Software Architecture

Practical Workshop

Course Logistics

- Introductions
  - Name
  - What’s your experience with OO?
  - What’s your experience with Software Architecture?
  - How have you ever used UML?
- Course Structure
  - Lecture—recap and expand
  - Main focus will be design session and discussions
  - Work in groups of three
- Choose your groups now

Exercise Outline

- Your choice of:
  - Car Rental
    - A “from scratch” car rental system
    - Needs web and local interfaces
  - Your own problem
    - Reasonably simple, simplified, or cut-down, so as to be manageable in class
    - Something you understand well
- Be thinking about which, or what combination you would like to work on

Course Outline

- Complex System Development
- Object Oriented Programming
- Modeling Complex Systems
- Unified Modeling Language
- Modeling Approaches
- Design Patterns
- Software Architecture Concepts
- Client-side Architectures
- Middle-tier Architectures
- Server-side Architectures

Complex System Development

Module 1

Advent of Complex Systems

- Evolution of computing
  - Hardware-based solutions
  - Automated solutions (mainframe age)
  - Software solutions (client-server age)
- Desire for business efficiency
  - Drove complexity
  - Drove expansion
  - Drove adoption
- Resulted in standardized
  - Programming languages
  - Processes and procedures
  - Support tools

Booch's Complex System Characteristics

- Hierarchical in structure
  - It is because of this hierarchical nature,
  - We can decompose into systems and subsystems
- Relative
  - One man's trash is another man's treasure
  - You may see "primitives" another may see "abstractions"
- Separation of Concern
  - Decomposition of a system into "parts"
  - Parts are either intra or inter dependent
- Patterns
  - Like in building architecture, systems follow patterns
  - Commonly classified as "Design Patterns" Modular

General Characteristics of Complex Systems

- Interactive
  - Humans and other “actors” interact with system
  - Some interactions are known, and some are unknown
- Integrative
  - Systems and sub-systems integrate with other systems
- Event Driven
  - Systems either have internal or external “actors”
  - “Actors” cause the system to change “states”
  - Notification of those changes

Enterprise Characteristics of Complex Systems

- Cross-organizational
  - Different groups have different expectations
- Mission-critical
  - New types of requirements effect the design

Building Complex Systems

- A modern approach
  - New software engineering processes
  - New programming paradigms
  - New programming languages
  - New technologies

- Required to manage
  - Complexity
  - Changing requirements
  - Business drivers and stakeholders

Bringing Order to Chaos

- Given today's systems are complex . . . There needs to be a way to digest, dissect, and manage the complexity

- How do you "digest and dissect" the complexity?

Purpose of Software Process

- Capture “big picture” and details of system
- So that it can be decomposed
- Manage the decomposition process to facilitate a design
- So that the design can be realized in software

Purpose of Software Design

- Satisfy a functional specification
- Meeting implicit or explicit criteria
- Provide “blueprint” for how to build the system

Purpose of OO Programming

- Implement software design in terms of “real world” entities
- Allows decomposition of complexity into “things”
- “Things” are represented as Object, Components, Modules

System Decomposition

- Does process guarantee design?

- Core “process” of the design process is System Decomposition
  - “Science and art” of designing a systems
  - Purpose of decomposition is to identify “ingredients”

- Ingredients will be:
  - “Core” elements
  - Abstractions
  - Components
  - Modules
  - Subsystems

How to Decompose a System

- GUI approach – how does the user interact with the system
- Service approach – how do other systems interact with the system
- Database-driven approach – how do we reverse engineer the db into a system

How to Decompose a System

- Functional approach – decompose system by “behaviors”
- Data approach – decompose system by “states”
- Object approach – decompose system by “states” and “behaviors”

System Composition of a Cobb Salad

I ate a cobb salad, it had:

- Blue cheese
- Bacon
- Tomatoes
- Egg
- Chicken
- Carrots
- Lettuce
- Dressing

Is this composition complete enough?

System Decomposition of Trip

I drive to the beach in a car:

- I got in the car
- I started the Engine
- I put the Transmission into drive
- I turned the steering wheel a bunch to change directions
- I stopped the car with the brakes
- I put the Transmission into Park
- I turned off the Engine
- I got out of the car

Is this composition complete enough?

When is Decomposition Done?

- Some would say “never” …
- Realistically, it is an iterative process that leads to design
- The moment the design is complete enough to build software, build it

Focus of This Class

- Object Oriented Decomposition
- Not focused on a single “software process”
- Focused on “practicing” decomposition

Lab 1: Value of Decomposition

As a group of 3, spend 20 minutes identifying:

the pros and cons of:

- Functional Decomposition
- Data decomposition
- “Top down” (UI)
- “Bottom up” (database drive)

Lab 2 : Project Writeup

- To prepare for our workshop, you'll need a project write up

- The project write up should describe
  - Primary goal of system you are building
  - Main Entities in system
  - Their interactions

Lab 3 : Logical Reflection

- Review your write up
  - Are there any missing entities
  - Are there enough details

- Consider correlations between “function” and “data”
  - Does some data go with some functions
  - Is the data “standalone”
  - Are the functions stand-alone
  - Group appropriately

Object Oriented Programming

Module 2

History of OO Programming

- Created out of need for description and simulation of complex systems

- Progression of Languages
  - SIMULA 1 (1965) and Simula 67 (1967)
  - Smalltalk (mid 1970s)
  - C++ (mid 1980s)
  - Eiffel (1985)
  - Java
  - C#

Definition of OOP

Object-oriented programming is..

… a method of implementation in which programs are organized as cooperative collections of objects

… each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Key Elements of OOP Definition

1. OOP uses objects as its fundamental building block

2. Each object is an instance of a class

3. Classes are related to another through a hierarchical relationship

OOP Concepts

- Classes
- Objects
- Instantiation
- Message Passing
- Inheritance*
- Polymorphism*

* We'll get to these later

OOP Concepts : Classes

- Description of real-world ‘thing’
  - Basic form would be data type
  - Or a struct
- More than just data ***type***; provides description
  - Of data (states)
  - Of functionality (behaviours)
- Commonly described as “Blueprints”
- Define how an object will look and act

OOP Concepts : Objects

- Concrete representations
  - Occupy memory
  - Have state
  - Perform actions
- Instance of class
  - Can be created
  - Can be destroyed

OOP Concept : Instantiation

- Process of creating an object from a template
- Translate idea into reality
- Typically performed in two step process:
  - Allocation memory
  - Initialize memory

OOP Concepts : Message Passing

- OO doesn't talk about calling a function, subroutine, procedure, or sub-program
  - Pass a message, or
  - Call a method
- Objects are considered as ‘autonomous’ things possibly with own compute power
  - Objects have their own state and own behaviour
  - Can simplify multi processor, or distributed system design

Lab : Organize Project

- Organize your project into a OO terms
  - Objects
  - Behaviors

- Are there similarities:
  - In state
  - In behavior
  - In functionality

- Simple approach:
  - Nouns – Objects
  - Verbs - Behaviors

The Object Model

Module 3

OO Analysis

Object-oriented analysis is …

… a method of analysis

… that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

OO Design

Object-oriented design is…

… a method of design

… encompassing the process of object-oriented decomposition

… and a notation for depicting the models of the system under design.

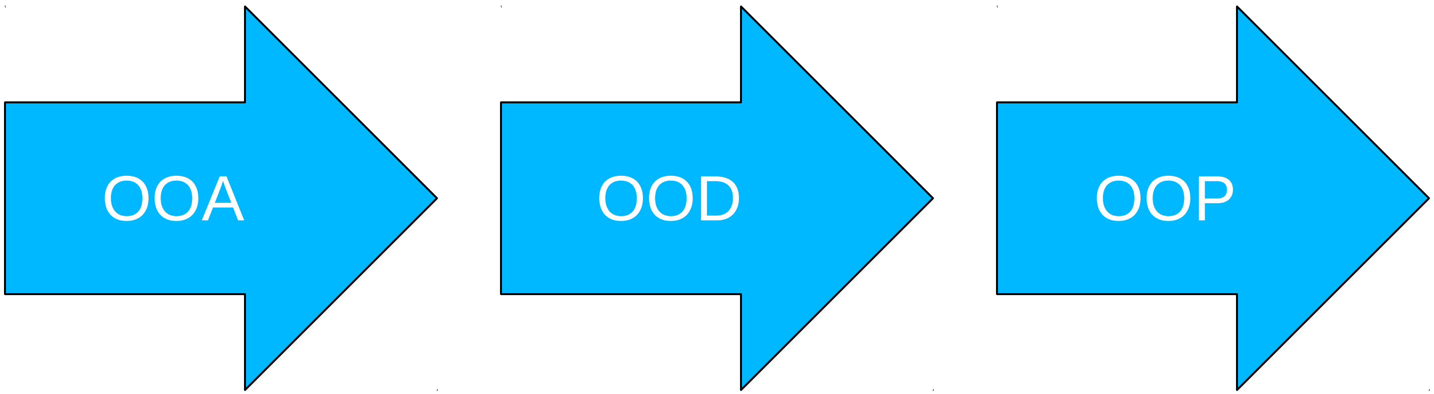
Definition of OOP

Object-oriented programming is..

… a method of implementation in which programs are organized as cooperative collections of objects

… each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

OOA – OOD - OOP



Premise of OO

- Model real-world entities (objects) in software

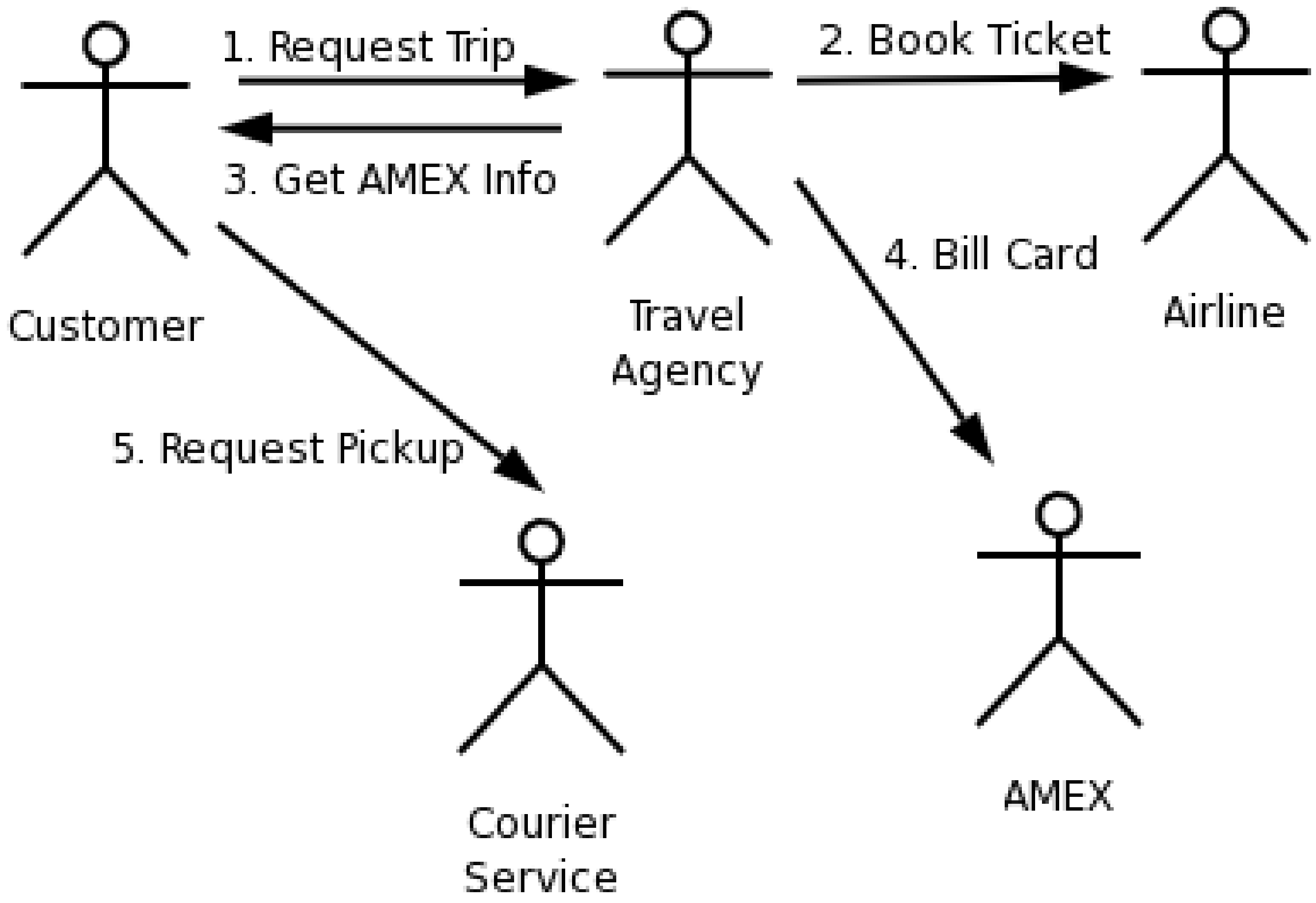
- Represent agented systems - Objects collaborate to achieve result

- Supporting real-world
  - Characteristics
  - Compositions
  - Hierarchies
  - Communication
  - Constraints

Agented Systems

- Object oriented programs are sometimes classified as “agented systems”
- An agented system is comprised of
  - Collections of autonomous and self-contained processing agents - Objects
  - Collaborations between agents that accomplish the system's objectives - Associations and compositions
  - Coordinated actions through exchanging messages - Method invocations

Example of Agented Systems

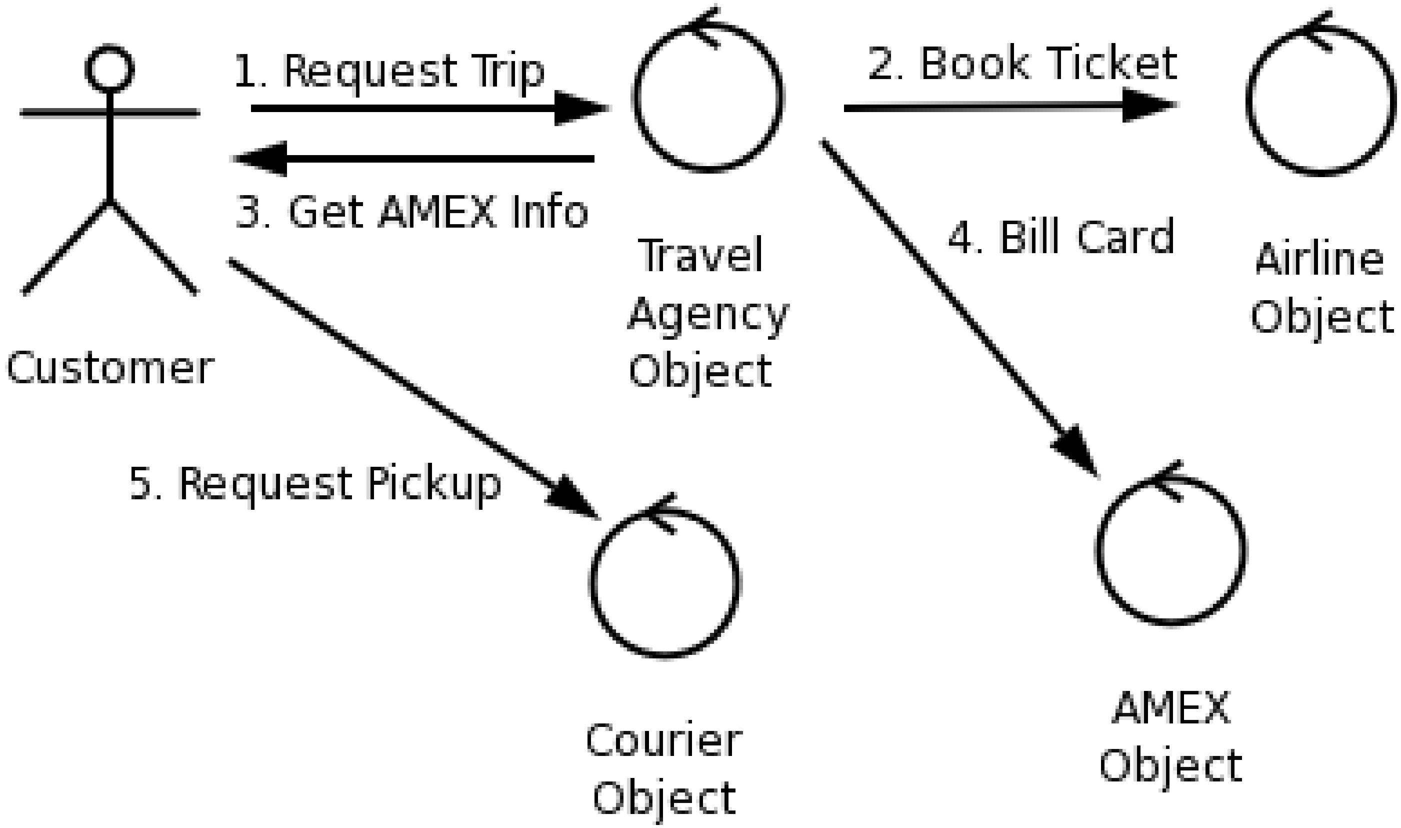


Agented Systems => OOP

- The challenge in OOP is to write programs that automates agented activity

- OOP does this by creating software *objects*, each object
  - Represents an agent
  - Possibly runs on its own processor
  - Possibly operates autonomously from the other objects
  - Possibly operates concurrently with the other objects

Agented Systems => OOP



Lab : Identifying Agents

- Look back at your Object chart

- Are there other things that could be added
  - Think in terms of Agented systems
  - Autonomous “objects”

Key Principles of OO Design

- Recursive design
- Alan Kay’s 5 Principles

Principle of Recursive Design

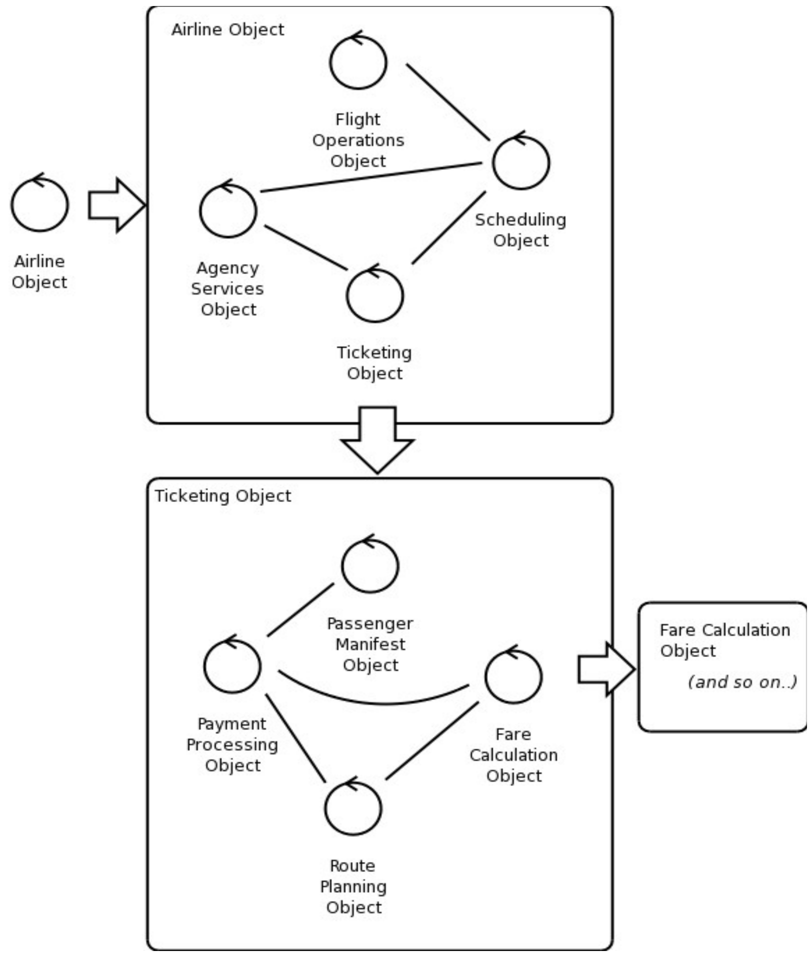
- *The structure of the part **mirrors** the structure of the whole*
  - Dividing a system (or computer) up into a collection of small processing engines, called objects
  - Each object will have similar computational power as the whole
  - Processing happens when the objects work together

- However, the recursive part is
  - Each object in turn is a collection of sub-objects
  - Each with similar computational power to the containing object
  - And so on . . .

Recursive Design

- Recursive design is at work in the ticket purchase system
- The system is made up of a collection of objects, one of which is an "Airline" object
- An Airline object is made up of a collection of objects
  - Each delivers part of the overall functionality of the Airline object
  - Ticketing object is responsible for doing ticketing
  - Ticketing object is made up of objects, each of which specializes in some aspect of the Ticketing object's functionality

Recursive Design



Kay’s 5 Principles

- Everything can be represented as an object
  - Objects are what we talk or think about
  - When we want to talk about something, we make it into an object
  - For example AMEX views the billing of the AMEX card for the flight to Paris as a credit card transaction - an object
- Systems are collections of objects collaborating for a purpose
- An object can have an internal structure composed of hierarchies of sub-objects

Kay's 5 Principles [cont.]

- An object presents an interface that
  - Specifies which requests can be made of it and what the results of those requests are
  - Is often referred to as the object's "contract"
- An object is of one or more types
  - Each type defines an interface
  - Each type may be derived from a hierarchy of types

Lab : Applying Recursive Design

- Apply the principle of recursive design to your system
  - Can you break apart objects
  - Into smaller objects

  - Do you need to combine objects
  - Into bigger parts

Going from OOD to OOP

- Goal of OOD is an Object Model

- A good Object Model will have:
  - Encapsulation
  - Modularity
  - Abstraction
  - Relationships

- OOP is process of coding Object Model

Designing Sound Objects

Keep related things together

- Objects should be “cohesive”
- Keep things that work together, together
- Keep things that change together, together
- Design the object to be a good model, not based on how it will be used

Designing Sound Objects [cont.]

Keep unrelated things separate

- Model one thing only, but model it well
- Separate things that change independently
- Reduce chance of change to one aspect breaking something else
- Usually using packages, rather than classes

OOD Concepts : Encapsulation

- Information hiding
  - Data format and meaning is purely internal
  - Cannot, and need not, access directly
  - Cannot misuse
- Ensures:
  - Safe data manipulation
  - Consistent data access
  - Behaviour is accessed only via interface
- Makes Abstraction viable

Types of Encapsulation

- State and Behavior – hidden fields and functions
- Type – hidden types
- Components – types hiding other types
- Modules – hiding components

Lab: Encapsulation

- Review your current design
- What can you hide to simplify the design?
  - Are there characteristics you should hide
  - Are there behaviors you should hide
- HINT: Most “states” should be hidden
  - Mark “public” things with a +
  - Mark “private” things with a -

OOD Concepts : Abstraction

- Generalized definition of some ‘thing’
  - General behaviours are known
  - But concrete implementations details aren’t

- Focuses on the “outside” view of an object
  - Think of a “black box”
  - Discussed in terms of an “interface

- Enables flexibility in solution – dependencies on “generalizations”

Types of Abstractions

- Entity – object represents a model in the problem domain
- Action – object that provides a generalized set of operations
- Virtual Machine – object that groups operations
- Coincidental – object that groups unrelated things together

OOD Concepts : Relationships

- Two types of relationships:
  - Hierarchy - Ranking or ordering of Abstractions
  - Composition – Grouping Abstractions as a Unit

- Both are needed for proper abstraction, encapsulation, and modularity

Hierarchy Relationships

- Described in terms of “Is A” relationship
- Rely on Logical ordering:
  - Top of hierarchy – most general
  - Bottom of hierarchy – most specific
- Referred to as:
  - Generalization or Parent
  - Specialization or Child

Classifying Hierarchies

- Defining Classifying hierarchies is hard

- Common ways to classify hierarchies
  - Natural ordering of things (plants, animals, etc)
  - Domain ordering of things (Manager, Employee, etc)
  - Behavioral ordering of things

Abstractions and Hierarchies

- Abstraction – generalized description
- Hierarchies – abstractions designed to create reusability & flexibility through inheritance

- Types of abstraction in hierarchies:
  - Concrete classes – parents & children
  - Abstract functions – one more functions in parent class are undefined
  - Abstract classes – all functions in parent class are undefined

Lab : Identify Abstractions

- Review your current design
- Are there any entities that could be abstracted into generalities?
  - Would this make your system more flexible
  - Or more brittle
- HINT: Look for “like” types or “like” behaviors
  - Mark abstractions with <<abstraction>> next to type

Composition Relationships

- Described in terms of “Has A” relationships
- Combine “simple” abstractions together to create “complex” abstractions
- Referred to in terms of:
  - General composition – implies full ownership
  - Aggregation – implies “borrowed” ownership
  - Containment – list, arrays, etc.

Object Model Relationships

- A Good Object Model will have both types of relationships
  - Think in terms of “IS A” (hierarchy)
  - And in terms of “HAS A” (composition)
- “IS A” relationships can have “HAS A” relationships
- “HAS A” relationships can have “IS A” relationships

Lab : Organizing Object Model

- Look at the system you’ve design so far

- Perform an *Iterative Decomposition*, checking to see if newly formed . . .
  - Abstractions
  - Encapsulation
  - Modularity
  - Relationships

  . . . still make sense

OOD Concepts : Modularity

- Modularity consists of dividing program into modules
  - Packages “abstractions” into discrete units
  - Relies on encapsulation

- Modularity is typically measured in terms of
  - How ***Cohesiveness*** the system is
  - How ***Coupled*** the system is

Cohesion

- GOAL: High-cohesion
  - All related functions and characteristics found in the class – good cohesion
  - Class contains functions and characteristics for multiple unrelated different types – poor cohesion

- Types of cohesion:
  - Functional (better) –all functions relate to one object
  - Procedural – grouped because functions operate in “sequence”
  - Coincidental – grouped aribtrarily

Coupling

- GOAL: Low Coupling
  - Objects have few dependencies on other objects – low coupling
  - Objects have many dependencies on other objects – high coupling

- Types of coupling:
  - Message (least) – all interactions done through “messaging” mechanism
  - Data – objects share data (globals?)
  - Content (most) – object relies on internal workings of another object

Lab: Modularity

- Review your current design
- Can you make your design less coupled with better cohesion?
- HINT:
  - Can your design interact with abstractions instead of implementations?
  - Count the number of external dependencies on each class; the higher the number the more dependent

OOP with PHP

Moving from OOD to OOP

- Is the design complete enough to code?
  - Primary and secondary entities designed
  - States and behaviors designed
  - Object interactions designed
- Then it’s time to code
  - Release early and often
  - Test earlier and more often
  - Refactor along the way

Quick Review of OO Concepts

OOD Concepts

- Abstraction
- Encapsulation
- Modularity
- Relationships

Quick Review of OOP Concepts

OOP Concepts

- Classes
- Objects
  - States
  - Behaviors
- Instantiation
- Message Passing
- Inheritance
- Polymorphism

PHP Classes

Implemented in Customer.php

```php
<?php
class Customer {
  public $firstName;
  public $lastName;
  public $address;

  function __construct($f, $l, $a) {
    $this->firstName = $f;
    $this->lastName = $l;
    $this->address = $a;
  }

  function getName() {
    return $this->firstName." ".$this->lastName;
  }

  function getAddress() {
    return $address;
  }

}
?>
```

Encapsulation

- Created through access modifiers
- Access modifiers – public, protected, private

```php
<?php
class Customer {
  private $firstName;
  private $lastName;
  private $address;

  function __construct($f, $l, $a) {
    $this->firstName = $f;
    $this->lastName = $l;
    $this->address = $a;
  }

  function getName() {
    return $this->firstName." ".$this->lastName;
  }

  function getAddress() {
    return $address;
  }

}
?>
```

Constructors

```php
<?php
class Customer {
  private $firstName;
  private $lastName;
  private $address;

  function __construct($f, $l, $a) {
    $this->firstName = $f;
    $this->lastName = $l;
    $this->address = $a;
  }

  function getName() {
    return $this->firstName." ".$this->lastName;
  }

  function getAddress() {
    return $address;
  }

}
?>
```

Instantiation & Message Passing

```php
<? include("Customer.php") ?>
<? include("Address.php") ?>
<?php
  $address = new Address();
  $address->city = "Louisville";
  $address->state = "Colorado";
  $address->streetAddress = "844 Main St.";
  $address->zipCode = "80027";
  $customer = new Customer("Bob","Jones", $adress);
  $mapUrl = "http://maps.google.com/maps?q=";
  $mapUrl.=$address->getFormattedAddress();
  $mapUrl.=";z=14&amp;iwloc=A&amp;output=embed";
?>
```

Abstraction

```php
<?php
class SeniorCustomer extends Customer {
	private $dob;

	function __construct($f, $l, $d, $a) {
		parent::__construct($f, $l, $a);
		$this->dob = $d;
	}

	function getDateOfBirth() {
		return $this->dob;
	}
}
?>
```

Abstract Classes

```php
<?php
abstract class Customer {
    protected $firstName;
    protected $lastName;
    protected $address;

    function __construct($f, $l, $a) {...}

    function getName() {...}

    function getAddress() {...}

    abstract function getCompleteDetails();

}
?>
```

Abstract Class Implementation

```php
<?php
class SeniorCustomer extends Customer {
    private $dob;

    function  __construct($f, $l, $d, $a) {
        parent::__construct($f, $l, $a);
        $this->dob = $d;
    }

    function getDateOfBirth() {
        return $this->dob;
    }

    function getCompleteDetails() {
        return $this->getName().",".$this->getDateOfBirth();
    }
}
?>
```

Lab : Pseudo Code

- Translated your OOD into OOP
- Use Pseudo code

Unified Modeling Language

Quick Overview of UML

History of UML

- “Specification language” created to support Unified Process
  - Released as separate entity in 1997
  - Managed by Object Management Group (OMG); UML 2.3 is current specification

- Intended to be usable by both humans and machines
  - Provides standardized notation used to depict an abstract model of a software systems
  - UML relies on concepts and terminologies for proper modeling

UML Concepts

- UML diagrams are classified in 3 primary categories
  - Structural diagrams
  - Behavioural diagrams
  - Relational diagrams

- Before can work with diagrams need to understand concepts found in each category

Structural Concepts

- ***Actor*** – someone who interacts with system
- ***Attribute*** - property in an object [*characteristic*]
- ***Class*** - template for defining objects
- ***Component*** - modules within software solution
- ***Interface*** - boundary definition
- ***Object*** - instance of class
- ***Package*** - collection of related classes [*namespace*]

Behavioural Concepts

- **Activity** - major task that must take place to fulfill an operation [*group of functions*]
- **Event** - notable occurrence at a particular point in time
- **Message** - communication means to pass control between objects [*method parameters*]
- **Method** - piece of code in a class, a [*function*]
- **Operation** - combining two things to get a third
- **State** - unique configuration of information
- **Use case** – way to capture functional requirements

Relational Concepts

- **Association** - relationship of one class or object to
- **Composition** - combining simple objects to create complex objects
- **Aggregation** - special form of composition allowing object to live beyond composition
- **Generalization / Specialization** - inheritance characteristics [parent/child]
- **Depends / Coupling** - degree to which there are dependencies across entities

Other UML Concepts

- **Stereotype** - qualifier
- **Multiplicity** - similar to cardinality in DB modeling
- **Role** - typically associated with actor

UML Diagrams

14 Diagrams broken into three categories

- **Structural** - focus on things that make up the system
- **Behavioral** - focus the things that happen in the system
- **Interaction** - focus on flow control to support the things that happen

Structural Diagrams

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

Behavior Diagrams

- Activity Diagram
- State Machine Diagram
- Use Case

Interaction Diagrams

- Communication Diagram
- Interaction Diagram
- Sequence Diagram
- Timing Diagram

Melding UML into Your Process

- UML works really well with RUP / USP
- But, can also work with other processes
- Traditional (Waterfall, Iterative, etc.)
  - Analysis Phase: Use Cases, Class diagrams, Activity Diagrams, State diagrams
  - Design Phase: Class diagrams, Sequence Diagrams, Package Diagrams, State Diagrams, Deployment Diagrams

Melding UML into Your Process [cont.]

- Agile is less structured
  - Not big on ceremony
  - Release early and often

- Common diagrams used in Agile
  - Analysis: User Stories, Class Diagrams
  - Design: Class Diagrams, Sequence Diagrams, Package Diagrams

Melding UML into Your Process [cont.]

Decisions you need to make:

- Where will you use diagrams?
- Which diagrams will you use?
- Which diagrams will you use where?
- How detailed will your diagrams be?
- Will you allow “illegal” notations?
- How will you manage the diagram’s “lifecycle”

Creating UML Artifacts

Typically, you'll use a UML tool to

- Create UML artifacts
- Manage UML artifacts
- Possibly generate code

Common UML Tools

- IBM Rational – Rational Rose
- SparxSystems – Enterprise Architect
- GentleWare – Poseidon UML

Structural Diagrams

Purpose of Structural Diagrams

- Emphasize structure of the system

- Described in terms of:
  - Static view [design time]
    - Class design
    - Components
    - Packaging and Deployment

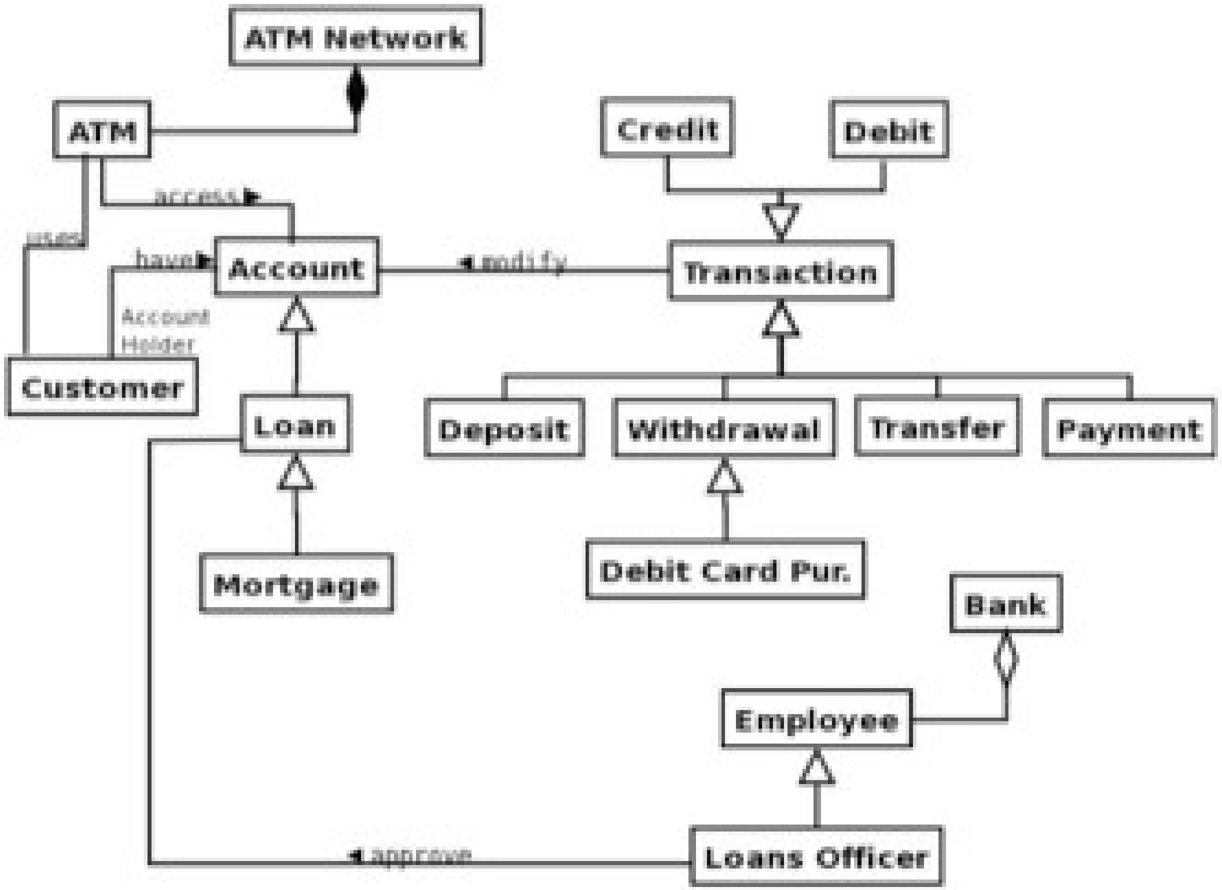
  - Dynamic view [run time]
    - Objects

- Used typically in Software Architecture

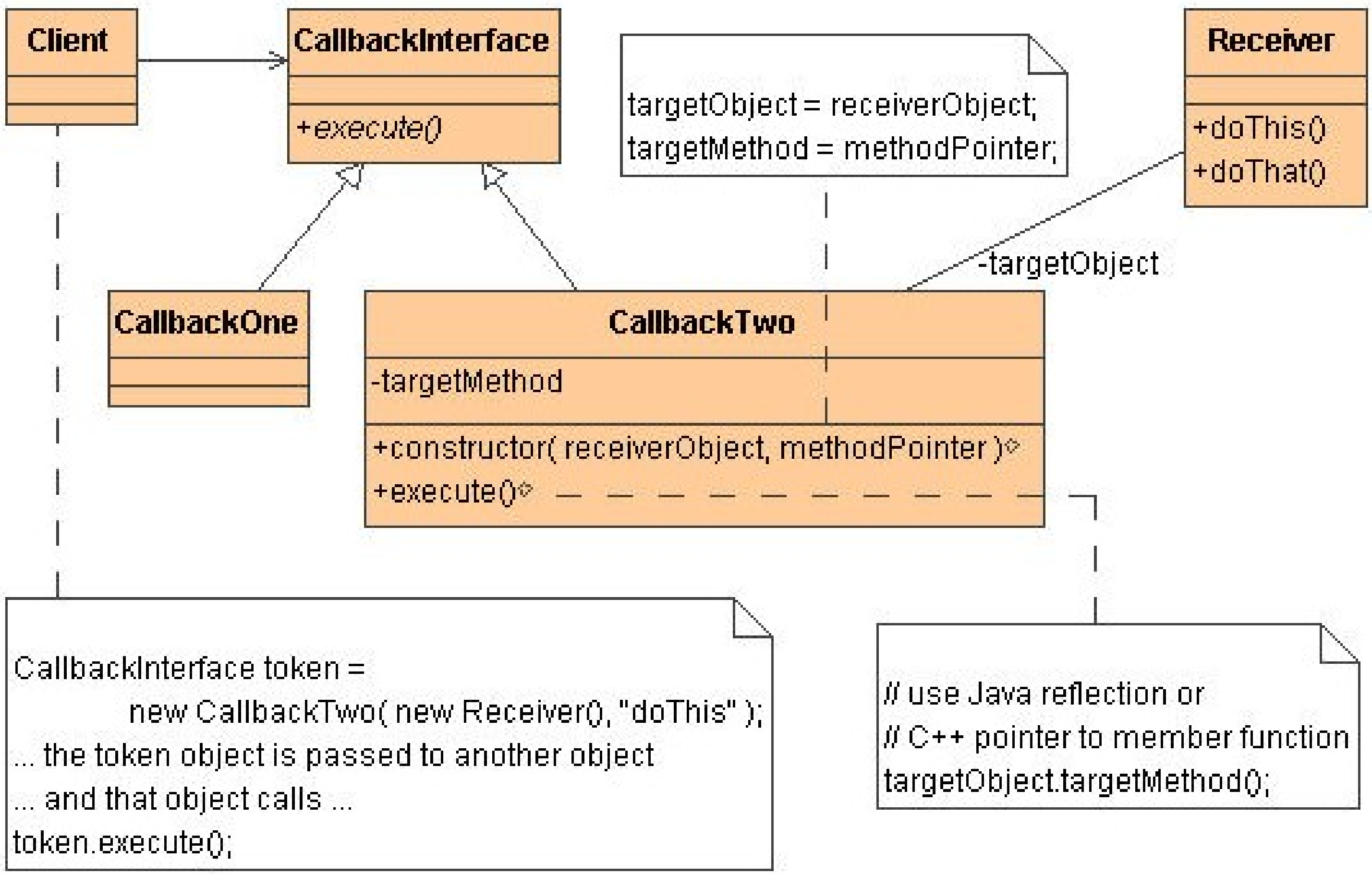
Class Diagram

- Static “type” oriented view of system
- Considered “building block” of object modeling
- Describes structure of system in terms of classes, attributes and relationships
- Many variations, from sketchy to comprehensive and complex

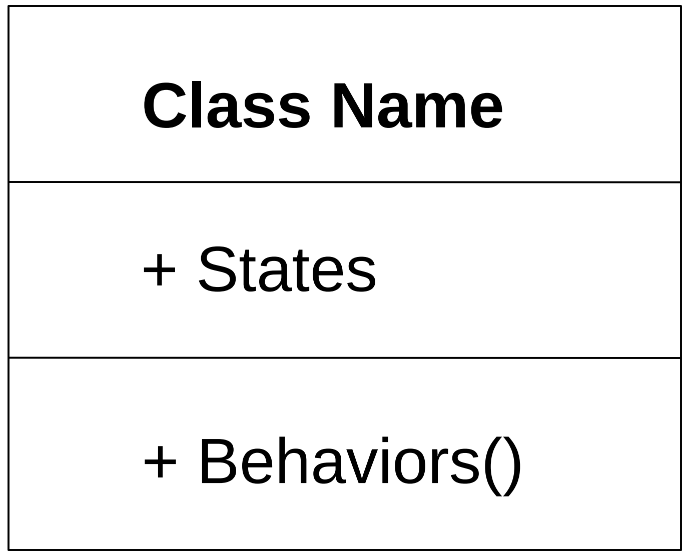
Sketchy Diagram



Comprehensive Class Diagram



General Class Diagram Form



Denoting States - Attributes

- Represented in second section of diagram
- Use notation:

  visibility name: type multiplicity = default {property-string}

- Where:
  - visibility is access modifier
  - name is variable name
  - type is datatype
  - multiplicity is number of internal instances
  - default is default value
  - {property-string} is additional information

Specifying Visibility

- UML is not-language specific

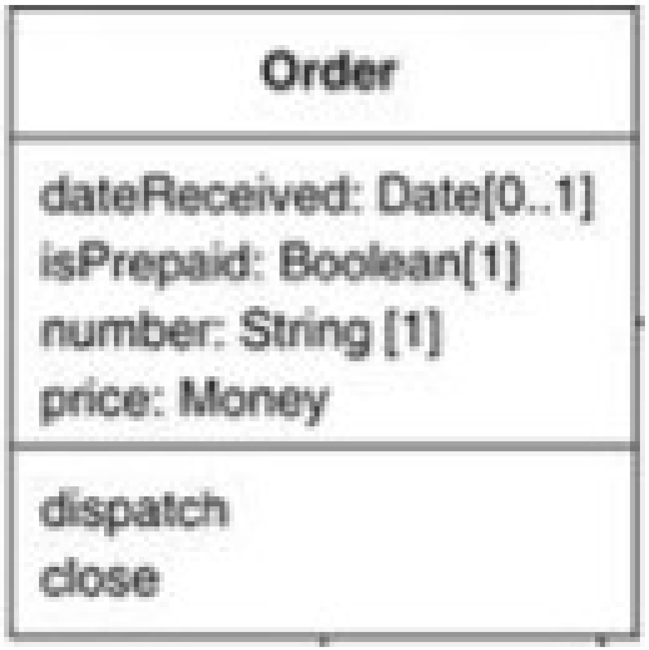
- Defines visibility in terms of:
  - **+** - stands for *public*
  - **~** - stands for *package*
  - **#** - stands for *protected*
  - **-** - stands for *private*

- You will need to agree on:
  - When to use
  - What ~ means

Specifying Multiplicity

- Multiplicity defines number of
- Borrowed from ERD (DB Modeling)
- Three primary notations:
  - ***n* –** exact count of
  - ***n..y* –** minimum through maximum
  - * - zero or more, with no top limit

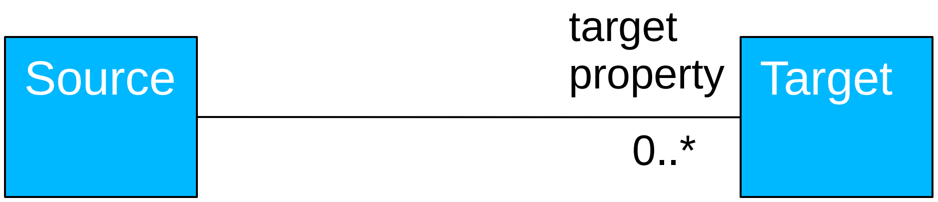
Class Diagram – Attribute Example



Denoting States - Associations

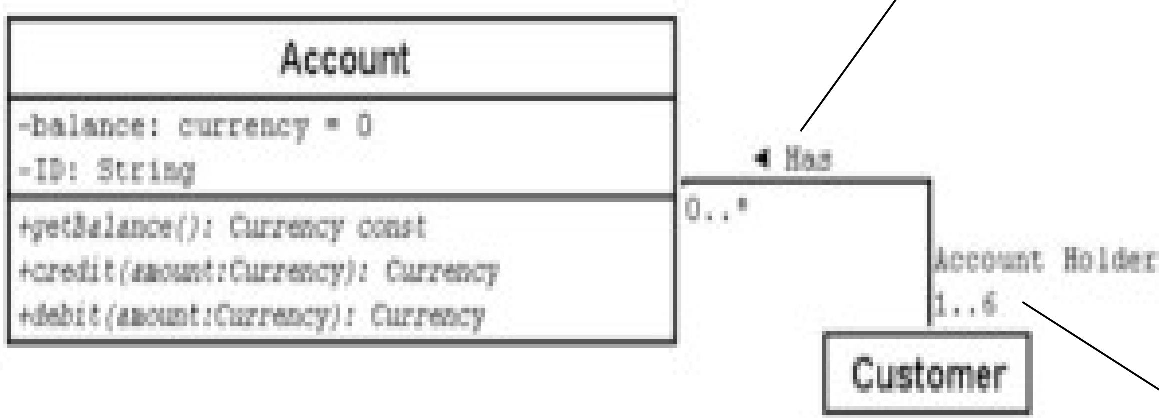
- Represented using Links (association lines)

- Use notation:



- Where:
  - association line runs from source to target
  - target property is called out by the target
  - with multiplicity requirements under property name
  - alternately, a navigation link could be used

Class Diagram : Property Example



Verb phrase
Direction

Bi-directional
Multiplicity

Denoting Operations

- Represented in third section of diagram
- Use notation:

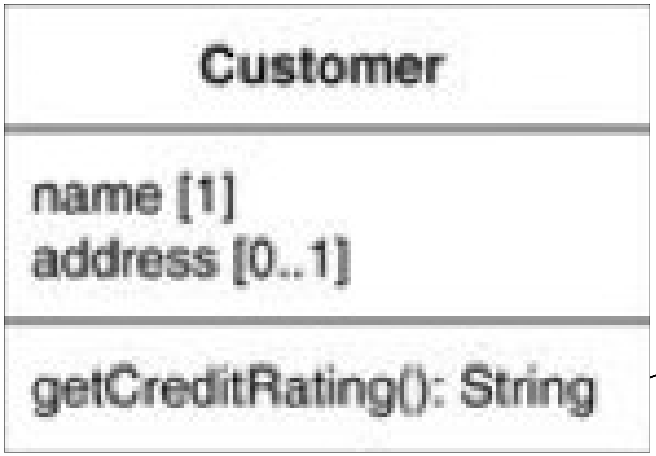
  visibility name (param list) : return-type {property-string}

- Where:
  - visibility is access modifier
  - name is operation name
  - (param list) is parameter list, specified using notation

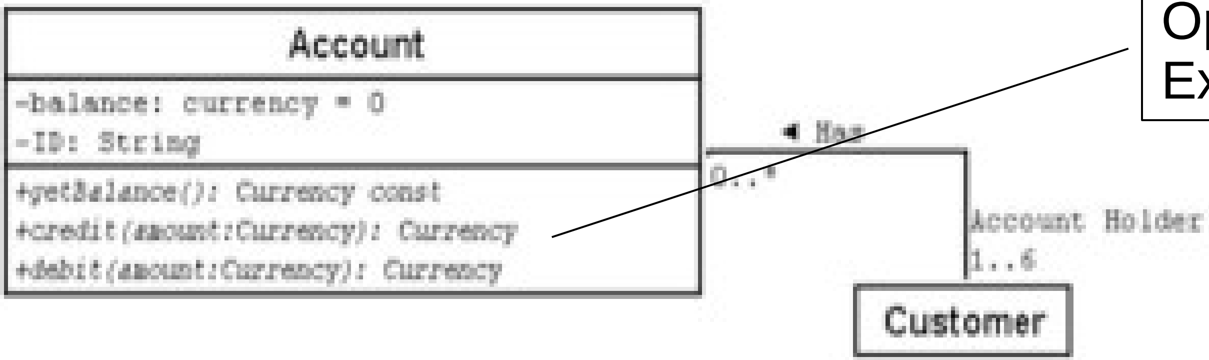
    direction name: type = default value

  - return-type is type of returned value
  - {property-string} is additional information

Class Diagram : Property Example



Operation Example

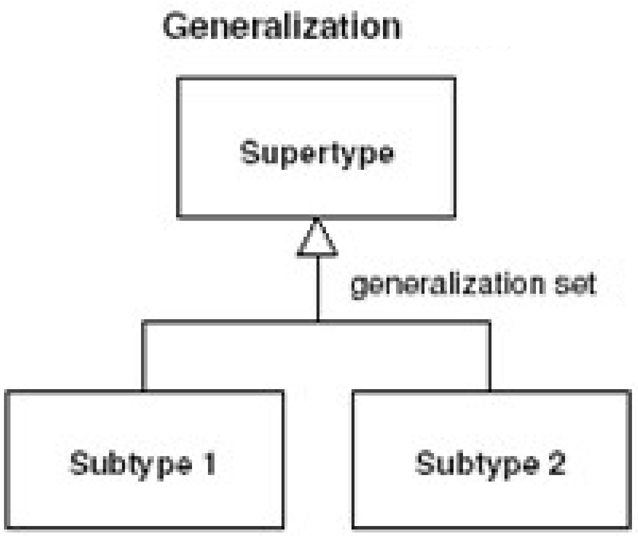


Operation Example

Specifying Hierarchies

Generalizations

- Describe Parent-Child relationships
- Open-arrow head points from child to parent
- Use notation for parents who are abstract or concrete classes



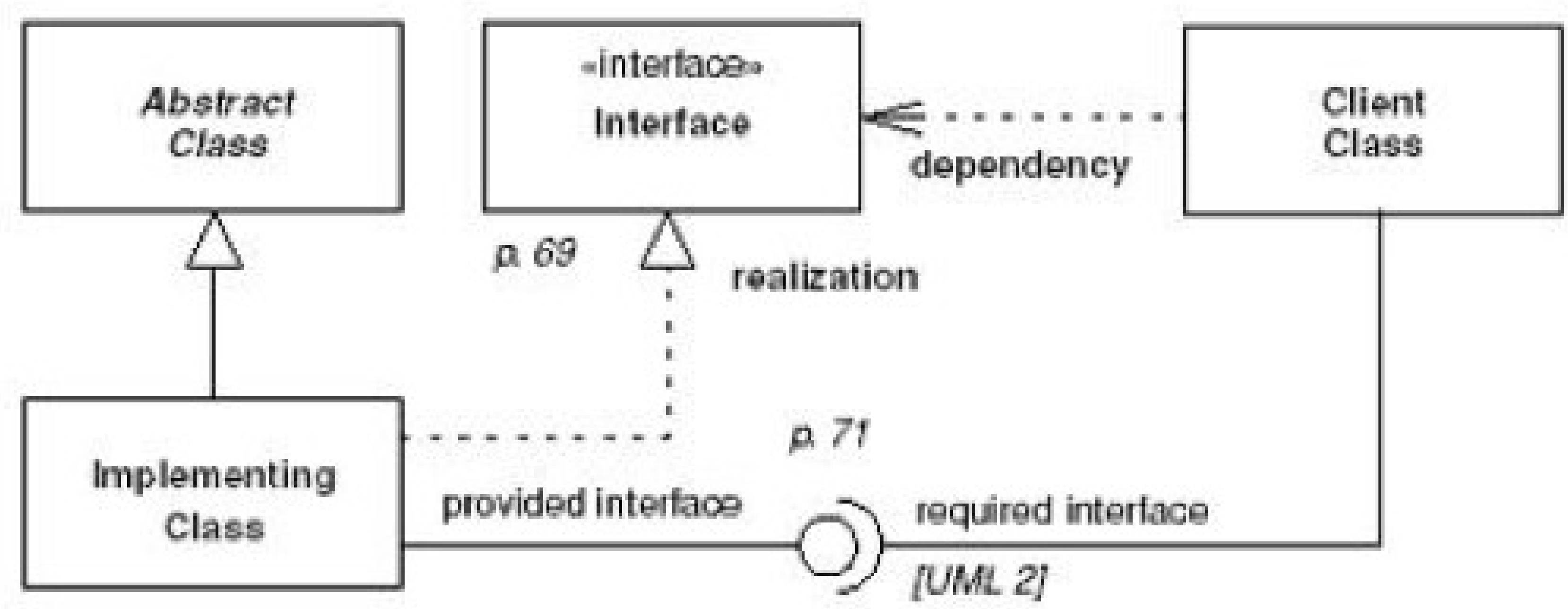
Specifying Realizations

- Describe Interface-Implementation relationships

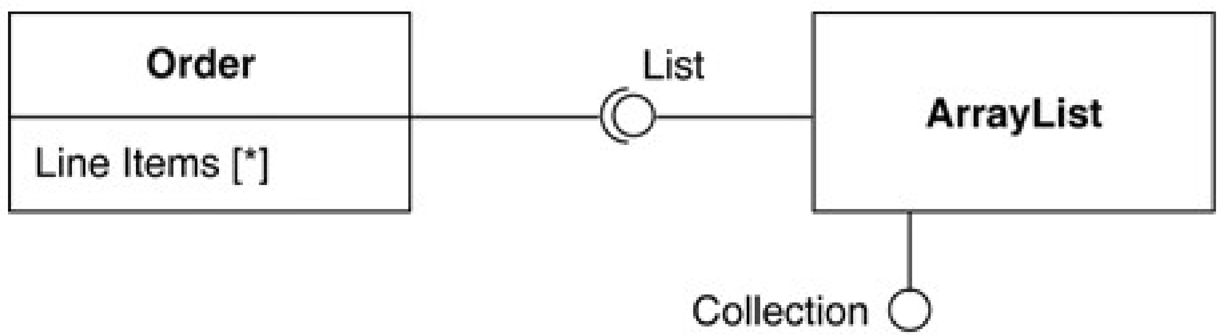
- Uses dotted-line notation
- Open-arrow head points from Implementor to Interface

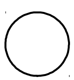
- Alternate notation use ball-socket notation

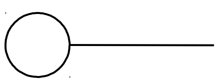
Realization Example

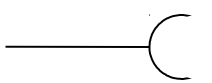


Realization Ball-and-Socket Notation

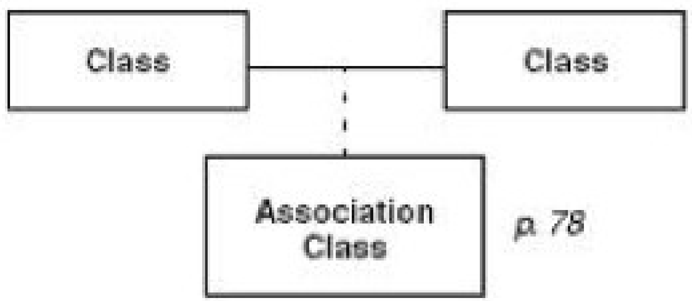


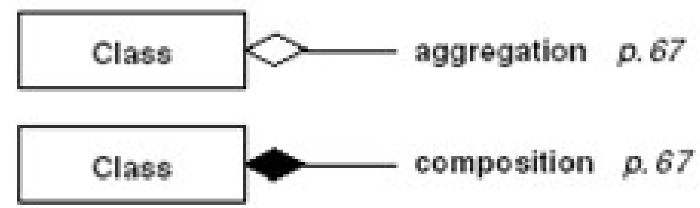
 Interface

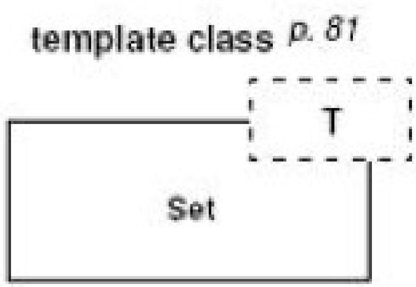
 Implementor

 User of Interface

Class Diagram Notations





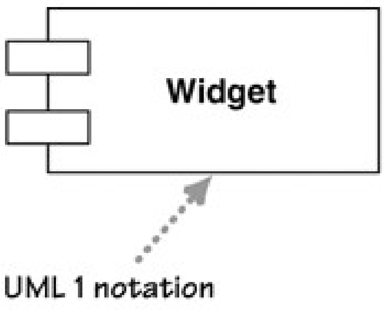


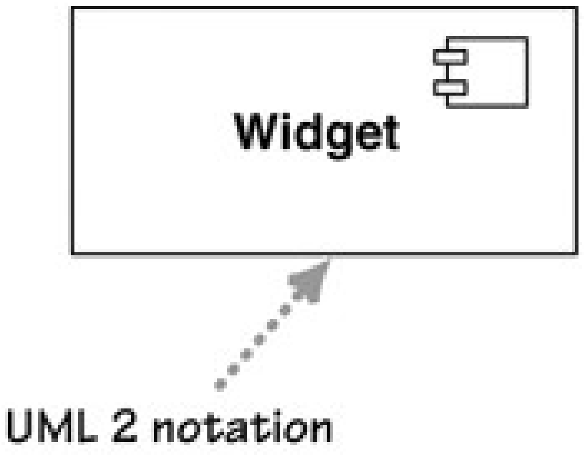
Component Diagram

- Static “type” oriented view of system
- Describes how structure of system is split up in terms of components; shows dependencies between components
- Relatively simple diagramming notation

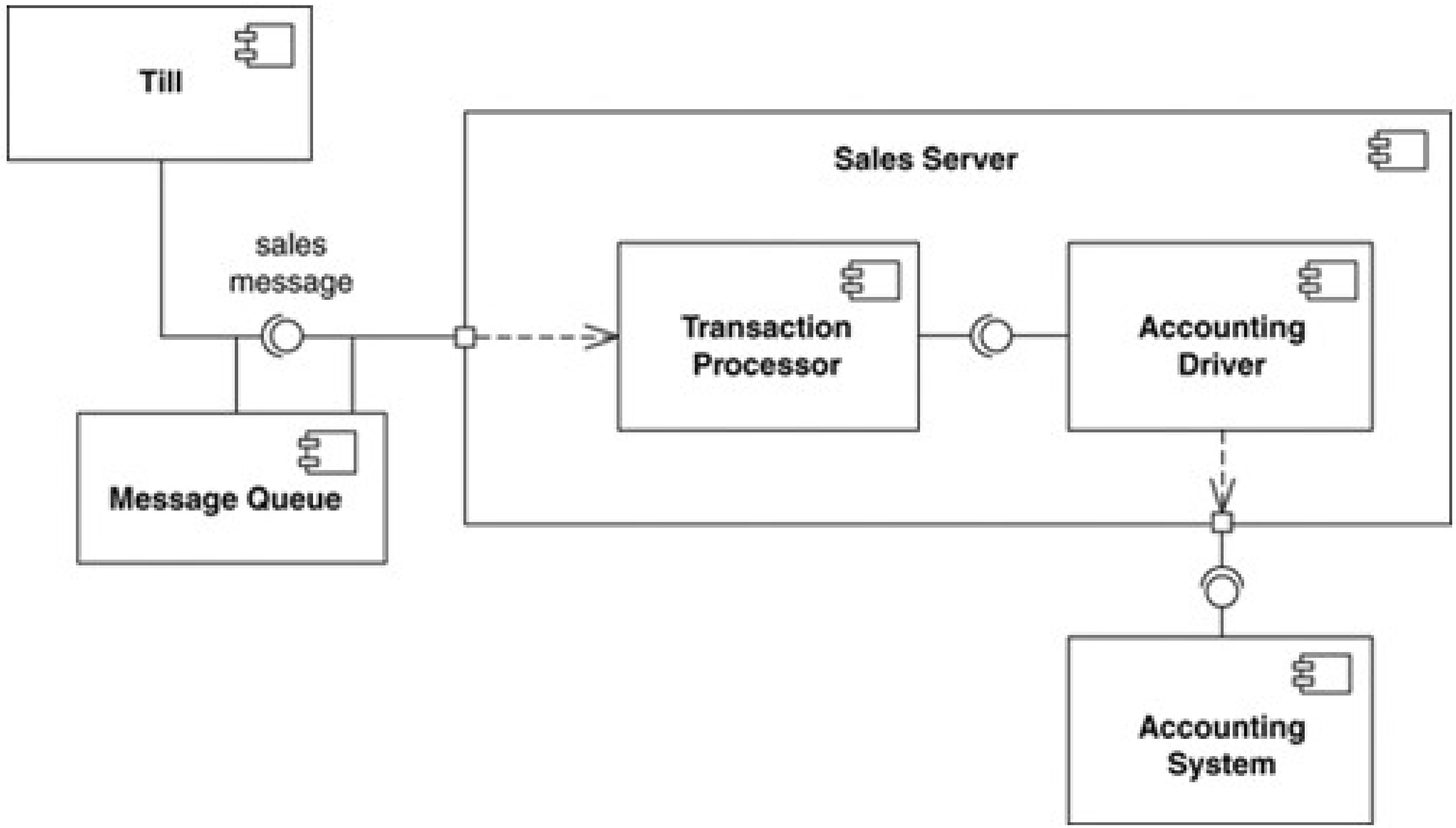
Component Notation

- High-level illustration
- Defined by:
  - Name
  - Realizations
  - Dependencies





Component Diagram

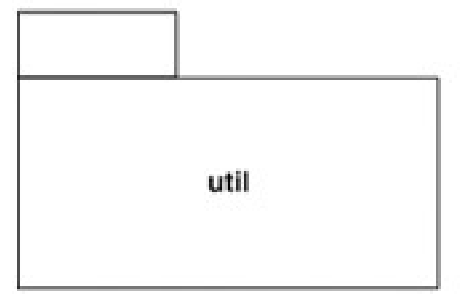


Package Diagrams

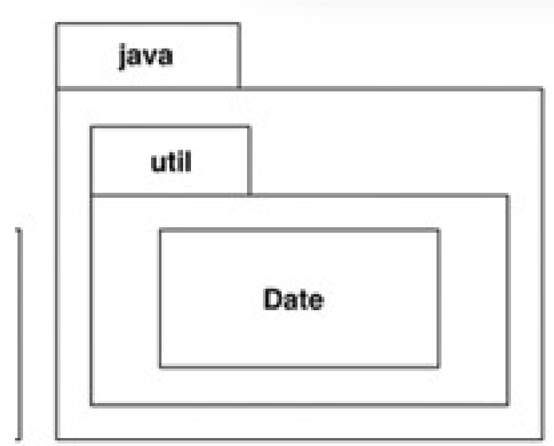
- Describes how system is split into logical groupings
- Shows dependencies across groupings
- Useful for robustness analysis and deployment analysis

Package Diagram Notation

- High-level illustration
- Defined by:
  - Name
  - Contents
  - Dependencies

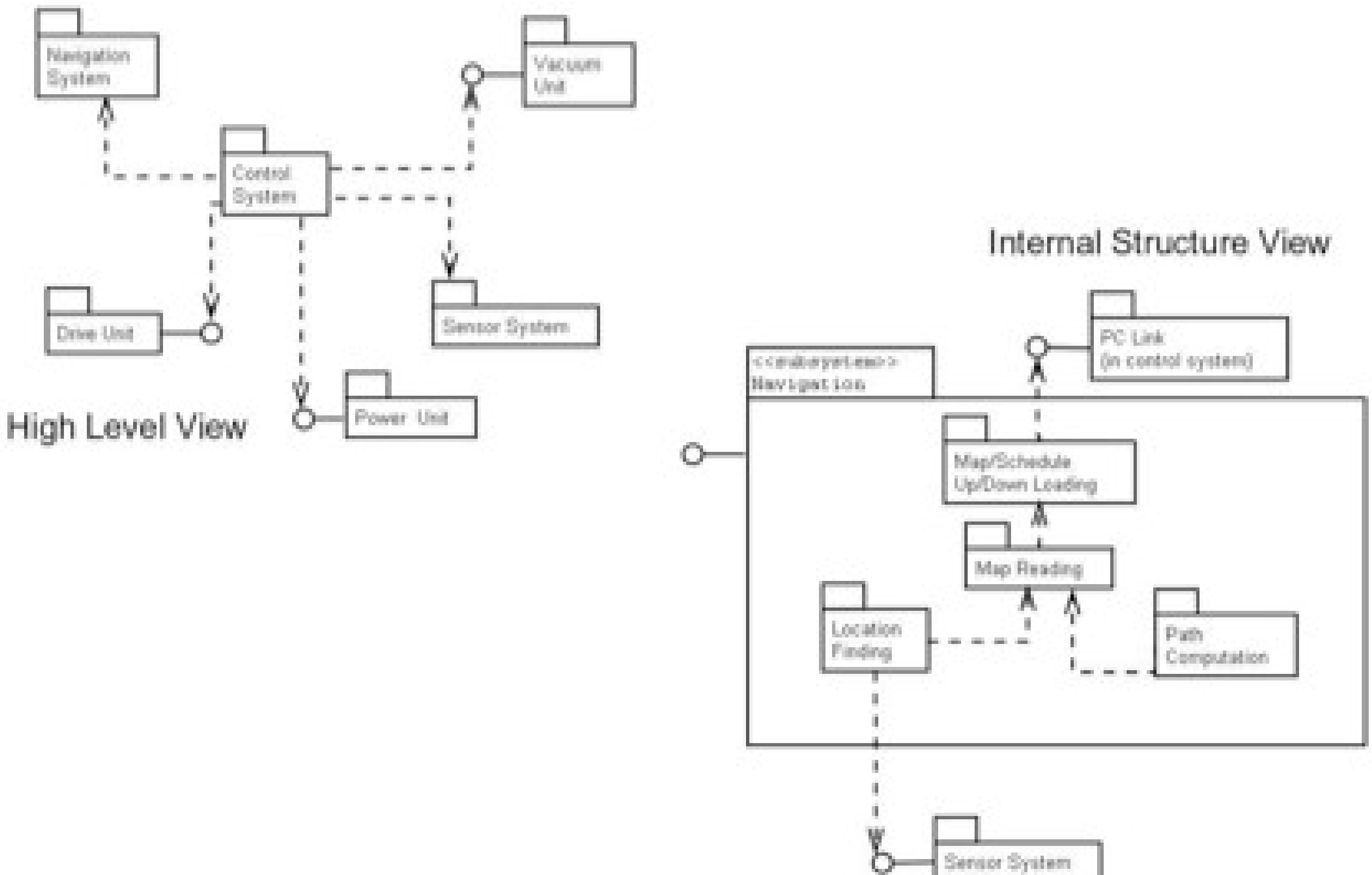


simple package



package + contents

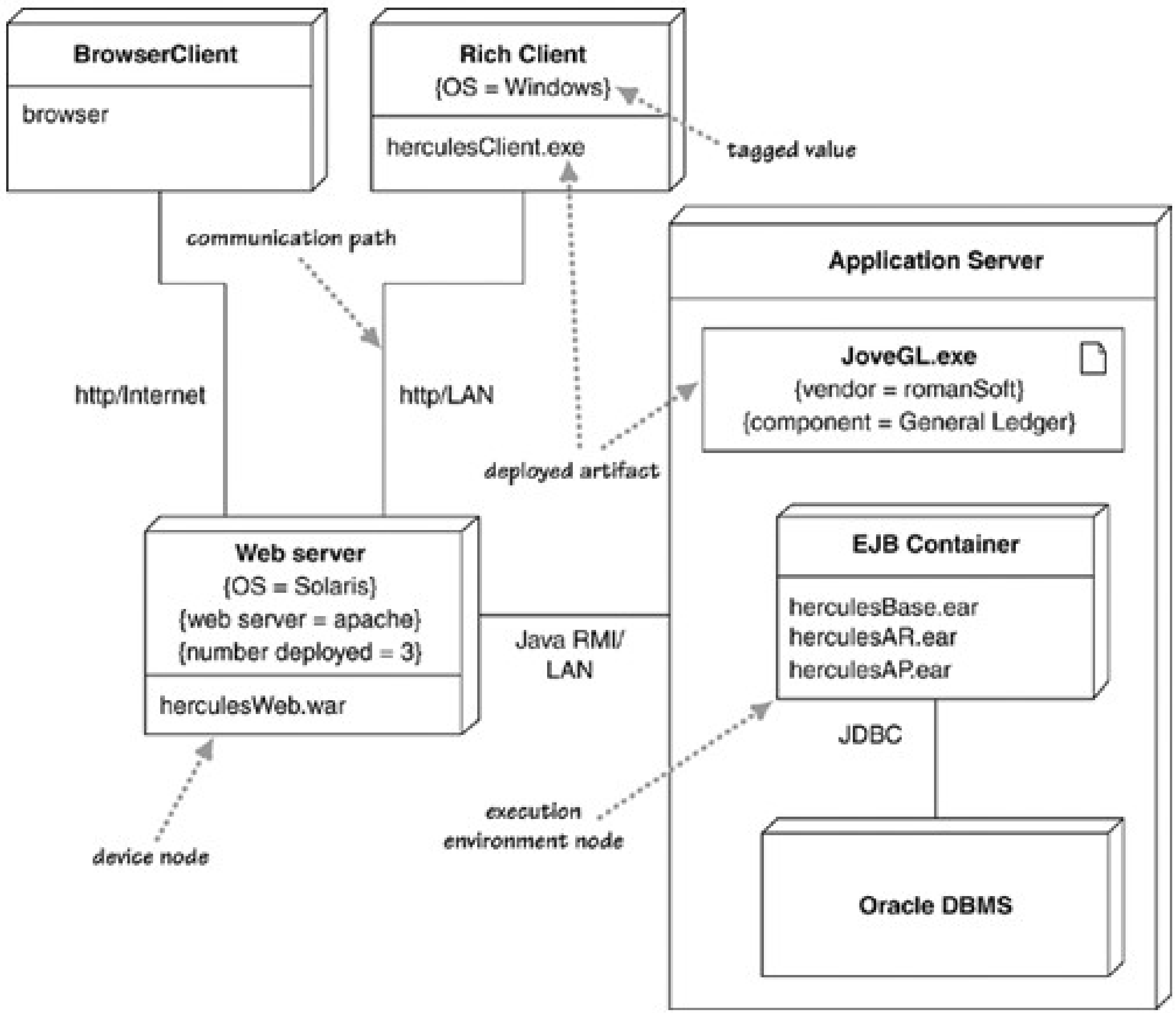
Package Diagram



Deployment Diagrams

- Used to describe nodes of system and deployment artifacts associated with nodes
- May incorporate package diagrams

Deployment Diagram



Object Diagram

- View of system at a specific time (run-time)
- Focuses on object instances, attributes, and links between instances
- Useful for showing examples of objects connected together
- Typically used to validate class diagrams

Object Diagram Notation

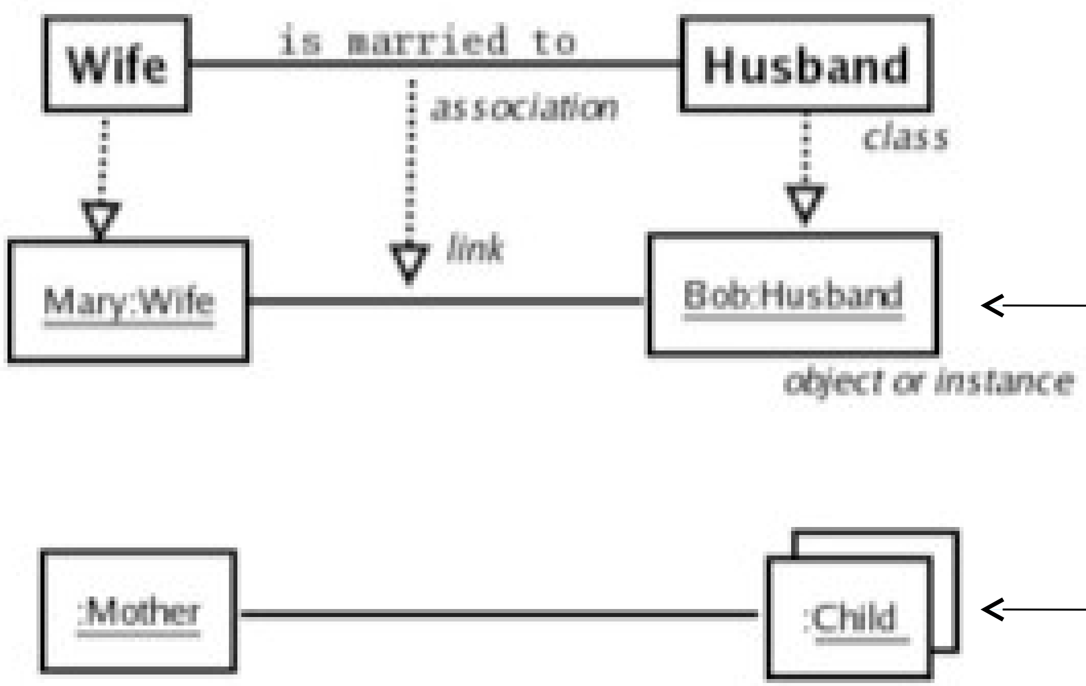
- Run-time illustration

- Defined by:
  - Object Instance
  - Type
  - Value

- Variations for:
  - Anonymous objects
  - Multiple instances

Object Diagram

Class Diagram



Object Diagram

Multiple Instances

Anonymous Instance

Behavioral Diagrams

Purpose of Behavioral Diagrams

- Emphasize what happens in system

- Focused on functionality of software
  - Actor stimulated functionality
  - Activity stimulated functionality
  - Object stimulated functionality

- Described in terms of:
  - Use Case Diagrams
  - Activity Diagrams
  - State Diagrams

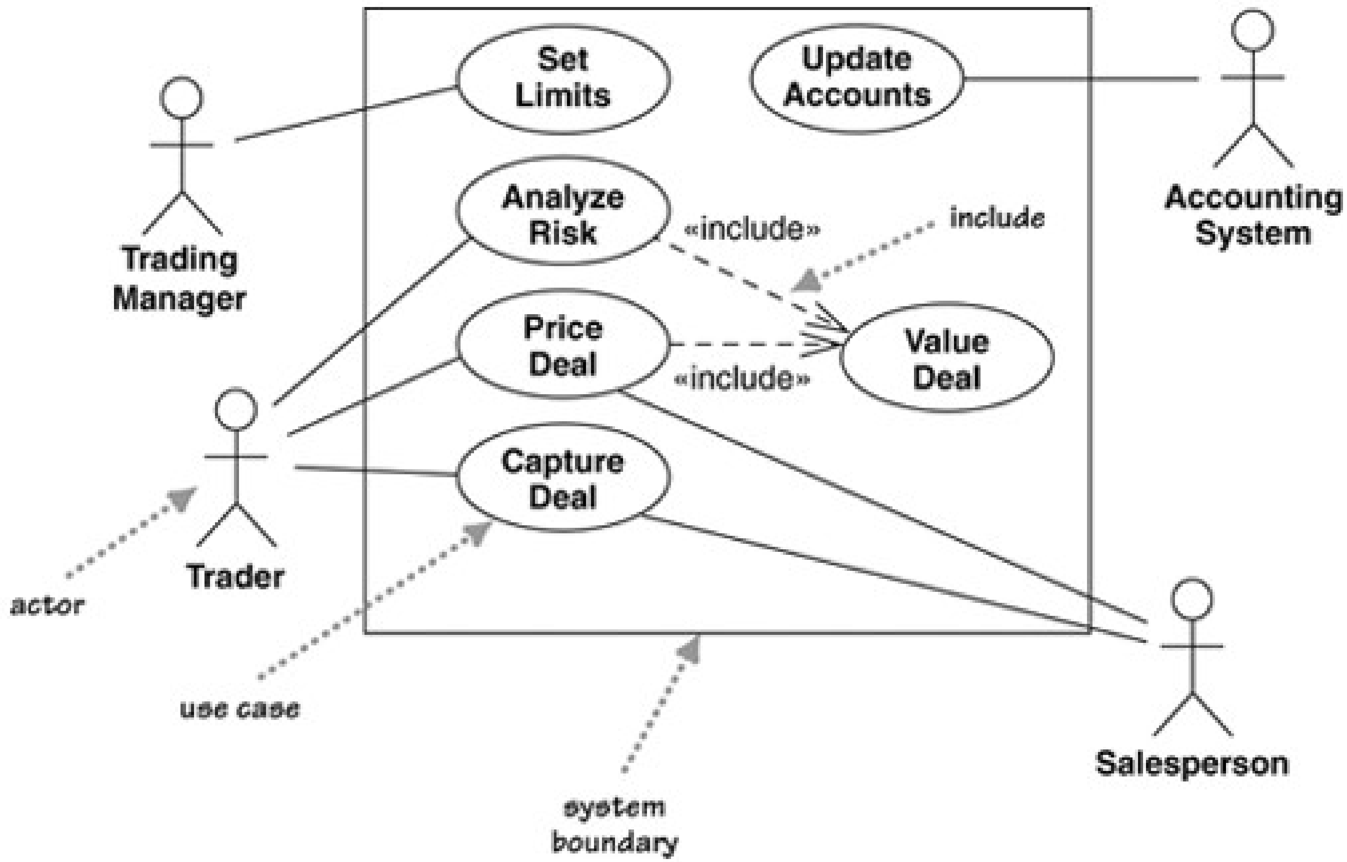
Use Case Diagram

- Represent stimuli on system that cause some operation to occur

- Typically requirement driven

- Provide user stories for system

Use Case Diagram



Use Case Description

Use Case	Withdraw Money
Use Case ID	ATM-001
Primary Actor:	ATM Customer
Goal:	To get cash from ATM
Scope	Customer Activity

1. Customer swipes card

2. Customer enters PIN at prompt

2a . If the customer is not from our bank, they select yes at the prompt that asks if they accept the extra service fee.

 2a-1 If they reply yes, we continue,

 2a-2 else the transaction is canceled.

3. Customer selects withdraw from presented options

4. Customer enters amount at prompt

5. Customer selects account from presented list

6. ATM dispenses cash and receipt

7. Customer takes cash and receipt and ATM resets

Activity Diagram

- Represent business and operational workflows
- Shows step-by-step operation across components
- Typically associated with a use-case

Activity Diagram

Start Point

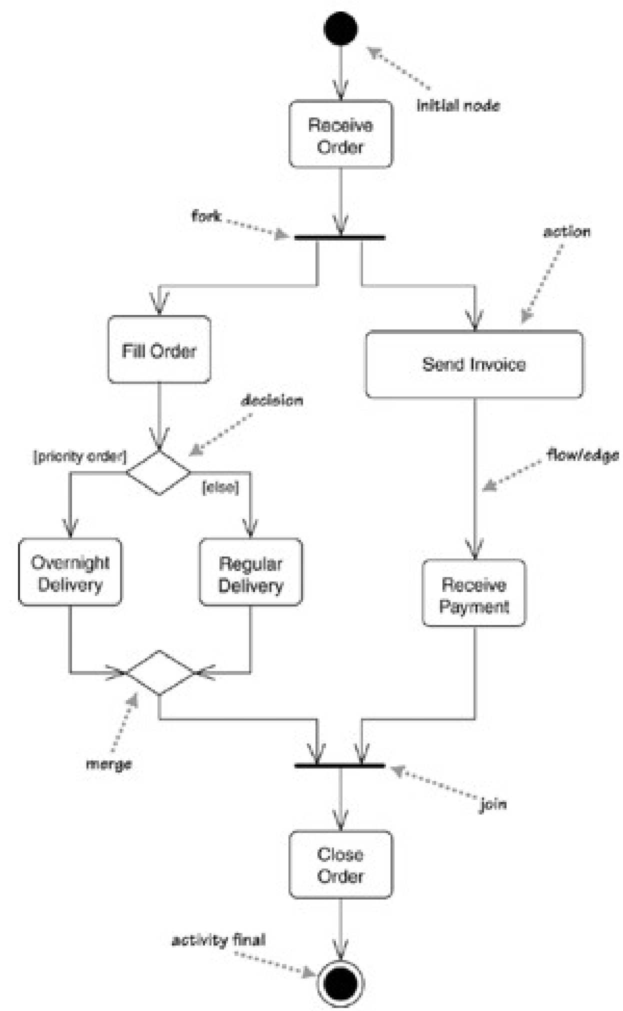
End Point

Activity

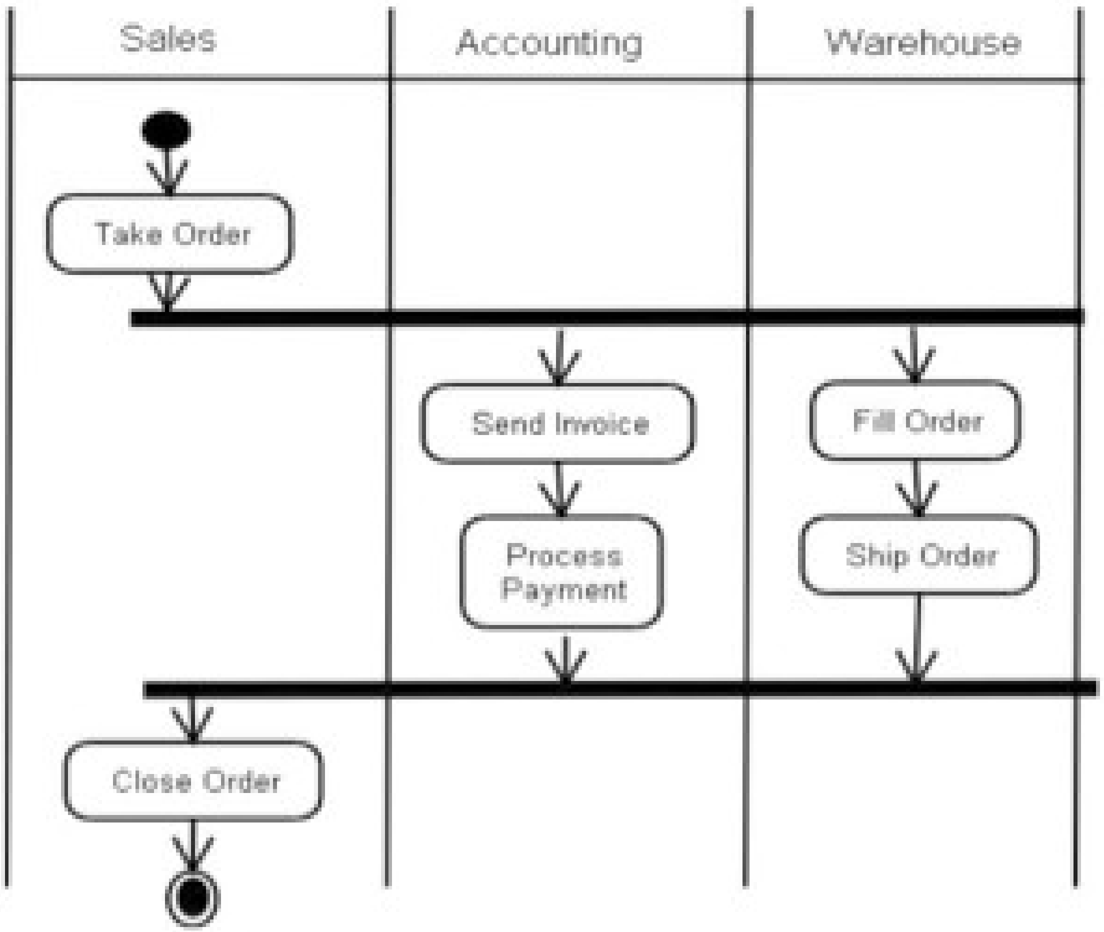
Fork / Join

Decision

Swimlane



Activity Diagram 2



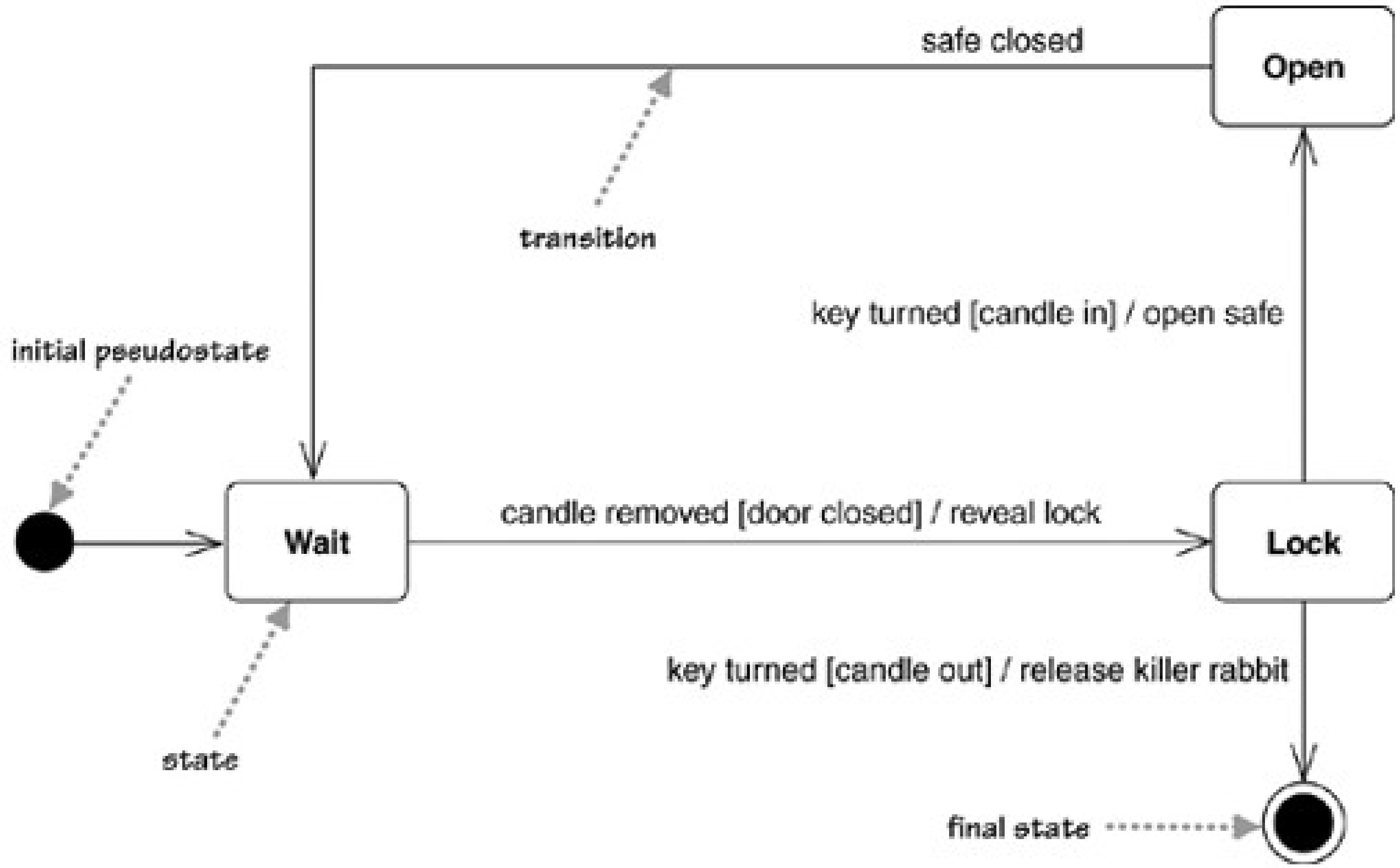
Statechart (State) Diagram

- Depict states of some aspect of the system

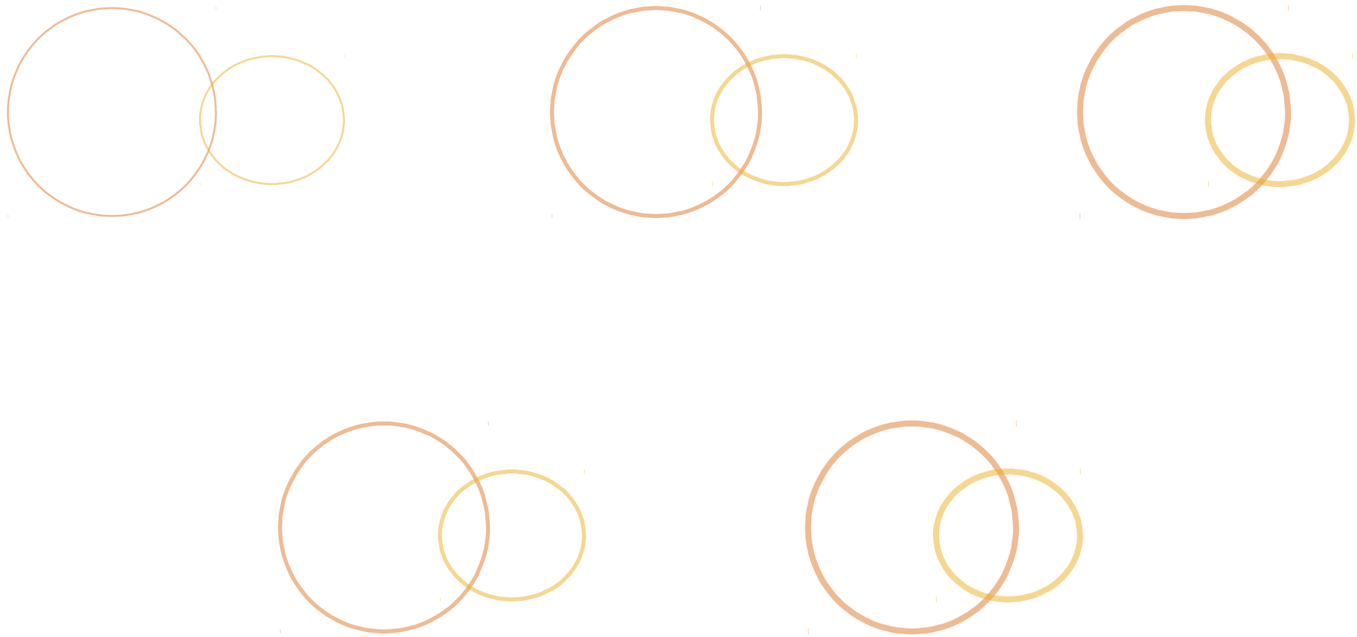
- Illustrates the states of an object
  - Represents the lifetime behavior of a single object
  - Shows triggers that cause transition from one state to another

- Described in terms of
  - State
  - Trigger - trigger-signature [guard]/activity
  - Navigation Arrows

State Diagram



Interaction Diagrams



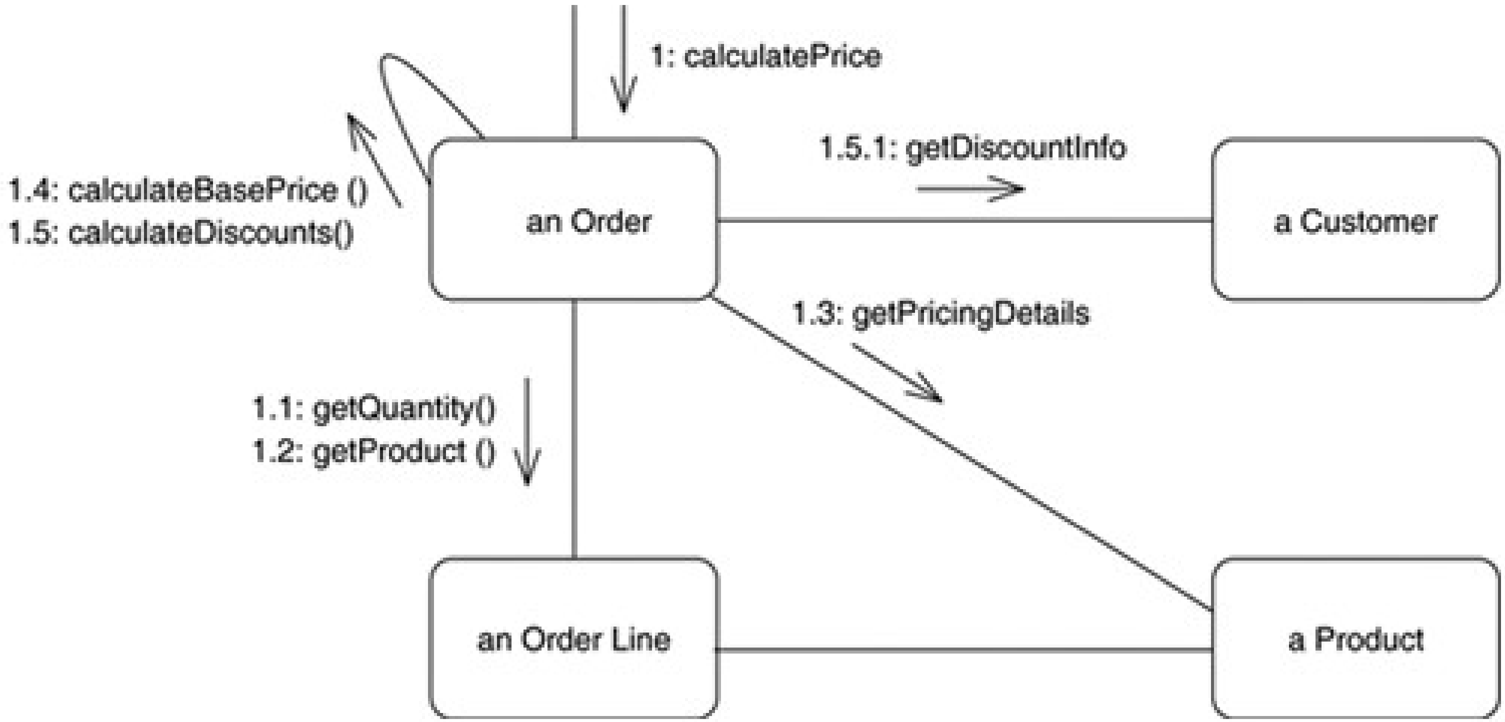
Purpose of Interaction Diagrams

- Emphasize how things flow through the system
- Focused on object interactions and data flow
- Described in terms of:
  - Communication Diagrams
  - Sequence Diagrams
  - Timing Diagrams

Communication Diagram

- Depicts interactions between objects in terms of sequence messages
- Are a combination of class, sequence, and use case diagrams

Communication Diagram



Sequence Diagram

- Object-based diagram
- Shows sequence of interacts across objects to achieve some task (a single scenario)
  - Address object creation and destruction
  - Inputs and outputs
  - Operation encapsulation
  - Exceptions
- 2nd Most commonly used UML diagram

Sequence Diagram Notation

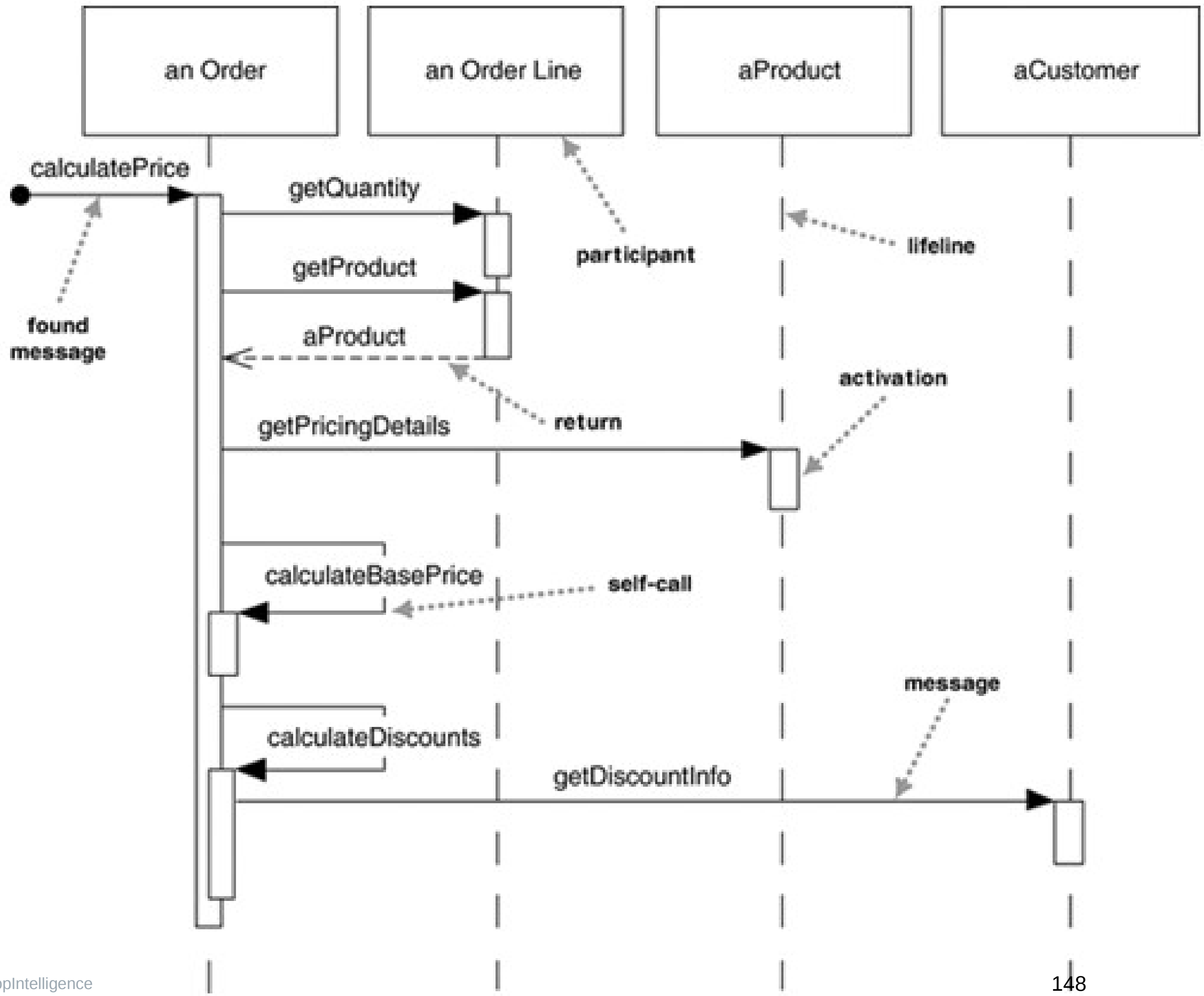
- Object Diagrams - Object notation
  - Illustrate objects participating in scenario
  - Objects diagram can be anonymous or specific
- Lifelife – Vertical dashed line
  - Illustrates the “lifespan” of the object
  - All objects are assumed to pre-exist and live beyond scenario
  - Can represent different lifespan if needed
- Activation Bars – Vertical rectangle
  - Identifies when object is actively participating
  - Activation bars may “overlap”
- Navigation arrows – illustrate function calls and returns

Sequence Diagram Notation

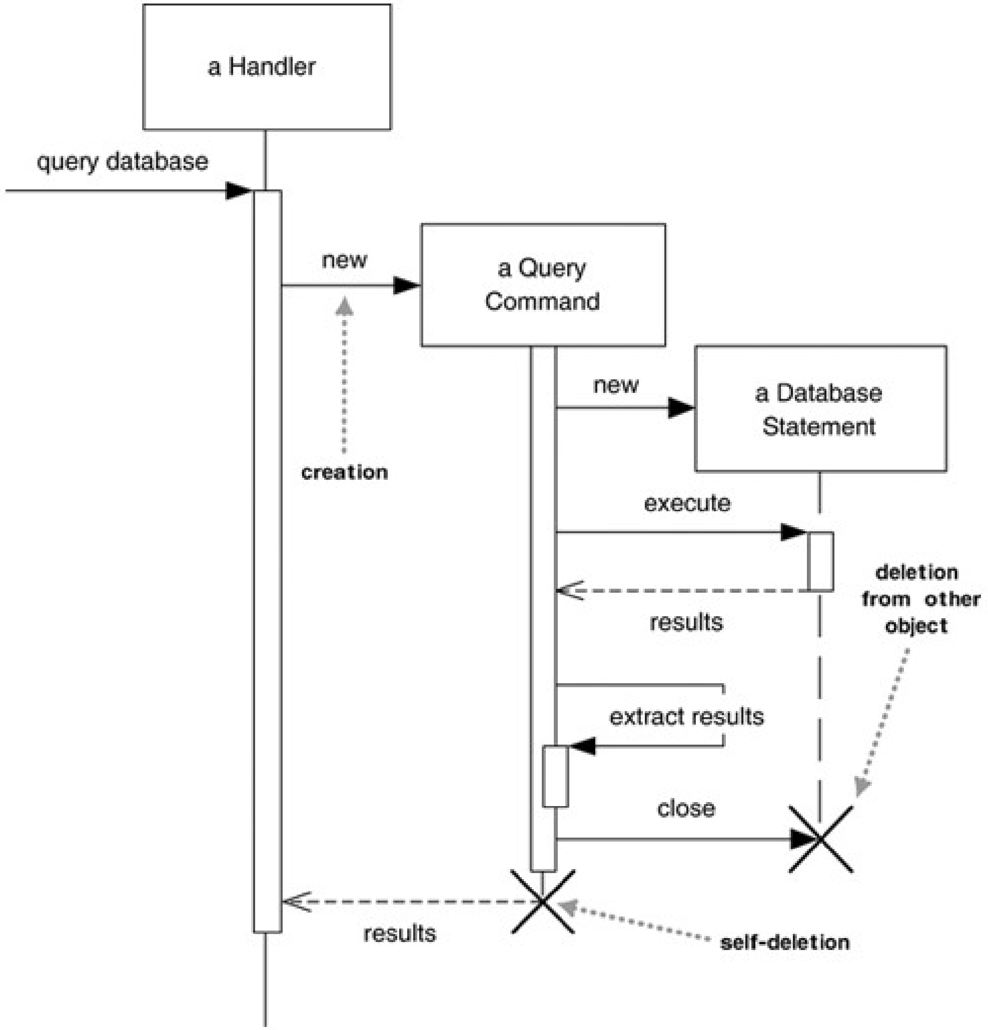
- Activation Bars – Vertical rectangle
  - Identifies when object is actively participating
  - Activation bars may “overlap”

- Navigation arrows – illustrate function calls and returns
  - Synchronous Function calls – solid line with solid arrow
  - Asynchronous Function calls – solid line with stick arrow head
  - Returns – dashed line with stick arrow head

Sequence Diagram



Sequence Diagram Example 2



Modeling Approaches

Module 5

UML Modeling Approaches

- UML-centric
  - Create functional, object, and dynamic models
  - Together represent system
- USP / RUP Approach
  - Outputs associated with workflows
  - System view grows over time
- 4+1 Architectural Approach
  - Use-case view +
  - Logical view, Implementation view, Process View, Deployment view

USP Approach

- Use-case driven
- Workflow artifacts
  - Requirements - Use Case Model
  - Analysis - Analysis Model
  - Design - Design Model, Deployment Model
  - Implementation - Implementation Model
  - Test - Test Model

4+1 - Logical View

- The design model
- Supports functional requirements
- Static aspects:
  - Class/Object diagrams
- Dynamic aspects:
  - Interaction, statechart, activity diagrams

4+1 - Implementation View

- Components and files that realize the system
- Supports configuration management
- Addresses testability
  - (Often—unwisely—overlooked)
- Static:
  - Component diagrams
- Dynamic:
  - Interaction, statechart, activity diagrams

4+1 - Process view

- Threads, concurrency, and synchronization
- Describes performance, scalability, and throughput
- Class/Object, Interaction, state-chart, activity diagrams
  - Diagrams emphasize active classes and concurrency issues

4+1 - Deployment view

- Hardware elements
- Addresses distribution, delivery, and installation
- Static aspects:
  - Deployment diagrams
- Dynamic aspects:
  - Interaction, statechart, activity diagrams

4+1 - Use-case View

- Use cases are the +1 aspect
- This is the lynch-pin of the views
- All functionality and work should be based on a real need, identified in, or traced back to, a Use case

Model Artifacts

- **Use Case Model** - defines and describes functionality
  - Use Cases
  - Use Case Diagrams
- **Analysis Model** - refines use cases and creates initial allocation of behaviors
  - Analysis Class Diagrams
  - Activity Diagrams
  - Package Diagrams
- **Design Model** - defines static structure of system
  - Class diagrams
  - Package diagrams

Model Artifacts (cont)

- **Implementation Model** - defines software components and mapping of classes to objects
  - Class diagrams
  - Object diagrams
  - Component diagrams
- **Deployment Model** - describes topology of solution and nodes
  - Component diagrams
  - Package diagrams
  - Deployment diagrams
- **Test Model** - defines set of test cases
  - Class diagrams
  - State diagrams

Modeling Complex Systems

Module 3

Modeling Complex Systems

- Successful creation of complex systems require
  - Careful control
  - Standardized processes and procedures
  - Standardized languages and techniques
  - Robust productivity and development tools
  - Stakeholder involvement

Software Development Lifecycle

- Formalized version of System Development Lifecycle targeted at software
- Drawn out of traditional engineering processes
- Software engineering is often expected to be infinitely flexible
  - This (unrealistic) aspect differs substantially from traditional engineering

Software Development Lifecycle

- Broken into four phases
  - Inception
  - Elaboration
  - Construction
  - Transition
- Phase completion occurs when milestones are achieved

Software Engineering Process

- Intended to manage software development process across SDLC
- Many differing variations
  - Waterfall
  - Spiral
  - Rational Unified Process
  - Unified Software Process
  - Agile process

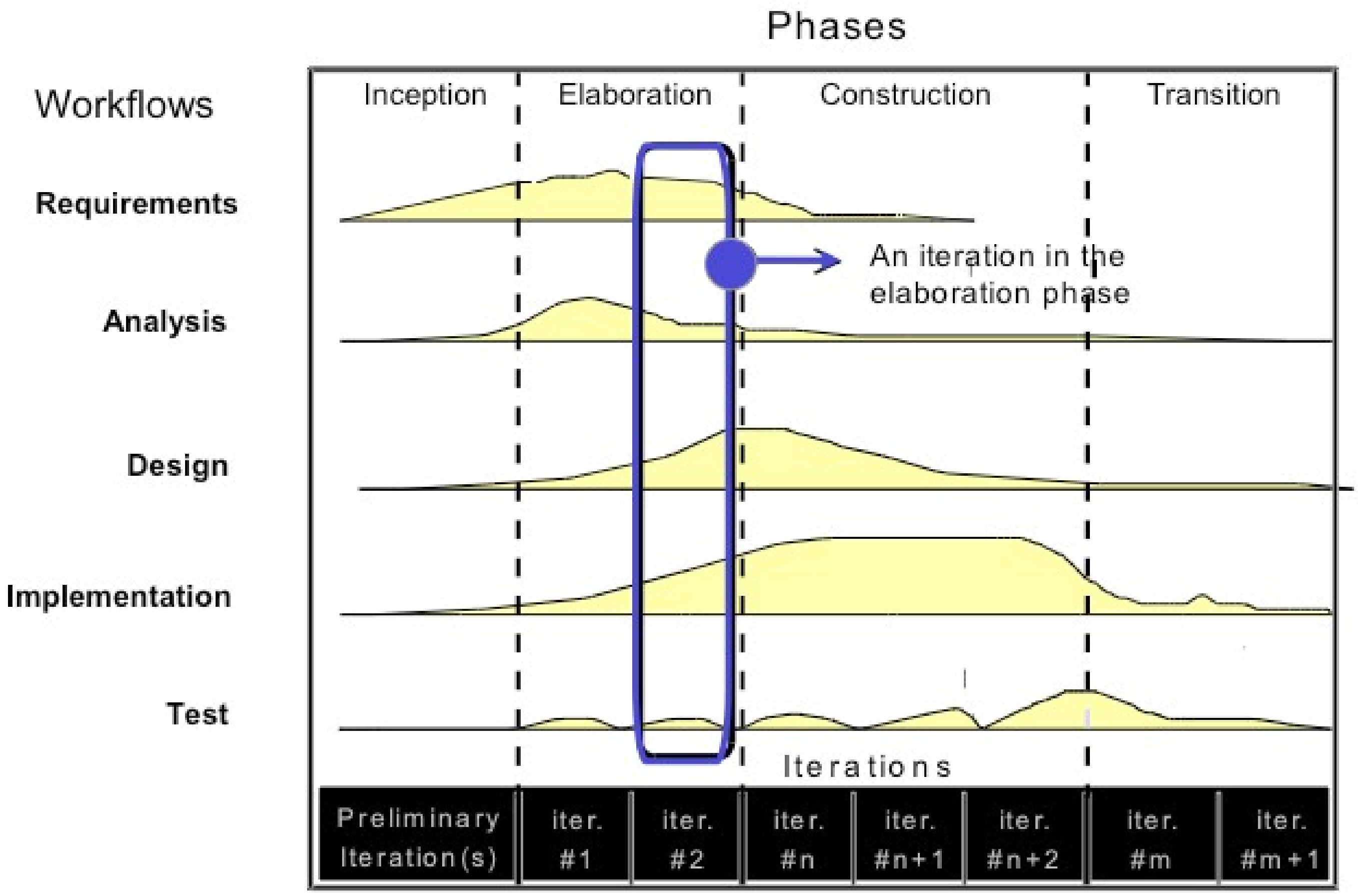
UP, USP, USDP / RUP

- Unified [Software [Development]] Process is standardized OO-focused SE process
- Rational Unified Process is a specific implementation of USP
- USP / RUP are:
  - Iterative
  - Stakeholder focused
  - Requirements driven
  - Use-Case driven
  - Architecture focused
  - Artifact centric

USP Workflows

- USP addresses effort across SDLC phases as workflows
- 5 standard workflows
  - Requirements
  - Analysis
  - Design
  - Implementation
  - Test
- Workflows occur in concert with one another and iteratively across SDLC phases

USP Workflows Diagram



USP Phase Deliverables

- USP phase transitions occur with delivery of artifacts
- Phase artifacts should support
  - Inception - system vision
  - Elaboration - baseline architecture
  - Construction - initial capability
  - Transition - user acceptance

Testing in Modern Software Development

- RUP, USP, Agile processes all emphasize testing
  - Verify understanding of problem and obtain user agreement with proposed solution
  - Verify functional and non-functional behavior at every step
  - Build only on thoroughly tested intermediate elements
- Very stable, well understood, projects can use waterfall approach

What is Architecture?

- Generally accepted as addressing quality of service, non-functional requirements
- A system that works but is unusable is poorly architected
  - too slow, unreliable, unmanagable, unmaintainable, unscalable
- Testing must include non-functional requirements

UML Workshop

Module 6

Workshop Logistics

- Work through OO/UML modeling
- Use UML modeling tool or pencil and paper
  - Pencil and paper have a shorter learning curve
  - Tools might enforce "correct" use and encourage more learning
- We will break and discuss at intervals
- Ask early, ask often!

“Deliverables”

- Use Case Model
- Activity Diagrams
- Class/Object Diagrams
- Sequence Diagrams
- Package and Deployment Diagrams
- Do not try to “complete”
  - Build a representative sample to ensure your comfort with the concepts and tools

Class/Object Diagrams

- Key/initial part of design model
- Derive from use cases and activity diagrams
- Identify generalizations/specializations
- Consider logical groupings
- Think about OO rules and guidance
- Verify using sequence diagrams
  - Do your objects and their relationships support the required behavior?

Sequence Diagrams

- Focus on object-to-object interactions
- Leverage class diagrams to identify objects
- Build necessary control flows to satisfy activity diagrams
  - Thereby closing the loop and verifying the model so far

Packaging and Deployment

- Focus on system structure
- Packaging should avoid circular dependencies
  - Group together related things, and things that change together
- Build Deployment Model
  - Identify network traffic, bottlenecks, single points of failure, and other architecturally significant elements

Have Fun

- Don't be afraid to experiment
  - You'll often learn more by mistakes than by doing it right
  - Learning is an active process; you can't learn unless you do
  - Ask early, ask often, discuss freely