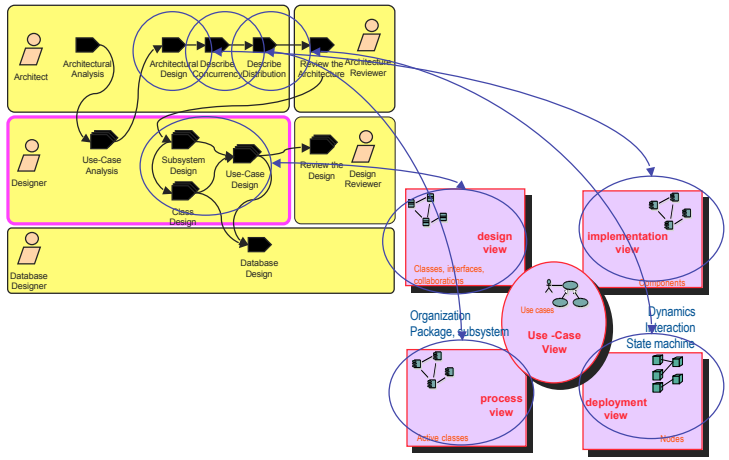
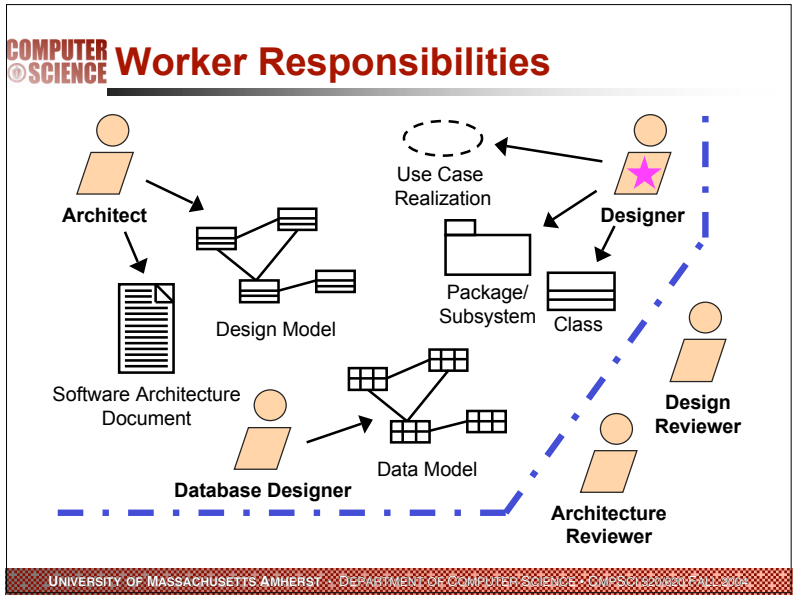
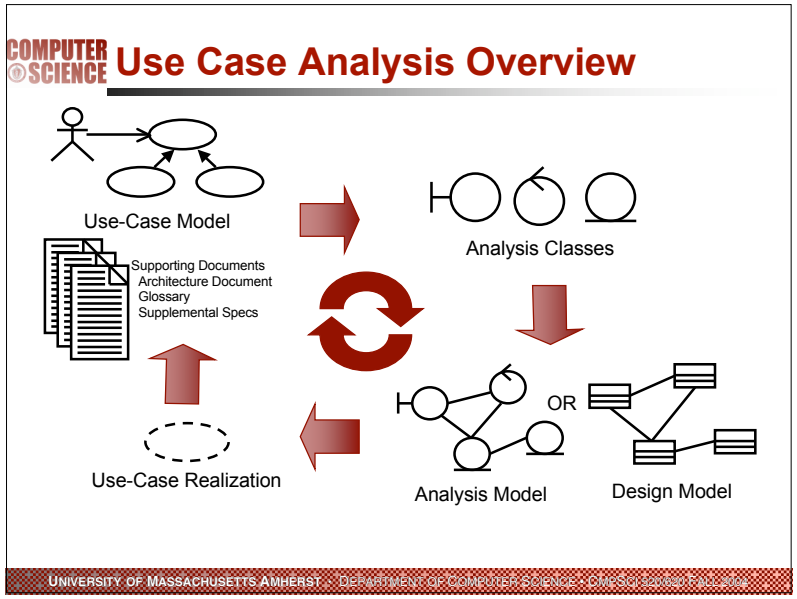
19-Design

- Readings
  - OOAD Using the UML
    - Copyright 1994-1998 Rational Software, all rights reserved
  - [Partly] posted

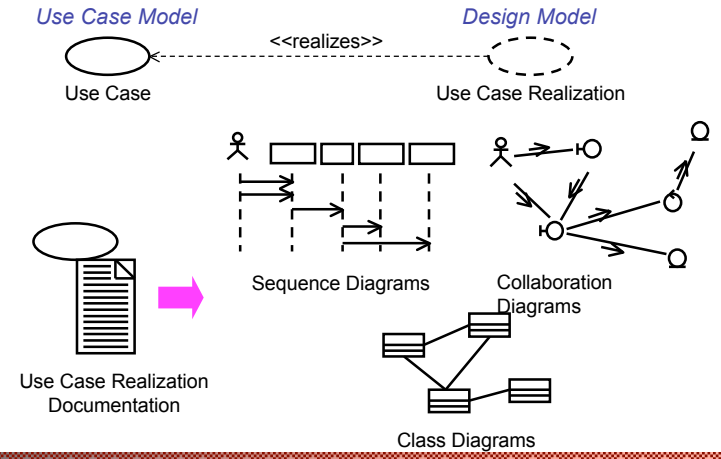
Views & Workflows







What is a Use-Case Realization?



Alternatives

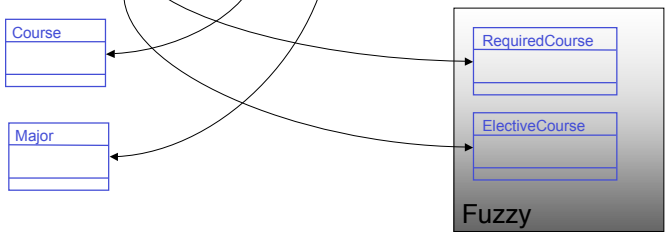
- RUP begins with Analysis classes we found by analyzing Collaboration & Sequence Diagrams (these derived from the Use Cases during Use Case Analysis), then is refined by defining Operations, States, Attributes, Associations and Generalizations (in Class Design)
- Some other approaches:
  - Noun Phrase Approach
  - Common Class Patterns
  - CRC (Class-Responsibility-Collaboration)

Noun Phrase Approach

- Examine the requirements and underline each noun
  - Each noun is a candidate class
- Divide list of candidate classes into
  - Relevant Classes
    - Part of the application domain; occur frequently in reqs
  - Irrelevant Classes
    - Outside of application domain
  - Fuzzy Classes
    - Unable to be declared relevant with confidence; require additional analysis
  - Experience will eventually enable designers to avoid generating irrelevant classes

Find Classes from requirements

- Consider the following University Enrollment system specification
  - each university major has a number of required courses and a number of elective courses.



Classes, Relationships & Attributes

- A **course** can be part of any number of **majors**
- Each **major** specifies minimum total credits required
- Students may combine **course offerings** into **programs of study** suited to their individual needs and leading to the degree/major in which enrolled

Relevant classes	Fuzzy classes
Course	CompulsoryCourse
Major	ElectiveCourse
Student	Sudyprogram
CourseOffering	

Noun Phrase Approach

- may help in identifying domain objects
- not good at identifying objects that live in the application domain
  - Thus, it can help at the beginning of analysis, but you will not return to it as you move into design
  - Finding good objects during design means identifying abstractions that are part of your application domain and its execution machinery
  - Objects that are part of your application domain will have a tenuous connection, at best, to real-world things
    - e.g. what's the correspondence of a scrollbar to the real world

Common Class Patterns

- Derive classes from the generic classification theory of objects
  - Concept class
    - a notion shared by a large community
  - Events class
    - captures an event that demarks intervals within a system
  - Organization class
    - a collection or group within the domain
  - People class
    - roles people can play
  - Places class
    - a physical location relevant to the system

Common Class Patterns

- Rumbaugh proposed a different scheme
  - Physical Class (Airplane)
  - Business Class (Reservation)
  - Logical Class (FlightTimeTable)
  - Application Class (ReservationTransaction)
  - Computer Class (Index)
  - Behavioral Class (ReservationCancellation)
- These taxonomies are meant to help a designer think of classes, however it is difficult to be systematic
- Probably only useful during early analysis

CRC Cards

- CRC = **Candidates, Responsibilities, Collaborators**
- Meant primarily as a brainstorming tool for analysis and design
- In place of use case diagrams ⇒ use index cards
- In place of attributes and methods ⇒ record responsibilities
  - See **Object Design** by Wirfs-Brock and McKean, © 2003

Index Cards

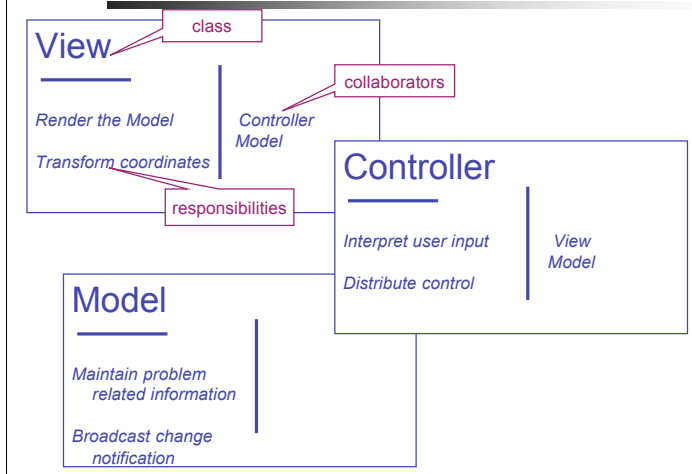
- On the unlined side of the index card
  - write an informal description of each candidate class' purpose and role
- On the lined side of the index card
  - identify responsibilities and collaborators

Document
Purpose: A Document acts as a container for graphics and text
Role: Container
Pattern: Composite

Document	←*candidate*
Knows contents	TextFlow
Knows storage location	
Inserts and removes	
text,graphics, other elements	

responsibilities *collaborators*

MVC using CRC cards



Not Just Index Cards

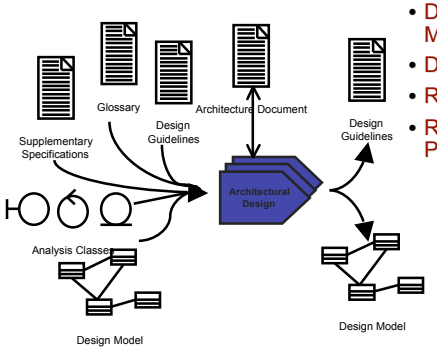
- Post-It Notes can be used for even less "structure";
  - might be easier when brainstorming

Document
Purpose: A document Represents a container that holds text and/or graphics that the user can enter and visually arrange on pages

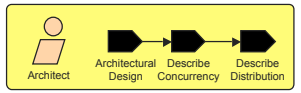
Why index cards?

- Forces you to be concise and clear and focus on major responsibilities since you must fit everything onto one index card
- Inherent Advantages
  - cheap, portable, readily available, and familiar
  - gives people a "feel" for the design
  - can propose and test changes to the design rapidly (all you have to do is make new cards)
  - focus on responsibilities as opposed to ”n:m attribute" design as promoted by OMT, Booch, etc
  - affords Spatial Semantics…
    - close collaborators can be overlapped
    - vertical dimension can be assigned meanings
    - abstract classes and specializations can form piles …which provides benefits
    - Beck and Cunningham report that they have seen designers talk about a new card by pointing at where it will be placed

Architectural Design Overview



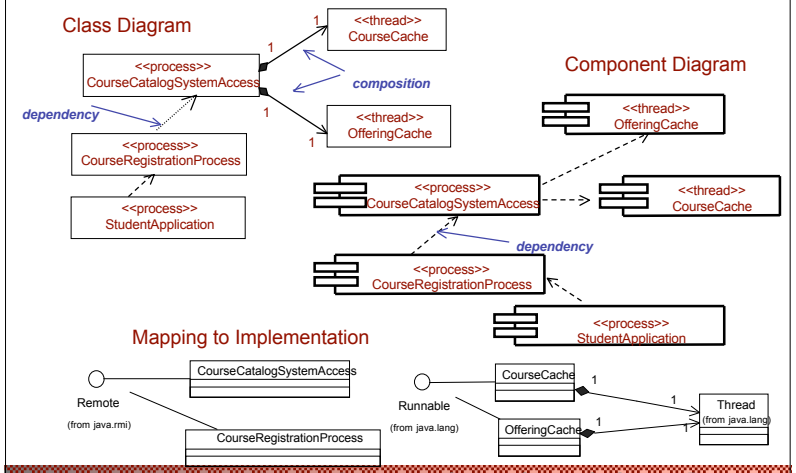
- Design and Implementation Mechanisms
- Design Classes and Subsystems
- Reuse Opportunities
- Refined Architectural Layers and Partitions



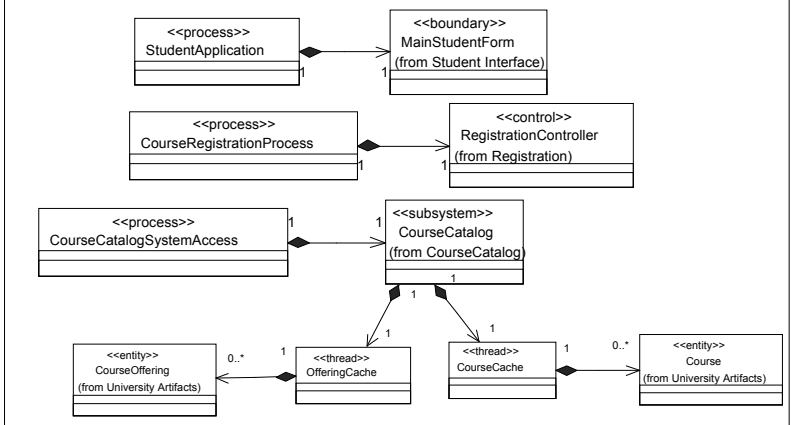
Describe Concurrency

- Concurrency Requirements driven by:
  - degree to which the system must be distributed
  - degree to which the system is event-driven
  - computational intensity of key algorithms
  - degree of parallel execution supported by the environment
- Modeling Processes map on independent threads of control supported by environment
  - Processes - stand-alone, heavyweight flow of control that may be divided into individual threads
  - Threads- lightweight flow of control which run in the context of an enclosing process
- Mapping Processes onto the Implementation Environment
- Distributing Model Elements Among Processes

Example

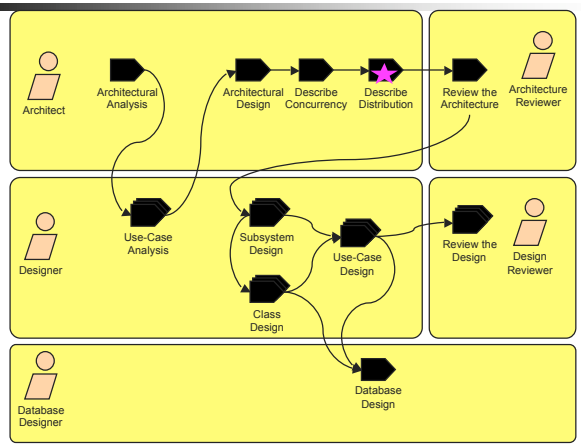


Example

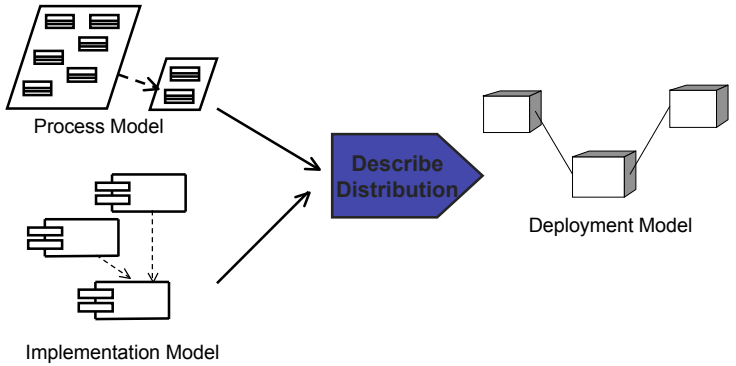


Mapping Design Elements to Processes

So Where Are We?



Describe Distribution Overview



Why Distribute?

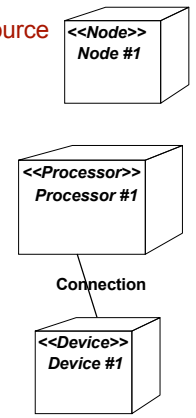
- Reduce processor load
- Special processing requirements
- Scaling concerns
- Economic concerns
- Distribution Patterns
  - Client/Server
    - 3-tier
    - Fat-Client
    - Web Application
    - Distributed Client/Server
  - Peer-to-peer

Distribution patterns

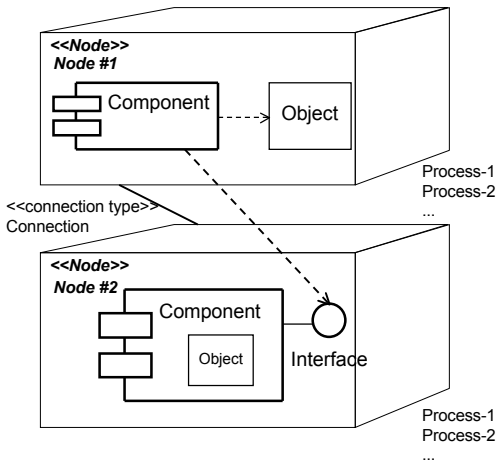
- Common services
  - Presentation Services
    - UI including, the visual appearance of output and how user input is handled
  - Business Services
    - Business rules and logic
  - Data Services
    - Data relationships, efficiency of storage, and data integrity
- Patterns
  - One-Tier
  - Two-Tier
    - Fat Client -- client has its presentation and business services; server has the data services
    - Thin Client -- client has the presentation services; server has the business and data services
  - Three-tier
    - client has presentation services; server has business services; separate (logical) server has data services.
  - Web-tier
    - client accesses a web server that at least handles presentation services; web server may have its own business and data services or it may utilize one or more servers that handle business and data services

Deployment Model Modeling Elements

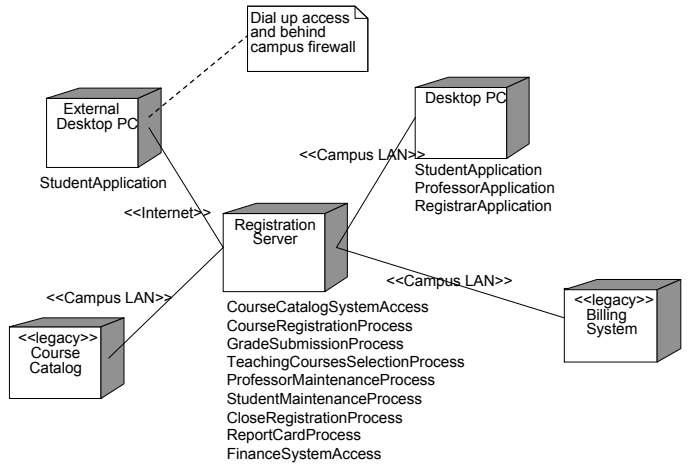
- Node
  - Physical run-time computational resource
- Processor
  - Execute system software
- Device
  - Support devices
  - Typically controlled by a Processor
- Connection
  - Communication mechanisms
  - Physical medium
  - Software protocol



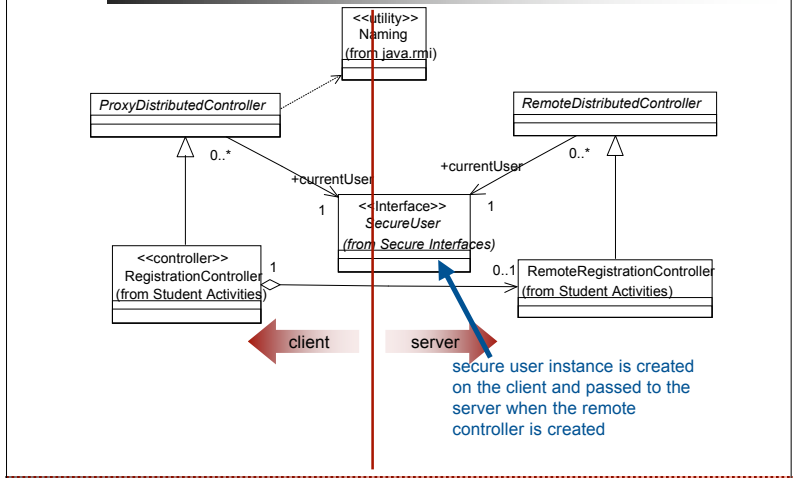
Deployment Diagrams



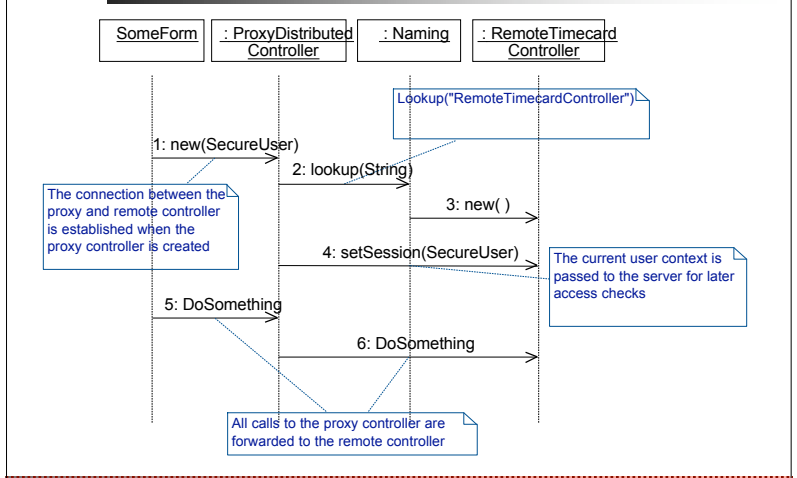
Process-to-Node Allocation



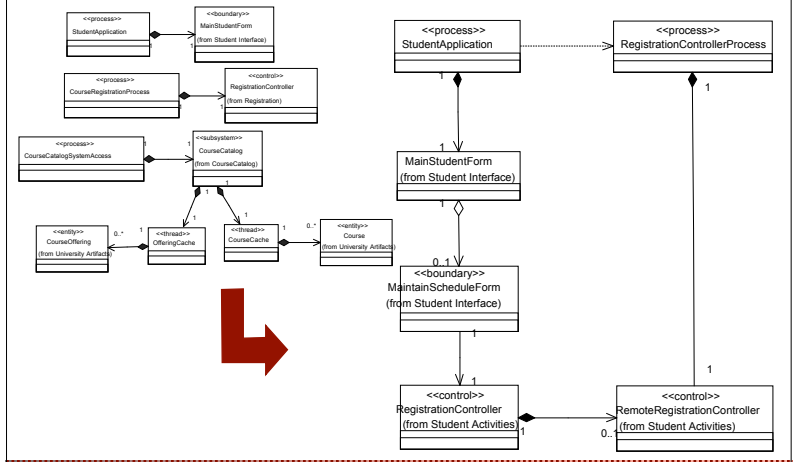
Design Distribution Pattern: Proxy



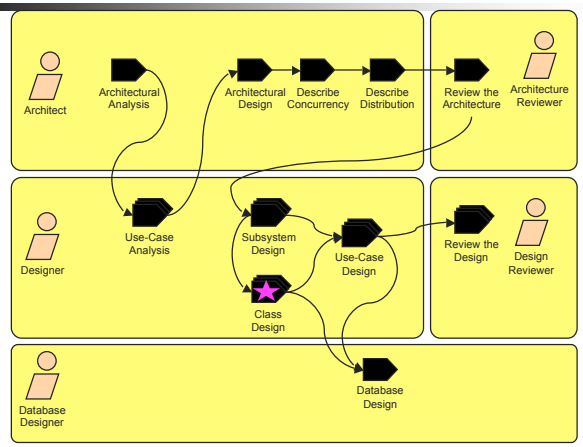
Design Distribution Pattern: Proxy



Affect on Process Model Associations

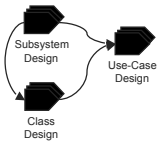


So Where Are We?

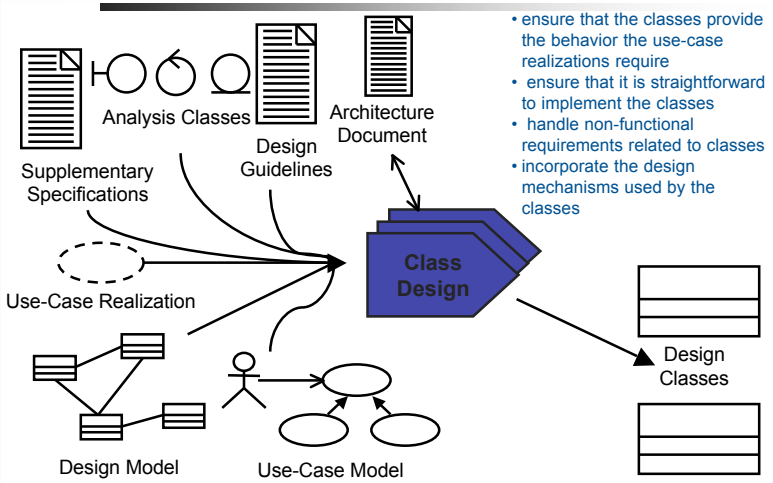


Design Iterations

- Architectural Design
  - decide what the infrastructure is (the pieces/parts of the architecture, if you will, and how they interact).
- Use Case Design
  - determine the responsibilities of the system are allocated to the pieces/parts.
- Subsystem and Class design
  - detail the specifics of the pieces/parts.
  - adjust the classes to the particular products in use, the programming languages, distribution, adaptation to physical constraints (e.g. limited memory), performance, use of component environments such as COM or CORBA, and other implementation technologies
- There is frequent iteration between Class Design, Subsystem Design, and Use Case Design.



Class Design Overview



- ensure that the classes provide the behavior the use-case realizations require
- ensure that it is straightforward to implement the classes
- handle non-functional requirements related to classes
- incorporate the design mechanisms used by the classes

Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define States
- Define Attributes
- Define Associations
- Define Generalizations

Class Design

- strategy:
  - how the analysis classes will be realized in the implementation
  - how design patterns can be used to help solve implementation issues
  - how the architectural mechanisms will be realized in terms of the defined design classes.
- boundary, control and entity stereotypes are most useful during Use Case Analysis
  - no longer need to make the distinction
- design patterns will be introduced, as needed, throughout Class Design.

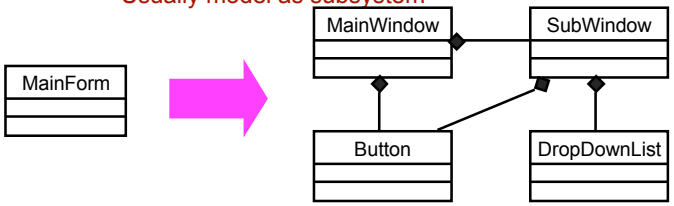
How Many Classes Are Needed?

- Many, simple classes means that each class
  - encapsulates less of the overall system intelligence
  - is more reusable
  - is easier to implement
- A few, complex classes means that each class
  - encapsulates a large portion of the overall system intelligence
  - is less likely to be reusable
  - is more difficult to implement
- A class should have a single well focused purpose
  - a class should do one thing and do it well!
  - how does this relate to my earlier suggestion that classes have multiple responsibilities?

- Class should have **multiple** responsibilities
  - Actions that object can perform
  - Knowledge object maintains
  - Non-functional requirements

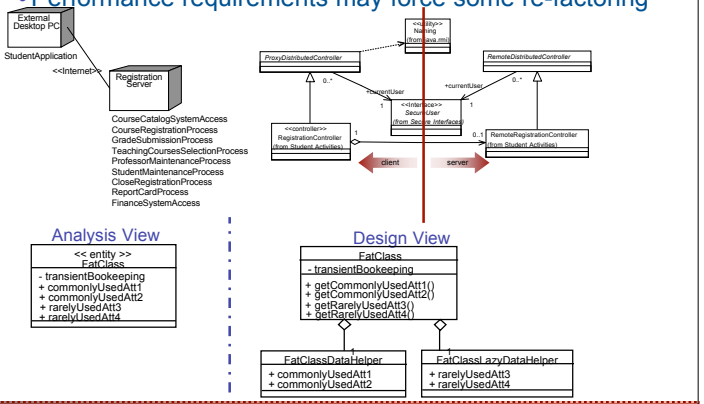
Designing Boundary Classes

- User interface (UI) boundary classes
  - What user interface development tools will be used?
  - How much of the interface can be created by the development tool?
  - "Reverse Engineering"
- External system interface boundary classes
  - Usually model as subsystem



Designing Entity Classes

- Entity objects are often passive and persistent
- Performance requirements may force some re-factoring

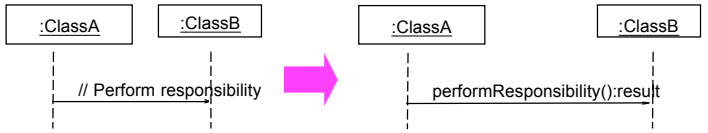


Designing Control Classes

- What Happens to Control Classes?
  - Are they really needed?
    - if just "pass-throughs" from the boundary classes to the entity classes, they may be eliminated.
  - Should they be split?
    - might depend on distribution, e.g., proxy-remote
- Control classes may become true design classes for any of the following reasons:
  - they encapsulate significant control flow behavior,
  - the behavior they encapsulate is likely to change
  - the behavior must be distributed across multiple processes and/or processors
  - the behavior they encapsulate requires some transaction management.

Operations

- Messages displayed in interaction diagrams



- Implement rules

Student
- name : String - dateOfBirth : Date
+ canEnroll() : Boolean # hasTakenPrerequisites() : Boolean # hasScheduleConflict() : Boolean

every class should have:
- *Manager functions*
- *Implementor functions*
- *Access functions*
- *Helping functions*

- Operations can lead to new class definitions

Utility Classes

- What is a Utility Class?
  - Utility is a class stereotype
  - Used for a class that contains a collection of free subprograms
- Why use it?
  - To provide services that may be (re)useful in a variety of contexts
  - To wrap non object-oriented libraries or applications

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

<<utility>> sql
+ bind() + execsql() + startTrans() + commit() + fetch() + getResults()

Identify and Define the States

- Significant, dynamic attributes

  The maximum number of students per course offering is 10

  numStudents < 10 → **Open**

  numStudents > = 10 → **Closed**

- Existence and non-existence of certain links

  Link to CourseOffering Exists → **Teaching**

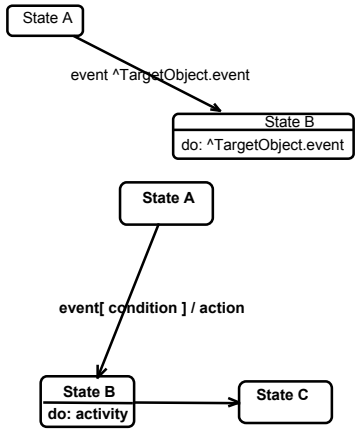
  Link to CourseOffering Doesn't Exist → **On Sabbatical**

  Professor — 0..1 — 0..* — CourseOffering

- explicitly define what it means to be in a particular state.

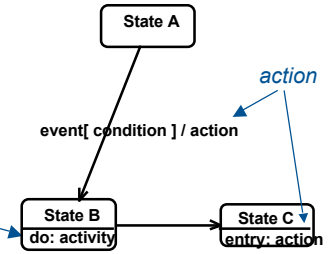
Identify the Events & Transitions

- Events
  - One event may trigger the sending of another event
  - An activity can also send an event to another object
- Transitions
  - For each state, determine what events cause transitions to what states, including guard conditions, when needed
  - Transitions describe what happens in response to the receipt of an event

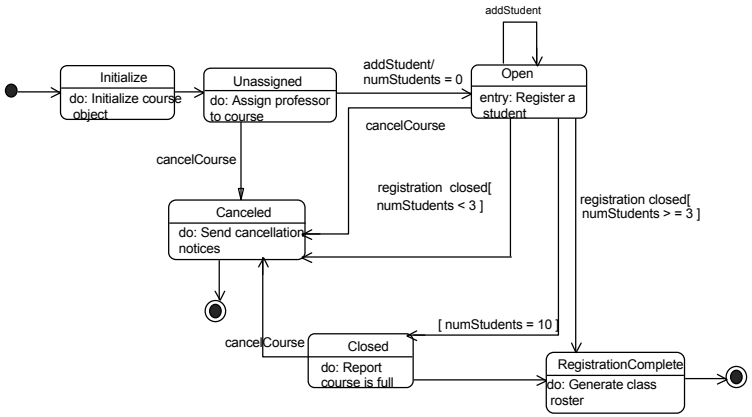


Add Activities and Actions

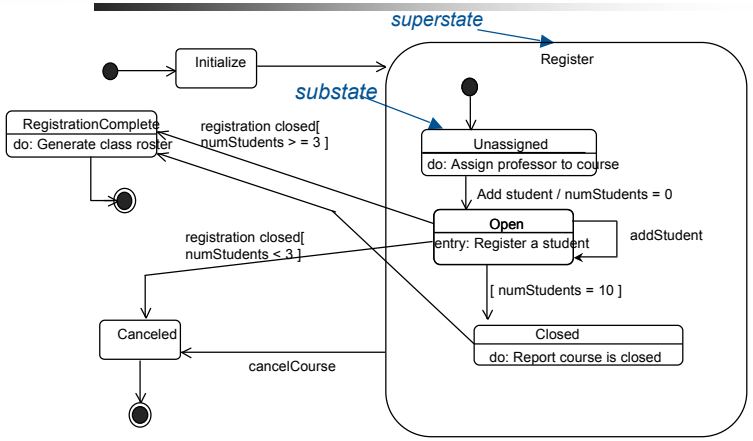
- Activities
  - Associated with a state
  - Start when the state is entered
  - Take time to complete
  - Interruptible
- Actions
  - Associated with a transition
  - Take an insignificant amount of time to complete
  - Non-interruptible



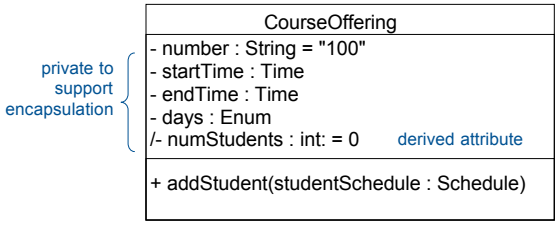
Statechart



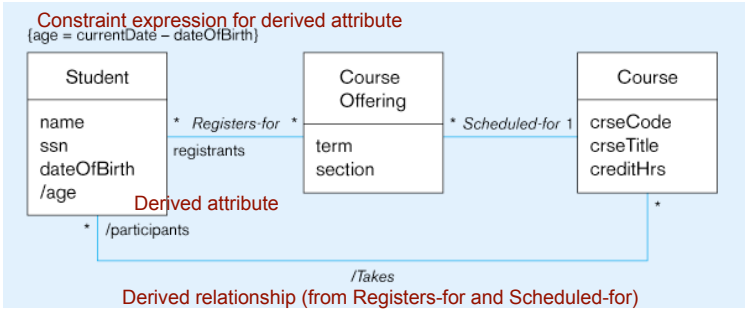
Statechart with Nested States



Example: Define Attributes



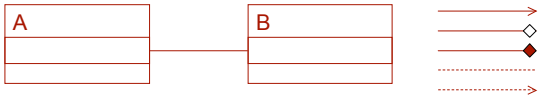
Derived attributes, associations, and roles



Derived attributes and relationships shown with / in front of the name

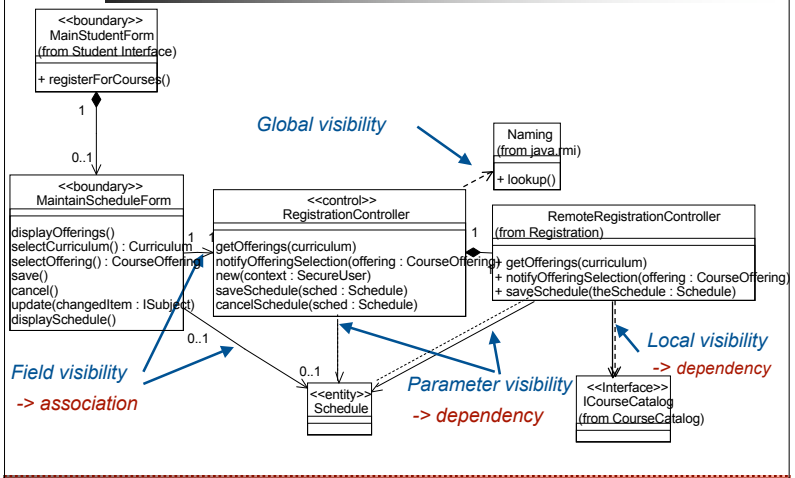
Associations & Dependencies

- association

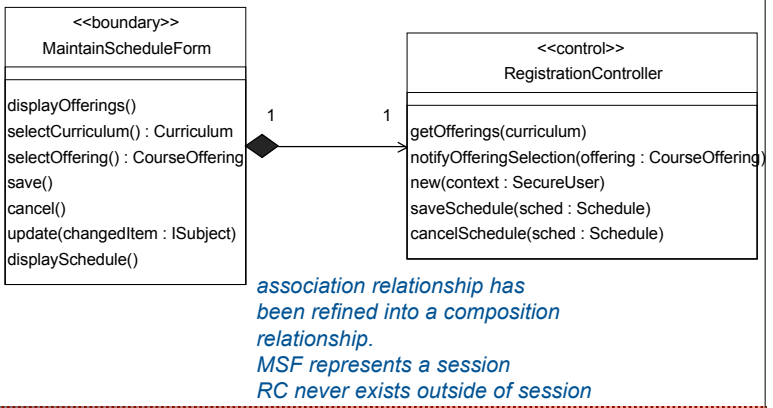


- visibility
  - attribute (field) visibility: B is an attribute of A
    - remains an association
  - parameter visibility: B is a parameter of a method A
    - becomes a dependency
  - local visibility: B is a (non-parameter) local object in a method of A
    - becomes a dependency
  - global visibility: B is in some way globally visible
    - becomes a dependency

Define Dependencies

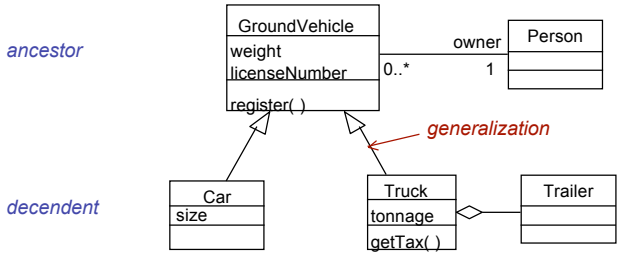


Example: Composition



Generalization (Inheritance)

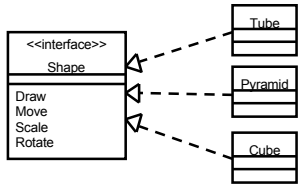
- One class inherits from another



- not just finding common attribute, operations and relationships
- more about the responsibilities and essence of the classes.
- avoid "skyscrapers"; the hierarchies should look like small, independent "forests

What is Polymorphism?

- The ability to hide many different implementations behind a single interface



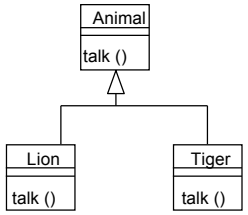
- Use generalization to support polymorphism

Without Polymorphism

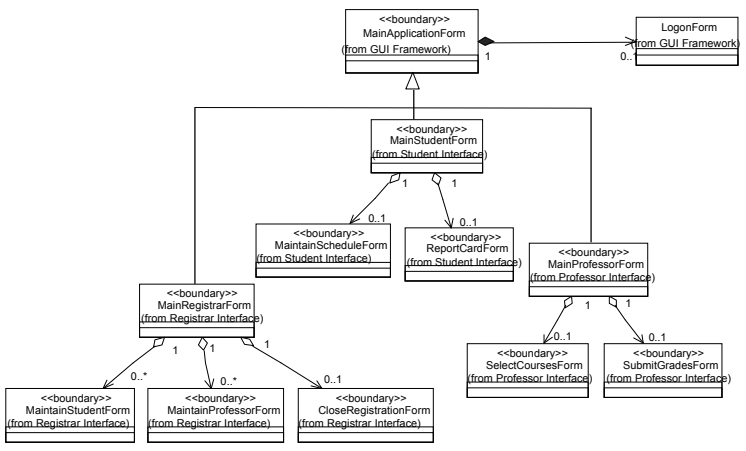
```
if animal = "Lion" then
      do the Lion talk
else if animal = "Tiger" then
      do the Tiger talk
end
```

With Polymorphism

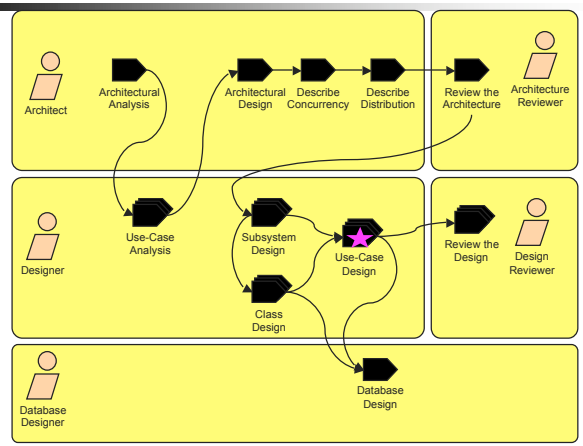
```
do the Animal talk
```



Define Generalizations



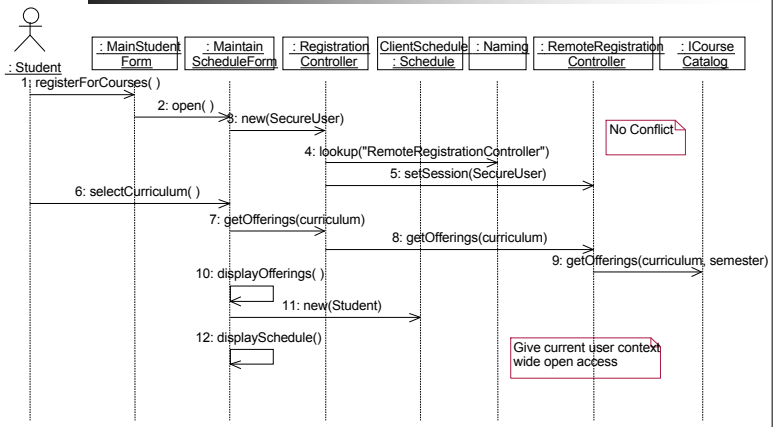
So Where Are We?

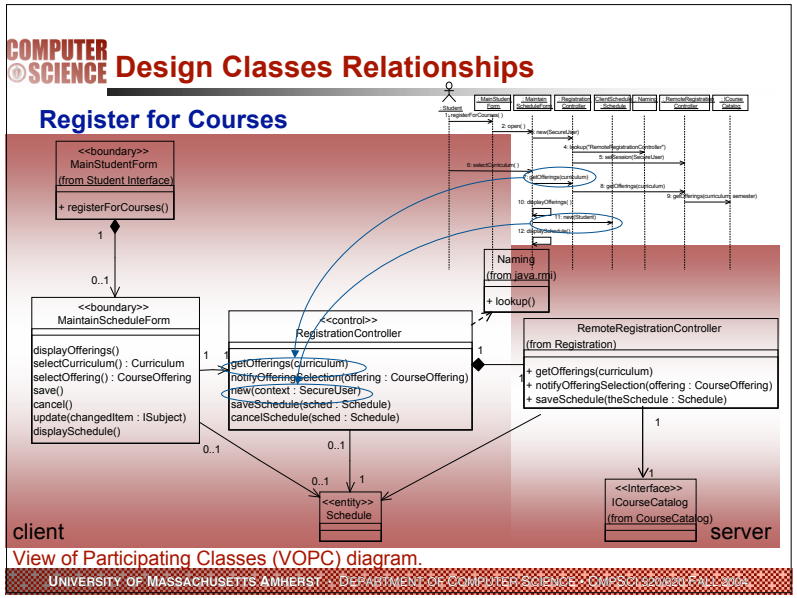


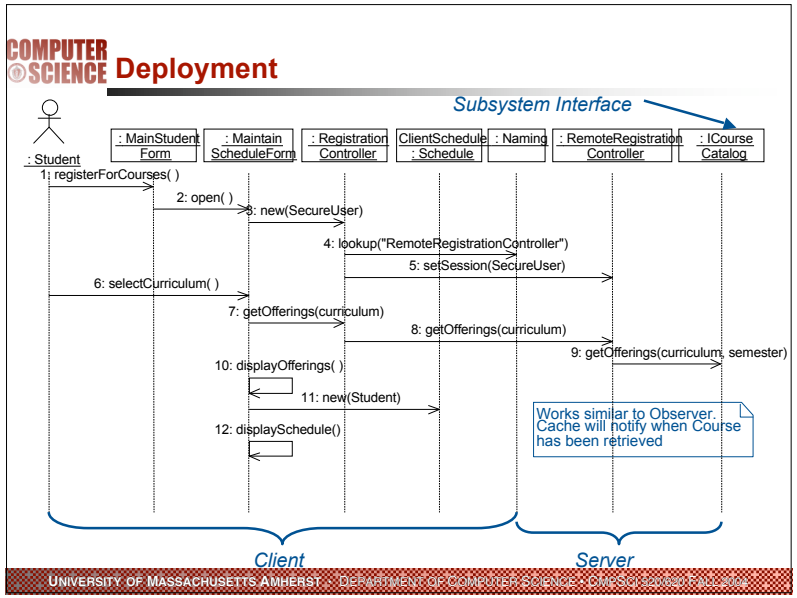
Strategy - where are we?

- made an initial attempt at defining the architecture
- defined the major elements of our system
  - the subsystems, their interfaces, the design classes, the processes and threads
  - relationships & how these elements map into the hardware on which the system will run.
- Now, concentrate on
  - making sure that there is consistency from beginning to end of use case implementation, i.e., that nothing has been missed (i.e., this is where we make sure that what we have done in the previous design activities is consistent with regards to the use case implementation).
- we do some Use Case Design before Subsystem Design
- Subsystem Design, Class Design and Use Case Design activities are tightly bound and tend to alternate between one another.

Design Element Interactions (Register For Courses - Set-Up)

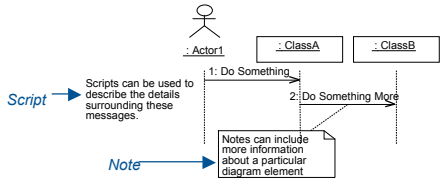




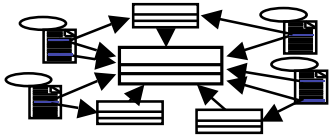


More steps

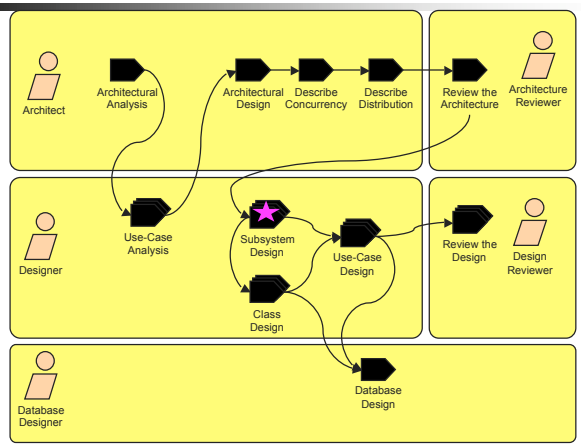
- Annotate the sequence diagrams



- Unify classes & subsystems
  - merge similar model elements
  - use inheritance to abstract model elements



So Where Are We?



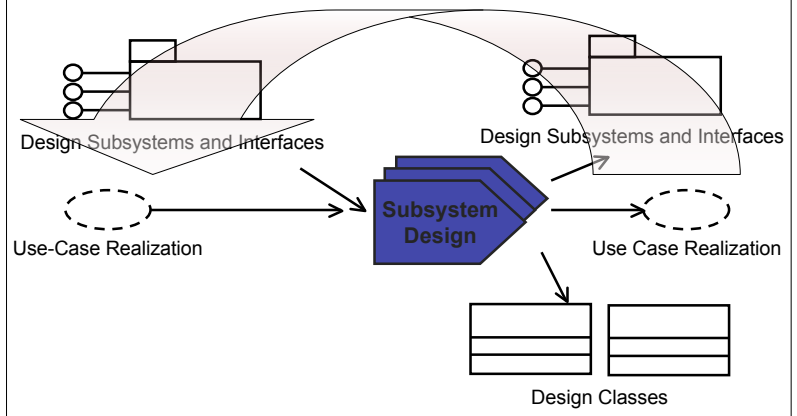
Strategy -- where are we?

- have defined subsystems, their interfaces, and their dependencies as “containers” of complex behavior that, for simplicity, we treat as a ‘black box’
- made an initial cut at some design classes, which have been allocated to subsystems
- need to flesh-out the details of the internal interactions
  - what classes exist in the subsystem to support?
  - how do they collaborate to support, the responsibilities documented in the subsystem interfaces?
  - In Subsystem Design, we look at the responsibilities of the subsystems in detail, defining and refining the classes that are needed to implement those responsibilities, refining subsystem dependencies, as needed. The internal interactions are expressed as collaborations of classes and possibly other components or subsystems

Strategy

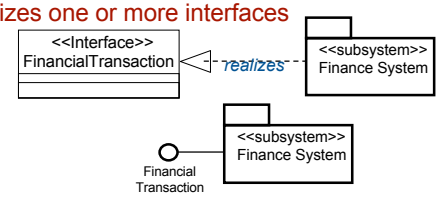
- need to do some Use Case Design before Subsystem Design
- after Analysis and Architectural Design
  - usually only have sketchy notions of responsibilities of classes and subsystems
  - details need to get worked out in Use Case Design, before one is really ready to design the classes and subsystems
- Reminder: there is frequent iteration between Use Case Design, Subsystem Design and Class Design.

Subsystem Design Overview



Interface and Subsystem

- What is an interface?
  - a model element which defines a set of behaviors (a set of operations) offered by a classifier model element (specifically, a class, subsystem or component)
- What is a subsystem?
  - Contains other model elements and has behavior
  - Realizes one or more interfaces

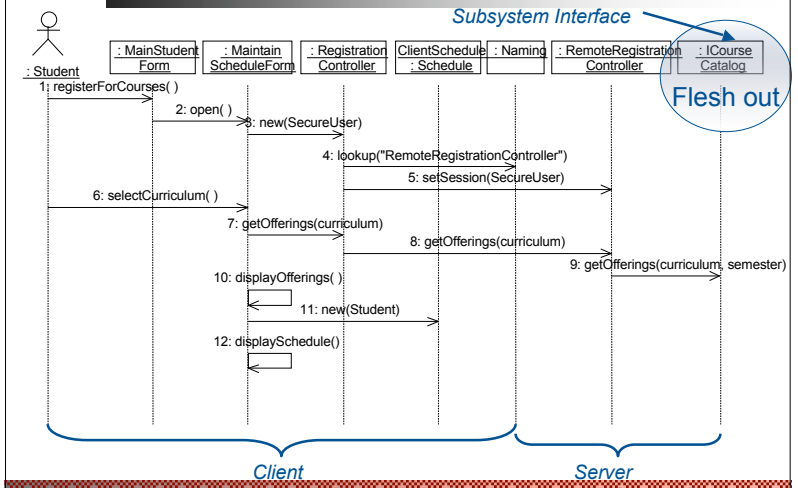


Recall: packages provide no behavior; they are simply containers of things which provide behavior

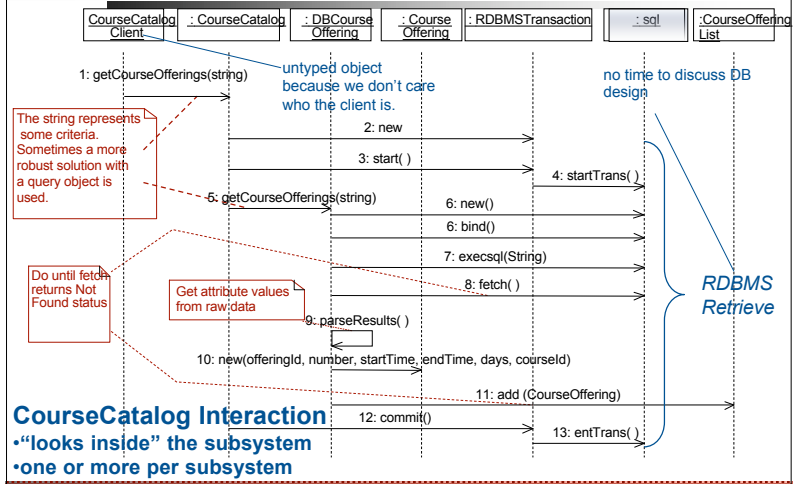
Distribute Subsystem Responsibilities

- Identify or reuse existing classes and/or subsystems
- Allocate subsystem responsibilities to classes and/or subsystems
- Incorporate the applicable mechanisms (e.g., persistence, distribution, etc.)
- Document collaborations with "interface realization" diagrams
  - 1 or more sequence diagrams per interface operation
- Revisit Architectural Design
  - Adjust subsystem boundaries and/or dependencies, as needed

Subsystem design



Local Subsystem Interaction

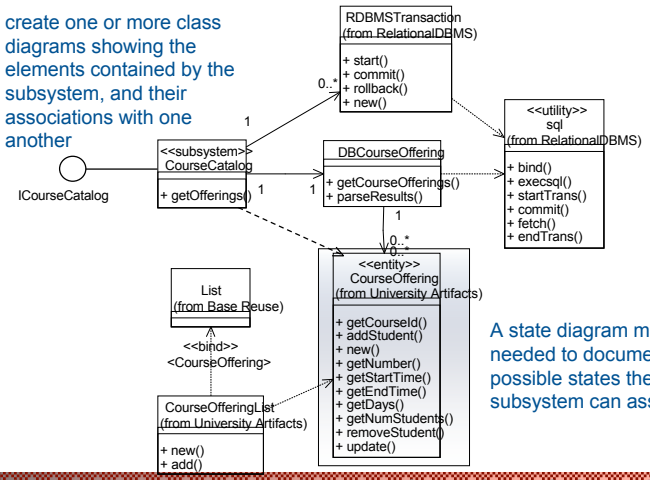


CourseCatalog Interaction

- "looks inside" the subsystem
- one or more per subsystem

Document Subsystem Elements

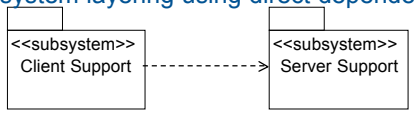
create one or more class diagrams showing the elements contained by the subsystem, and their associations with one another



A state diagram may be needed to document the possible states the subsystem can assume

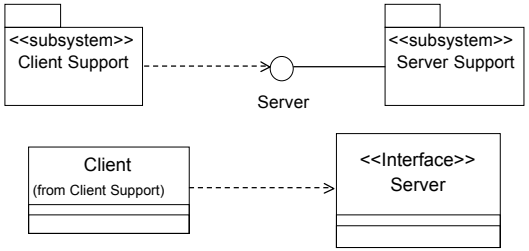
Describe Subsystem Dependencies

- Subsystem layering using direct dependency



Not recommended

- Subsystem layering using interface dependency



More flexible

Describe Subsystem Dependencies