Software Engineering
G22.2440-001

Session 7 - Main Theme
From Analysis and Design
to Software Architectures
(Part II)

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

- Review
- OOAD Using UML
  - Use Cases Review
  - Use Case Design
  - GRASP Patterns
  - System Design
  - UML and OOAD Summarized
  - UML to Java Mapping
  - Sample OCL Problem
- Introduction to Design and Architectural Patterns
- Micro/Macro Architecture
- Design and Architectural Patterns
- Summary
  - Individual Assignment #4 - ongoing
  - Project (Part 2) - ongoing

Summary of Previous Session

- Towards Agile Enterprise Architecture Models
- Building an Object Model Using UML
- Architectural Analysis
- Design Patterns
- Architectural Patterns
- Architecture Design Methodology
  - Achieving Optimum-Quality Results
  - Selecting Kits and Frameworks
  - Using Open Source vs. Commercial Infrastructures
- Summary
  - Individual Assignment #4
  - Project (Part 2)

Part I

Review

See: Sub-Topic 6 Presentation on
"Introduction to OOAD Modeling and UML"
and
Sub-Topic 7 Presentation on
"Use Case Modeling"

Part II

OOAD Using UML

UML Notation Baseline

(review)

Diagram Name	**Type**	**Phase**
Use Case	Static*	Analysis
Class	Static	Analysis
Activity	Dynamic**	Analysis
State-Transition	Dynamic	Analysis
Event Trace (Interaction)	Dynamic	Design
Sequence	Dynamic	Design
Collaboration	Dynamic	Design
Package	Static	Delivery
Deployment	Dynamic	Delivery

***Static describes structural system properties**
****Dynamic describes behavioral system properties.**

What can you Model with UML?

UML defines twelve types of diagrams, divided into three categories

- Four diagram types represent static application structure:
  - Class Diagram
  - Object Diagram
  - Component Diagram
  - Deployment Diagram
- Five represent different aspects of dynamic behavior
  - Use Case Diagram
  - Sequence Diagram
  - Activity Diagram
  - Collaboration Diagram
  - Statechart Diagram
- Three represent ways you can organize and manage your application modules
  - Packages
  - Subsystems
  - Models

Source: http://www.omg.org/gettingstarted/what_is_uml.htm

Part II.1

Use Cases Review

Use Cases: Scenario Based Requirements Modeling

- Recommended: UML distilled...

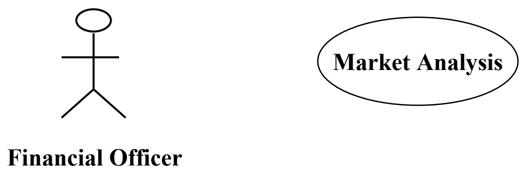
Use Cases

Use case

- specifies the behavior of a system
- sequence of *actions* to yield an observable result of value to an *actor*
- Capture the *intended* behavior (the what) of the system omitting the implementation of the behavior (the how)
- customer requirements/ early analysis

What is a Use Case?

- Description of a sequence of *actions*, including variants (specifies desired behavior)
- Represents a functional requirement on the system
- Use case involves interaction of actors and the system



Use Cases: Terms and Concepts

- Unique name
- Sequence of actions (event flows)
  - textual (informal, formal, semi formal)

    Main flow of events: The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a pin number...
  - interaction diagrams

Use Cases: Summary

- A use case encodes a typical user interaction with the system. In particular, it:
  - captures some user-visible function.
  - achieves some concrete goal for the user.
- A complete set of use cases largely defines the requirements for your system: everything the user can see, and would like to do.
- The granularity of your use cases determines the number of them (for you system). A clear design depends on showing the right level of detail.
- A use case maps actors to functions. The actors need not be people.

13

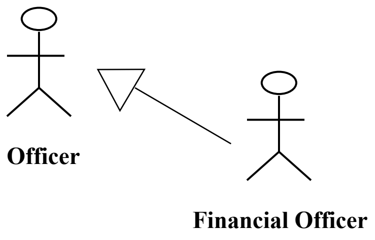
Actors

- *Role* that a user plays with respect to the system
- Actors carry out use cases
  - look for actors, then their use cases
- Actors do not need to be humans!
- Actors can get value from the use case or participate in it

14

Actors

- Actors can be specialized



- connected to use cases only by association
- association = communication relationship (each one sending, or receiving messages)

15

Use Case Description

- Generic, step-by-step written description of a use case’s event flow
- Includes interactions between the actor(s) and a use case
- May contain extension points
- Clear, precise, short descriptions

16

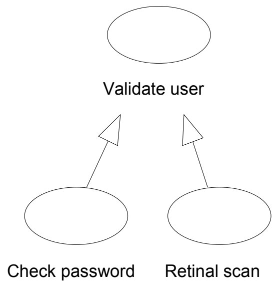
Example Use Case Description

- Capture deal
  1. Enter the user name & bank account
  2. Check that they are valid
  3. Enter number of shares to buy & share ID
  4. Determine price
  5. Check limit
  6. Send order to NYSE
  7. Store confirmation number

Organizing Use Cases

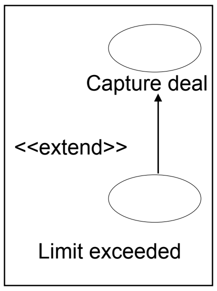
- Generalization
- Use/Include
- Extend

Generalization Relationship



- child use case inherits behavior and meaning of the parent use case
- child may add or override the parent's behavior
- child may substitute any place the parent appears

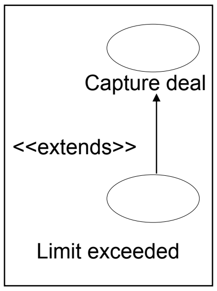
Extends Relationship



- Allows to model the part of a use case the user may see as optional
- Allows to model conditional subflows
- Allows to insert subflows at a certain point, governed by actor interaction

- represented by an *extend* dependency
- extension points (in textual event flows)

Extends Relationship



- Allows to model the part of a use case the user may see as optional
- Allows to insert subflows at a certain point, governed by actor interaction
- Allows to model conditional subflows

⇨ Capture the base use case

⇨ For every step ask

  what could go wrong

  how might this work out differently

⇨ Plot every variation as an extension of the use case

Example: Extension Points

- Capture deal

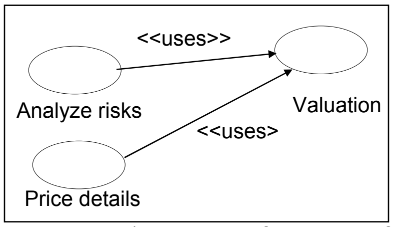
  1. Enter the user name & bank account
  2. Check that they are valid

  extension point: reenter data in case they are invalid

  3. Enter number of shares to buy & share ID
  4. Determine price
  5. Check limit
  6. Send order to NYSE
  7. Store confirmation number

Uses/Includes Relationship

- Used to avoid describing
  the same flow of events several times, by putting the common behavior in a use case of its own

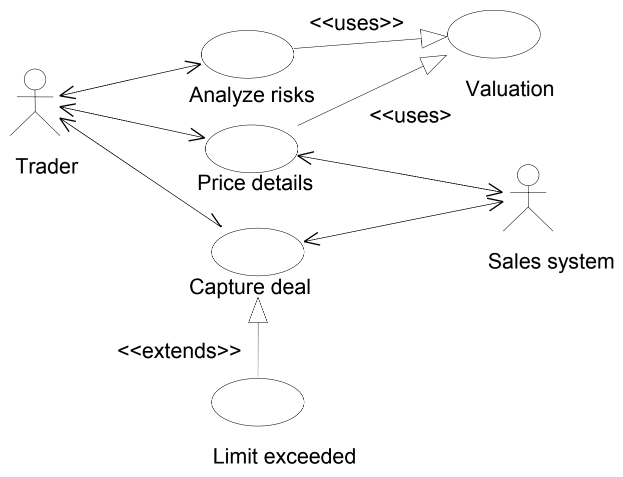


- Avoids copy-and-paste of parts of use case descriptions

Comparing Extends/Uses

- Different intent
  - extends
    - to distinguish variants
    - set of actors perform use case and all extensions
    - actor is linked to “base” case
  - uses/includes
    - to extract common behavior
    - often no actor associated with the common use case
    - different actors for “caller” cases possible

A Use Case Diagram



Use Case Diagrams (Functional)

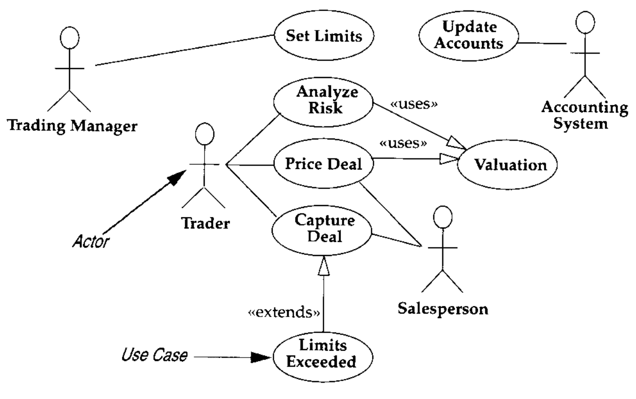


Figure 3-1: *Use Case Diagram*

Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Properties of Use Cases

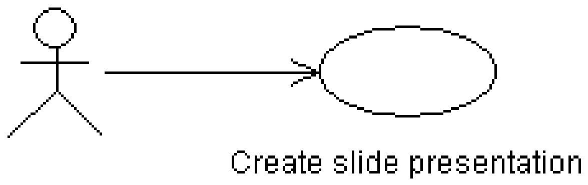
- Granularity: fine or course
- Achieve a discrete goal
- Use cases describe externally required functionality
- Often: Capture user-visible function

When and How

- Requirements capture - first thing to do
- Use case: Every discrete thing your customer wants to do with the system
  - give it a name
  - describe it shortly (some paragraphs)
  - add details later

Use Case Examples, 1

(High-level use case for powerpoint.)

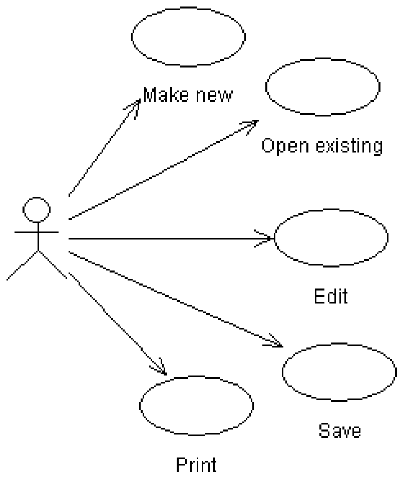


About the Last Example...

- Although this is a valid use case for powerpoint, and it completely captures user interaction with powerpoint, it’s too vague to be useful.

Use Case Examples, 2

(Finer-grained use cases for powerpoint.)

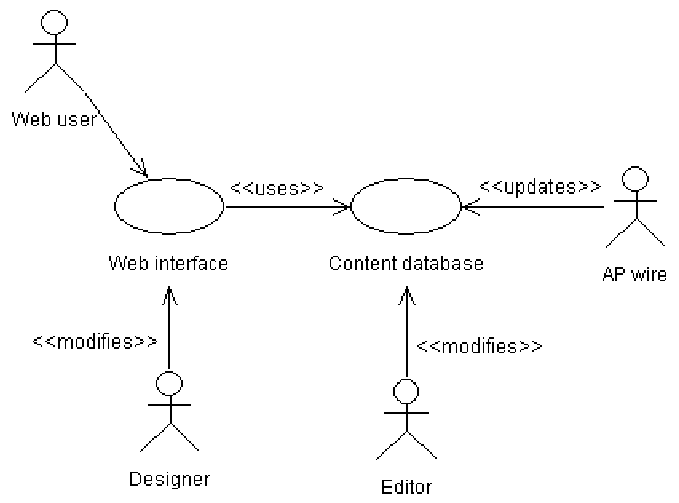


About the Last Example...

- The last example gives a more useful view of powerpoint (or any similar application).
- The cases are vague, but they focus your attention the key features, and would help in developing a more detailed requirements specification.
- It still doesn’t give enough information to characterize powerpoint, which could be specified with tens or hundreds of use cases (though doing so might not be very useful either).

Use Case Examples, 3

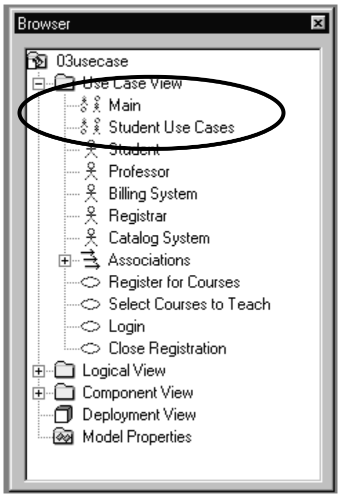
(Relationships in a news web site.)



About the Last Example...

- The last is more complicated and realistic use case diagram. It captures several key use cases for the system.
- Note the multiple actors. In particular, ‘AP wire’ is an actor, with an important interaction with the system, but is not a person (or even a computer system, necessarily).
- The notes between << >> marks are *stereotypes:* identifiers added to make the diagram more informative. Here they differentiate between different roles (i.e., different meanings of an arrow in this diagram).

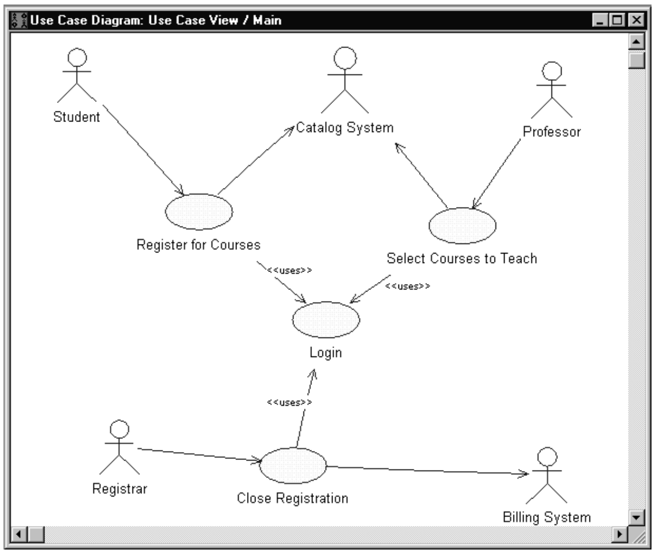
Use Case Diagrams



About the Next Slide…

- The next slide shows a full use case diagram.
- The stick figures denote actors, and the ovals are use cases (a function or behavior or interface your software provides).
- The arrows indicate 'use' or dependency. For example, the "Student" uses the function "Register for Courses", which in turn uses the external "Catalog System".
- The <<uses>> tokens attached to some of the associations (arrows) are *stereotypes*, an indication of what the association means. In this diagram, <<uses>> indicates that the association means a direct software link, i.e., that the function "Register for Courses" will directly use the function "Login". This is different than the unmarked arrows, which indicate "use" in the vague sense of manipulating or interacting.

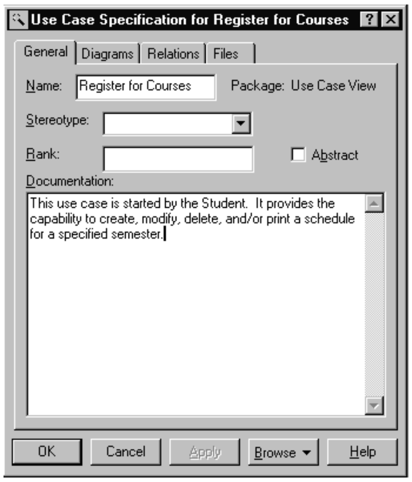
Use Case Relationships



About the Next Slide…

- The next slide shows how documentation (notes, etc) can be added to a particular element.
- Here, they're adding the documentation via the Specification dialogue.

Brief Description -- Register for Courses


Use Case Specification for Register for Courses

Name: Register for Courses

Package: Use Case View

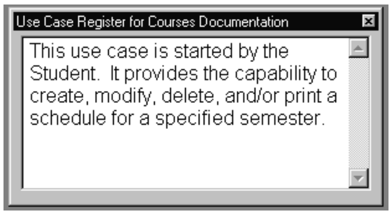
Stereotype:

Rank:

☐ Abstract

Documentation:

This use case is started by the Student. It provides the capability to create, modify, delete, and/or print a schedule for a specified semester.

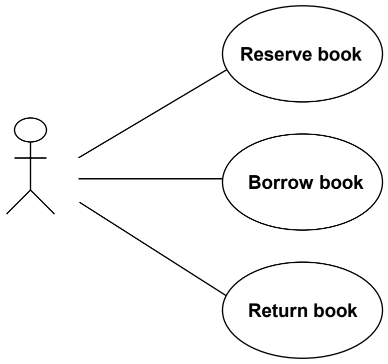

Use Case Register for Courses Documentation

This use case is started by the Student. It provides the capability to create, modify, delete, and/or print a schedule for a specified semester.

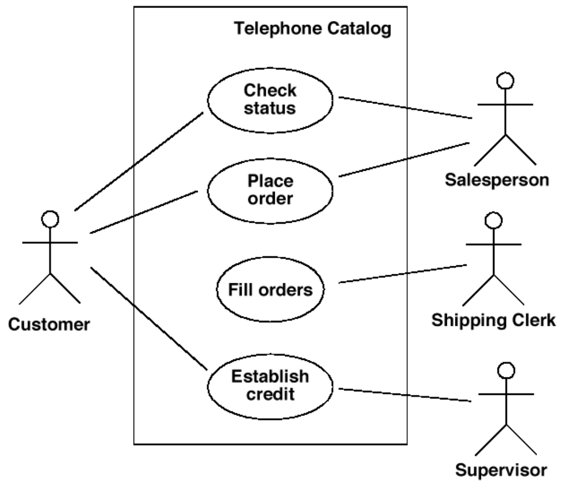
More Examples Use Case Diagrams

- Means of capturing requirements
- Document interactions between user(s) and the system
  - User (actor) is not part of the system itself
  - But an actor can be *another* system
- An individual use case represents a task to be done with support from the system (thus it is a ‘coherent unit of functionality’)

Simple Use Case Diagram



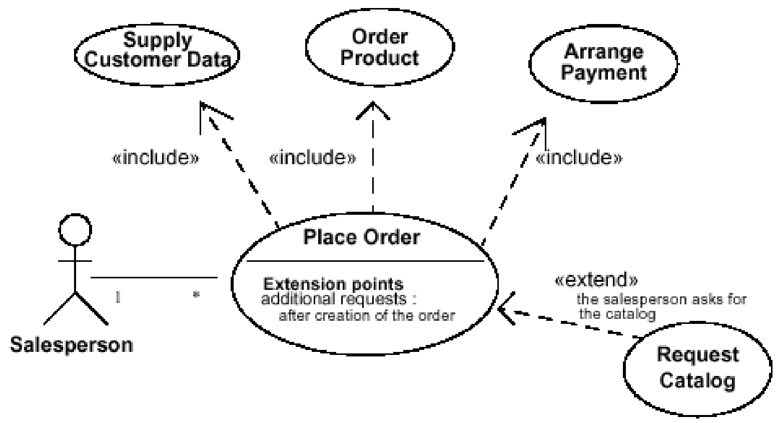
Use Case Diagram with Multiple Actors



Use Cases

- Are actually defined as text, including descriptions of all of the normal and exception behavior expected
- Do not reveal the structure of the system
- Collectively define the boundaries of the system to be implemented
- Provide the basis for defining development iterations

Example Use Case Diagram (Advanced Features)



Practical (Agile) UML

- You don't typically use *all* the diagrams
  - You'll choose between them based on preference and particular situation
- You typically use *many* diagrams
  - A single use case may not capture all scenarios
  - If you are going to use statecharts, there are probably *lots* of objects with states
  - Each sequence/collaboration diagram only shows *one* interaction

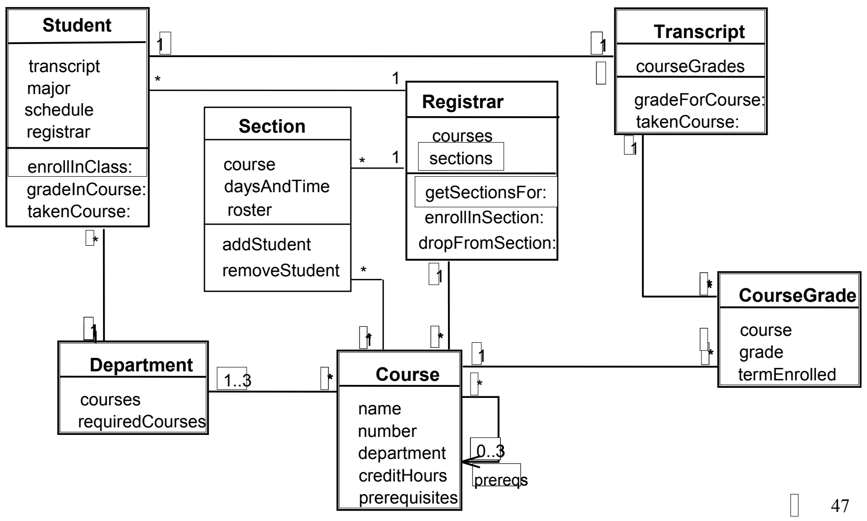
45

Example: Student Registration System

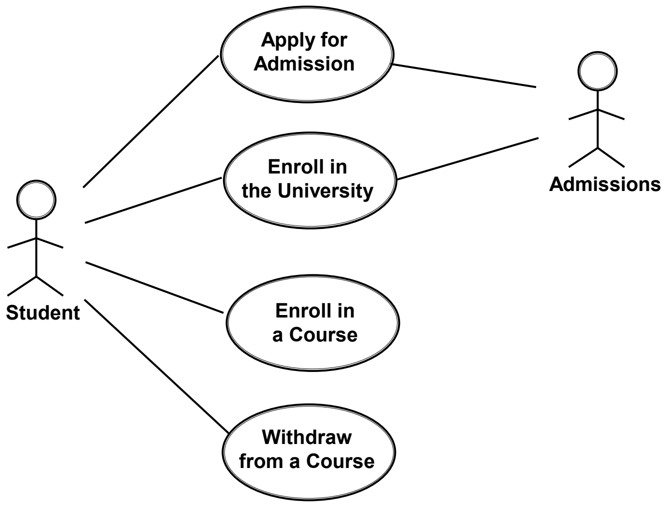
- Not going to do *all* the diagrams
  - Not all types, not even all that completely specify the system
- But this is an application you know, so the examples may help make sense

46

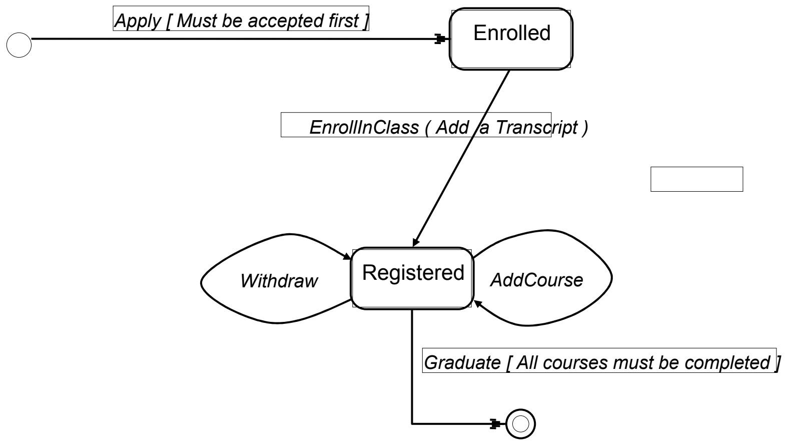
Student Registration Class Diagram



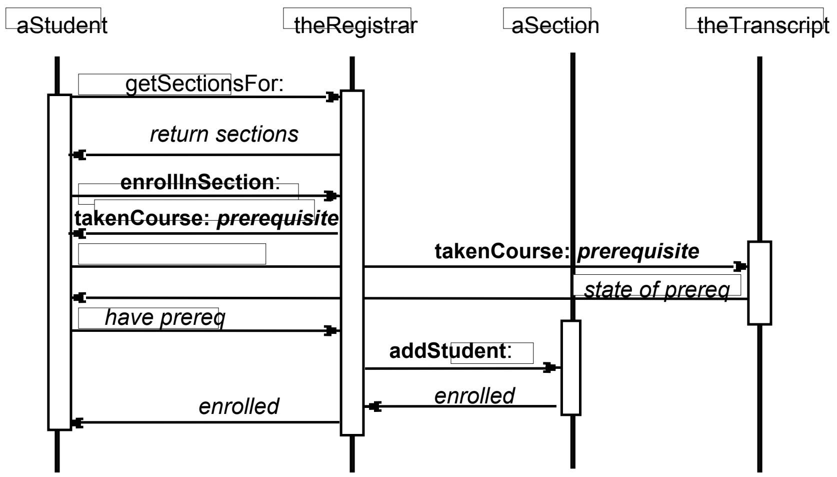
Partial Use Case Diagram



States of a Student


Apply [Must be accepted first]
Enrolled
EnrollInClass (Add a Transcript)
Withdraw
Registered
AddCourse
Graduate [All courses must be completed]

Sequence Diagram: Registering for Course



Process to Representations

- OOA
  - CRC Cards (but they're not officially UML)
  - Use Cases
- OOD
  - Just about all of the rest
  - But variations—some detail is later
- OOP
  - Can actually go UML->code with some tools!

Summary

- This lesson only describes how to write UML
- It does not answer the big question on how to come up with the model !
- A System Sequence Diagram is an excellent first cut
- Both sets of artifacts are used in the sequence and collaborations diagrams

Part II.2

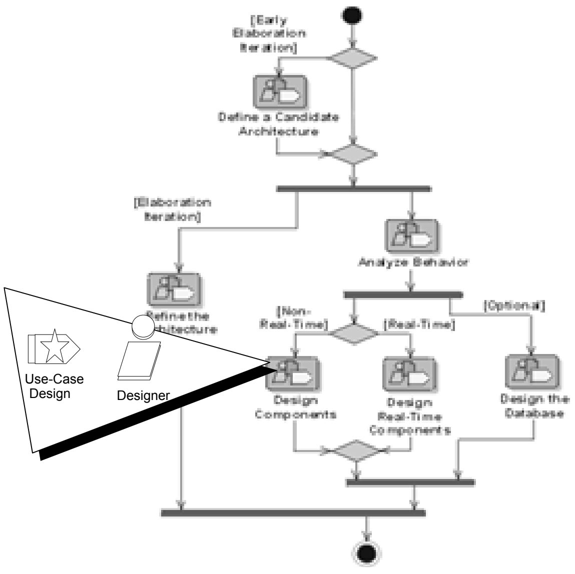
Use Case Design

See: Sub-Topic 1 Presentation on "UML and the SDLC"

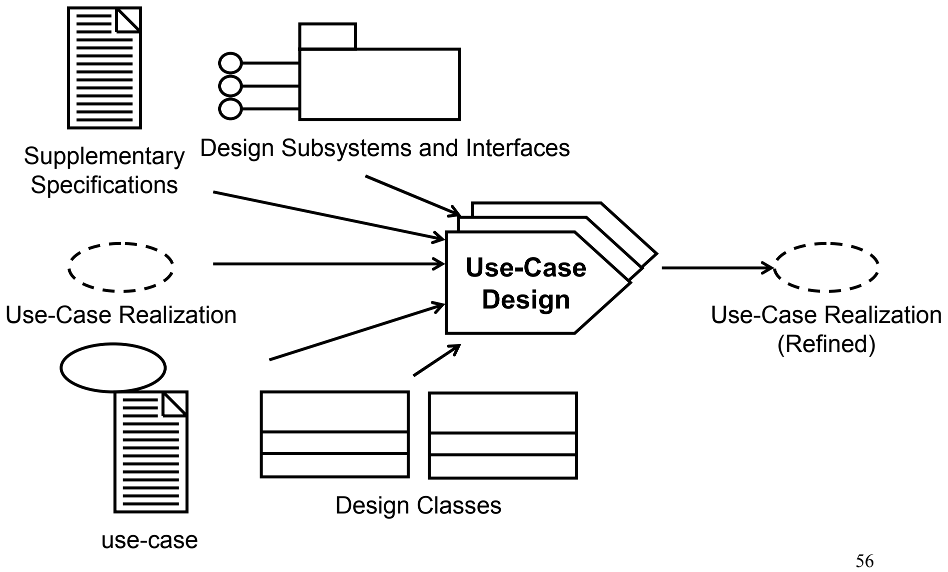
Objectives: Use-Case Design

- Understand the purpose of Use-Case Design and when in the lifecycle it is performed
- Verify that there is consistency in the use-case implementation
- Refine the use-case realizations from Use-Case Analysis using defined design model elements

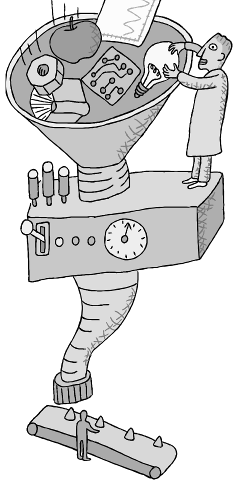
Use-Case Design in Context



Use-Case Design Overview


Supplementary Specifications
Design Subsystems and Interfaces
Use-Case Design
Use-Case Realization
Use-Case Realization (Refined)
use-case
Design Classes

Use-Case Design Steps



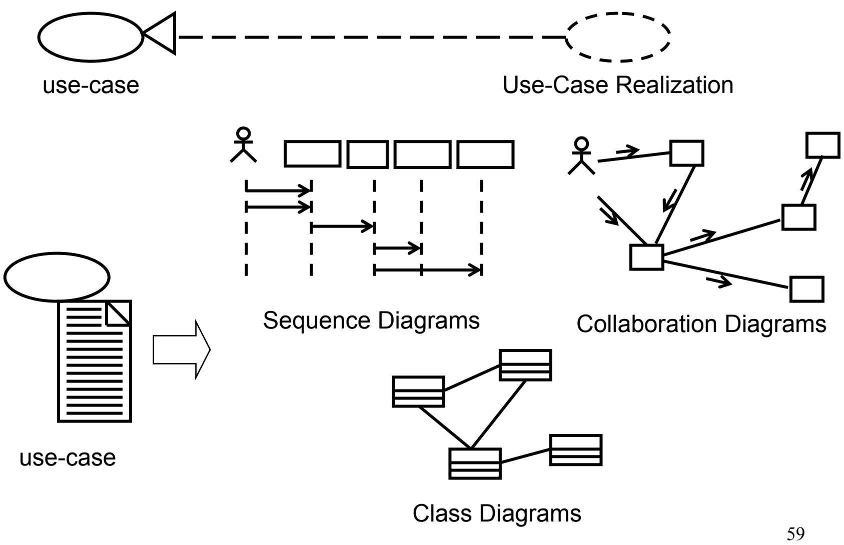
- Describe interaction between design objects
- Simplify sequence diagrams using subsystems
- Describe persistence related behavior
- Refine the flow of events description
- Unify classes and subsystems

Use-Case Design Steps

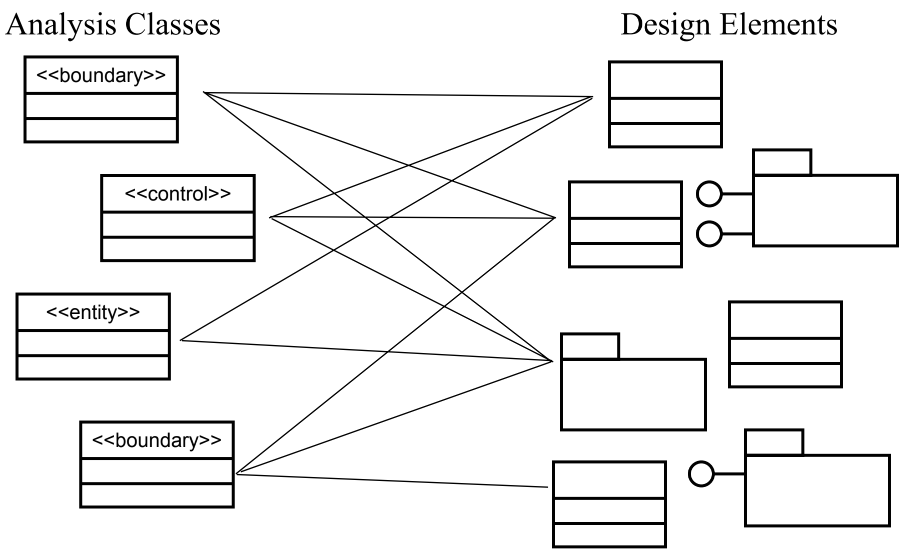
☆

- Describe interaction between design objects
- Simplify sequence diagrams using subsystems
- Describe persistence related behavior
- Refine the flow of events description
- Unify classes and subsystems

Review: Use-Case Realization



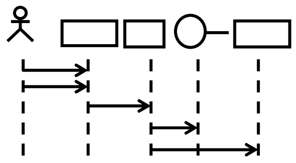
Review: From Analysis Classes to Design Elements

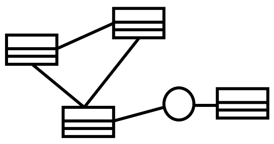


Many-to-Many Mapping

Use-Case Realization Refinement

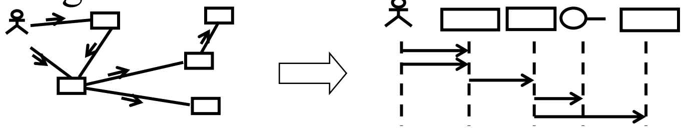
- Identify participating objects
- Allocate responsibilities amongst objects
- Model messages between objects
- Describe processing resulting from messages
- Model associated class relationships


Sequence Diagrams


Class Diagrams

Use-Case Realization Refinement Steps

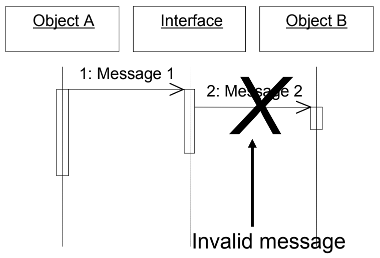
- Identify each object that participates in the flow of the use-case
- Represent each participating object in a sequence diagram

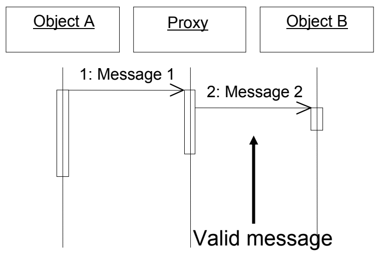


- Incrementally incorporate applicable architectural mechanisms

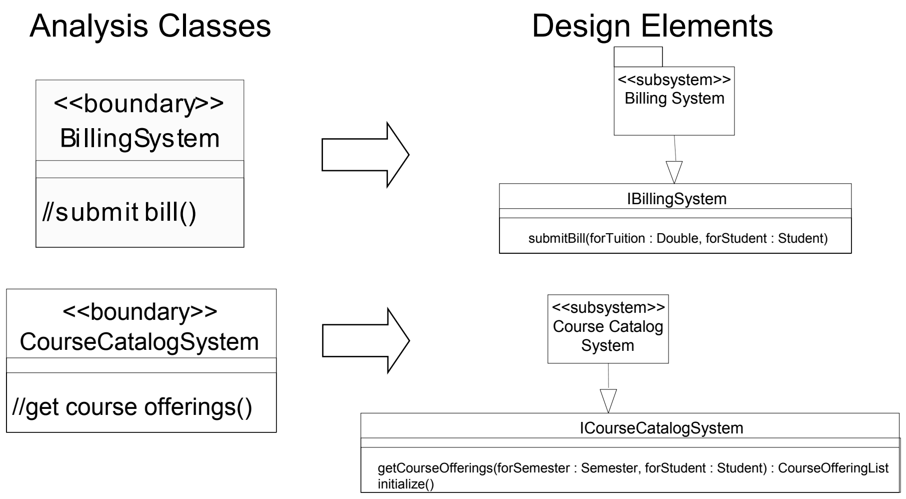
Representing Subsystems on a Sequence Diagram

- Interfaces
  - Represents any model element that realizes the interface
  - No message should be drawn from the interface
- Proxy class
  - Represent a specific subsystem
  - Messages can be drawn from the proxy



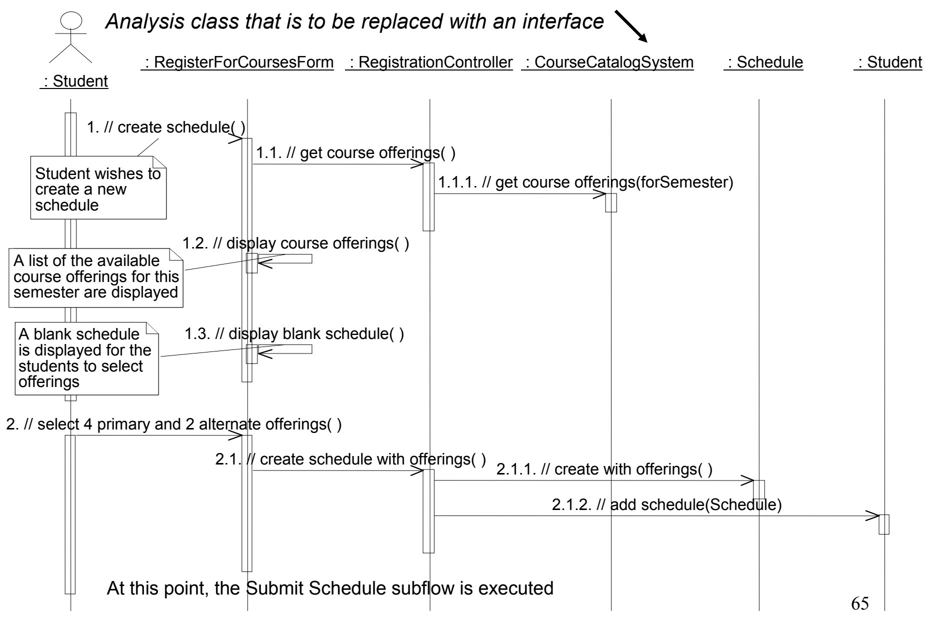


Example: Incorporating Subsystem Interfaces



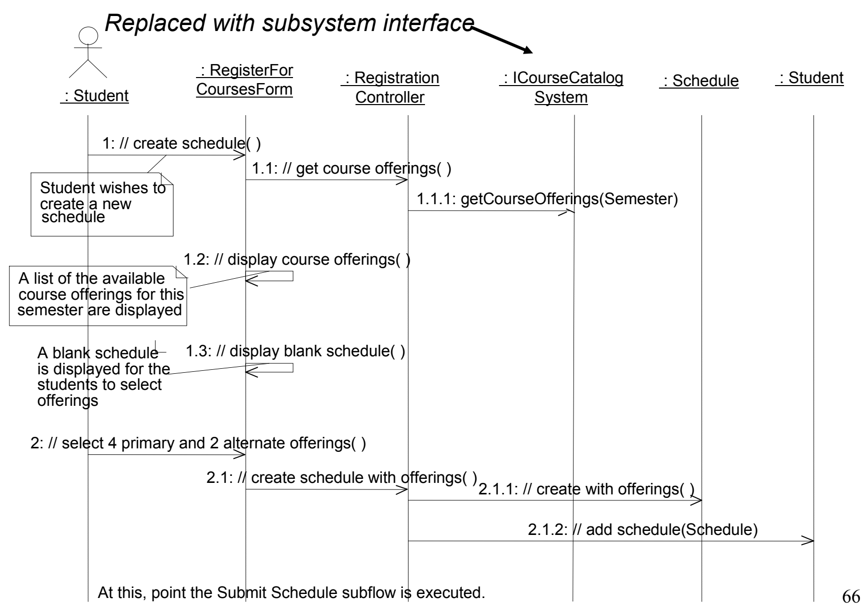
All other analysis classes mapped directly to design classes

Example: Incorporating Subsystem Interfaces (Before)



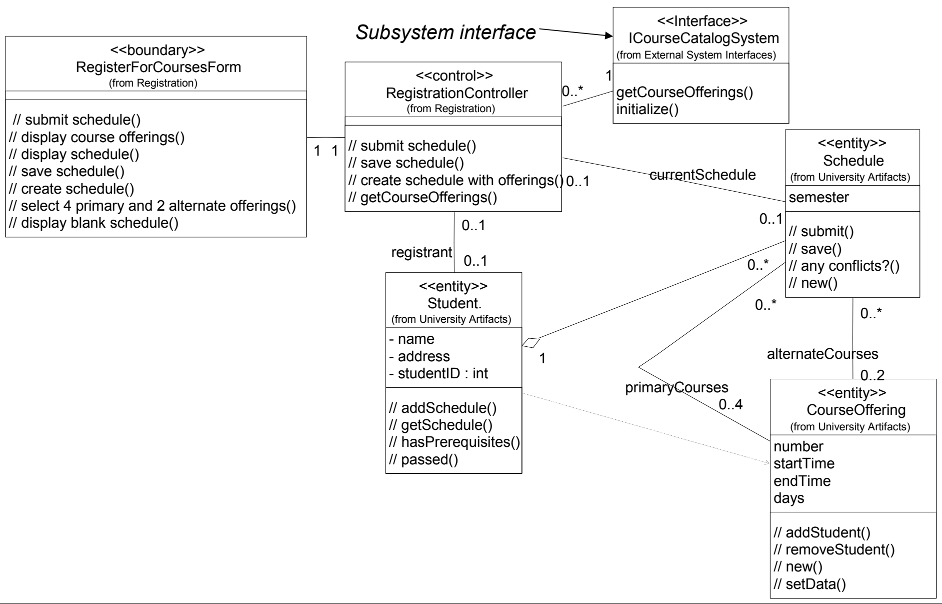
Analysis class that is to be replaced with an interface
: Student
: RegisterForCoursesForm
: RegistrationController
: CourseCatalogSystem
: Schedule
: Student
1. // create schedule( )
Student wishes to create a new schedule
1.1. // get course offerings( )
1.1.1. // get course offerings(forSemester)
1.2. // display course offerings( )
A list of the available course offerings for this semester are displayed
1.3. // display blank schedule( )
A blank schedule is displayed for the students to select offerings
2. // select 4 primary and 2 alternate offerings( )
2.1. // create schedule with offerings( )
2.1.1. // create with offerings( )
2.1.2. // add schedule(Schedule)
At this point, the Submit Schedule subflow is executed

Example: Incorporating Subsystem Interfaces (After)



Replaced with subsystem interface
: Student
: RegisterFor CoursesForm
: Registration Controller
: ICourseCatalog System
: Schedule
: Student
1: // create schedule( )
Student wishes to create a new schedule
1.1: // get course offerings( )
1.1.1: getCourseOfferings(Semester)
1.2: // display course offerings( )
A list of the available course offerings for this semester are displayed
1.3: // display blank schedule( )
A blank schedule is displayed for the students to select offerings
2: // select 4 primary and 2 alternate offerings( )
2.1: // create schedule with offerings( )
2.1.1: // create with offerings( )
2.1.2: // add schedule(Schedule)
At this, point the Submit Schedule subflow is executed.

Example: Incorporating Subsystem Interfaces (VOPC)



Incorporating Architectural Mechanisms: Security

- Analysis-Class-to-Architectural-Mechanism Map from Use-Case Analysis

Analysis Class	Analysis Mechanism(s)
Student	Persistency, *Security*
Schedule	Persistency, *Security*
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	Distribution

Incorporating Architectural Mechanisms: Distribution

- Analysis-Class-to-Architectural-Mechanism Map from Use-Case Analysis

Analysis Class	Analysis Mechanism(s)
Student	Persistency, Security
Schedule	Persistency, Security
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	*Distribution*

Review: Incorporating RMI: Steps

√ • Provide access to RMI support classes (e.g., Remote and Serializable interfaces, Naming Service)
  - *java.rmi and java.io package in Middleware layer*

√• For each class to be distributed:
  - *Controllers to be distributed are in Application layer*
  - √ *Need dependency from Application layer to Middleware layer to access java packages*
  - Define interface for class that realizes Remote
  - Have class inherit from UnicastRemoteObject

(continued)

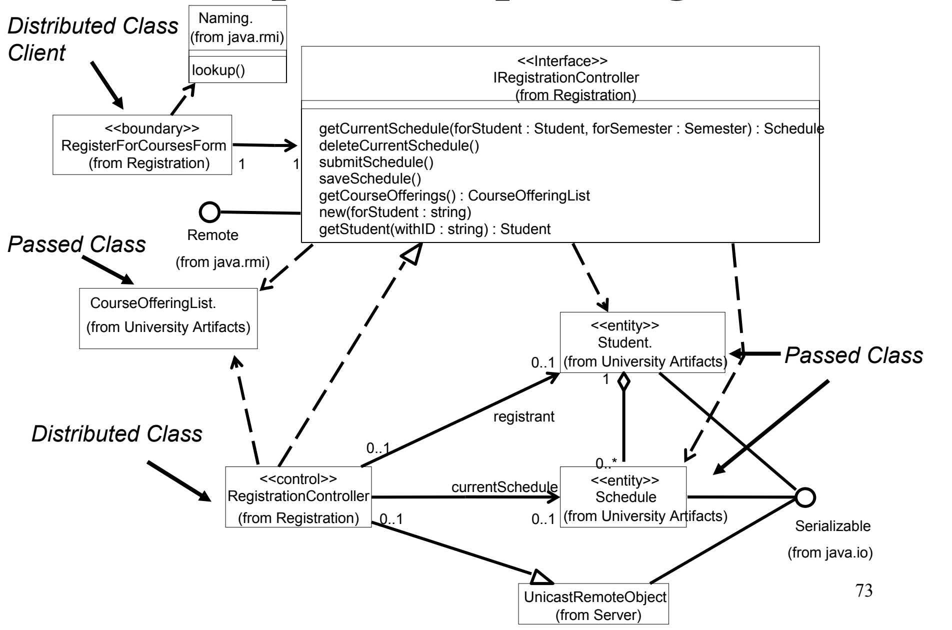
Review: Incorporating RMI: Steps (contd)

- √ Have classes for data passed to distributed √ objects realize the Serializable interface
  - *Core data types are in Business Services layer*
  - *Need dependency from Business Services layer to Middleware layer to get access to java.rmi* $\}$ Out of scope
  - Add the realization relationships
- Run pre-processor

Review: Incorporating RMI: Steps

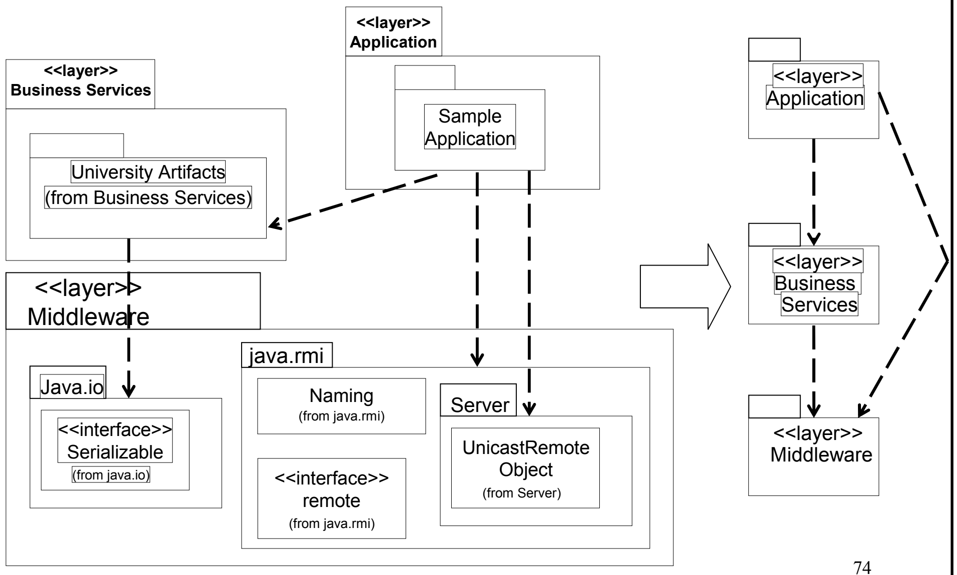
- Have distributed class clients lookup the √ remote objects using the Naming service
  - √ *Most Distributed Class Clients are Forms*
  - √ *Forms are in Application layer*
  - *Need dependency from Application layer to Middleware layer to get access to java.rmi*
  - Add relationship from Distributed Class Clients to Naming Service
- Create/update interaction diagrams with distribution processing (optional)

Example: Incorporating RMI



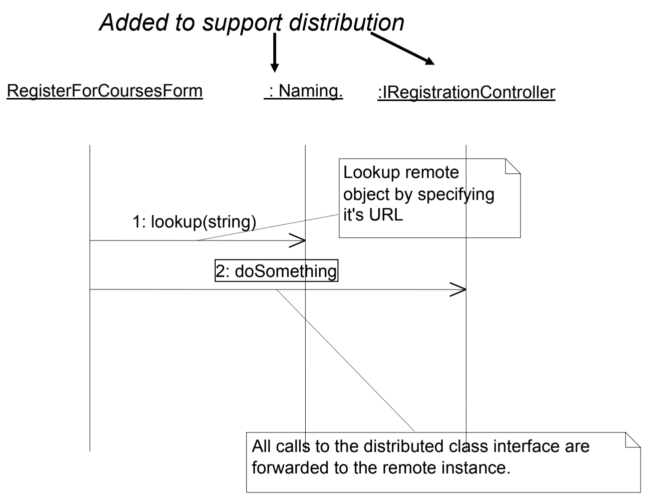
Distributed Class Client
Naming. (from java.rmi)
lookup()
<<Interface>> IRegistrationController (from Registration)
getCurrentSchedule(forStudent : Student, forSemester : Semester) : Schedule
deleteCurrentSchedule()
submitSchedule()
saveSchedule()
getCourseOfferings() : CourseOfferingList
new(forStudent : string)
getStudent(withID : string) : Student
<<boundary>> RegisterForCoursesForm (from Registration)
1
1
Remote (from java.rmi)
Passed Class
CourseOfferingList. (from University Artifacts)
<<entity>> Student. (from University Artifacts)
0..1
1
Passed Class
registrant
Distributed Class
0..1
0..*
<<control>> RegistrationController (from Registration)
currentSchedule
<<entity>> Schedule (from University Artifacts)
0..1
0..1
Serializable (from java.io)
UnicastRemoteObject (from Server)

Example: Incorporating RMI (contd)



<<layer>> Business Services
University Artifacts (from Business Services)
<<layer>> Application
Sample Application
<<layer>> Middleware
Java.io
<<interface>> Serializable (from java.io)
java.rmi
Naming (from java.rmi)
<<interface>> remote (from java.rmi)
Server
UnicastRemote Object (from Server)
<<layer>> Application
<<layer>> Business Services
<<layer>> Middleware

Example: Incorporating RMI (contd)



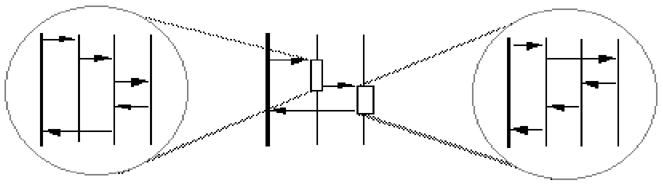
Added to support distribution
RegisterForCoursesForm
: Naming.
:IRegistrationController
Lookup remote object by specifying it's URL
1: lookup(string)
2: doSomething
All calls to the distributed class interface are forwarded to the remote instance.

Use-Case Design Steps

- Describe interaction between design objects
- ☆ Simplify sequence diagrams using subsystems
- Describe persistence related behavior
- Refine the flow of events description
- Unify classes and subsystems

Encapsulating Subsystem Interactions

- Interactions can be described at several levels
- Subsystem interactions can be described in their own interaction diagrams

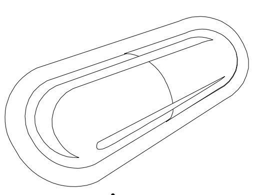


Raises the level of abstraction

When to Encapsulate Sub-Flows in a Subsystem

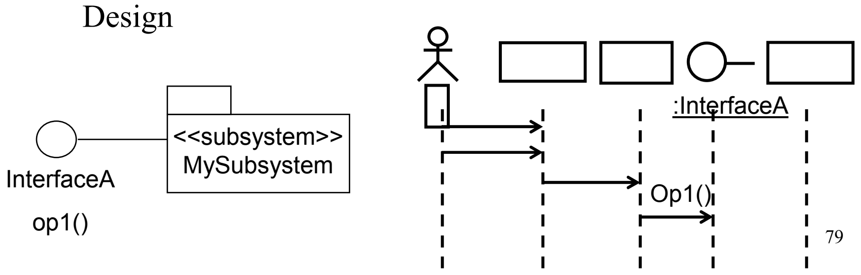
Encapsulate a Sub-flow when it:

- occurs in multiple use-case realizations
- has reuse potential
- is complex and easily encapsulated
- is responsibility of one person/team
- produces a well-defined result
- is encapsulated within a single Implementation Model component



Guidelines: Encapsulating Subsystem Interactions

- Subsystems should be represented by their interfaces on interaction diagrams
- Messages to subsystems are modeled as messages to the subsystem interface
- Messages to subsystems correspond to operations of the subsystem interface
- Interactions within subsystems modeled in Subsystem Design



Advantages of Encapsulating Subsystem Interactions

Use-case realizations:

– Are less cluttered

– Can be created before the internal designs of subsystems are created (parallel development)

– Are more generic and easier to change (subsystems can be substituted)

Parallel Subsystem Development

- Concentrate on requirements that affect subsystem interfaces
- Outline required interfaces
- Model messages that cross subsystem boundaries
- Draw interaction diagrams in terms of subsystem interfaces for each use-case
- Refine the interfaces needed to provide messages
- Develop each subsystem in parallel

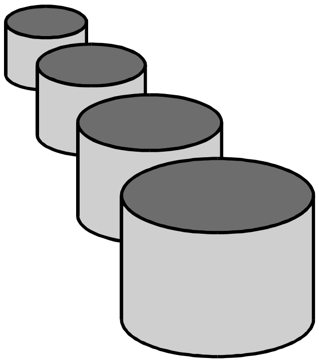
Use subsystem interfaces as synchronization points

Use-Case Design Steps

- Describe interaction between design objects
- Simplify sequence diagrams using subsystems
- ☆ Describe persistence related behavior
- Refine the flow of events description
- Unify classes and subsystems

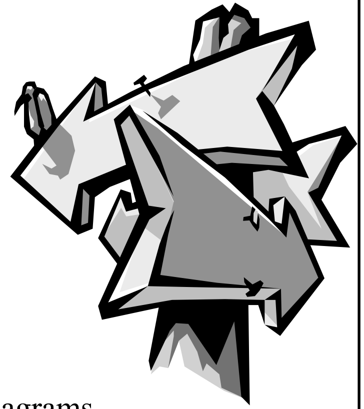
Use-Case Design Steps: Describe Persistence-related Behavior

- Describe Persistence-related Behavior
  - Modeling Transactions
  - Writing Persistent Objects
  - Reading Persistent Objects
  - Deleting Persistent Objects



Modeling Transactions

- What is a Transaction?
  - Atomic operation invocations
  - “All or nothing”
  - Provide consistency
- Modeling Options
  - Textually (scripts)
  - Explicit messages
- Error conditions
  - Rollback
  - Failure modes
  - May require separate interaction diagrams



Incorporating the Architectural Mechanisms: Persistency

- Analysis-Class-to-Architectural-Mechanism Map from Use-Case Analysis

Analysis Class	Analysis Mechanism(s)
Student	*Persistency*, Security
Schedule	*Persistency*, Security
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	Distribution

OODBMS Persistency

RDBMS Persistency

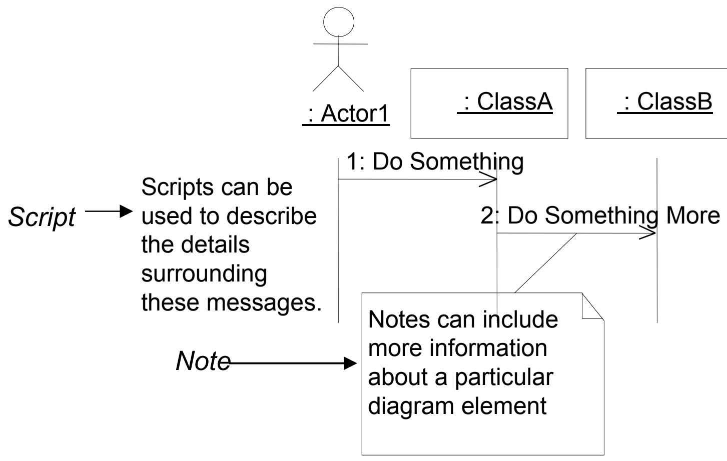
Legacy Persistency (RDBMS) deferred to Subsystem Design

Use-Case Design Steps

- Describe interaction between design objects
- Simplify sequence diagrams using subsystems
- Describe persistence related behavior
- ☆ Refine the flow of events description
- Unify classes and subsystems

Detailed Flow of Events Description Options

- Annotate the interaction diagrams



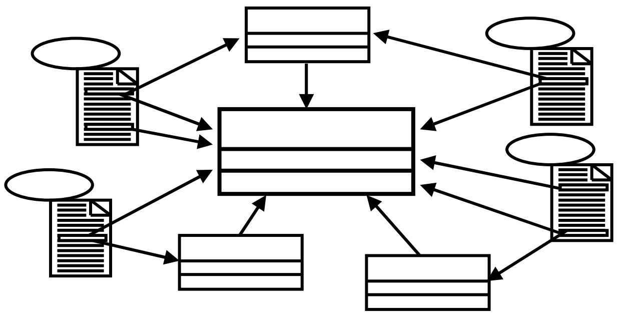
: Actor1
: ClassA
: ClassB
1: Do Something
2: Do Something More
Script
Scripts can be used to describe the details surrounding these messages.
Note
Notes can include more information about a particular diagram element

Use-Case Design Steps

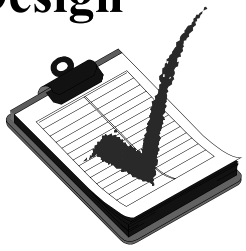
- Describe interaction between design objects
- Simplify sequence diagrams using subsystems
- Describe persistence related behavior
- Refine the flow of events description
- ☆ Unify classes and subsystems

Design Model Unification Considerations

- Model element names should describe their function
- Merge similar model elements
- Use inheritance to abstract model elements
- Keep model elements and flows of events consistent

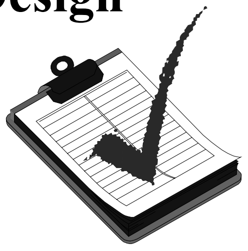


Checkpoints: Use-Case Design



- Is package/subsystem partitioning logical and consistent?
- Are the names of the packages/subsystems descriptive?
- Do the public package classes and subsystem interfaces provide a single, logically consistent set of services?
- Do the package/subsystem dependencies correspond to the relationships between the contained classes?
- Do the classes contained in a package belong there according to the criteria for the package division?
- Are there classes or collaborations of classes which can be separated into an independent package/subsystem?

Checkpoints: Use-Case Design



- Have all the main and/or sub-flows for this iteration been handled?
- Has all behavior been distributed among the participating design elements?
- Has behavior been distributed to the right design elements?
- If there are several interaction diagrams for the use-case realization, is it easy to understand which collaboration diagrams relate to which flow of events?

Review: Use-Case Design

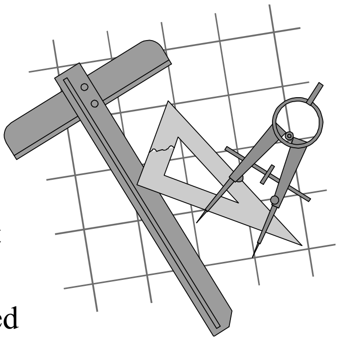
- What is the purpose of Use-Case Design?
- What is meant by encapsulating subsystem interactions? Why is it a good thing to do?

Exercise: Use-Case Design

- Given the following:
  - Analysis use-case realizations (VOPCs and interaction diagrams)
  - The analysis-class-to-design-element map
  - The analysis-class-to-analysis-mechanism map
  - Analysis-to-design-mechanism map
  - Patterns of use for the architectural mechanisms

(continued)

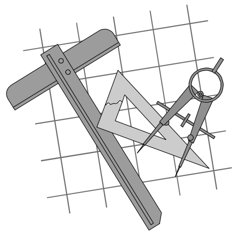
Exercise: Use-Case Design (cont.)



- Identify the following:
  - The design elements that replaced the analysis classes in the analysis use-case realizations
  - The architectural mechanisms that affect the use-case realizations
  - The design element collaborations needed to implement the use-case
  - The relationships between the design elements needed to support the collaborations

(continued)

Exercise: Use-Case Design



- Produce the following:
  - Design use-case realization
    - Interaction diagram(s) per use-case flow of events that describes the design element collaborations required to implement the use-case
    - Class diagram (VOPC) that includes the design elements that must collaborate to perform the use-case, and their relationships

(continued)

Exercise: Review

- Compare your Use-Case Realizations
  - Have all the main and/or sub-flows for this iteration been handled?
  - Has all behavior been distributed among the participating design elements?
  - Has behavior been distributed to the right design elements?
  - Are there any messages coming from the interfaces?

Part II.3

Grasp Patterns

GRASP Patterns I

- What are Patterns
- The Five GRASP Patterns

What are Patterns

- Pattern - Principles and idiomatic solutions codified in a structured format describing a problem and a solution
- Or
- A named problem/solution pair that can be applied in new contexts

What are Patterns

- It is advice from previous designers to help designers in new situations
- There are many books on the subject
- The best of all the books Design Patterns by Erich Gamma which covers 23 patterns

What are Patterns

Some definitions

- Coupling - How strongly one class has knowledge of another class
- It is a measure of how related one class is to another
- Can you guess it is best for classes to be independent. Why ?

What are Patterns

Some definitions

- Cohesion - How strongly related the responsibilities are
- It is a measure of how related all the attributes and behavior in a class are
- Can you guess it is best attributes and behavior in classes to be related. Why ?

What are Patterns

Some definitions

- Responsibilities - An obligation of a class
- Note a responsibility is really a behavior the other classes depend on
- Thus it must be a responsible behavior
- Thus many authors refer to behaviors as responsibilities as they are of little value otherwise. Why ?

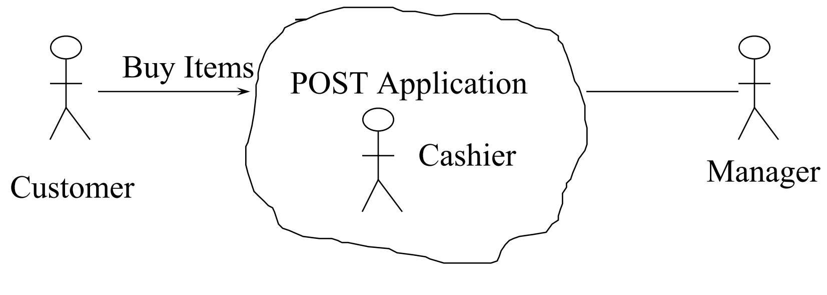
What are Patterns

- **GRASP**
  - **General Responsibility Assignment Software Patterns**

The Five GRASP Patterns

- There first five GRASP Patterns are
  - Expert
  - Creator
  - Low Coupling
  - High Cohesion
  - Controller

An Example



The GRASP Pattern Expert

- Any responsibility must be accomplished by the class that has or will have the data
- This means a class must populate all of its attributes
- It also must trigger the creation of all things that it contains

The GRASP Pattern Expert

- The conceptual diagram becomes important because it shows the attributes of each concept which is a potential class
- Also an expert should be responsible to make calculated values that are required by the class

The GRASP Pattern Expert

Computing calculated values:

- Sale knows how to total itself
- Sales Line Item knows how to compute the extended value of price times quantity

The GRASP Pattern Creator

- Each object-oriented language has a message called new that is sent to the class
- Thus the class must create its own objects
- In UML this message is called '***create'***
- It was felt the using ***'new'*** would show a bias to a particular object-oriented language

The GRASP Pattern Creator

Clarification:

- An expert is responsible to create its attributes
- But each attribute is itself a class
- This is accomplished by the expert sending a message to the class of the attribute
- The attribute’s class knows how to create itself

The GRASP Pattern Low Coupling

- Assign a responsibility so that the coupling remains low
- Recall coupling is how strongly one class has knowledge of another class
- The best example of this is not to have POST (buy-item use case) talking to every class
- POST (buy-item use case) should not have knowledge of any more classes than its has to

The GRASP Pattern High Cohesion

- Assign a responsibility so that all the responsibilities in a class are related
- Recall cohesion is how strongly related the responsibilities are
- Best example is to not use POST (buy-item use case) to create payment
- Sale has all the similar responsibilities that create its attributes

The GRASP Pattern Controller

- Every business system should have a controller
- A controller is class whose job it is to coordinate the system events
- It sees to it the messages are sent to the correct expert in the model
- The most common one is a Use Case controller which we will use

The GRASP Pattern Controller

- The reason to have a controller is to separate the business model called domain objects from the visual logic called view objects
- This is often called a MVC (Model View Controller) separation

The GRASP Pattern Controller

- Advantage - is that the changes to the model (domain) do not affect the GUI (view) objects
- Advantage - is that the changes to the GUI (view) do not affect the model (domain) objects
- It provides a buffer between the visual and the business logic

The GRASP Pattern Controller

- Since the decision to have a controller is decided, the real question is choosing which concept is to be the controller class
- This is relatively easy as the concept that processes the system events is the logical choice
- In the Buy Item use case, it is the POST

The GRASP Pattern Controller

- Other choices are Store and Cashier
- POST was chosen over store as it is conceptually closer to the system event
- POST was chosen over Cashier as cashier is really an actor the runs the POST
- Thus POST is the logical choice

Summary

- There a five initial GRASP Patterns that complement each other
- MVC (Model View Controller) separation builds systems that are maintainable
- This is still an art as it is not yet a science
- A good idea is to practice by creating this example from scratch

Part II.4

System Design

Designing the System

- Collecting Artifacts
- Using Patterns

Collecting Artifacts

- One needs the following deliverables from previous steps
- They are:
  - Use Case
  - Conceptual Diagram
  - System Sequence Diagram
  - Contracts

Using Patterns

- The following patterns are used:
- First a controller must be chosen
- Second all creator objects must be identified
- Third all experts must be identified
- Fourth adjustments are made to reduce coupling

Example

- It is easiest to build a sequence diagram
- This is because the System Sequence Diagram is already completed
- We will break this up into five steps

Example

- Step One - Choosing a controller
- Since the use case works with the POST the POST makes the most sense
- It serves as a bridge between the GUI and the Model
- Then change the class System to POST in the System Sequence Diagram

Example

- Step Two - Look for containment's in the Conceptual diagram that suggests that the parent will create the child
- Thus the creator pattern will be used
- The containment's are
  – POST contains (creates) Sale
  – Sale contains (creates) SaleLineItem
  – Sale contains (creates) Payment

Example

- Step Three - Look for attributes that need to modified or calculated by their experts
- Thus the expert pattern will be used
- The experts are
  - Sale knows how to set the attribute isComplete to true - becomeComplete ( )
  - Sale knows how to compute the total - total ( )
  - SaleLineItem knows how to compute the subtotal - subtotal ( )

Example

- Step Four - Look for ways to lower coupling thus the Low Coupling pattern will be used
- The concerns are
  - To make SalesLineItem one must send a message to Sale to contain it and a Message to to SalesLineItem to create it
  - Better to send one message to Sale and have it create it - makeLineItem (productSpecification, quantity)

Example

- Step Four Continued - Look for ways to lower coupling
- The concerns are
  - To make Payment one must send a message to Sale to contain it and a Message to Payment to create it
  - Better to send one message to Sale and have it create it - makePayment (amount)

Example

- Step Five - Look for problems
- A concern is
  - To make SalesLineItem POST must send a message to Sale to create it
  - But it needs productSpecification which POST does not know
  - Thus POST must send a message to the expert ProductCatalog to get the information - getProductSpecification (upc)

Example

- Step Five Continued - Look for problems
- Another concern is
  - To place the sale in the sales ledger someone must do it
  - The expert (who has the information) is POST
  - POST must send a message to SalesLedger to which will know how to add it to itself - addSale(sale)

Summary

- Patterns help us with the design
- All previous work is used
  - Use Case
  - Conceptual Diagram
  - System Sequence Diagram
  - Contracts

Part II.5

UML and OOAD Summarized

See: Sub-Topic 2 Presentation on "UML Review"

Section Outline

- Introduction
- Software Development
- Object-Oriented Analysis
- Object-Oriented Design
- Summary

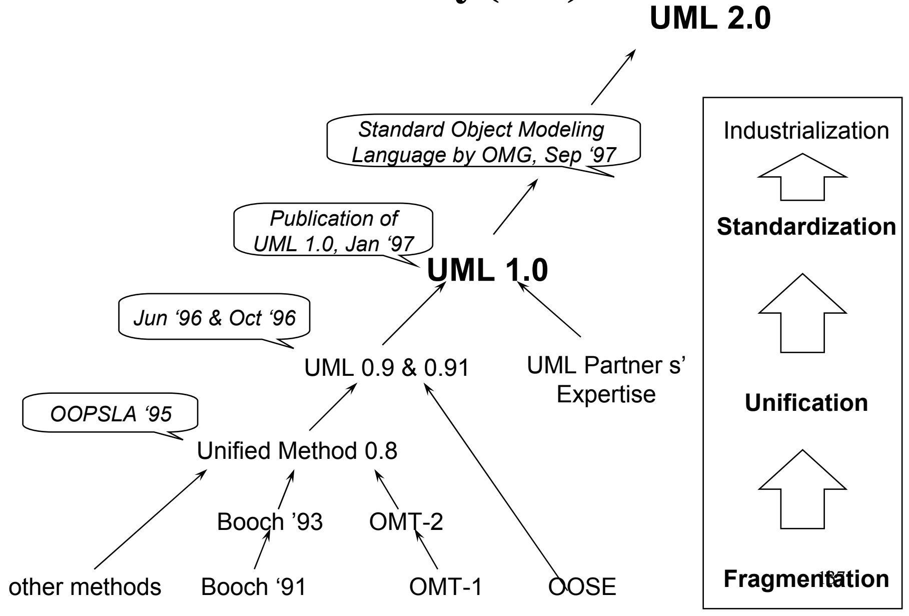
Introduction

- UML?
  - The successor to the wave of OOA&D methods that appeared in the late ’80s and early ’90s
  - Unification of the methods of Booch, Rumbaugh, and Jacobson
  - The standard of OMG (Object Management Group)

History(1/2)

- UML motivation
  - To create a set of semantics and notation that can adequately address all scales of architectural complexity across all domains
- Each of OO methods were recognized as having certain strengths
- Base methods
  - Booch, Rumbaugh(OMT), Jacobson(OOSE)

History(2/2)



Scope of the UML(1/2)

- Language for specifying constructing, visualizing, and documenting the artifacts of a software-intensive system
- Fusing the concepts of Booch, OMT, and OOSE
- Envelop of what can be done with existing methods
- Standard object modeling language adopted by OMG, Sep., 1997

Scope of the UML(2/2)

- Object-Oriented Analysis
  - Use Case diagram
  - Object interaction diagram
  - Class diagram
  - State diagram
  - Activity diagram
- Object-Oriented Design
  - Process diagram
  - Architecture diagram
  - Deployment diagram

Comparing to Others(1/2)

- Not a radical departure from Booch, OMT, and OOSE
- Legitimate successor to all three methods
- More expressive yet cleaner and more uniform
- Value in moving to the UML
  - It will allow projects to model things they could not have done before
  - It removes the unnecessary differences in notation and terminology

Comparing to Others(2/2)

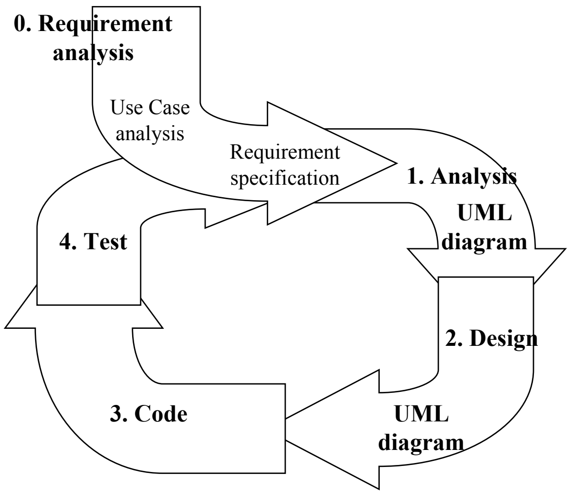
Class Diagram Terminology

UML	Class	Association	Generalization	Aggregation
Booch	Class	Uses	Inherits	Containing
Coad	Class & Object	Instance Connection	Gen-Spec	Part-Whole
Jacobson	Object	Acquaintance Association	Inherits	Consists of
Odell	Object type	Relationship	Subtype	Composition
OMT	Class	Association	Generalization	Aggregation
Shlaer/Mellor	Object	Relationship	Subtype	N/A

Section Outline

- Introduction
- Software Development
- Object-Oriented Analysis
- Object-Oriented Design
- Summary

Phases of a Development Cycle



Phase of Development

- Requirement Analysis
- Analysis
- Design
- Code
- Test

Section Outline

- Introduction
- Software Development
- <u>Object-Oriented Analysis</u>
- Object-Oriented Design
- Summary

OO Analysis

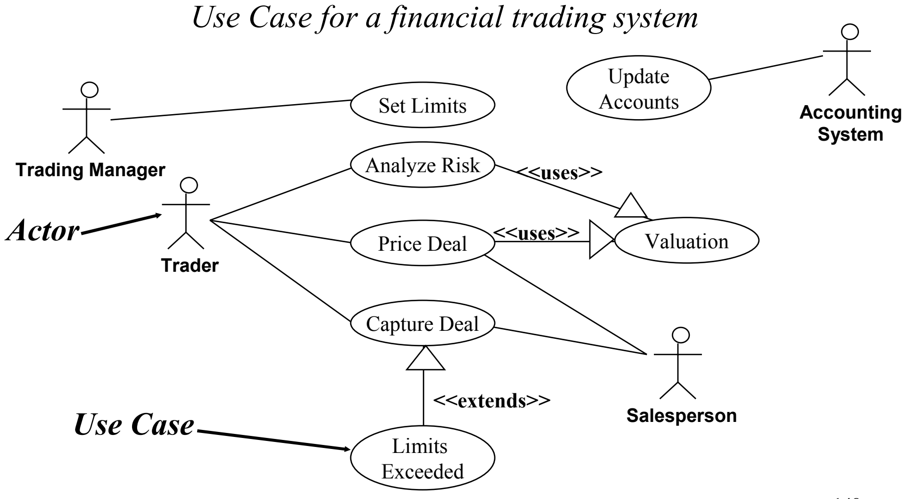
Phase 1. Use Case Modeling
Phase 2. Class Finding & Refinement
Phase 3. Object Finding
Phase 4. Class Relationship
Phase 5. Class Specification
Phase 6. Analysis Refinement

OOA Phase 1. Use Case Modeling

- Introduced by Jacobson(1994)
- Use Case modeling by user requirement
- Relationship among actors and *Use Cases*
- *Actors* carry out *Use Cases*
- *Use Case* is a typical interaction between a user and a computer system
  - <<extends>> relationship : similar but do a bit more
  - <<uses>> relationship : a chunk of behavior similar across more than one use case

Use Case Diagram

Use Case for a financial trading system



OOA Phase 2. Class Finding & Refinement

- For each *Use Case*, finding *classes*
  - Class Finding
    - *Class Diagram*
  - Class Refinement
    - Remove redundant
    - Name same, semantics different

Class

- The name of a class has scope within the package in which it is declared and the name must be unique (among class names) within its package
- References
  - Show a reference to a class defined in another package
  - *Package-name::class-name*

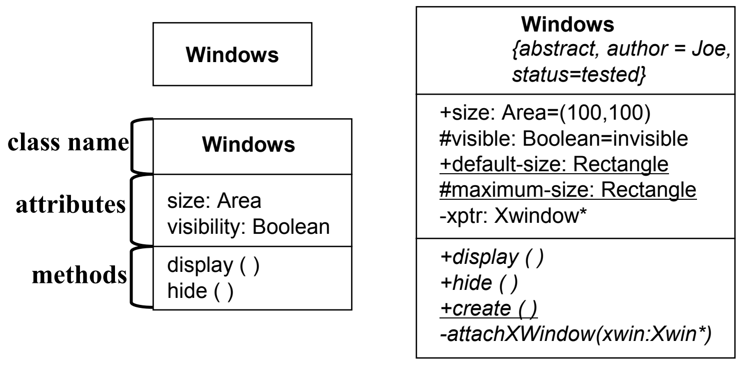
Class Diagram

- Showing the static structure of the system
- A graph of modeling elements shown on a two-dimensional surface
- A collection of (static) declarative model elements, such as classes, types, and their relationships, connected as a graph to each other and to their contents

Type

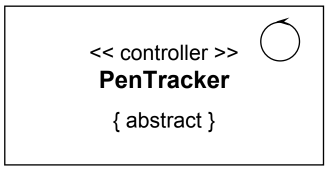
- A type is descriptor for objects with abstract state, concrete external operation specification, and no operation implementations
- Classes implement types
- Shown as a stereotype of a class symbol with the stereotype <<type>>
- May contain lists of abstract attributes and of operations
- May contain a context and specifications of its operations accordingly

Examples of Class Diagram



Name Compartment

- Displays the name of the class and other properties in up to 3 sections
- An optional *stereotype* keyword, the name, a property list



OOA Phase 3. Object Finding

- For each class, finding objects, and making object interaction diagram
  - *Sequence Diagram*
  - *Collaboration Diagram*
- Finding messages within objects

Object Diagram

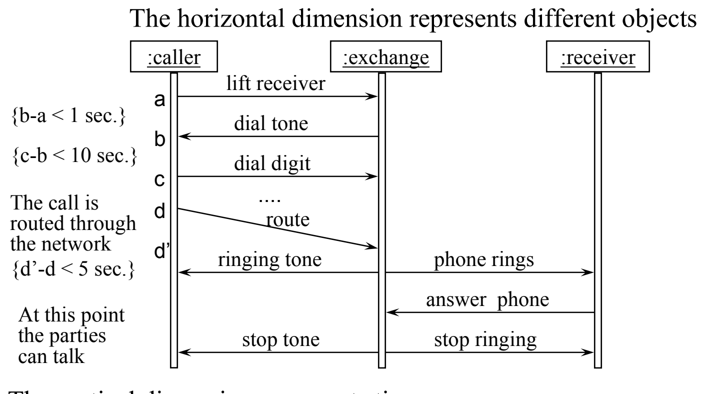
- A graph of instances
- Static object diagram is an instance of a class diagram
- Dynamic object diagram shows the detailed state of a system over some period of time
- Class diagrams can contain objects, so a class diagram with objects and no classes is an "object diagram"

Sequence Diagram(1/2)

- Showing an interaction arranged in time sequence
- Showing the explicit sequence of messages
- Better for real-time specifications and for complex scenarios

Sequence Diagram(2/2)

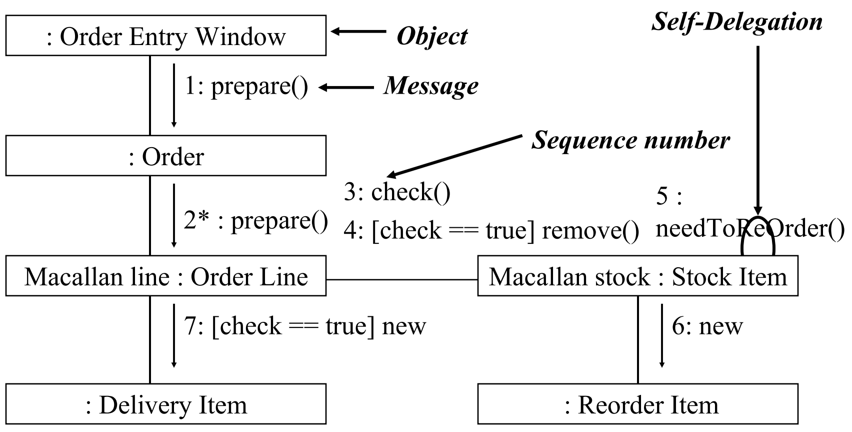
Sequence diagram for concurrent objects



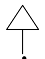
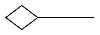
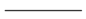
Collaboration Diagram(1/2)

- Showing the relationships among objects
- Better for understanding all of the effects on a given object and for procedural design
- Showing an interaction organized around the objects in the interaction and their links to each other
- Not showing time as a separate dimension

Collaboration Diagram(2/2)

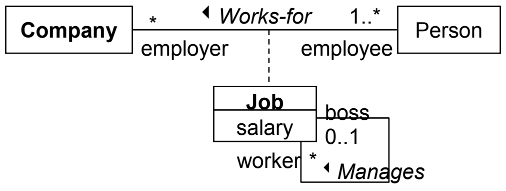


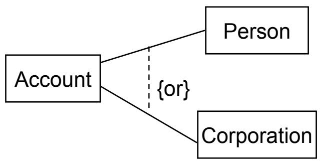
OOA Phase 4. Class Relationship

- Finding class relationships from *Object Diagrams*
- Is-A 
- Aggregation 
- Link 
  - Relationship which is not Is-A, nor Aggregation
  - Between classes that exchange messages

Association Relationship

- Association relationship is drawn as a solid path connecting two class symbols
- OR-association
  - Only one of several potential associations may be instantiated at one time for any single object



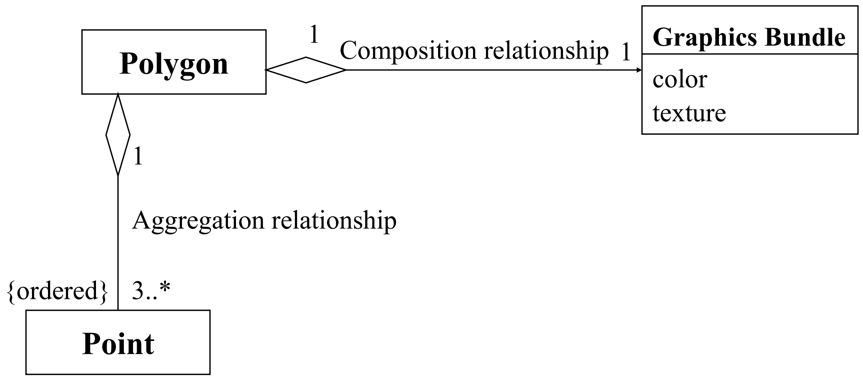


Aggregation & Composition

- Aggregation
  - Part of relationship
- Composition Relationship
  - Form of aggregation with strong ownership and coincident lifetime of part with the whole
  - The multiplicity of the aggregate end may not exceed one

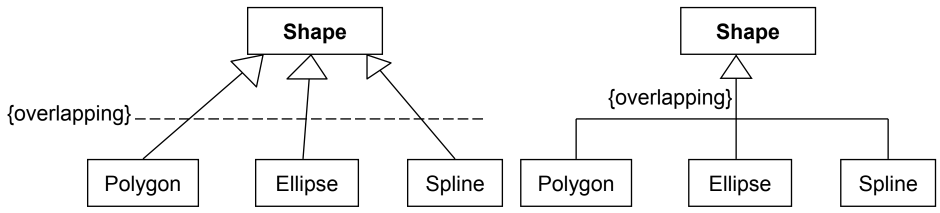
Example of Aggregation and Composition

Aggregation and Composition



Generalization

- Taxonomic relationship between a more general element and a more specific element



OOA Phase 5. Class Specification

- Attribute
- Behavior
  - By object interaction diagrams
  - Operation

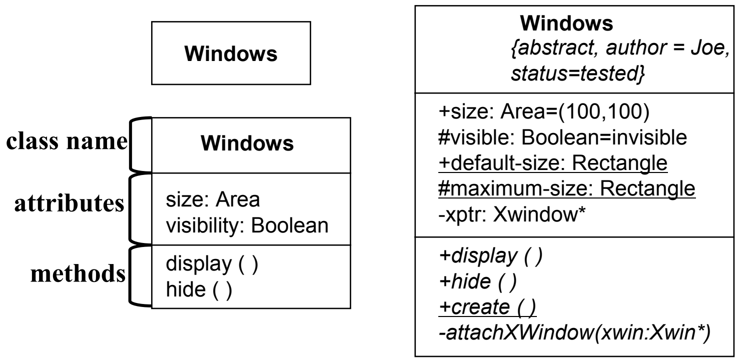
Attribute

- A text string that can be parsed into the various properties of an attribute model element
- The default syntax
  - *Visibility name:type-expression=initial-value{property-string}*
  - Visibility
    - + public
    - # protected
    - - private
  - A class-scope attribute is shown by underlining the entire string

Operation

- A text string that can be parsed into the various properties of an operation model element
- *Visibility name* (parameter-list): return-*type-expression=initial-value{property-string}*
  - Visibility
    - + public
    - # protected
    - - private
  - Parameter-list
    - *name*: : *type-expression = default-value*

Examples of Attributes and Operations



OOA Phase 6. Analysis Refinement

- Review all processes of analysis

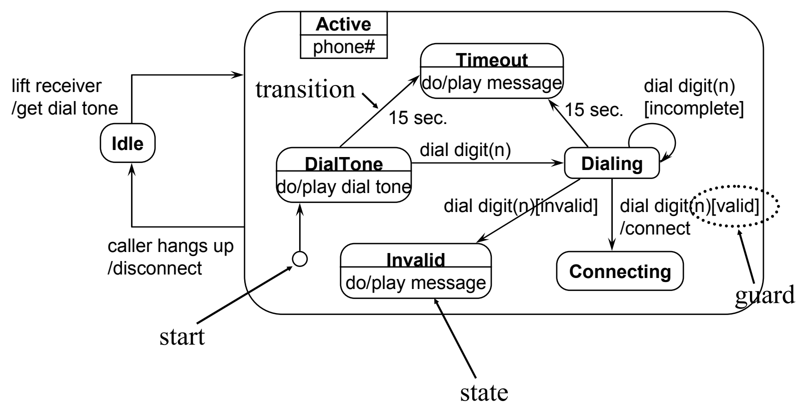
Other UML Diagrams

- By user needs
- State Diagram
- Activity Diagram

State Diagram(1/2)

- A familiar technique to describe the behavior of a system
- Describe all the possible states a particular object can get into and how the object’s state changes as a result of events
- Drawn for a single class to show the lifetime behavior of a single object

State Diagram(2/2)

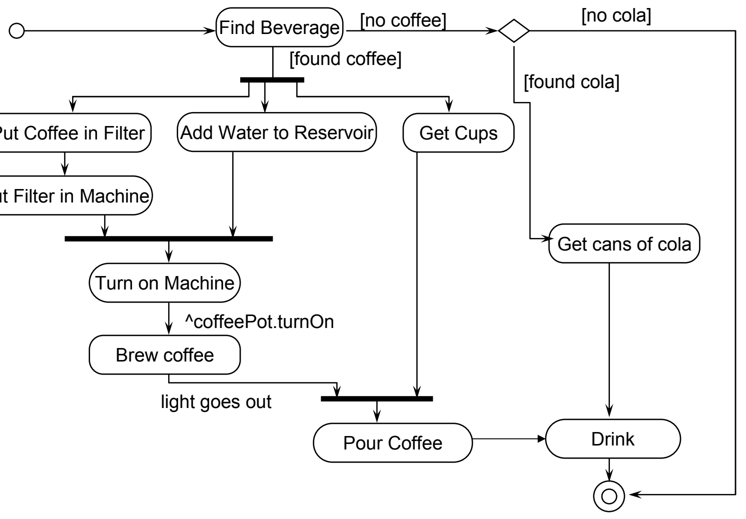


transition : event[guard(logical condition)]/action

Activity Diagram(1/2)

- A special case of a state diagram in which all of the states are action states and in which all of the transitions are triggered by completion of the actions in the source states
- Use activity diagrams in situations where all or most of the events represent the completion of internally-generated actions
- Use ordinary state diagrams in situations where asynchronous events occur

Activity Diagram(2/2)



Section Outline

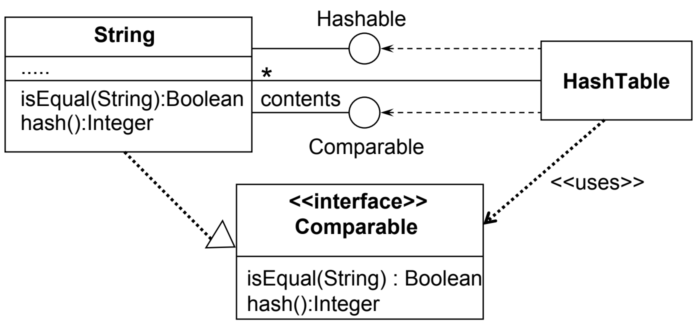
- Introduction
- Software Development
- Object-Oriented Analysis
- Object-Oriented Design
- Summary

OO Design

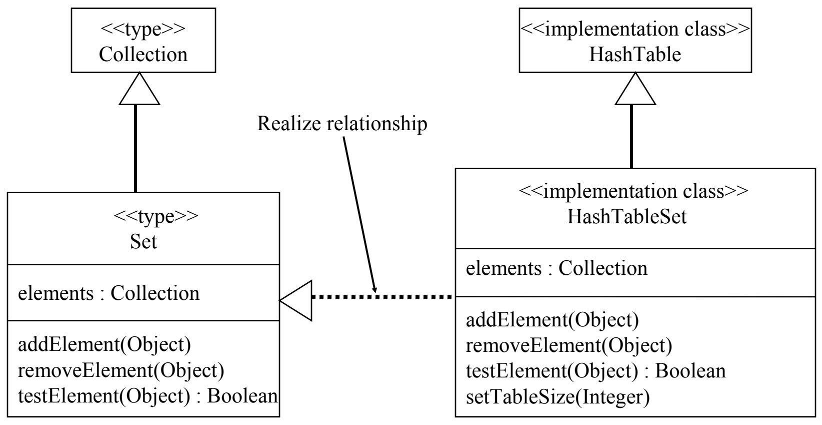
- Add implementation classes (DLL, …)
- Process
  - Consider the performance
- Deployment Diagram
  - Architecture diagram
  - System placement
- Component Diagram
  - Interface

Interface

- The use of type to describe the externally-visible behavior of a class component, or other entity



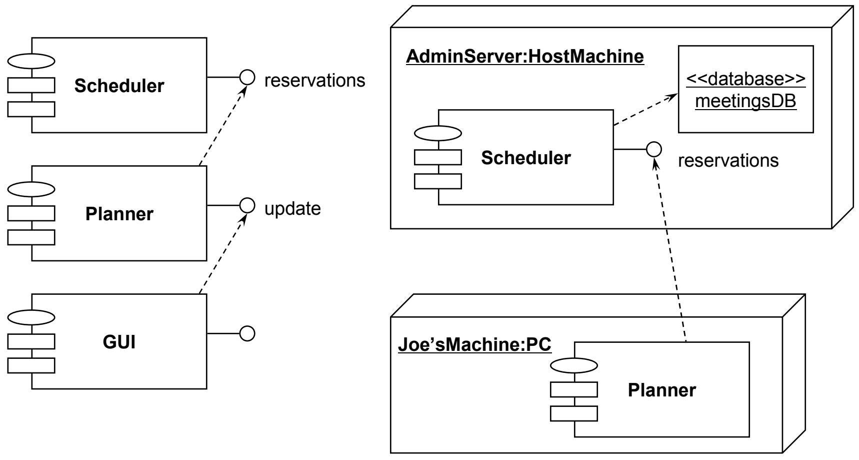
Types and Implementation Class



Implementation Diagram

- Show aspects of implementation, including source code structure and run-time implementation structure
- Component Diagrams
  - Show the structure of the code itself
- Deployment Diagrams
  - Show the structure of the runtime system

Deployment Diagram

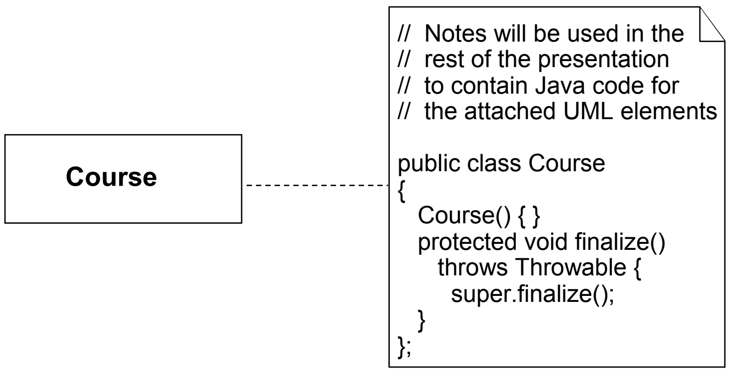


Summary

- UML is a standard for OOA&D
- Software Development
  - Requirement Analysis
  - **Object-Oriented Analysis**
  - **Object-Oriented Design**
  - Code
  - Test

Part II.6
UML to Java Mapping

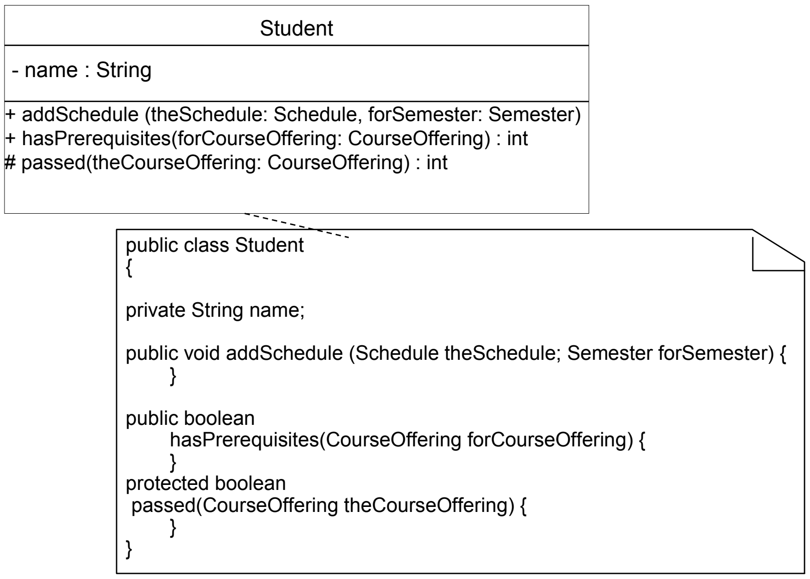
Mapping Representation: Notes



```
//  Notes will be used in the
//  rest of the presentation
//  to contain Java code for
//  the attached UML elements

public class Course
{
   Course() { }
   protected void finalize()
      throws Throwable {
        super.finalize();
   }
};
```

Visibility for Attributes and Operations


Student
- name : String
+ addSchedule (theSchedule: Schedule, forSemester: Semester)
+ hasPrerequisites(forCourseOffering: CourseOffering) : int
# passed(theCourseOffering: CourseOffering) : int

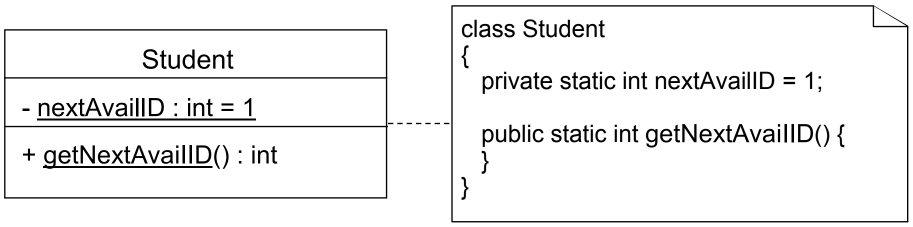
```
public class Student
{

private String name;

public void addSchedule (Schedule theSchedule; Semester forSemester) {
        }

public boolean
        hasPrerequisites(CourseOffering forCourseOffering) {
        }
protected boolean
 passed(CourseOffering theCourseOffering) {
        }
}
```

Class Scope Attributes and Operations

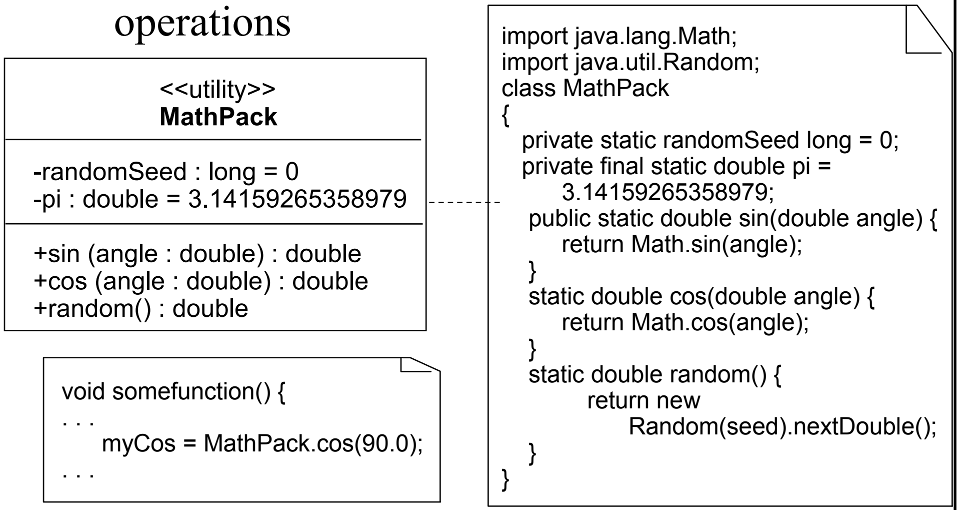


```
class Student
{
   private static int nextAvailID = 1;

   public static int getNextAvailID() {
   }
}
```

Utility Class

- A grouping of global attributes and operations



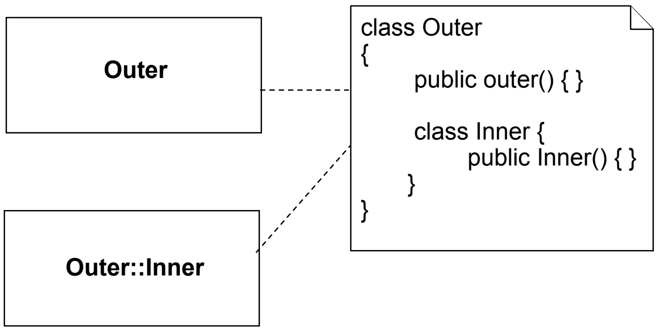
<<utility>>
MathPack
-randomSeed : long = 0
-pi : double = 3.14159265358979
+sin (angle : double) : double
+cos (angle : double) : double
+random() : double

```
void somefunction() {
. . .
    myCos = MathPack.cos(90.0);
. . .
```

```
import java.lang.Math;
import java.util.Random;
class MathPack
{
  private static randomSeed long = 0;
  private final static double pi =
       3.14159265358979;
   public static double sin(double angle) {
       return Math.sin(angle);
   }
   static double cos(double angle) {
       return Math.cos(angle);
   }
   static double random() {
           return new
                 Random(seed).nextDouble();
   }
}
```

Nested Class

- Hide a class that is relevant only for implementation

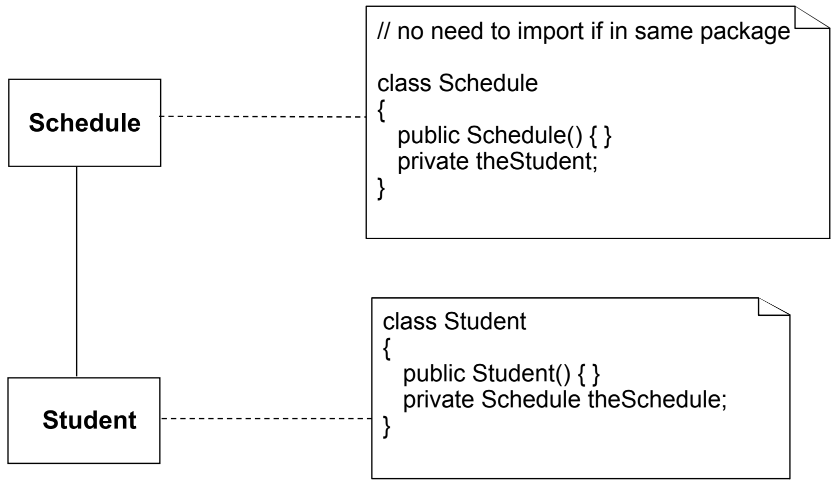


```
class Outer
{
      public outer() { }

      class Inner {
             public Inner() { }
      }
}
```

Associations

- Bi-directional associations



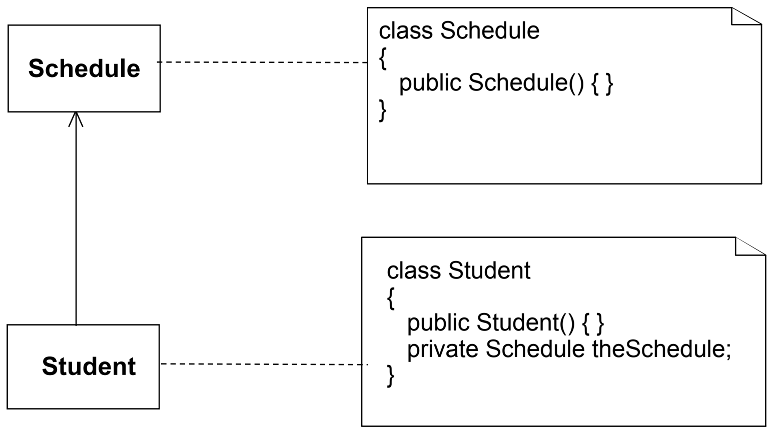
```
// no need to import if in same package

class Schedule
{
   public Schedule() { }
   private theStudent;
}
```

```
class Student
{
   public Student() { }
   private Schedule theSchedule;
}
```

Association Navigability

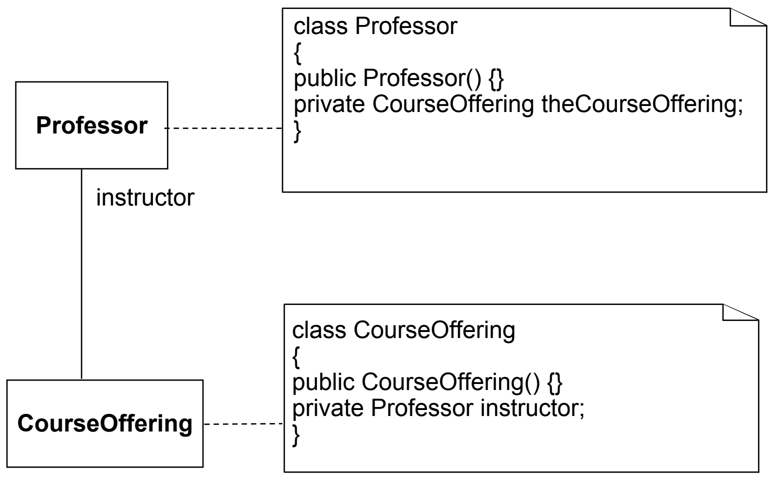
- Uni-directional associations



```
class Schedule
{
   public Schedule() { }
}
```

```
class Student
{
   public Student() { }
   private Schedule theSchedule;
}
```

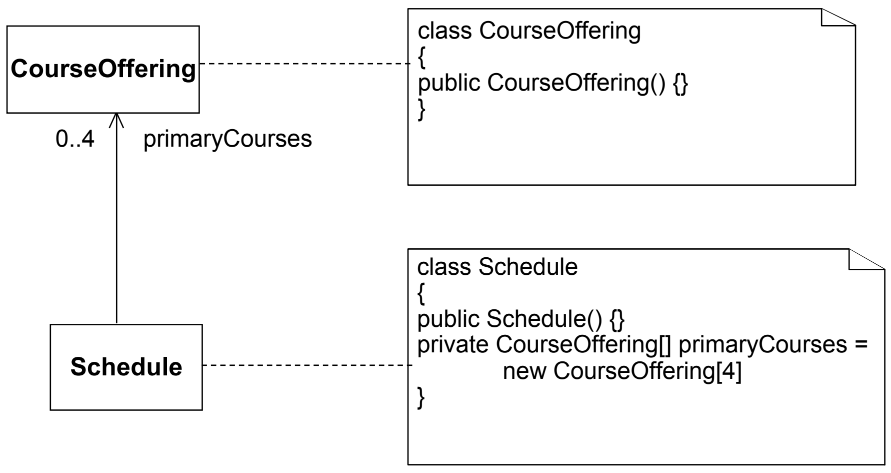
Association Roles



```
class Professor
{
public Professor() {}
private CourseOffering theCourseOffering;
}
```

```
class CourseOffering
{
public CourseOffering() {}
private Professor instructor;
}
```

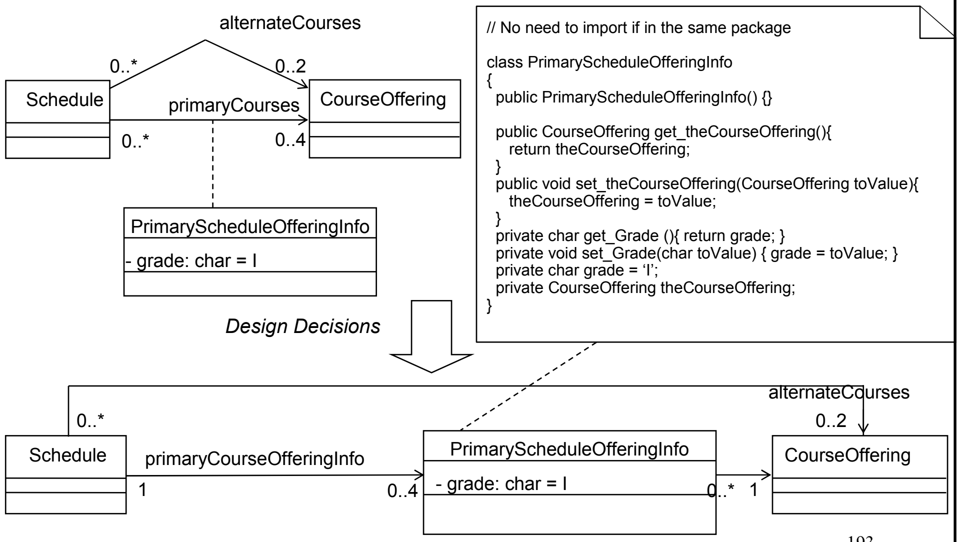
Association Multiplicity



```
class CourseOffering
{
public CourseOffering() {}
}
```

```
class Schedule
{
public Schedule() {}
private CourseOffering[] primaryCourses =
            new CourseOffering[4]
}
```

Association Class

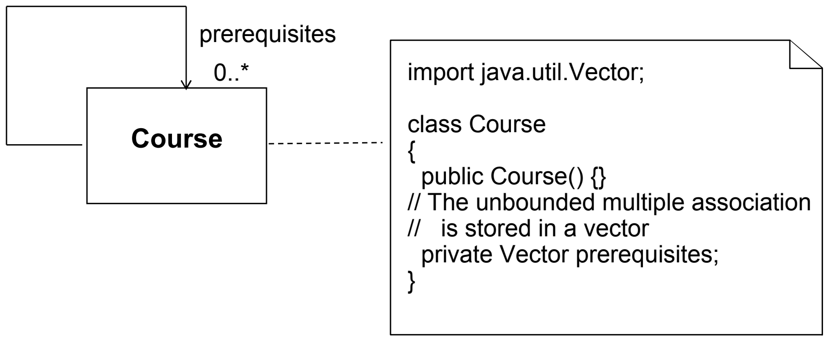


```
// No need to import if in the same package

class PrimaryScheduleOfferingInfo
{
  public PrimaryScheduleOfferingInfo() {}

  public CourseOffering get_theCourseOffering(){
     return theCourseOffering;
  }
  public void set_theCourseOffering(CourseOffering toValue){
     theCourseOffering = toValue;
  }
  private char get_Grade (){ return grade; }
  private void set_Grade(char toValue) { grade = toValue; }
  private char grade = 'I';
  private CourseOffering theCourseOffering;
}
```

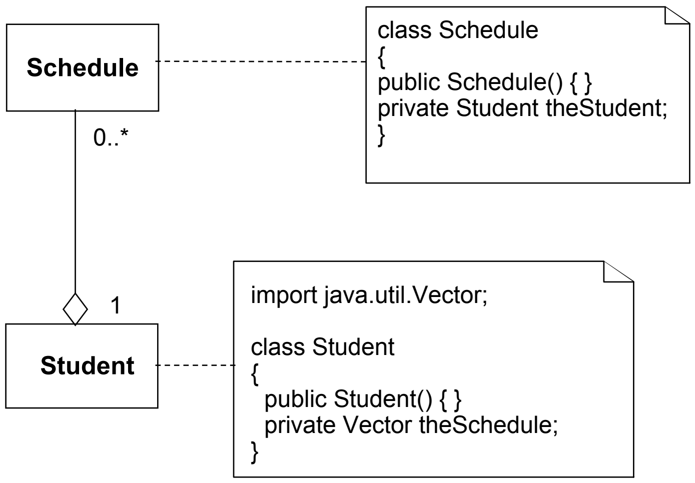
Reflexive Associations



```
import java.util.Vector;

class Course
{
  public Course() {}
// The unbounded multiple association
//   is stored in a vector
  private Vector prerequisites;
}
```

Aggregation

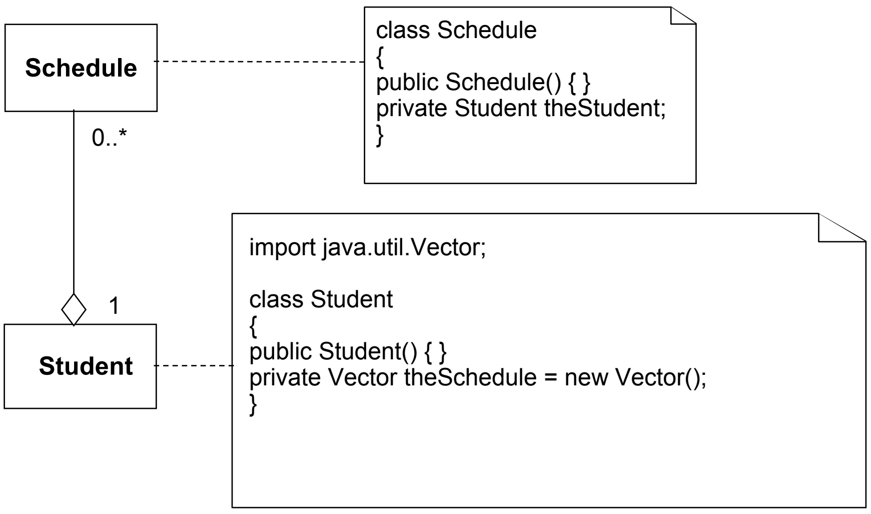


```
class Schedule
{
public Schedule() { }
private Student theStudent;
}
```

```
import java.util.Vector;

class Student
{
  public Student() { }
  private Vector theSchedule;
}
```

Composition

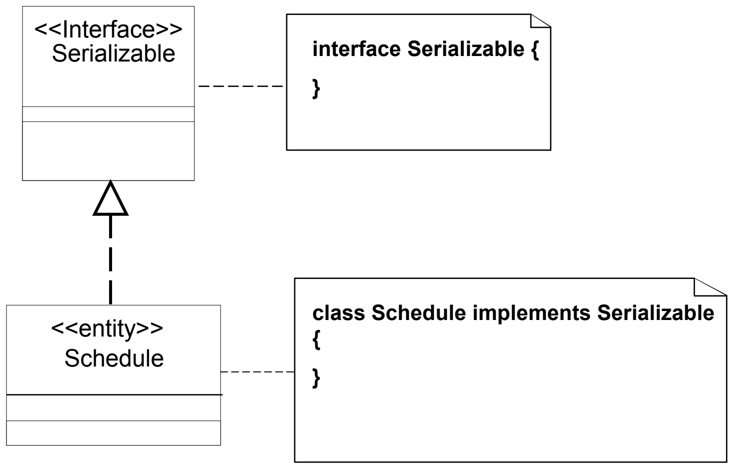


```
class Schedule
{
public Schedule() { }
private Student theStudent;
}
```

```
import java.util.Vector;

class Student
{
public Student() { }
private Vector theSchedule = new Vector();
}
```

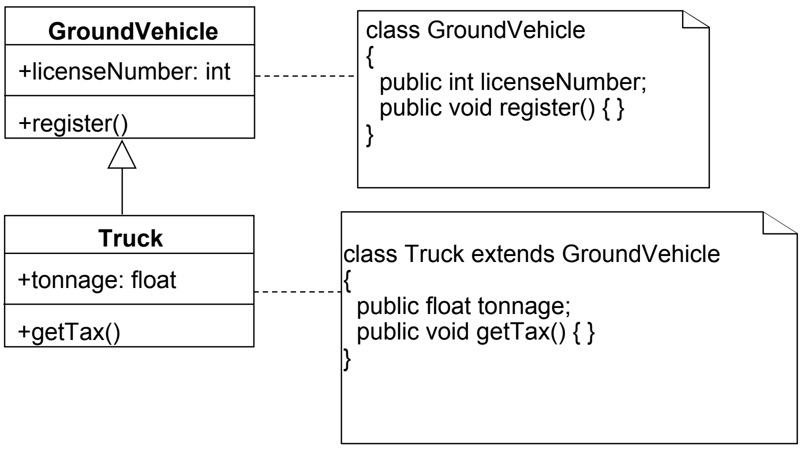
Interfaces and Realizes Relationships



```
interface Serializable {
}
```

```
class Schedule implements Serializable
{
}
```

Generalization

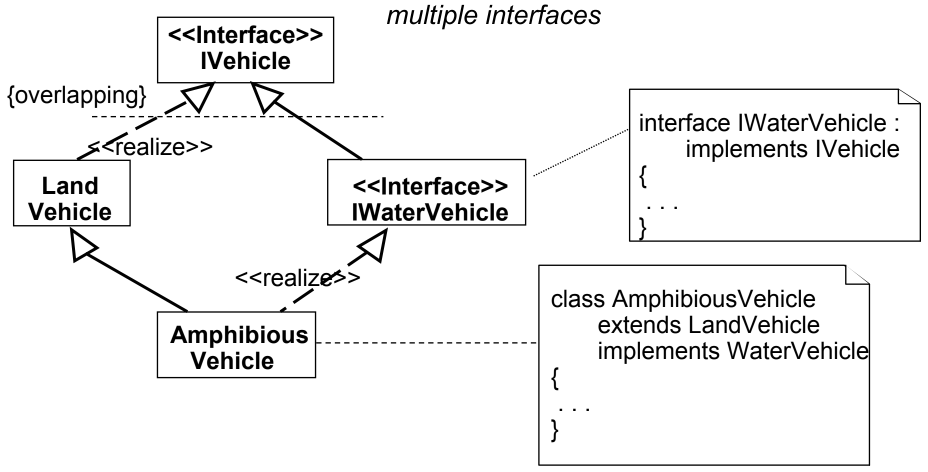


```
class GroundVehicle
{
  public int licenseNumber;
  public void register() { }
}
```

```
class Truck extends GroundVehicle
{
  public float tonnage;
  public void getTax() { }
}
```

Multiple Inheritance

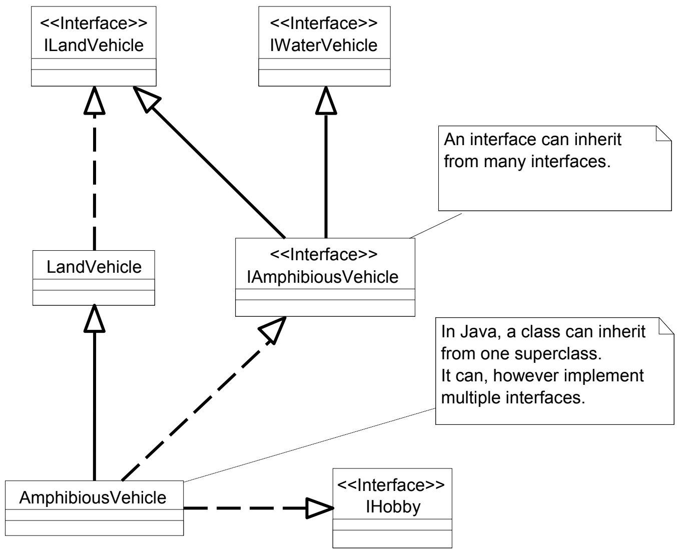
In Java, a class can only inherit from one superclass. It can, however implement multiple interfaces



```
interface IWaterVehicle :
      implements IVehicle
{
 . . .
}
```

```
class AmphibiousVehicle
      extends LandVehicle
      implements WaterVehicle
{
 . . .
}
```

Multiple Inheritance (contd)

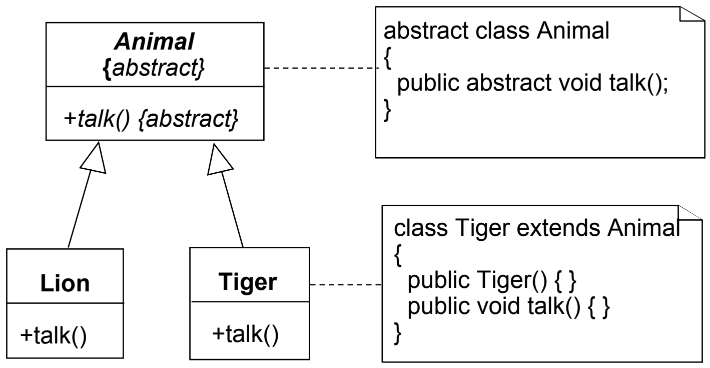


<<Interface>> ILandVehicle
<<Interface>> IWaterVehicle
LandVehicle
<<Interface>> IAmphibiousVehicle
AmphibiousVehicle
<<Interface>> IHobby

An interface can inherit from many interfaces.

In Java, a class can inherit from one superclass. It can, however implement multiple interfaces.

Abstract Class

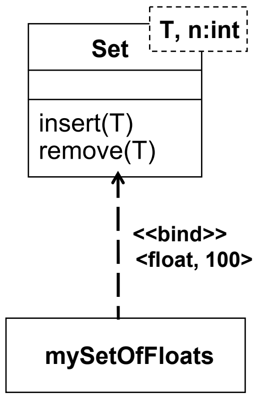


```
abstract class Animal
{
  public abstract void talk();
}
```

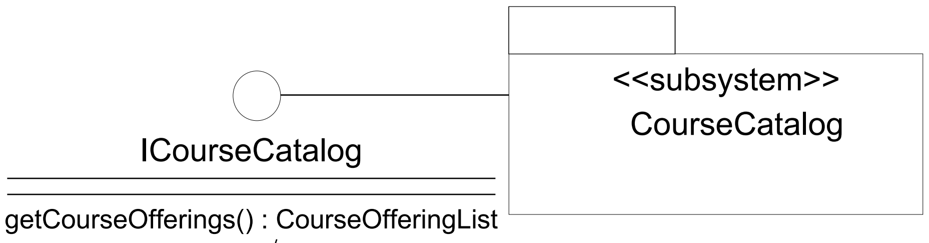
```
class Tiger extends Animal
{
  public Tiger() { }
  public void talk() { }
}
```

Parameterized Class

Java does not (yet) support parameterized classes



Subsystems



```
package CourseCatalog;


public interface ICourseCatalog {
  public CourseOfferingList
getCourseOfferings();
}
```

Part II.7

Sample OCL Problem

Sample Problem

- Problem Statement (Tower of Hanoi)

  There are 3 needles and a tower of 5 disks on the first one, with the smaller on the top and the bigger on the bottom. The purpose of the puzzle is to move the whole tower from the first needle to the second, by moving only one disk every time and by observing not to put a bigger disk atop of a smaller one.

- Typical OCL Deliverables - annotate the appropriate diagrams with:
  1. Invariants
  2. Pre-conditions
  3. Post-conditions
  4. Guards

Scenarios 1 and 2

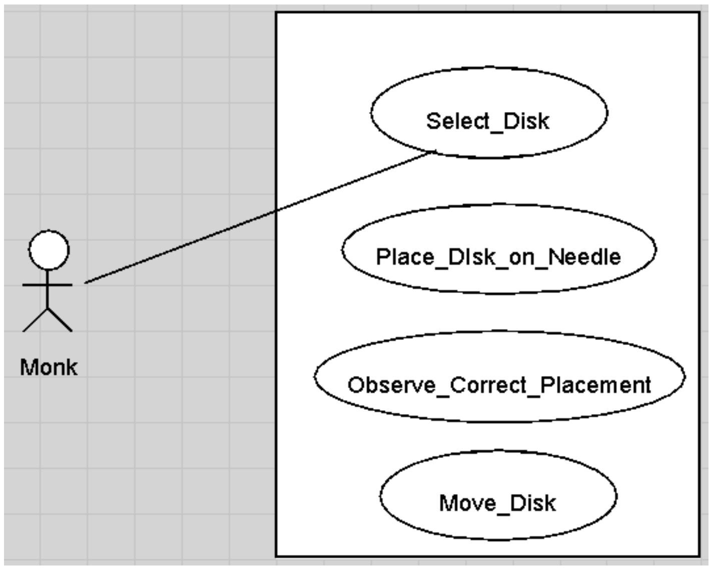
Normal Scenario

1. Monk selects top most disk
2. Places disk on third needle
3. Selects next disk
4. Places disk on second needle
5. Moves disk on third needle to second needle
6. Selects next ring
7. Places ring on third needle
8. Continues until done

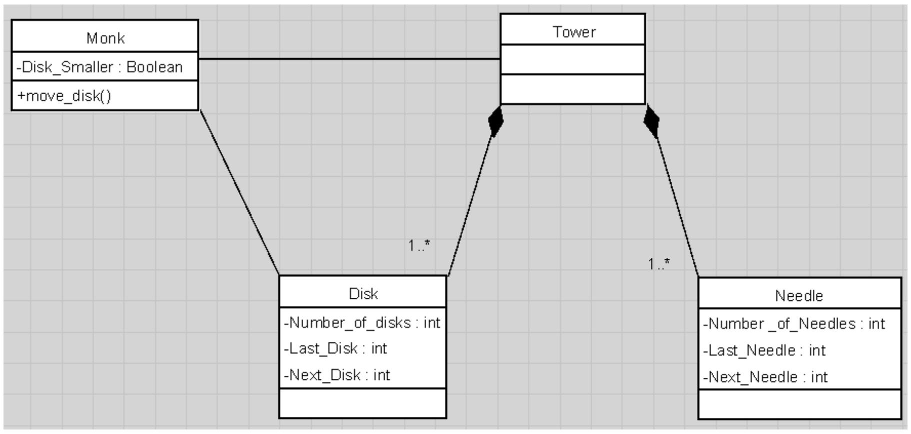
Abnormal Scenario

1. Monk selects top most disk
2. Places disk on third needle
3. Selects next disk
4. Places disk on third needle
5. Larger disk on smaller disk

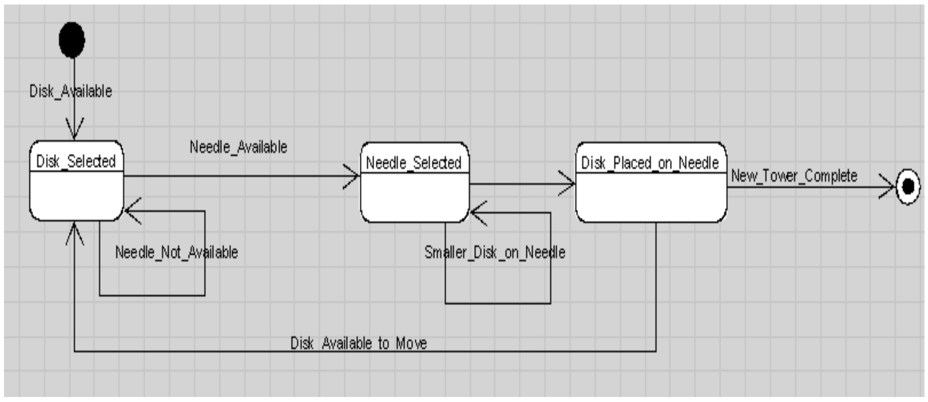
Use Case Diagram



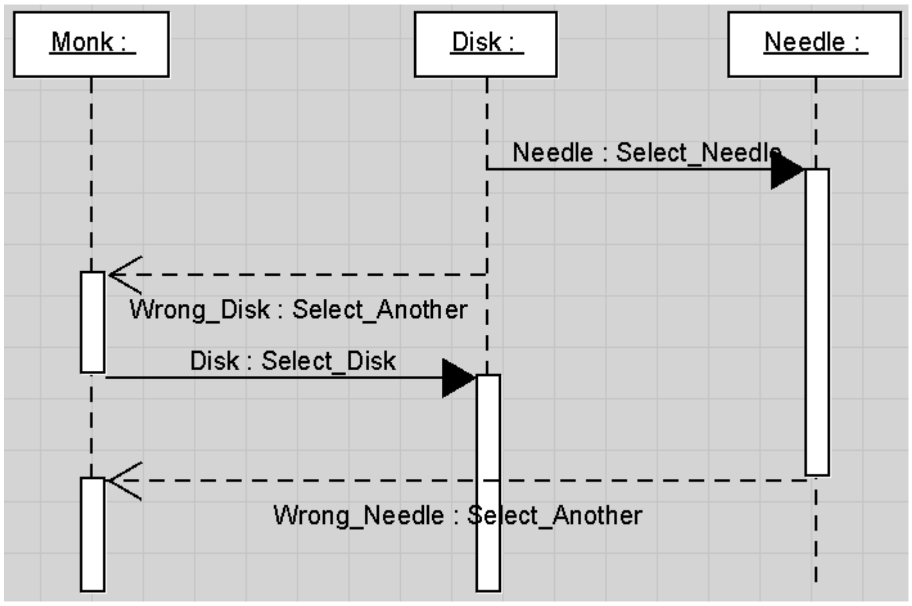
Class Diagram



State Transition Diagram



Sequence Diagram



Problem Analysis

- Numeric Values Available
  - 3 needles
  - 5 disks
- Rules
  - Move tower from disk 1 to 2
  - 1 disk per move
  - Never put a bigger disk on a smaller disk

Object Constraint Language (OCL)

- Invariants - ALWAYS coupled to classes
  - Number of Needles
  - Number of Disks
- Pre-Conditions
- Post Conditions
- Guards - only on state transition diagrams

Scenarios 1 and 2

(NO OCL goes here!)

Normal Scenario

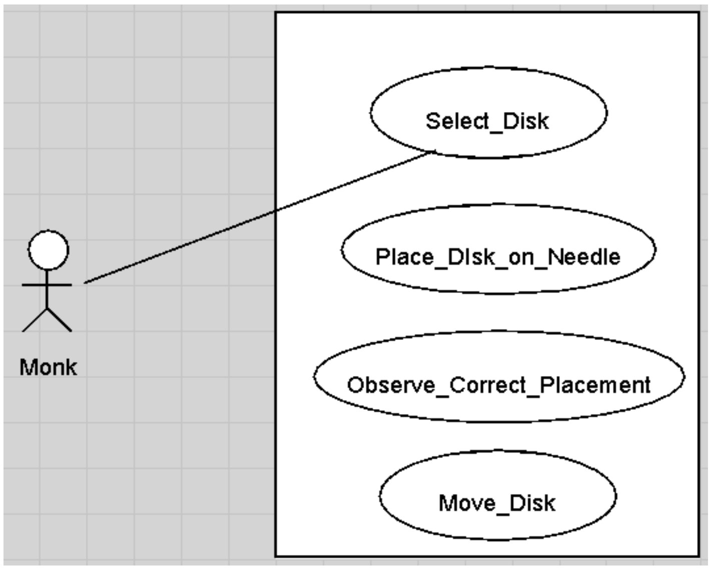
1. Monk selects top most disk
2. Places disk on third needle
3. Selects next disk
4. Places disk on second needle
5. Moves disk on third needle to second needle
6. Selects next ring
7. Places ring on third needle
8. Continues until done

Abnormal Scenario

1. Monk selects top most disk
2. Places disk on third needle
3. Selects next disk
4. Places disk on third needle
5. Larger disk on smaller disk

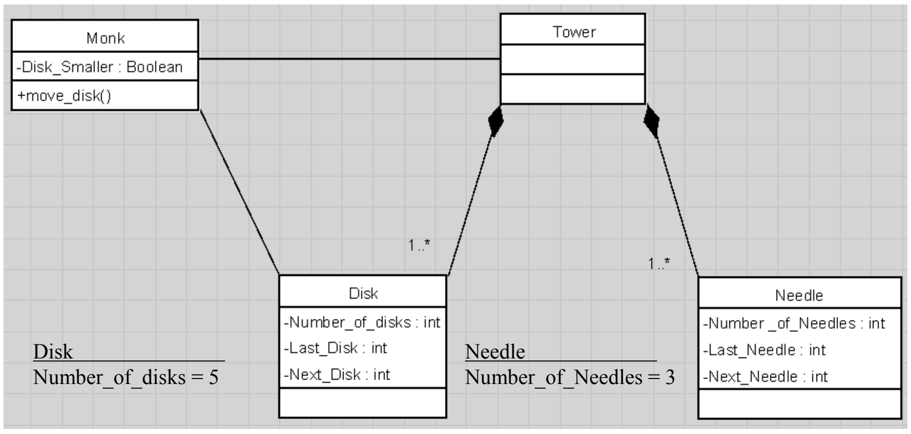
Use Case Diagram

(NO OCL goes here!)



Class Diagram

(Invariants go here!)

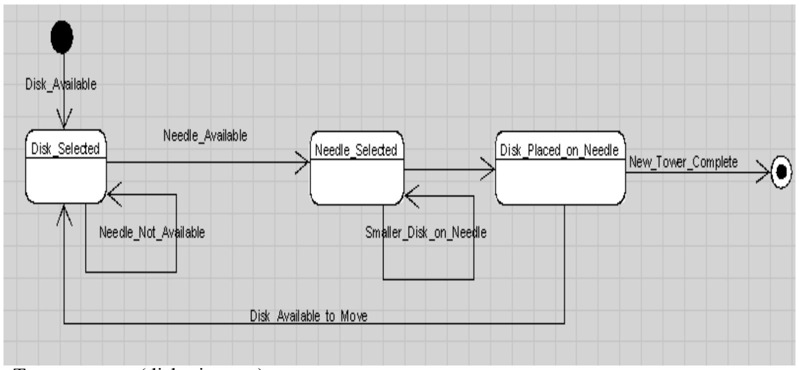


<u>Disk</u>
Number_of_disks = 5

<u>Needle</u>
Number_of_Needles = 3

State Transition Diagram for Tower

(Guards go here!)

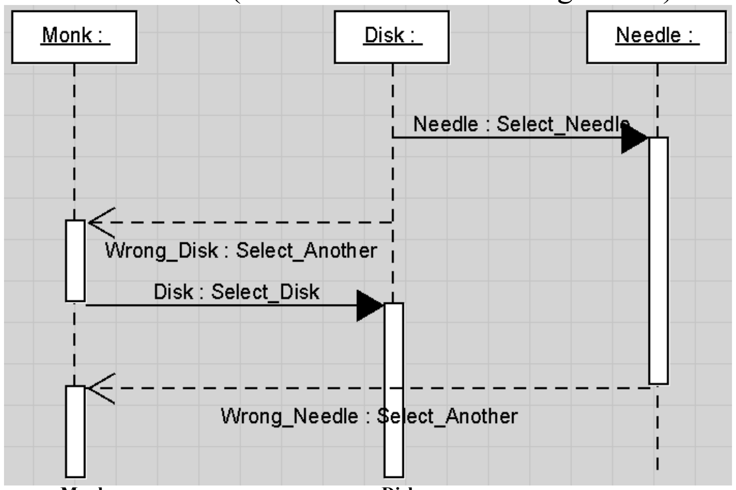


Note: All state names and events are transformed into Bool attributes

<u>Tower :: move(disk : integer)</u>
Pre: not Disk_Selected and not Needle_Selected and not Disk_Placed_on_Needle
Post: New_Tower_Complete = True
<u>Tower :: Needle_Available()</u>
Pre: Disk_Selected = True
Post: Needle_Selected = False
<u>Tower :: Needle_Not_Available</u>
Pre: Disk_Selected = True
Post: Disk_Selected = False

Sequence Diagram

(Pre and Post Conditions go here!)



Note: The context of pre- and post conditions is always an operation or a method.
OCL text pg. 23

Monk :
Pre: None
Post: Select_Disk = True

Disk :
Pre: Select_Disk = True
Post: Select_Needle = True
Or
Select_Another = True

Needle :
Pre: Select_Needle = True
Post: None
Or
Select_Another = True

Part III

Introduction to Design and Architectural Patterns

See: Sub-Topic 3 Presentation on "Introduction to Design and Architectural Patterns"

Part IV

Micro / Macro Architecture

See: Sub-Topic 4 Presentation on "Micro / Macro Architecture"

Part V

Design and Architectural Patterns

See: Sub-Topic 5 Presentation on "Design and Architectural Patterns"

Part VI

Conclusion

Course Assignments

- Individual Assignments
  - Problems and reports based on case studies or exercises
- Project-Related Assignments
  - All assignments (other than the individual assessments) will correspond to milestones in the team project.
  - As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consists inter-related deliverables which are due on a (bi-) weekly basis.
  - There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
  - A sample project description and additional details will be available under handouts on the course Web site.

Course Project

- Project Logistics
  - Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
  - Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may not be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

Sample Project Methodology Very eXtreme Programming (VXP)

- After teams formed, 1/2 week to Project Concept
- 1/2 week to Revised Project Concept
- 2 to 3 iterations
- For each iteration:
  - 1/2 week to plan
  - 1 week to iteration report and demo

Sample Project Methodology Very eXtreme Programming (VXP)

(continued)

- Requirements: Your project focuses on two application services
- Planning: User stories and work breakdown
- Doing: Pair programming, write test cases before coding, automate testing
- Demoing: 5 minute presentation plus 15 minute demo
- Reporting: What got done, what didn't, what tests show
- 1st iteration: Any
- 2nd iteration: Use some component model framework
- 3rd iteration: Refactoring, do it right this time

Revised Project Concept (Tips)

1. Cover page (max 1 page)
2. Basic concept (max 3 pages): Briefly describe the system your team proposes to build. Write this description in the form of either user stories or use cases (your choice). Illustrations do <u>not</u> count towards page limits.
3. Controversies (max 1 page)

First Iteration Plan (Tips)

- Requirements (max 2 pages):
- Select user stories or use cases to implement in your first iteration, to produce a demo by the last week of class
- Assign priorities and points to each unit - A point should correspond to the amount of work you expect one pair to be able to accomplish within one week
- You may optionally include additional medium priority points to do “if you have time”
- It is acceptable to include fewer, more or different use cases or user stories than actually appeared in your Revised Project Concept

First Iteration Plan (Tips)

- Work Breakdown (max 3 pages):
- Refine as *engineering tasks* and assign to pairs
- Describe specifically what will need to be coded in order to complete each task
- Also describe what unit and integration tests will be implemented and performed
- You may need additional engineering tasks that do not match one-to-one with your user stories/use cases
- Map out a *schedule* for the next weeks
- Be realistic – demo has to been shown before the end of the semester

2nd Iteration Plan (Tips): Requirements

- Max 3 pages
- Redesign/reengineer your system to use a component framework (e.g., COM+, EJB, CCM, .NET or Web Services)
- Select the user stories to include in the new system
  - Could be identical to those completed for your 1st Iteration
  - Could be brand new (but explain how they fit)
- Aim to maintain project velocity from 1st iteration
- Consider what will require new coding vs. major rework vs. minor rework vs. can be reused "as is"

2nd Iteration Plan (Tips): Breakdown

- Max 4 pages
- Define engineering tasks, again try to maintain project velocity
- Describe new unit and integration testing
- Describe regression testing
  - Can you reuse tests from 1st iteration?
  - If not, how will you know you didn't break something that previously worked?
- 2nd iteration report and demo to be presented before the end of the semester

2nd Iteration Report (Tips): Requirements

- Max 2 pages
- For each engineering task from your 2nd Iteration Plan, indicate whether it succeeded, partially succeeded (and to what extent), failed (and how so?), or was not attempted
- Estimate how many user story points were actually completed (these might be fractional)
- Discuss specifically your success, or lack thereof, in porting to or reengineering for your chosen component model framework(s)

2nd Iteration Report (Tips): Testing

- Max 3 pages
- Describe the general strategy you followed for unit testing, integration testing and regression testing
- Were you able to reuse unit and/or integration tests, with little or no change, from your 1st Iteration as regression tests?
- What was most difficult to test?
- Did using a component model framework help or hinder your testing?

Project Presentation and Demo

- All Iterations Due
- Presentation slides (optional)

Readings

- Readings
  - Slides and Handouts posted on the course web site
  - Documentation provided with business and application modeling tools (e.g., Popkin Software Architect)
  - SE Textbook: Chapters 8-12 (Part 2), 18-19 (Part 3), 30 (Part 5)
- Project Frameworks Setup (ongoing)
  - As per references provided on the course Web site
- Individual Assignment
  - See Session 6 Handout: “Assignment #4”
- Team Assignment
  - See Session 6 Handout: “Team Project” (Part 2)

Next Session:

From Analysis and Design to Software Architectures (Part III)

- Enterprise Architectural Patterns
- Sample Middleware Reference Architectures
- Architectural Capabilities
- Object-Oriented Design Guidelines
- Summary
  - Individual Assignment #5
  - Project (Part 3)