COMPUTER SCIENCE

22- Design: RUP

Rick Adrion

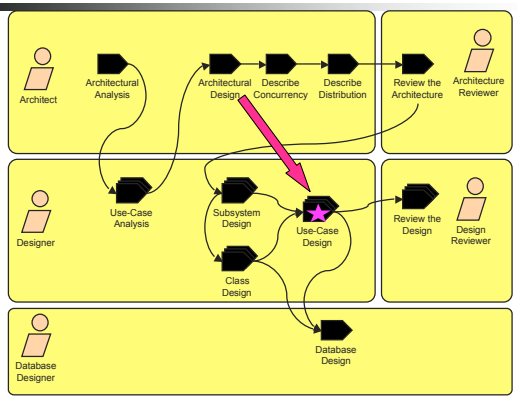
COMPUTER SCIENCE

Grading

- Homework 45%
  - Average of 4 assignments
- Project 45%
  - Project #1 = report (50%) + presentation (50%)
  - Project #2 = report (80%) + questions (10%) + interviews (10%)
  - Project #3 = report (50%) + review (50%)
  - Project = average of #1-#3
- Class participation 10%

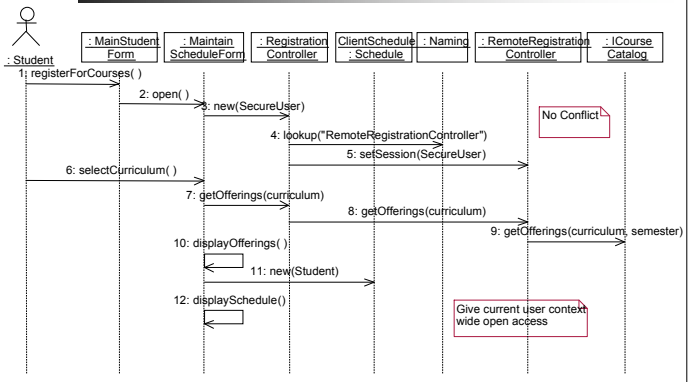
COMPUTER SCIENCE

So Where Are We?



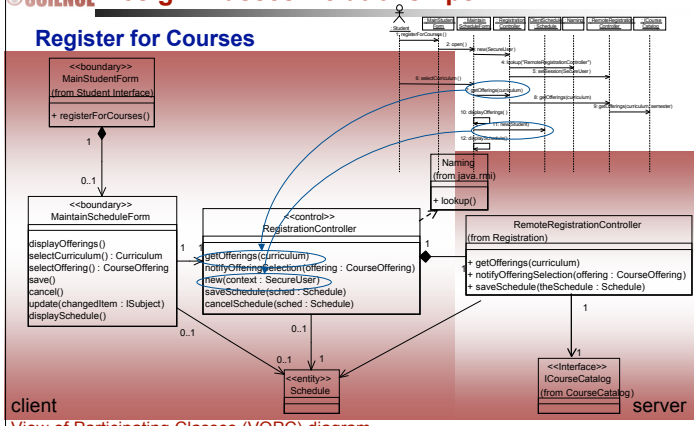
COMPUTER SCIENCE

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



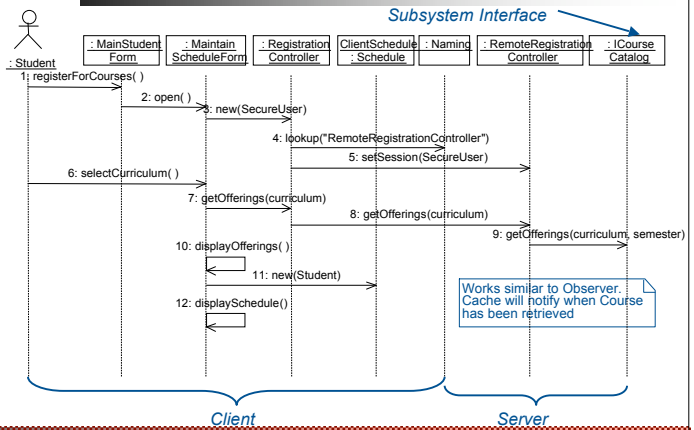
Design Classes Relationships

Register for Courses



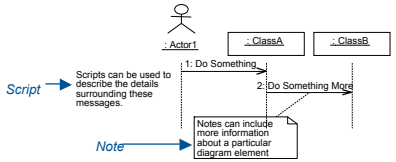
View of Participating Classes (VOPC) diagram.

Deployment

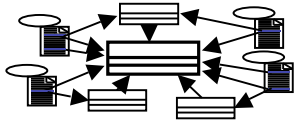


More steps

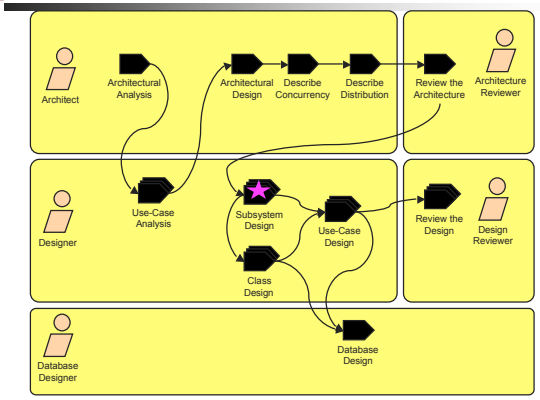
- Annotate the sequence diagrams



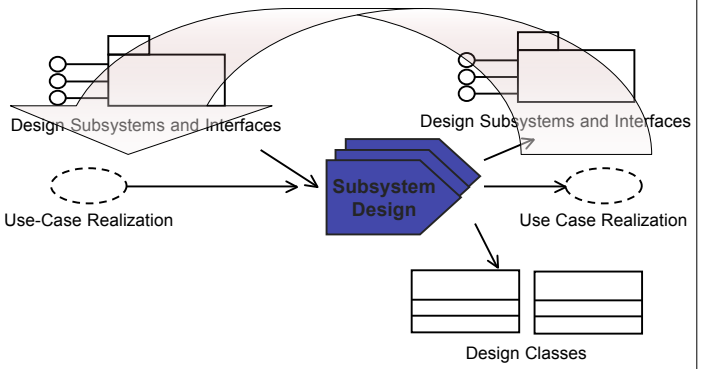
- Unify classes & subsystems
  - merge similar model elements
  - use inheritance to abstract model elements



So Where Are We?



Subsystem Design Overview

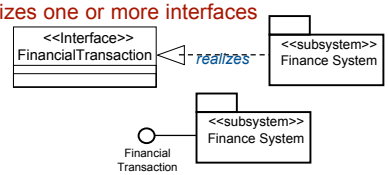


Subsystem design

- we have
  - defined the subsystems, their interfaces, and their dependencies
  - made an initial cut at some design classes, which have been allocated to subsystems
  - identified components or subsystems: “containers” of complex behavior that, for simplicity, we treat as a ‘black box’.
- in Subsystem Design, we look at
  - responsibilities of the subsystems in detail
  - defining and refining the classes that are needed to implement those responsibilities
  - refining subsystem dependencies, as needed
  - internal interactions are expressed as collaborations of classes and possibly other components or subsystems

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

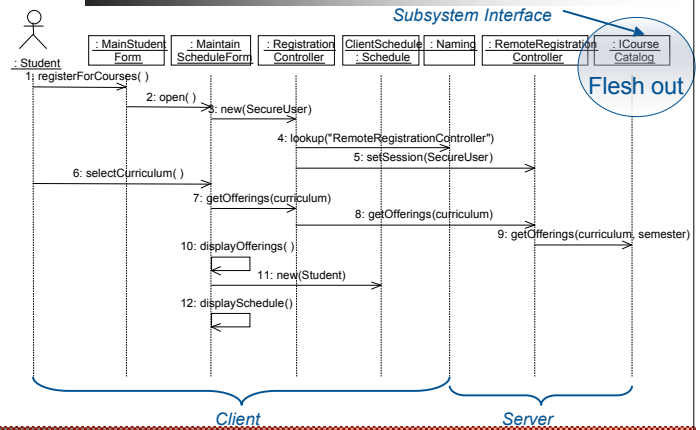


Recall: packages provide no behavior; they are simply containers of things which provide behavior

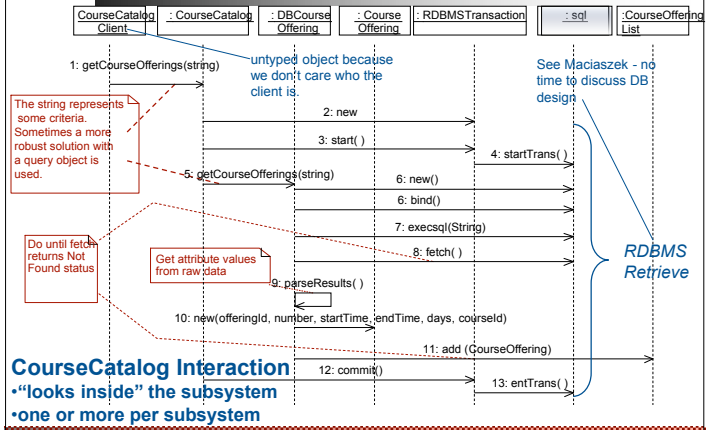
Distribute Subsystem Responsibilities

- 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

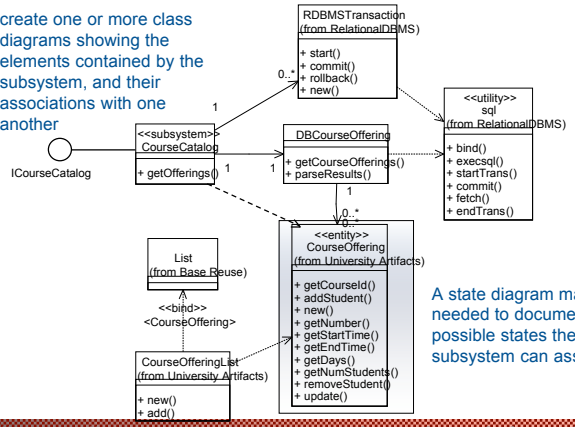


Local Subsystem Interaction



Document Subsystem Elements

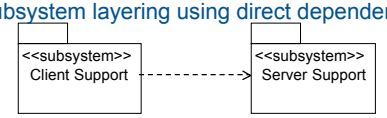
create one or more class diagrams showing the elements contained by the subsystem, and their associations with one another



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

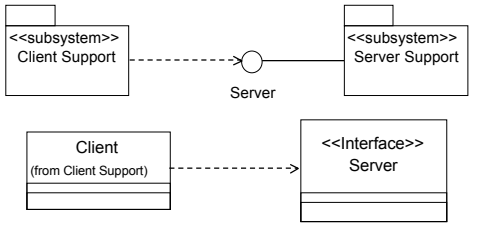
Describe Subsystem Dependencies

- Subsystem layering using direct dependency



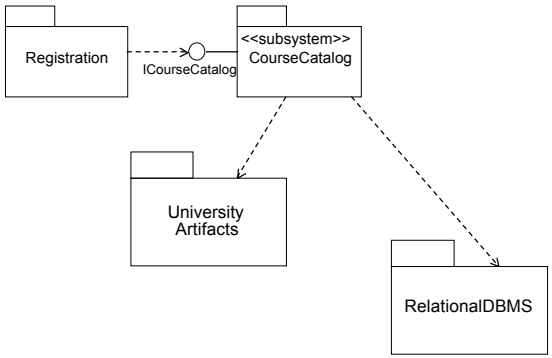
Not recommended

- Subsystem layering using interface dependency

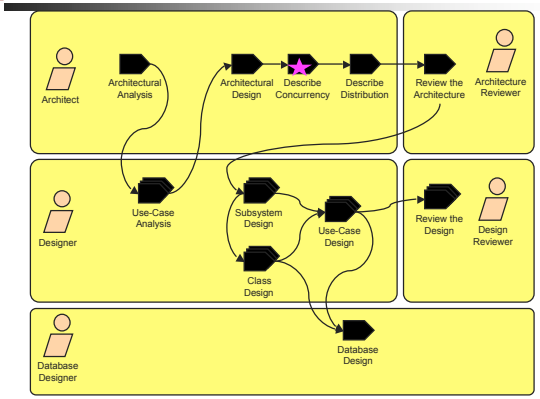


More flexible

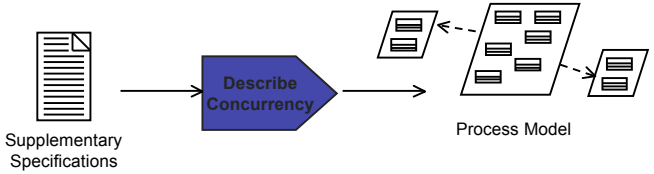
Describe Subsystem Dependencies



So Where Are We?



Describe Concurrency Overview



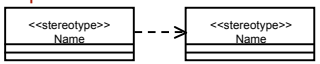
- the independent threads of control are identified and the identified model elements (subsystems and classes) are mapped to these threads of control.

Modeling Processes

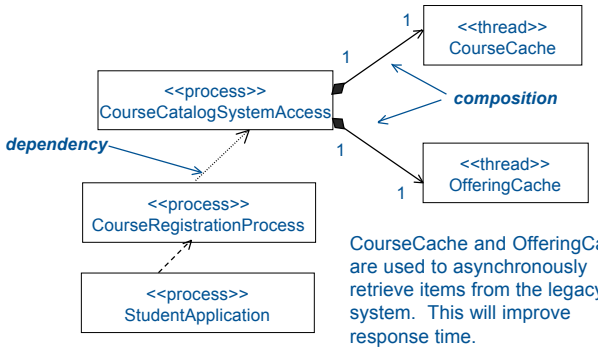
- Processes may be modeled using
  - Active classes (Class Diagrams) and Objects (Interaction Diagrams)
    - "owns" it's own thread of execution and can initiate control activity
    - may execute in parallel (i.e., concurrently) with other active classes.

  - Components (Component Diagrams)

  - Relationships



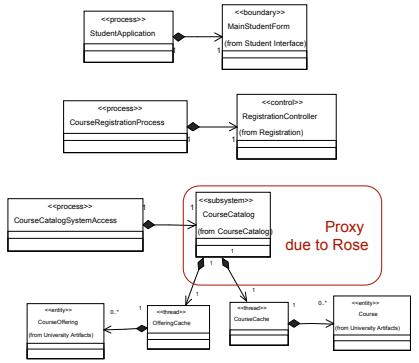
Modeling Processes: Class Diagram



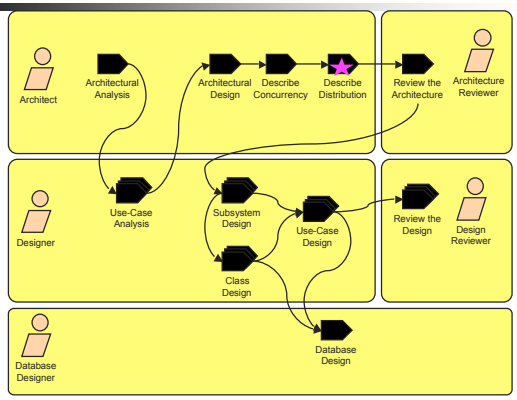
CourseCache and OfferingCache are used to asynchronously retrieve items from the legacy system. This will improve response time.

Mapping Design Elements to Processes

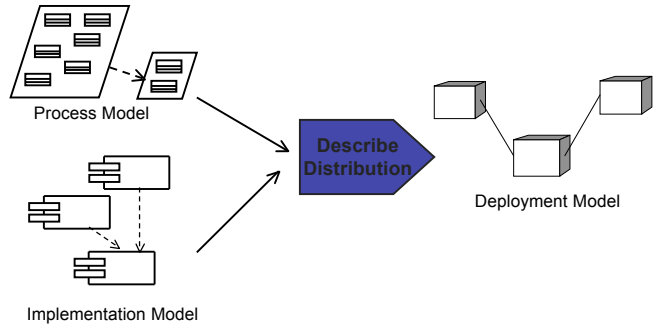
- The classes associated with the individual user interfaces were mapped to the application processes.
- The classes associated with the individual business services were mapped to the controller processes.
- The classes associated with access to the external systems were mapped to the access processes
- For the threads, composition is used to show what design elements have been mapped to them.



So Where Are We?



Describe Distribution Overview



Why Distribute?

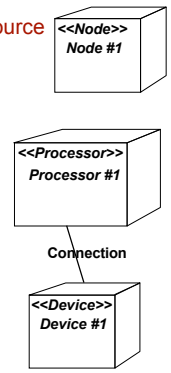
- Reduce processor load
- Special processing requirements
- Scaling concerns
- Economic concerns
- Distribution Patterns
  - Client/Server
    - 3-tier
    - Fat-Client
    - Web Application
    - Distributed Client/Server
  - Peer-to-peer

Distribution patterns

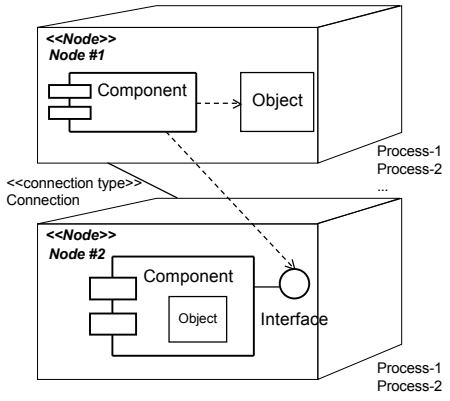
- Common services
  - Presentation Services
    - UI including, the visual appearance of output and how user input is handled
  - Business Services
    - Business rules and logic
  - Data Services
    - Data relationships, efficiency of storage, and data integrity
- Patterns
  - One-Tier
  - Two-Tier
    - 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

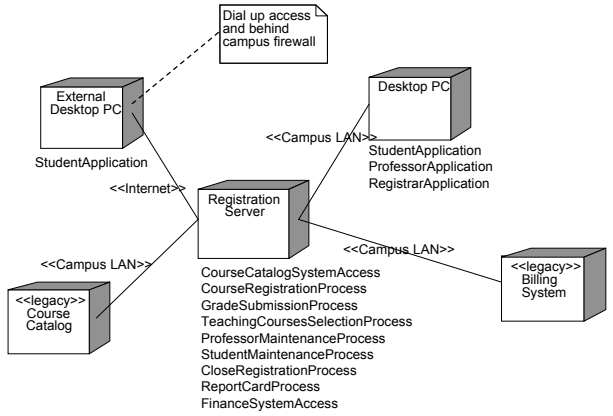
- Node
  - Physical run-time computational resource
- Processor
  - Execute system software
- Device
  - Support devices
  - Typically controlled by a Processor
- Connection
  - Communication mechanisms
  - Physical medium
  - Software protocol



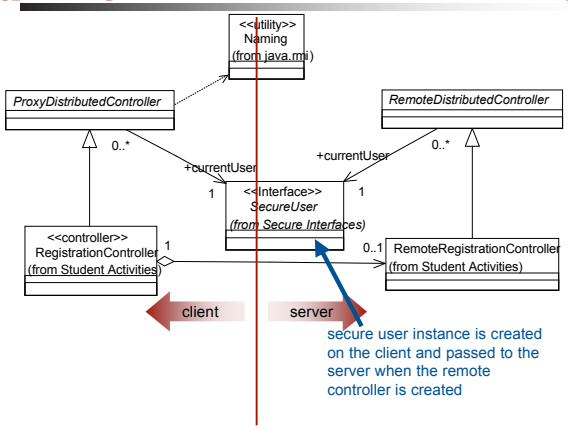
Deployment Diagrams



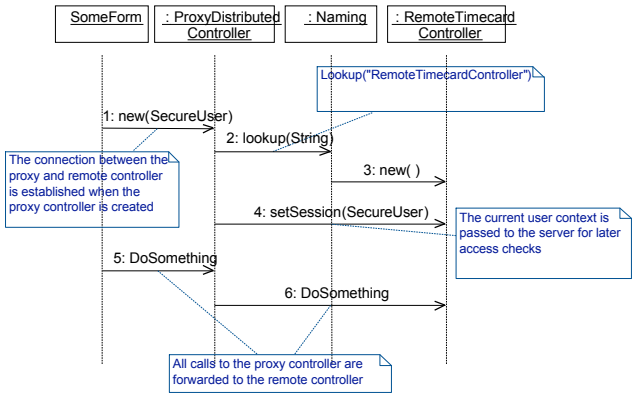
Process-to-Node Allocation



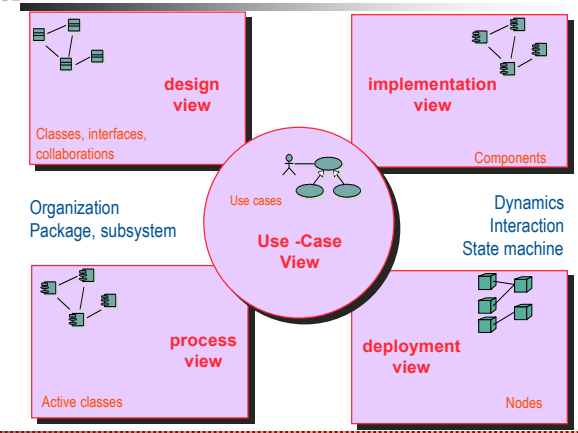
Design Distribution Pattern: Proxy



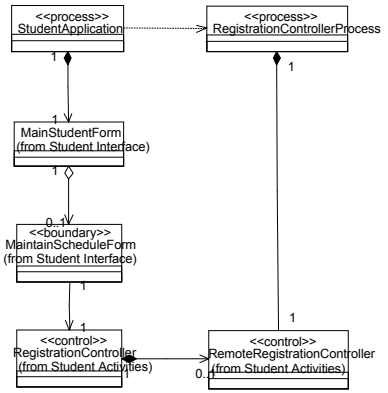
Design Distribution Pattern: Proxy



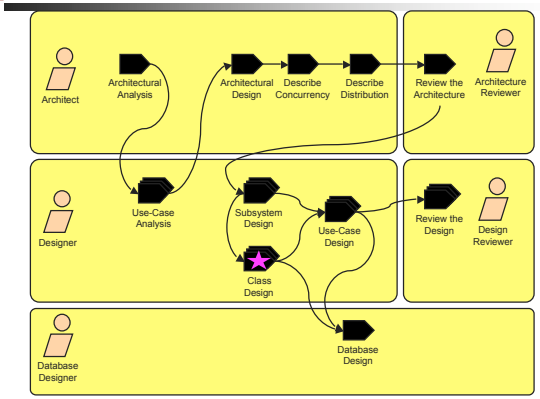
The Rational 4+1 Views



Affect on Process Model Associations



So Where Are We?



Class Design

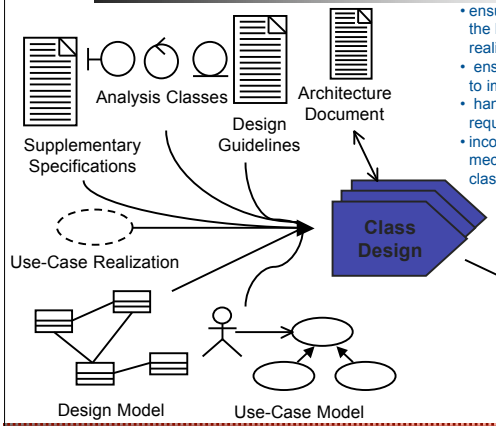
Architectural Design
decide what the infrastructure is (the pieces/parts of the architecture, if you will, and how they interact)

Use Case Design
define responsibilities of the system are allocated to the pieces/parts

Subsystem and Class design
detail the specifics of the pieces/parts.

- Class Design
  - take into account the implementation and deployment environments
  - 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
  - performed for each class in the current iteration.

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

How Many Classes Are Needed?

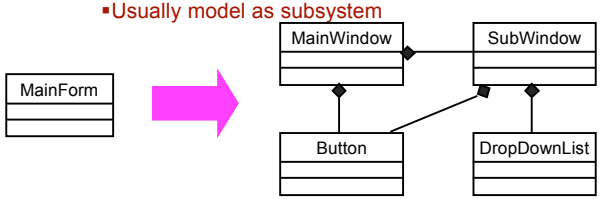
- Many, simple classes means that each class
  - encapsulates less of the overall system intelligence
  - is more reusable
  - is easier to implement
- A few, complex classes means that each class
  - encapsulates a large portion of the overall system intelligence
  - is less likely to be reusable
  - 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

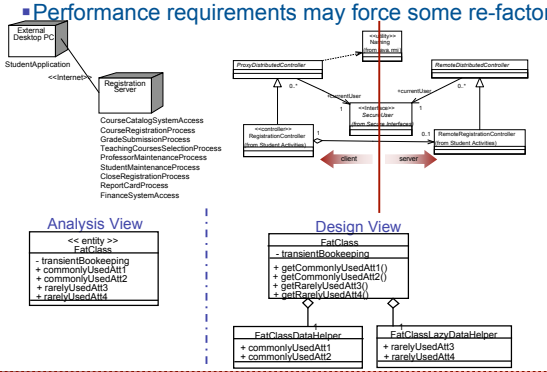
- User interface (UI) boundary classes
  - What user interface development tools will be used?
  - How much of the interface can be created by the development tool?
  - "Reverse Engineering"
- External system interface boundary classes
  - Usually model as subsystem

See Maciaszek - no time to discuss UI design



Designing Entity Classes

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

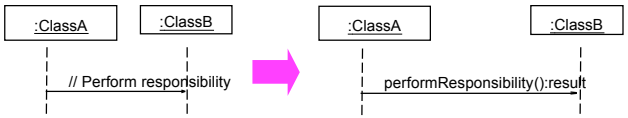


Designing Control Classes

- 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

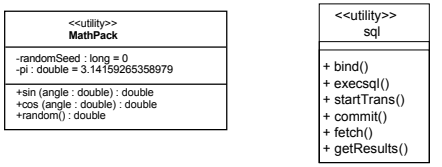
Student
- name : String - dateOfBirth : Date
+ canEnroll() : Boolean # hasTakenPrerequisites() : Boolean # hasScheduleConflict() : Boolean

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

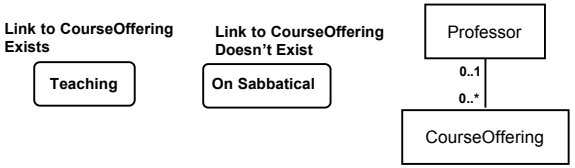


Identify and Define the States

- Significant, dynamic attributes

  The maximum number of students per course offering is 10

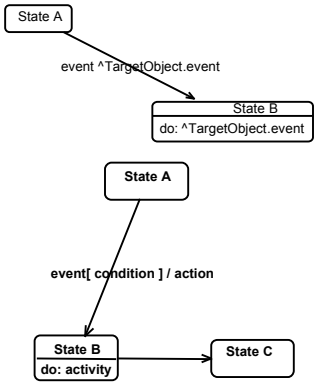
- Existence and non-existence of certain links



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

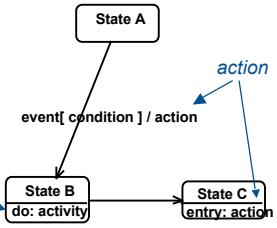
Identify the Events & Transitions

- 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

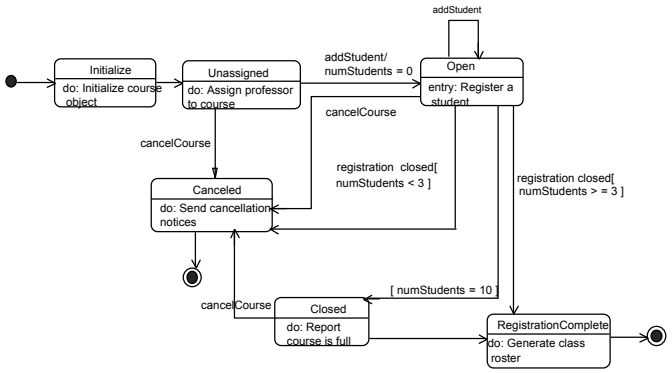


Add Activities and Actions

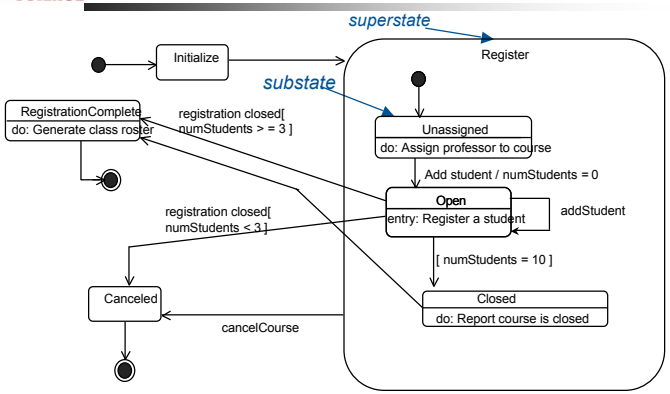
- Activities
  - Associated with a state
  - Start when the state is entered
  - Take time to complete
  - Interruptible
- Actions
  - Associated with a transition
  - Take an insignificant amount of time to complete
  - Non-interruptible



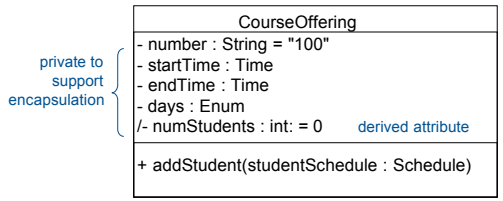
Statechart



Statechart with Nested States

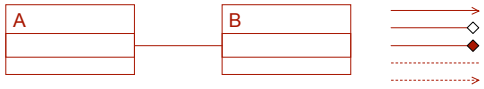


Example: Define Attributes



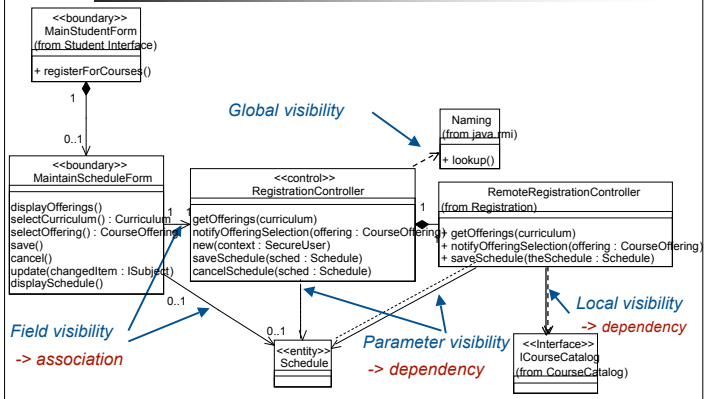
Associations & Dependencies

- association

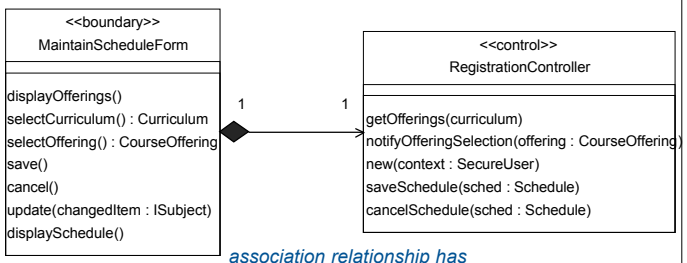


- visibility
  - attribute (field) visibility: B is an attribute of A
    - remains an association
  - parameter visibility: B is a parameter of a method A
    - becomes a dependency
  - local visibility: B is a (non-parameter) local object in a method of A
    - becomes a dependency
  - global visibility: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

Define Dependencies



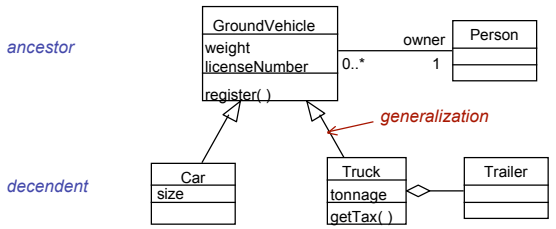
Example: Composition



association relationship has been refined into a composition relationship.
MSF represents a session
RC never exists outside of session

Generalization (Inheritance)

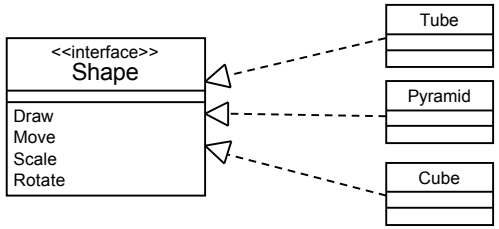
- One class inherits from another



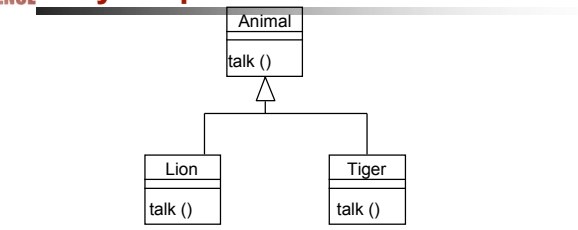
- not just finding common attribute, operations and relationships
- more about the responsibilities and essence of the classes.
- avoid “skyscrapers”; the hierarchies should look like small, independent “forests

Review: What is Polymorphism?

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



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