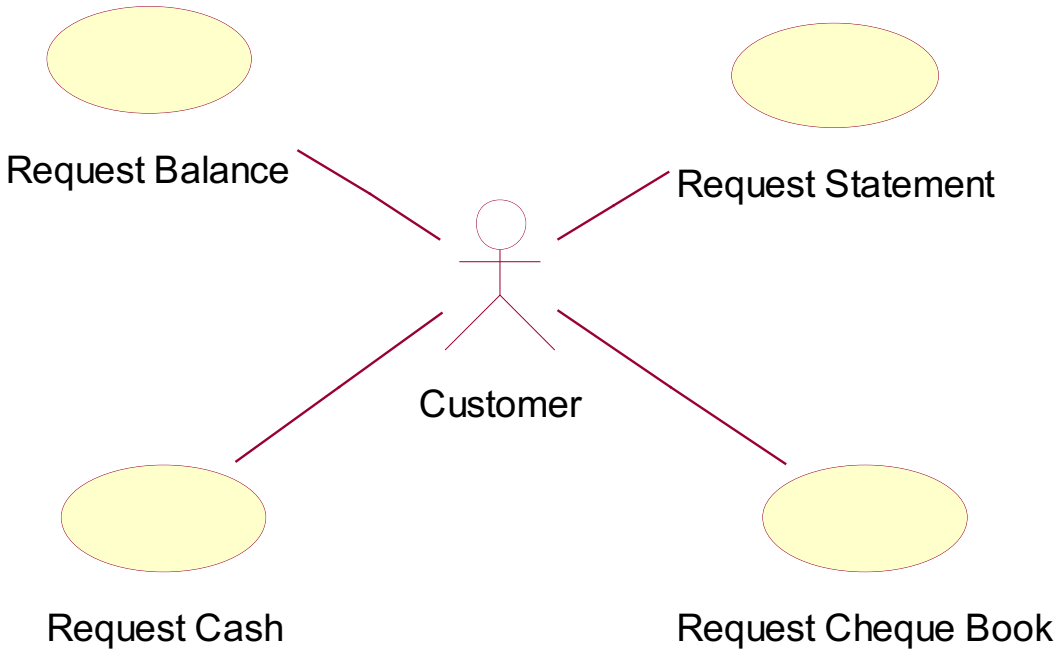
ATM, Use-Case Development

Identify a user **goal** and **interaction** with an ATM

In essence, this system (completely automated) provides the following functionalities:

- Request Cash
- Request Balance
- Request Statement
- Request Checkbook

Capturing Use-Cases

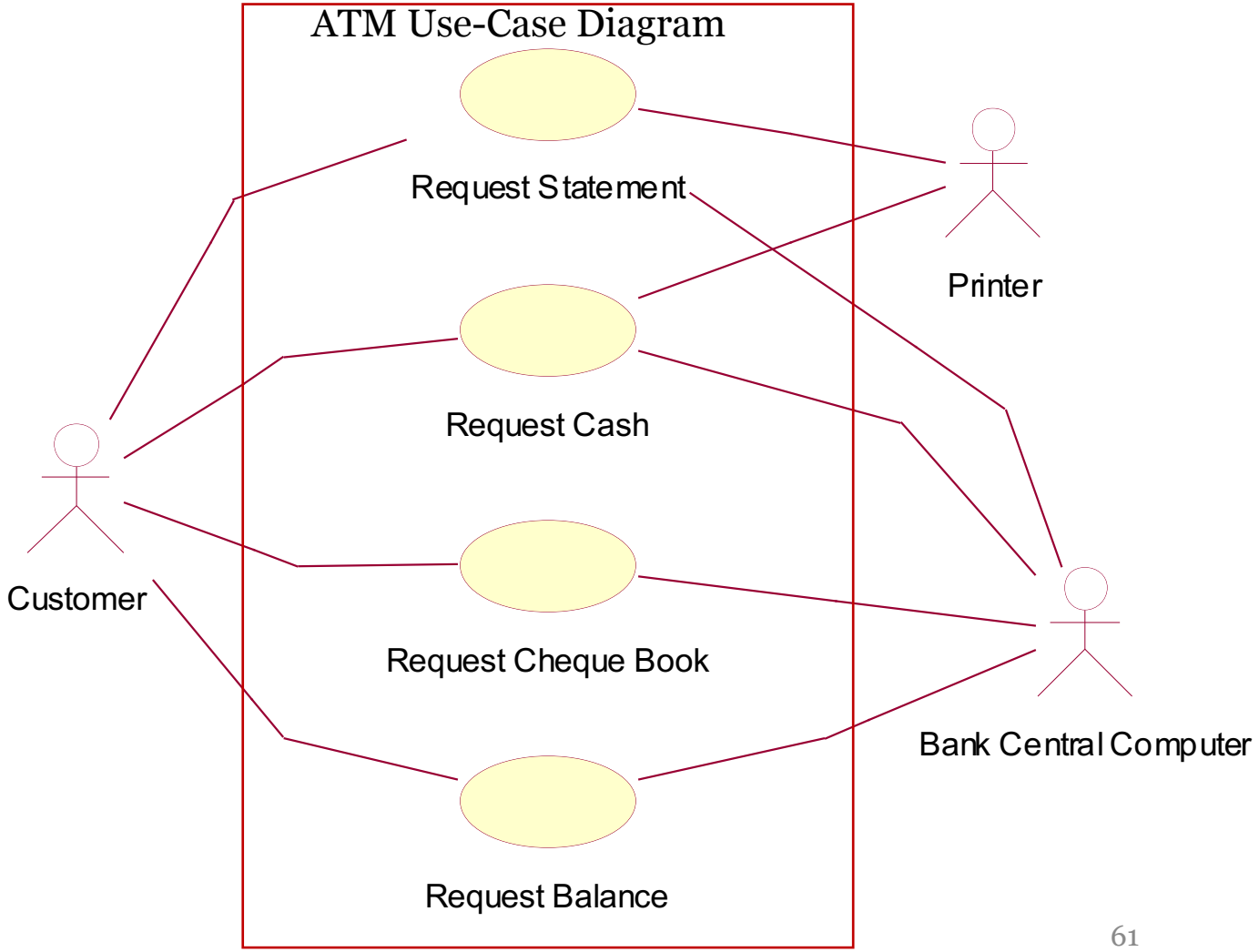


ATM Use-Case Diagram

Would we show the Bank Computer and Printer as Actors?

Possibly,

but because they do not initiate a use-case, the UML refers to them as "secondary" actors



A Use Case Diagram Looks So Simple, Why Bother With It?

Not many details of design and implementation

Focuses on what needs to be done, not on how it will be done

BIG-PICTURE

Use Case Diagrams Looks So Simple, Why Bother With them?

Customers can:

Easily relate to a use-case diagram and

Identify the major objectives of their business within it

Developers can:

Immediately assess the functionality required

And assess the risk involved in implementing each use case and

Assess resources

Use-Case Scenarios

A different scenario whereby the customer incorrectly enters their PIN and the transaction is aborted

A **different use-case scenario** since the user interaction and outcome are not the same as someone who interacts and successfully obtains money, not the goal!

'What-Ifs' type questions that can be posed during analysis. For example:

- What if the user's PIN is incorrectly entered?
- What if the user has insufficient funds in their account?
- What if the cash dispenser cannot read the card's magnetic strip?
- What if the cash dispenser is out of money?
- What if the bank's central computer is offline?

Use Case Scenarios in the Use-Case Request Cash

Start of Transaction

1. The user inserts their ID card into the system.
2. The system reads the magnetic strip from the card.
3. If the system cannot read the card, then **<<A1>>**
4. The system contacts the bank's central computer to request the PIN for the card and their account details.
5. If the bank's central computer cannot access users account then **<<A1>>**
6. The system prompts the user for their PIN.
7. The user enters their PIN.
8. If PIN cannot be authenticated **<<A2>>**
9. The user is prompted for the amount of the withdrawal.
10. The user enters the amount of withdrawal.
11. The system checks with the bank's central computer
12. If the user has insufficient funds **<<A3>>**
13. The cash is dispensed and the customer's account at the Bank Central Computer is debited with the withdrawal amount.
14. The card is returned to the user and a receipt is issued.

End of Transaction

A1: The user's card is returned. **End of Transaction**

A2: The user is given two more attempts to enter a correct PIN. If this fails, the card is kept and the transaction ends. Otherwise, resume the primary scenario.

A3: The user is allowed to enter a lesser amount or cancel the transaction. If cancel is chosen, the card is returned and the transaction ends. If the lesser amount is acceptable then resume the primary scenario.

Use Case Relationships —Includes

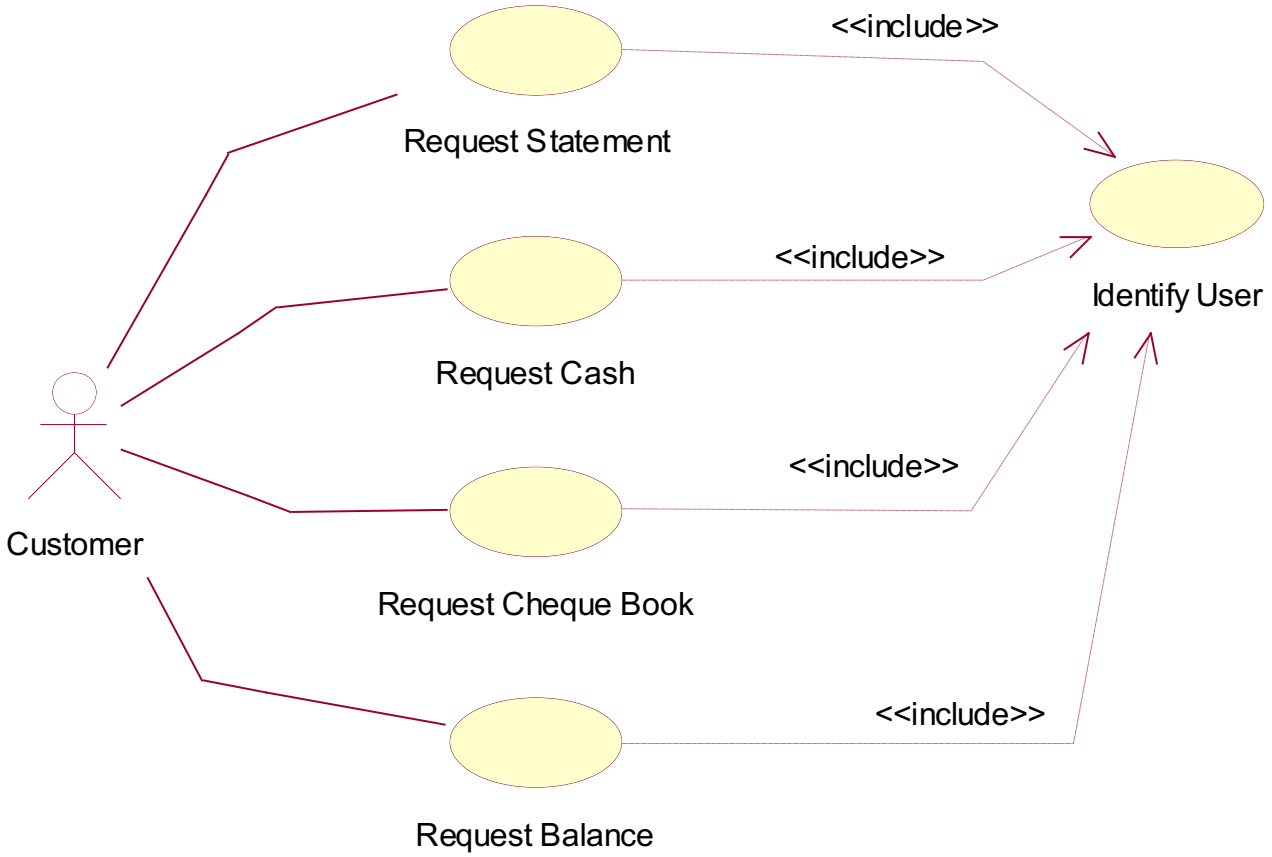
Sometimes apparent that there exists some commonality or replication between the steps involved in the execution of one or more use cases

Request Cash, Request Balance, Request Statement, Request Check Book: User is required to insert their ID card and enter their PIN, which is then verified by the bank central computer

Rather than duplicate, chose to represent with a use case Identify User

Dashed line indicates a dependency relationship

Arrow points to the use-case that will be included


<<include>>
Request Statement
Identify User
<<include>>
Request Cash
<<include>>
Customer
Request Cheque Book
<<include>>
Request Balance

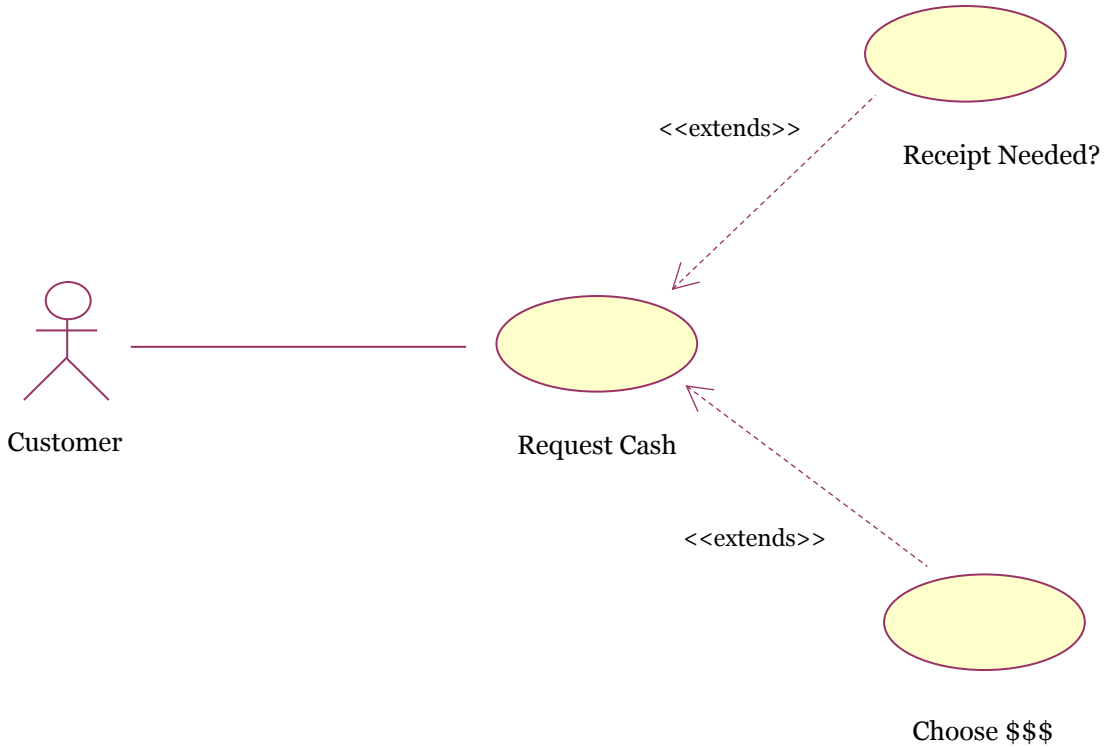
Use Case Relationships – Extends

The final use-case relationship is one that allows extensions to be made to a use-case.

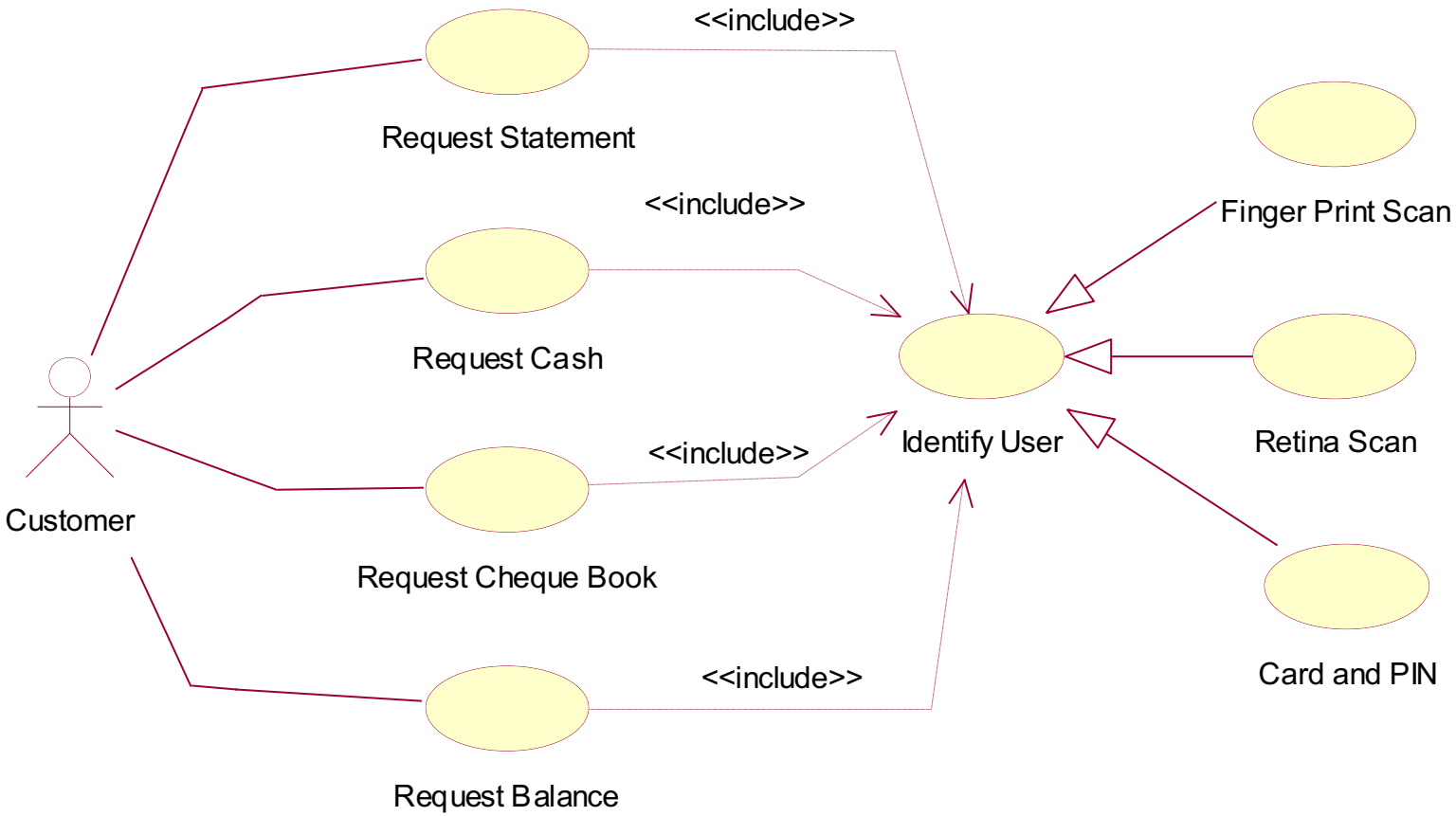
This can be used to model optional behavior, particularly interesting and important scenarios within a use-case.

Document them in their own use cases and add an extends relationship to their parent or containing use-case

Use Case Relationships – Extends



Use Case Relationships —Generalization



Finger Print Scan,

Retina Scan

Card and Pin

each **is an** identify user use case

UML Part III

Object Oriented Analysis

- Classes & Objects
- Class Diagrams

Requirements to Analysis to Design

From Use Case diagrams (**Requirements**) an initial set of objects and classes can be identified

This is the first step of **OO Analysis**

The second step is to refine the use cases through interaction diagrams (**OO Design**)

What is a Model?

A model is a simplification of reality

Modeling achieves four aims:

- Visualize a system
- Specify the structure or behavior of a system
- Template guides you in constructing a system
- Documents the decisions you have made

We build models of complex systems because you **cannot comprehend** a system in its entirety

We build models to better **understand** the system we are developing

Principles of Visual Modeling

Model we create influences how the problem is attacked

Every model may be expressed at different levels of precision

Best models are connected to reality

No single model is sufficient

Basic Principles of Object Orientation

Abstraction

Encapsulation

Modularity

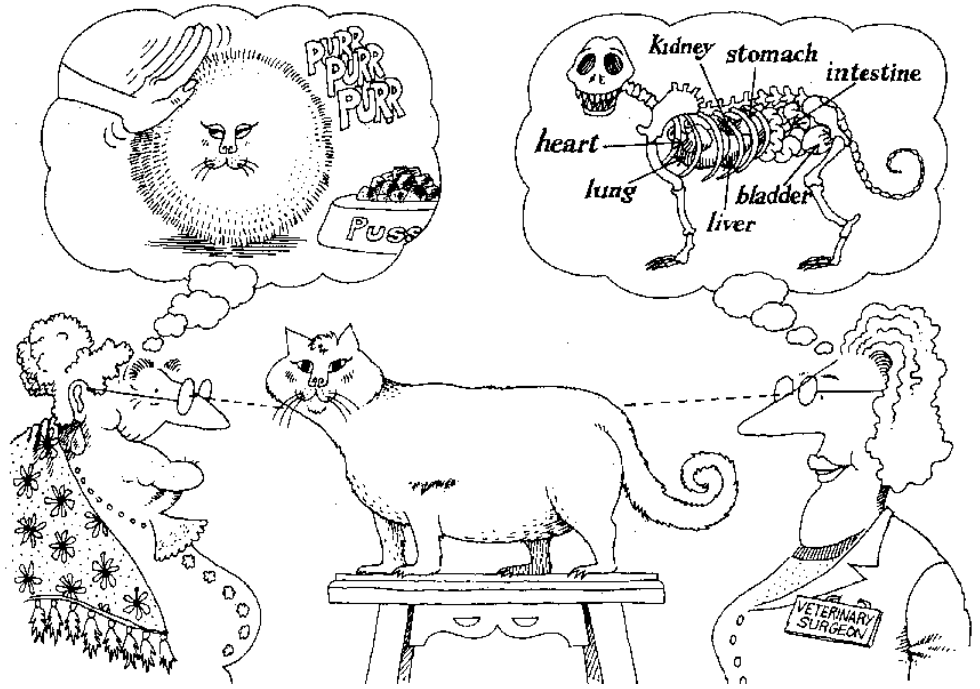
Hierarchy

What Is Abstraction?

Essential characteristics of an entity that distinguishes it from all other kinds of entities

Depends on the perspective of the viewer, not a concrete manifestation

Abstraction: Is the process of removing characteristics from something in order to reduce it to a set of essential characteristics
[Dictionary of IT Terms By S.k. Bansal]

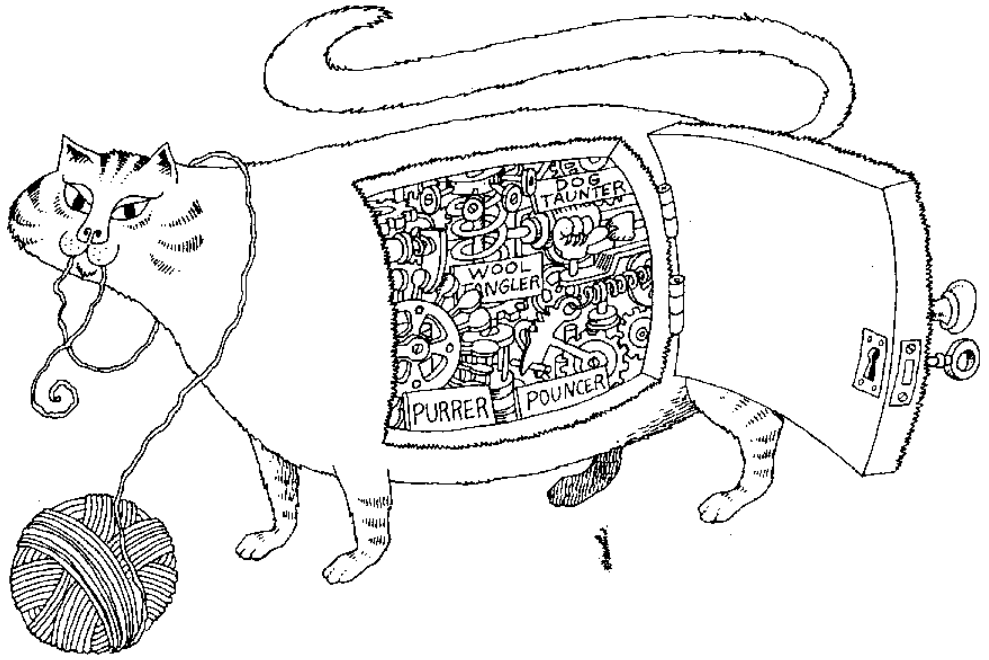


From *Object-Oriented Analysis and Design with Applications* by Grady Booch, 1994

What Is Encapsulation?

Hides implementation from clients

- Clients depend on an interface
- Improves the resiliency of the system,
  i.e. its ability to adapt to change

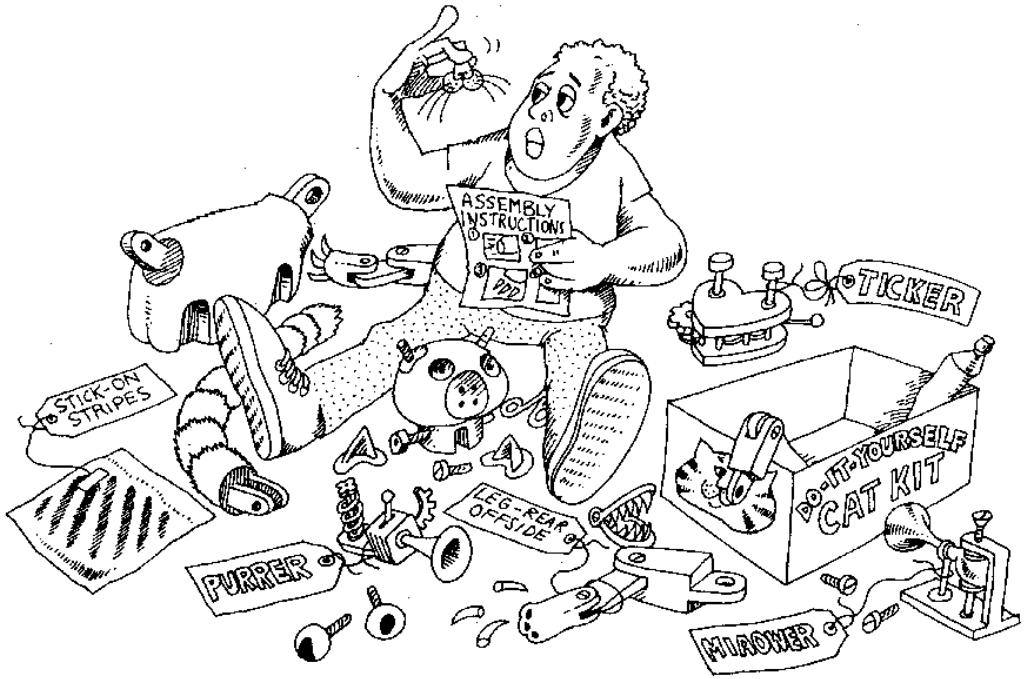


From *Object-Oriented Analysis and Design with Applications* by Grady Booch , 1994

What Is Modularity?

Breaks up something complex into manageable pieces

Helps people understand complex systems



From *Object-Oriented Analysis and Design with Applications* by Grady Booch , 1994

Object Oriented Decomposition

Identifying objects derived from the vocabulary of the problem domain

Algorithmic view highlights the ordering of events

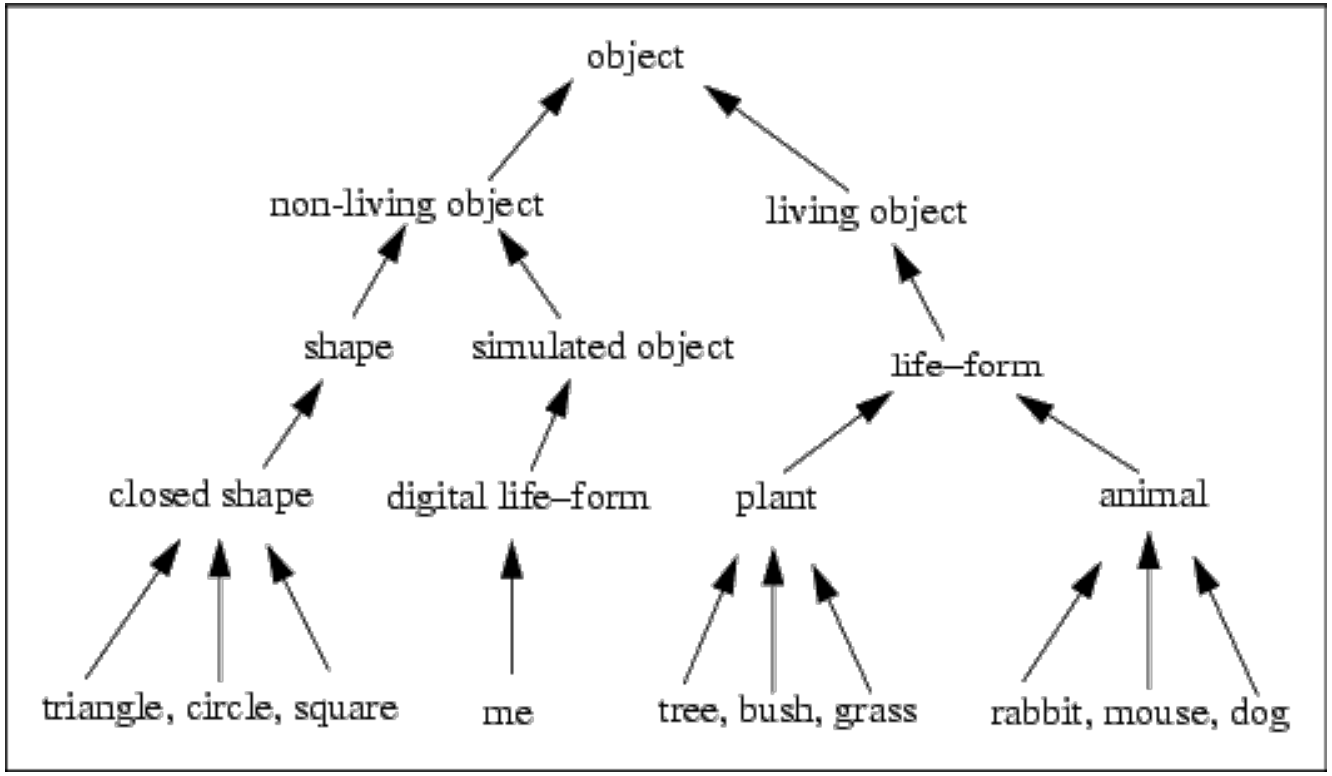
What Is Hierarchy?

Activity of pushing commonality up the inheritance structure

A generalization and specialization, a **hierarchy**

In OO, classes at the top are more general (or abstract) and the classes at the bottom are more specific (or concrete)

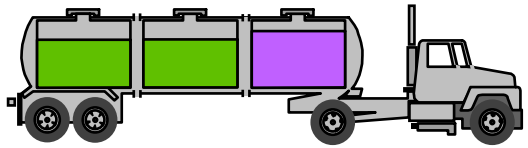
Hierarchy, Example .



What Is an Object?

Informally, an object represents an entity, either physical, conceptual, or software

- Physical entity

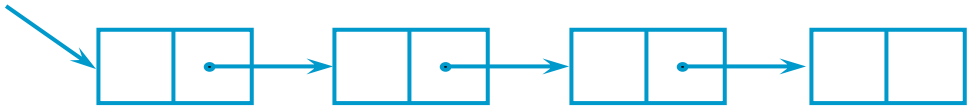


Truck

- Conceptual entity

Chemical Process

- Software entity



Linked List

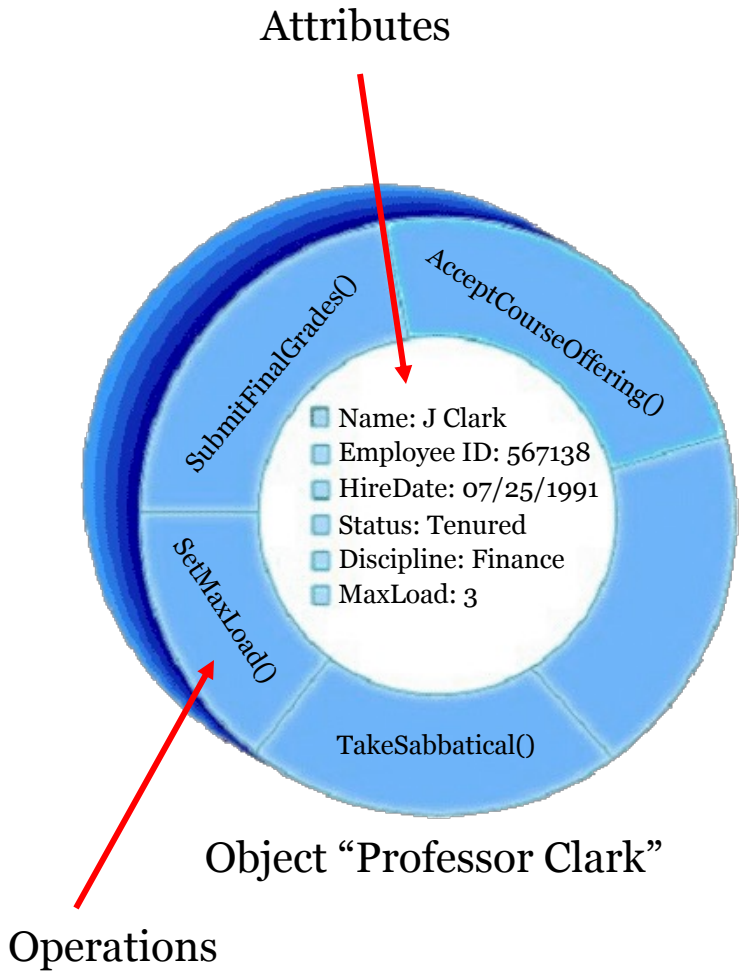
Object in OO

An object is an

entity with a well-defined boundary and

identity that encapsulates state and behavior

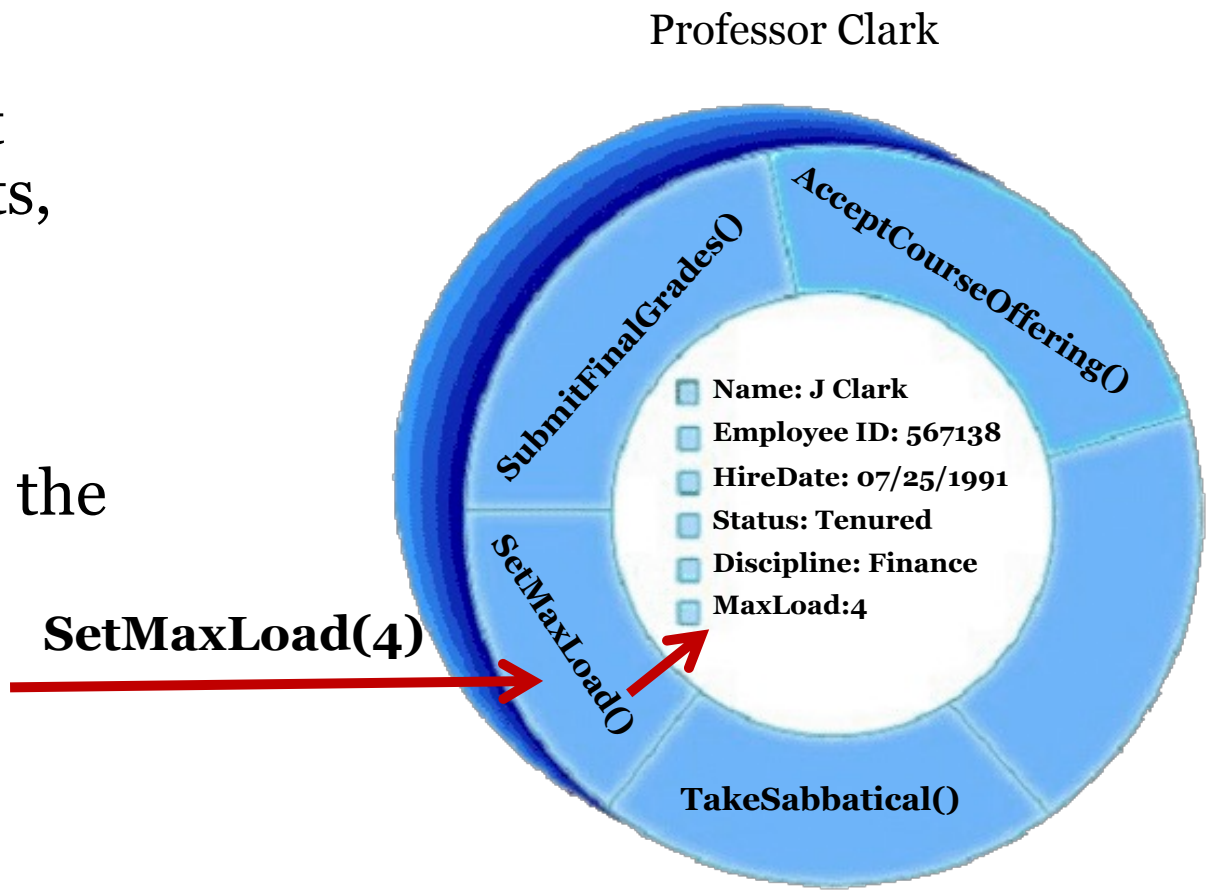
- State is attributes and relationships
- Behavior is operations and methods



Objects and Encapsulation

Encapsulation:

- Protects the supplier object from incorrect use by clients, **Hiding**
- Protects the clients from implementation changes in the supplier, **Coupling**



What Is a Class?

A class is a description of a set of objects that share the same *attributes*, *operations*, *relationships*, and *semantics*

- An object is an instance of a class

A class is an abstraction in that it

- Emphasizes relevant characteristics (public)
- Suppresses other characteristics (private)

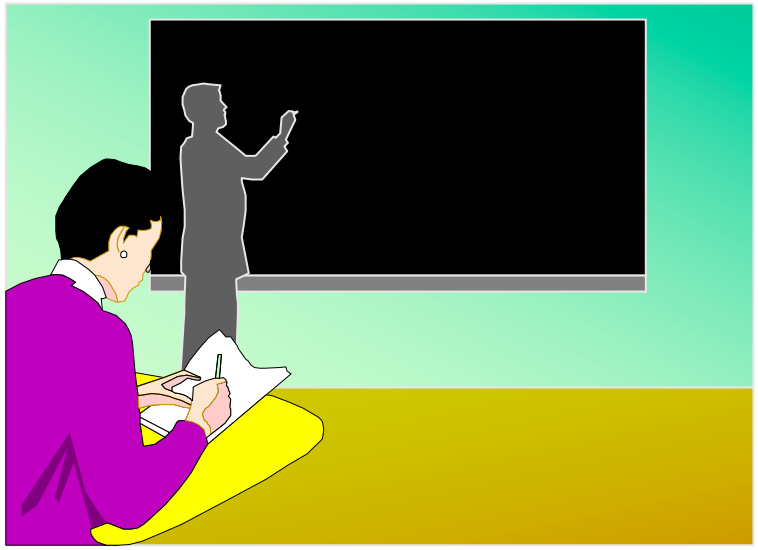
A Sample Class

Class

Course

Attributes

Name
Location
Days offered
Credit hours
Start time
End time



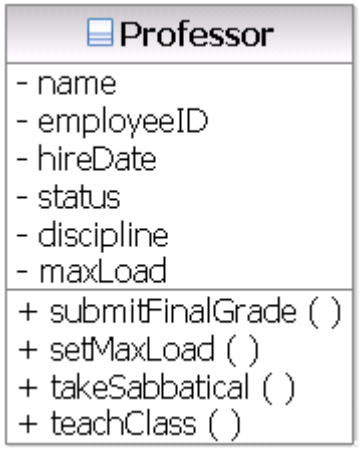
Behavior

Add a student
Delete a student
Get course roster
Determine if it is full

Representing Classes in UML

A class is represented using a rectangle often with three compartments:

- Class name
  - It should be a simple name and it should reflect exactly what the class is and does
- Structure (attributes)
- Behavior (operations)



Representing Classes in UML

Style Guidelines

- Capitalize the first letter of class names
- Begin attribute and operation names with a lowercase letter

Compartments

- Only the name compartment is mandatory
- Additional compartments may be supplied to show other properties

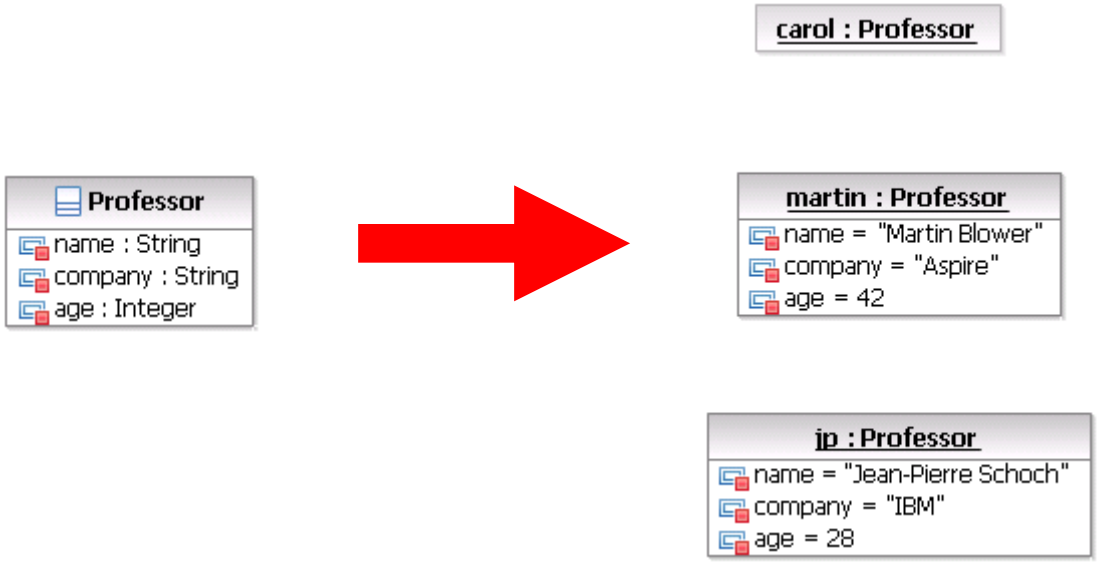
- Attribute compartment suppressed,
- visibility shown as “decoration”,
- and operation compartment filtered to show only two operations

Relationship between Classes and Objects

A class is an abstract definition of an object

- Defines structure and behavior of each object
- A template for creating objects

Classes are not collections of objects



Classes Collaborate

Objects collaborate by sending messages

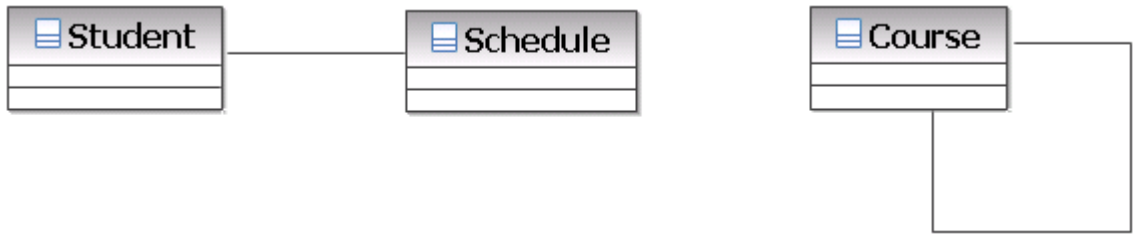
- Client object must have a link with the supplier

A link between two objects means a relationship:

- The relationship may be a structural relationship, an association
  - aggregation, composition
- Or a non-structural relationship called a dependency

What Is an Association?

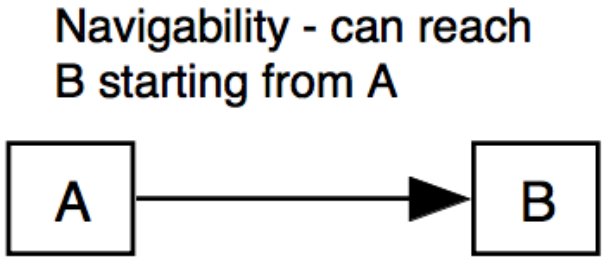
Semantic and Structural relationships between two or more classifiers specifies connections among their instances



What is Navigability? .

A can see (has an attribute referencing) **B**

In contrast **B** has no idea about/referencing to **A**

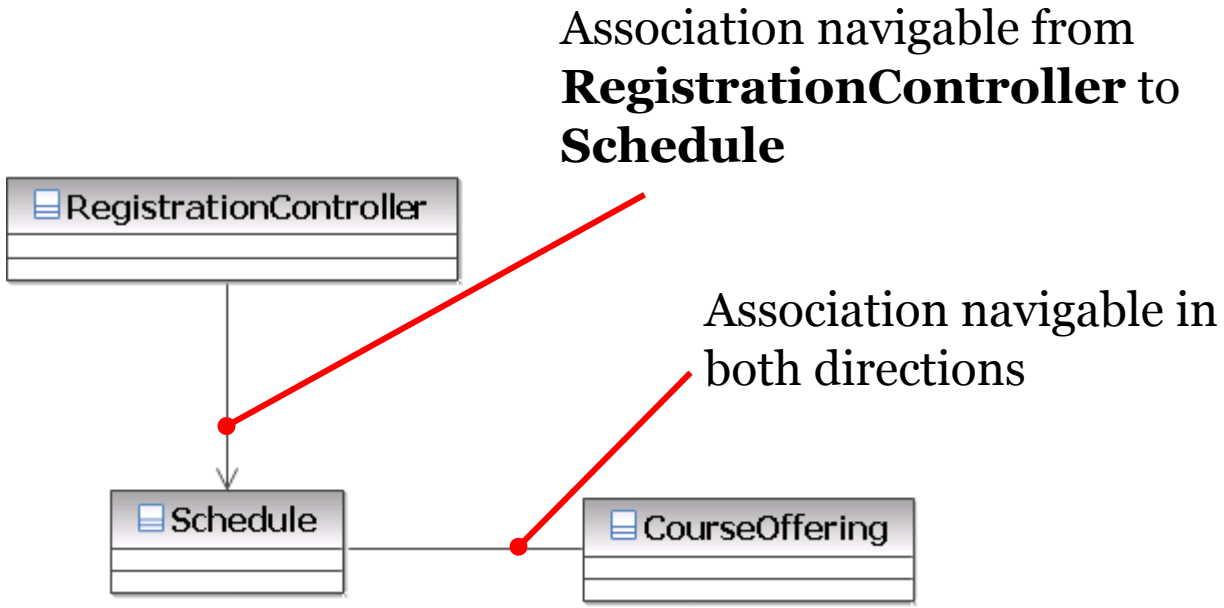


What is Navigability?

Possibility to navigate from one class to another

One instance of a class navigable to another instance of the same class or of another class

- Bi-directional association
- One-way association



Naming Associations

To clarify its meaning, an association can be named

The name is represented by a label as shown below

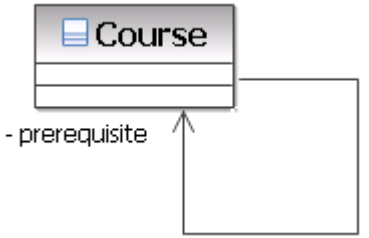
Usually, a verb or an expression starting with a verb

Association Roles

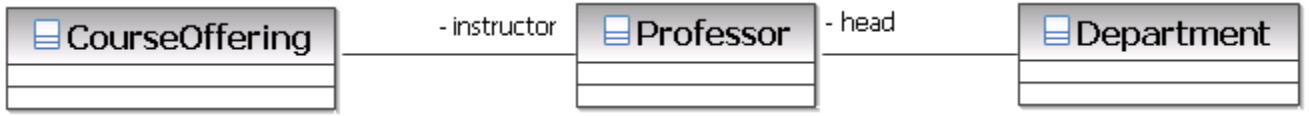
A role name specifies the role that a class plays in its relationship with another class

Role names are typically names or noun phrases

A role name is placed at the association end next to the class to which it applies



For more on associations, names, and roles … .

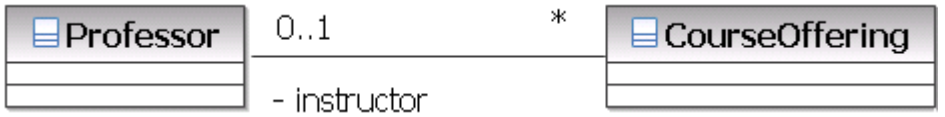


What Is Multiplicity?

Multiplicity is the number of instances one class relates to ONE instance of another class

For each association, two multiplicity of decisions to make, one on each end, e.g.:

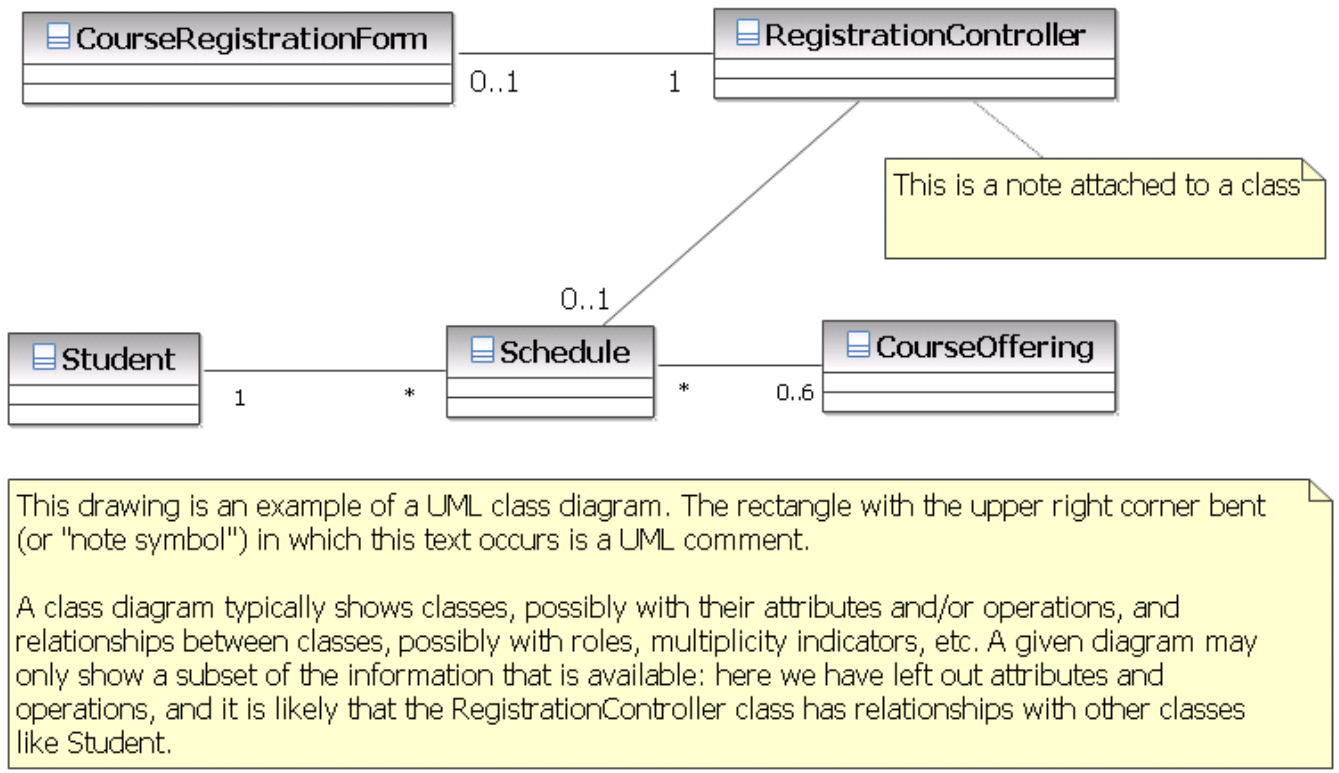
- For each instance of Professor, many Course Offerings may be taught
- For each instance of Course Offering, there may be either one or zero Professor as the instructor



Multiplicity Indicators

Multiplicity	Option	Cardinality
0..1		No instances or one instance
1..1	**1**	Exactly one instance
0..*	*****	Zero or more instances
1..*		At least one instance
5..5	**5**	Exactly 5 instances
m..n		At least m but no more than n instances

Multiplicity Example

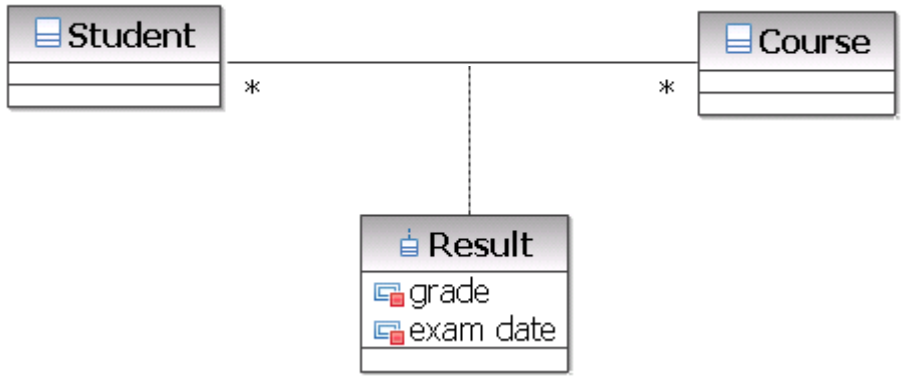


Association Classes

n-n relationship between **Student** and **Course**

If you have to capture the grade and related attributes received by a student for a given course, where would you place the grade? On *Student*? On *Course*?

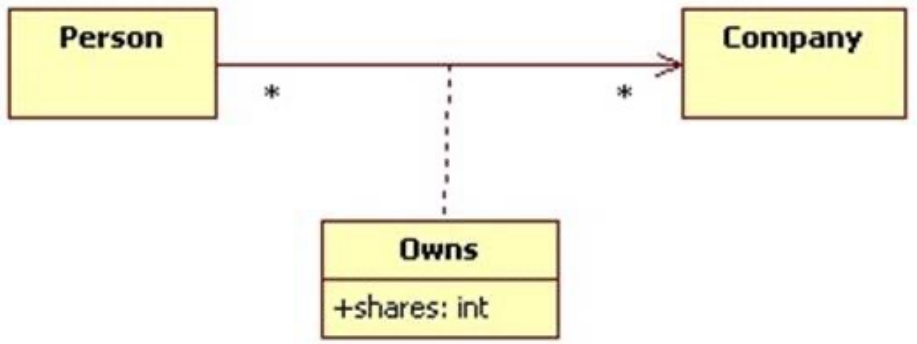
- The answer is on the association itself, association class **Result**
- Represents the association of exactly one student and one course



Association Classes .

Person X owns N shares of **Company Y**. **Y** has people. Where will N be stored?

N is neither an attribute of **Company** nor **Person**.



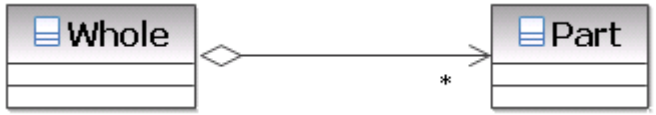

```
public class Company { … }

public class Person {
  private List<Owns> investments;
  public void add(Company c, int shares) {
    investments.add(new Owns(c, this, shares));
  }
  // etc.
}

class Owns {
  private Person owner;
  private Company company;
  private int shares;
  public Owns(Company c, Person p, int num) {
    comnpany = c;
    person = p;
    shares = num;
  }
  // etc.
}
```

What Is an Aggregation?

A whole-part strong relationship between the aggregate (the whole) and its parts

"is a part-of" relationship

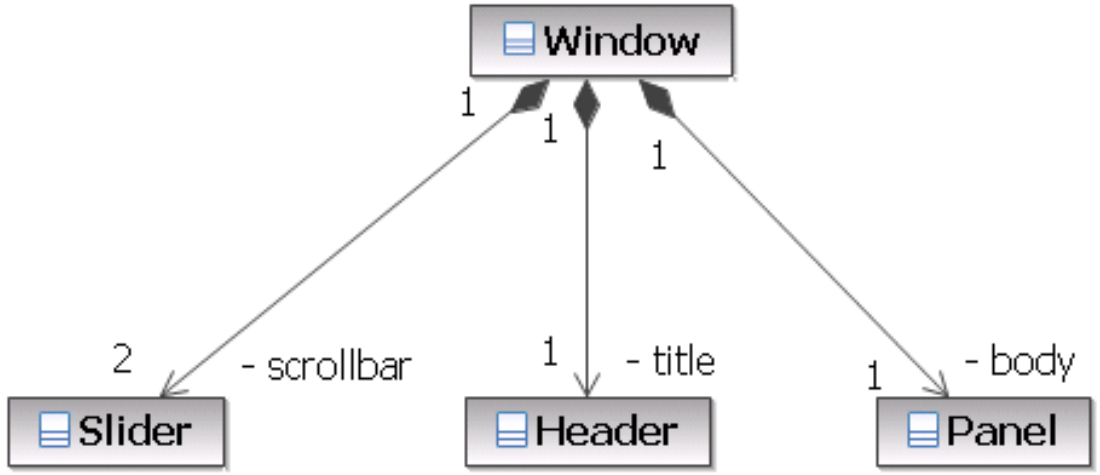
Multiplicity is represented like other associations



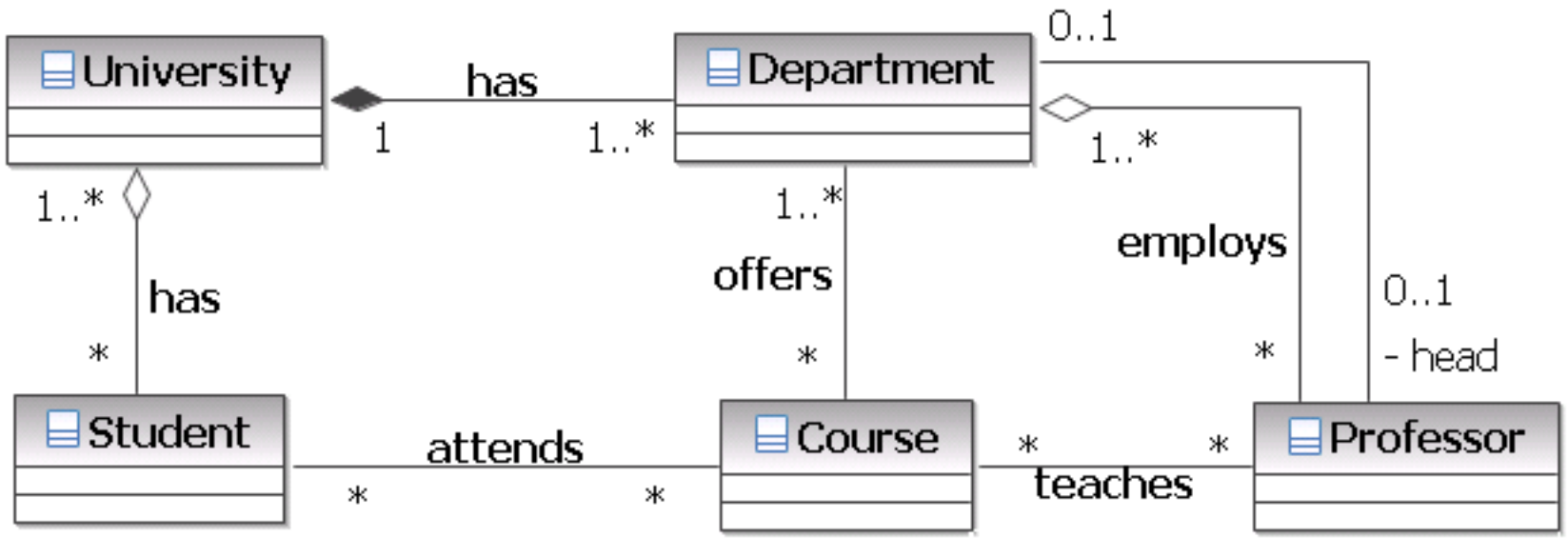
What Is a Composition?

A form of aggregation with very strong ownership and coincident lifetimes

- The parts cannot survive the whole/aggregate



Example


University
has
Department
0..1
1
1..*
1..*
1..*
has
offers
employs
0..1
- head
*
*
*
Student
attends
Course
teaches
Professor
*
*
*
*

What Is A Dependency?

A non-structural relationship between two classes

- Client needs access to services provided by the supplier
- But doesn't need to maintain a permanent relationship with the supplier objects (transient relationship)

A **dependency** may result from:

- A local declaration within the body of an operation (see ***op1***)
- Supplier appears as a parameter type (***op2***)