COMPUTER SCIENCE

21- Design: RUP

Rick Adrion

COMPUTER SCIENCE

CMPSCI 520/620
Advanced Software Engineering:
Synthesis and Development
online material: http://www-edlab.cs.umass.edu/cs520/
Calendar 11/16/03 3:47 PM

Lec	Scheduled Lecture	520/620 Reading	620 Reading	Assignment	Due Date	Date
19a	SIS Interviews					11/10/03
20	Design	Maciaszek Ch.7, 8		HW #3		11/12/03
19b	SIS Interviews					11/14/03 11/17/03
21	Design			Project #3		11/17/03
22	Design; Analyzing Products	Maciaszek Ch.10				11/19/03
23	Analyzing Products				Project #2	11/24/03
24	Analyzing Products			HW #4	HW #3	11/26/03
25	Representing & Managing Processes					12/1/03
26	Analyzing Processes					12/3/03
27	Guest Lecture or Rescheduled Class					12/8/03
28	Reuse, Evolution & Maintenance					12/10/03
	Scheduled Final Exam (**there will be no final**)				HW #4 Project #3	12/18/03

COMPUTER SCIENCE

JSD and JSP

- In JSD, the principles of JSP are extended into the areas of systems analysis, specification, design and implementation
- In JSP, a simple program describes a sequential process that communicates by means of sequential data streams; its structure is determined by the structure of its input and output data stream
- In JSD, the real world is modeled as a set of sequential model processes that communicate with the real world and with each other by sequential data streams (as well as by a second read-only communication called state vector connection). The structure of a model process is determined by the structure of its inputs and outputs.
- The JSD implementation step embodies the JSP implementation technique, program inversion, in which a program is transformed into a procedure
- Other JSP techniques, such as the single read-ahead rule and backtracking, and principles, such as implementation through transformation, are used in JSD

COMPUTER SCIENCE

Comments/Evaluation

- Focus on conceptual design
  - But difficult to build a system this way
- Based upon model of real world
- Careful (and experienced) analysis of the model generally points suggested implementation tactics, though
  - Parnas notions of module not perceptible here
  - Not an iterative refinement approach either
- Treatment of data is very much subordinated/secondary
- Does a good job of suggesting possible parallelism
- Contrasts strongly with Objected Oriented notions (eg. Booch, UML)

A Minimal Iterative Process

Getting Started: (do this once)

1. Capture the major functional and non-functional requirements for the system.
   - Express the functional requirements as use cases, scenarios, or stories.
   - Capture non-functional requirements in a standard paragraph-style document.
2. Identify the classes which are part of the domain being modeled.
3. Define the responsibilities and relationships for each class in the domain.
4. Construct the domain class diagram.
   - This diagram and the responsibility definitions lay a foundation for a common vocabulary in the project.
5. Capture use case and class definitions in an OO CASE tool (e.g., Rose) only when they have stablilized.

A Minimal Iterative Process

Getting Started: (do this once)

6. Identify the major risk factors and prioritize the most architecturally significant use cases and scenarios.
   - It is absolutely imperative that the highest risk items and the most architecturally significant functionality be addressed in the early iterations. You must not pick the "low hanging fruit" and leave the risks for later.
7. Partition the use cases/scenarios across the planned iterations.
8. Develop an Iteration plan describing each "mini-project" to be completed in each iteration.
   - Describe the goals of each iteration, plus the staffing, the schedule, the risks, inputs and deliverables.
   - Keep the iterations focused and limited (2-3 weeks per iteration). In each iteration, conduct all of the software activities in the process: requirements, analysis, design, implementation and test.

A Minimal Iterative Process

For each iteration: (repeat until done)

1. Merge the functional flow in the use cases/scenarios with the classes in the domain class diagram
   - Produce sequence (and collaboration) diagrams at the analysis level.
2. Test and challenge the sequence diagrams on paper, or whiteboard
   - Discover additional operations and data to be assigned to classes
   - Validate the business process captured in the flow of the sequence diagram
3. Develop statechart diagrams for classes with "significant" state
   - Statechart events, actions, and most activities will become operations on the corresponding class
4. Enhance sequence diagrams and statechart diagrams with design level content
   - Identify and add to the class diagram and sequence diagrams any required support or design classes (e.g. collection classes, GUI and other technology classes, etc.)
5. Challenge the sequence diagrams on paper/whiteboard, discovering additional operations and data assigned to classes.

A Minimal Iterative Process

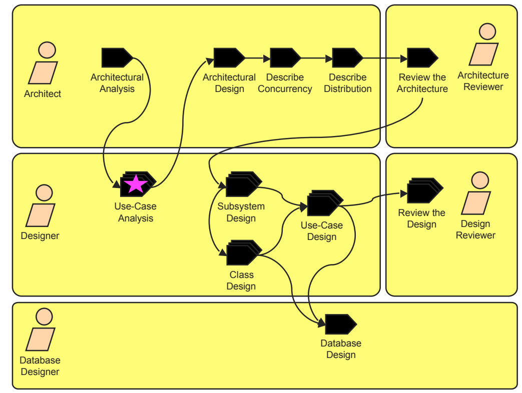
For each iteration: (repeat until done)

6. Update the OO CASE tool information as models stabilize, and if the there is a good reason to save them.
   - Update class diagrams: add in discovered datatypes, message names, actual functions and arguments, actual return types. These are discovered especially in the design level sequence and statechart diagrams.
   - Add or modify classes as necessary
   - Republish system reports for team members
7. Develop the code for the use cases/scenarios in the current iteration from the current diagrams
8. Test the code in the current iteration. (In a test-then-code approach this step precedes #7.)
9. Conduct an Iteration review:
   - What went wrong? What went right? Re-evaluate the iteration plan, and content of next iteration
   - Revise the next iteration plan if necessary
   - Revise the Project Plan if necessary
10. Conduct the next iteration, adding in the next set of use cases/scenarios, until the system is completely built.

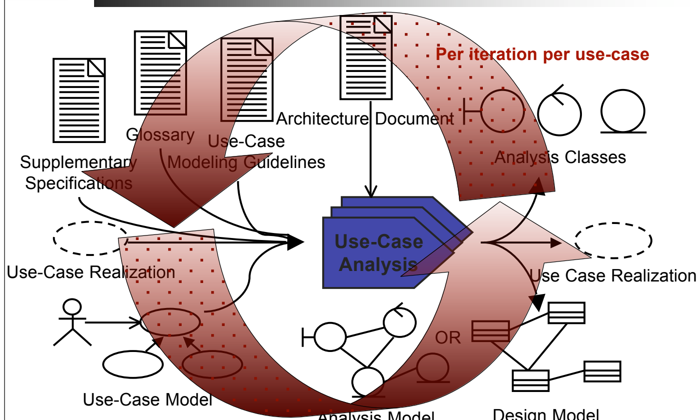
Rational Unified Process

adapted from
OOAD Using the UML

So where do we start?



Use Case Analysis Overview

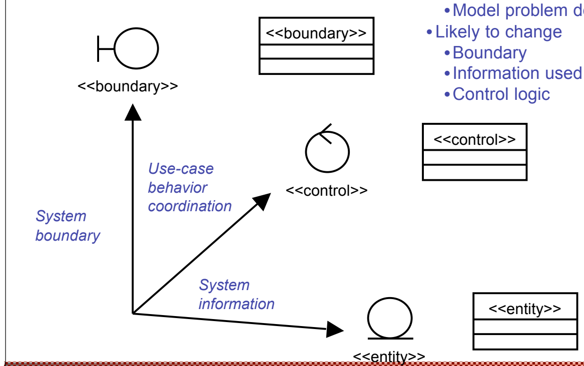


Use Case Analysis Steps

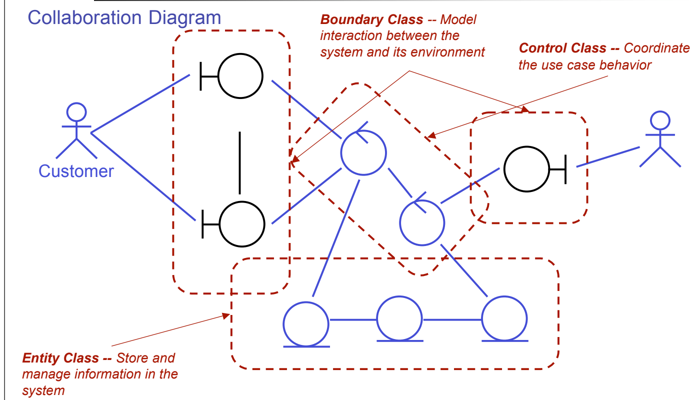
- Supplement the Descriptions of the Use Case
- For each use case realization
  - Find Classes from Use-Case Behavior
  - Distribute Use-Case Behavior to Classes
- For each resulting analysis class
  - Describe Responsibilities
  - Describe Attributes and Associations
  - Qualify Analysis Mechanisms
- Unify Analysis Classes

What is an Analysis Class?

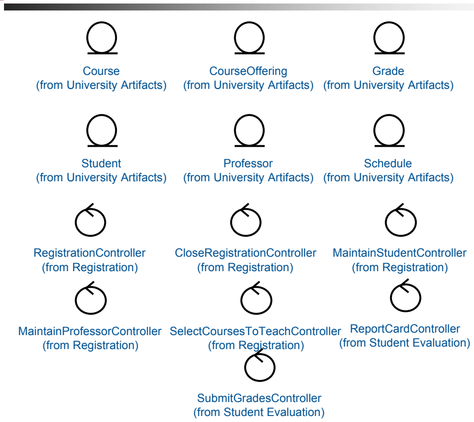
- Early conceptual model
  - Functional requirements
  - Model problem domain
- Likely to change
  - Boundary
  - Information used
  - Control logic



The Roles



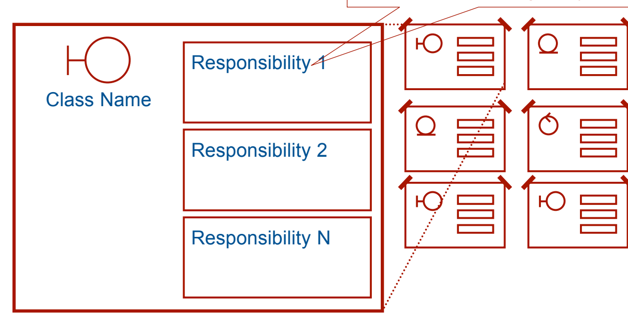
Example: Entity & Control Classes



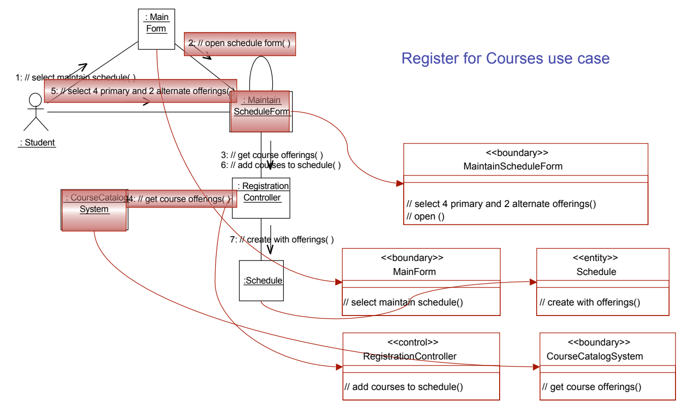
Describe Responsibilities

- What are responsibilities?
- How do we find them?

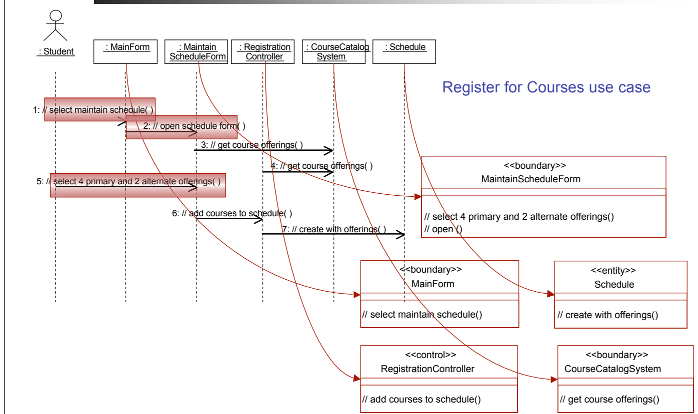
- First cut at class operations
  - Actions that object can perform
  - Knowledge object maintains
  - Non-functional requirements
- Class should have **multiple** responsibilities



Class Responsibilities from a Collaboration Diagram

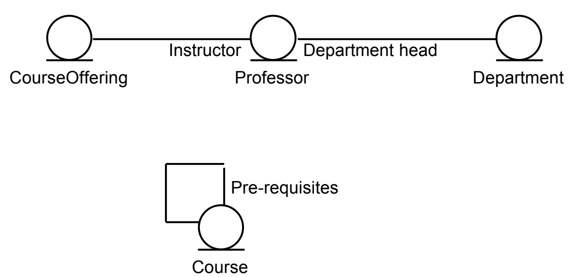


Class Responsibilities from a Sequence Diagram

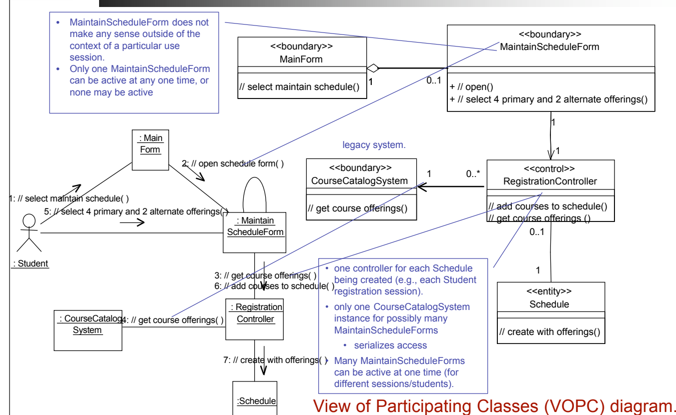


What are Roles?

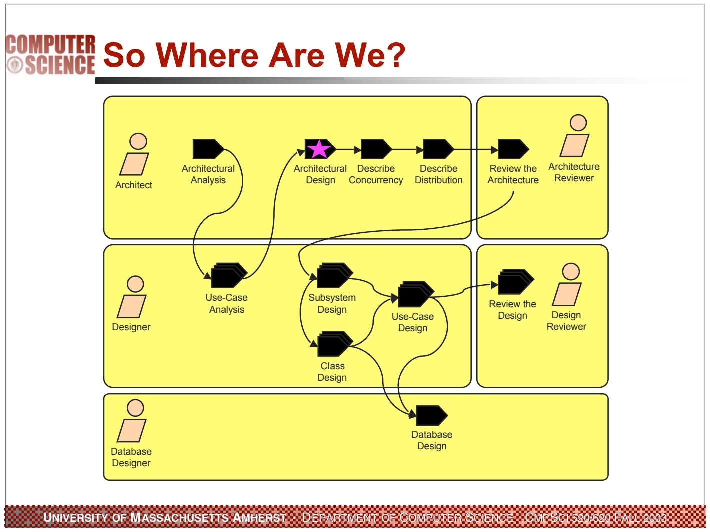
- The "face" that a class plays in the association

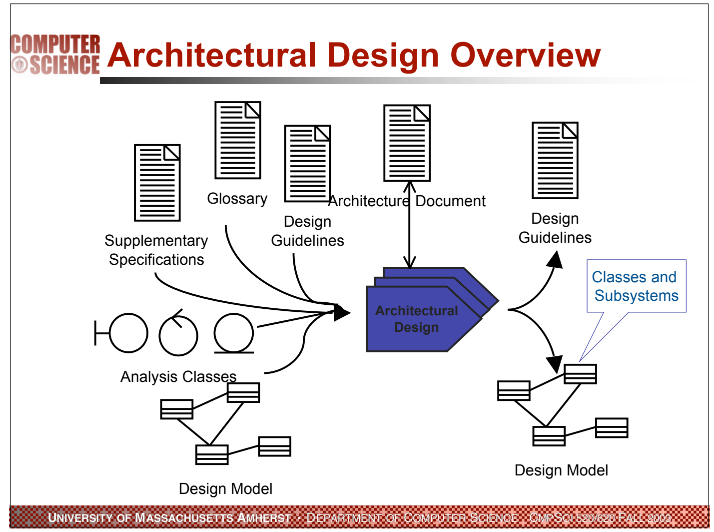


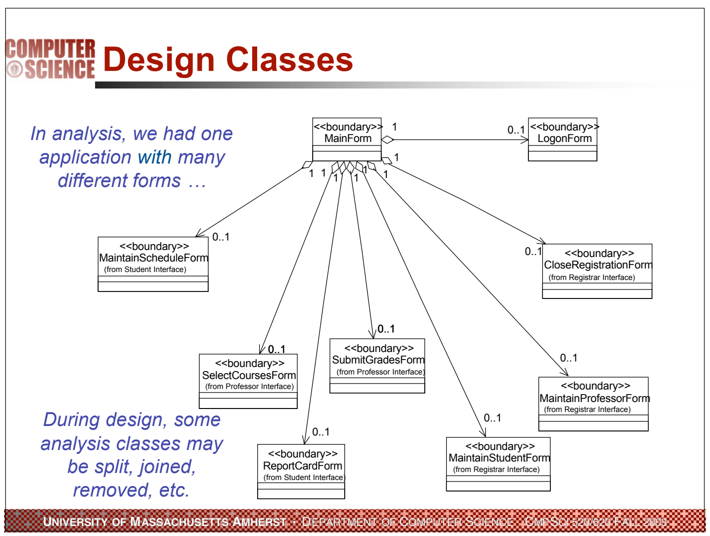
Example: Finding Relationships

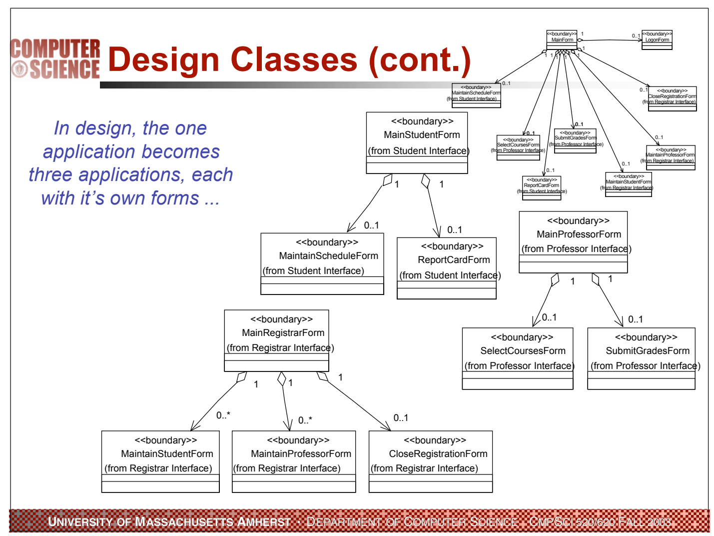


View of Participating Classes (VOPC) diagram.


COMPUTER SCIENCE
So Where Are We?
Architect
Architectural Analysis
Architectural Design
Describe Concurrency
Describe Distribution
Review the Architecture
Architecture Reviewer
Designer
Use-Case Analysis
Subsystem Design
Use-Case Design
Class Design
Review the Design
Design Reviewer
Database Designer
Database Design
UNIVERSITY OF MASSACHUSETTS AMHERST

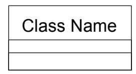

COMPUTER SCIENCE
Architectural Design Overview
Glossary
Architecture Document
Supplementary Specifications
Design Guidelines
Design Guidelines
Architectural Design
Classes and Subsystems
Analysis Classes
Design Model
Design Model
UNIVERSITY OF MASSACHUSETTS AMHERST


COMPUTER SCIENCE
Design Classes
In analysis, we had one application with many different forms …
<<boundary>> MainForm
<<boundary>> LogonForm
<<boundary>> MaintainScheduleForm (from Student Interface)
<<boundary>> CloseRegistrationForm (from Registrar Interface)
<<boundary>> SelectCoursesForm (from Professor Interface)
<<boundary>> SubmitGradesForm (from Professor Interface)
<<boundary>> MaintainProfessorForm (from Registrar Interface)
<<boundary>> ReportCardForm (from Student Interface)
<<boundary>> MaintainStudentForm (from Registrar Interface)
During design, some analysis classes may be split, joined, removed, etc.
UNIVERSITY OF MASSACHUSETTS AMHERST

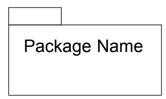

COMPUTER SCIENCE
Design Classes (cont.)
In design, the one application becomes three applications, each with it's own forms …
<<boundary>> MainStudentForm (from Student Interface)
<<boundary>> MaintainScheduleForm (from Student Interface)
<<boundary>> ReportCardForm (from Student Interface)
<<boundary>> MainProfessorForm (from Professor Interface)
<<boundary>> SelectCoursesForm (from Professor Interface)
<<boundary>> SubmitGradesForm (from Professor Interface)
<<boundary>> MainRegistrarForm (from Registrar Interface)
<<boundary>> MaintainStudentForm (from Registrar Interface)
<<boundary>> MaintainProfessorForm (from Registrar Interface)
<<boundary>> CloseRegistrationForm (from Registrar Interface)
UNIVERSITY OF MASSACHUSETTS AMHERST

Classes & packages

- What is a class?
  - A description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.



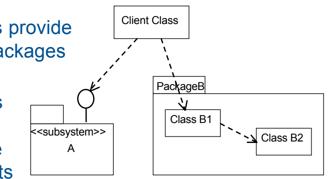
- What is a package?
  - A general purpose mechanism for organizing elements into groups
  - A model element which can contain other model elements



Packages Vs. Subsystems

- Packages provide no behavior
- Packages are simply containers of things which provide behavior
- Packages help organize and control sets of classes that are needed in common, but which aren't really subsystems
- Dependencies are on specific elements within the Package

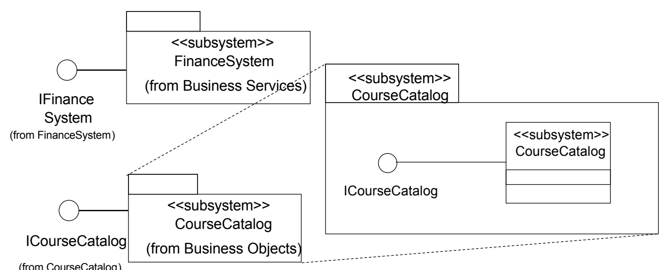
- Subsystems provide behavior, packages do not
- Subsystems completely encapsulate their contents
- Dependencies are on the interface of the subsystem
- Subsystems are easily replaceable



*Encapsulation is the key! But note for packages dependencies should be on **public** classes*

Modeling Design Subsystems

Note: Rose does not fully support subsystems

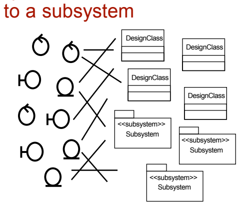


<<subsystem>> package = package with a stereotype of <<subsystem>>

<<subsystem>> proxy class = class with a stereotype of <<subsystem>>

Design Classes and Subsystems

- Identifying Design Classes
  - analysis class is simple and already represents a single logical abstraction-> design class
  - entity classes survive relatively intact into design.
- Identifying Subsystems
  - analysis class is complex, such that it appears to embody behaviors that cannot be the responsibility of a single class acting alone, or the responsibilities may need to be reused, the analysis class should be mapped to a subsystem
  - may take a few iterations to stabilize.
- Analysis classes which evolve into subsystems might include:
  - complex services and/or utilities
  - user interfaces and external system interfaces.



Design goals

- Properties of a system which make it flexible, maintainable
  - Abstraction
  - Modularity
    - Cohesion
      - how clearly-defined a particular module or procedure is
      - a module with high cohesion does one or a few things exceedingly well.
    - Coupling
      - strength of connections between modules
      - what information needs to be communicated between modules
    - Goal: High cohesion, low coupling
  - Information hiding
  - Complexity

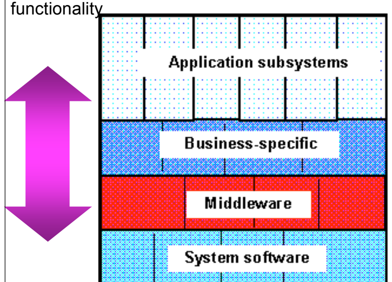
Partitioning Considerations

- Coupling and cohesion
  - design elements with tight coupling/cohesion (e.g., lots of relationships and communication) should be should be placed in the same partition
  - design elements with loose coupling/cohesion should be placed in separate partitions.
- User organization
  - not a good long-term strategy because the organizational structure may change
  - you want the software and the business organization to be independent
- System distribution
  - partitioning to reflect distribution can help to visualize the network communication which will occur as the system executes., but can make the system more difficult to change if the Deployment Model changes significantly.
- Secrecy & access control
  - functionality requiring special clearance must be partitioned into subsystems that will be developed independently, with the interfaces to the secrecy areas the only visible aspect of these subsystems.
- Variability
  - partition “optional” functionality

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Typical Layering Approach

Specific functionality



Distinct application subsystem that make up an application - contains the value adding software developed by the organization.

Business specific - contains a number of reusable subsystems specific to the type of business.

Middleware - offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.

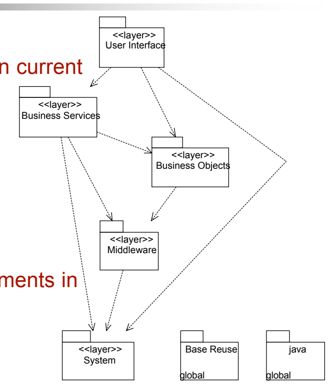
System software - contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers and so on.

General functionality

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Layering Guidelines

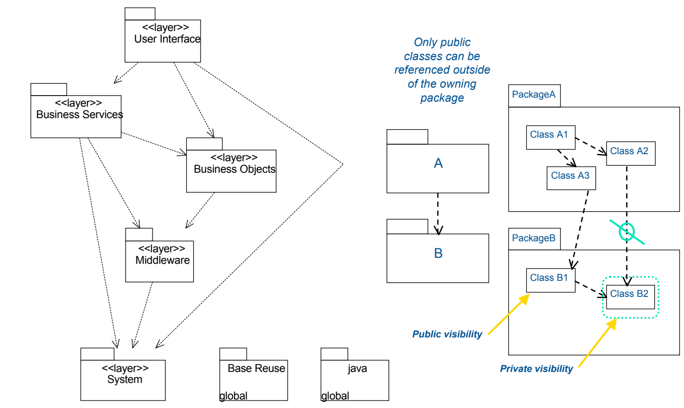
- Visibility
  - Dependencies only within current layer and below
- Volatility
  - Upper layers affected by requirements changes
  - Lower layers affected by environment changes
- Generality
  - More abstract model elements in lower layers
- Number of layers
  - Small system: 3 layers
  - Complex system: 5-7 layers



Goal is to reduce coupling and to ease maintenance effort

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

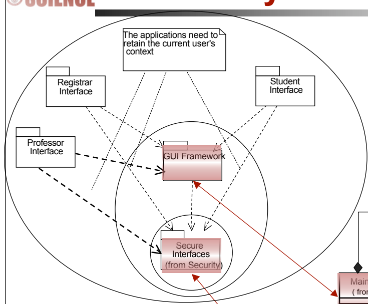
Layers & Visibility



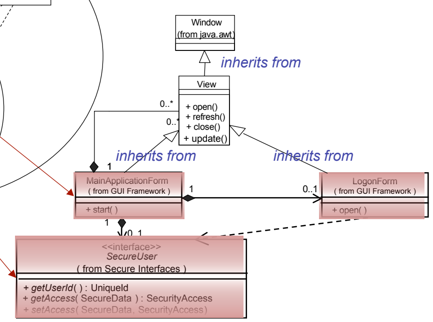
Layering

- Concentrate on encapsulating change
- Package dependencies are not transitive, thus one layer can shield another from change
- Upward dependencies should be resolved in design
  - e.g., call backs can be replaced with the “subscribes to” association whose source is a class (called the subscriber) and whose target is a class (called the publisher)
    - subscriber specifies a set of events and is notified when one of those events occurs in the target

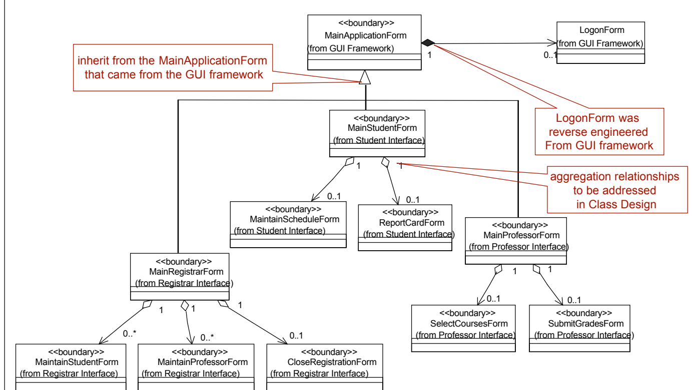
Back to layers



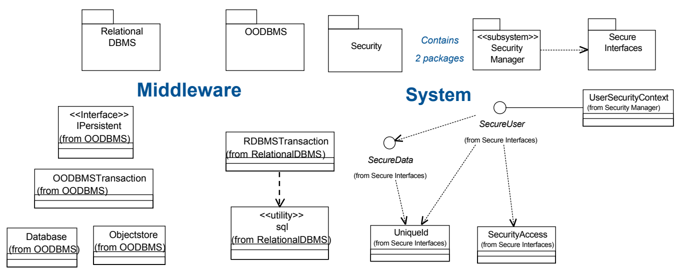
A bi-directional relationship exists between the GUI Framework and the other interface packages because the Logon Form needs to be able to notify the application forms



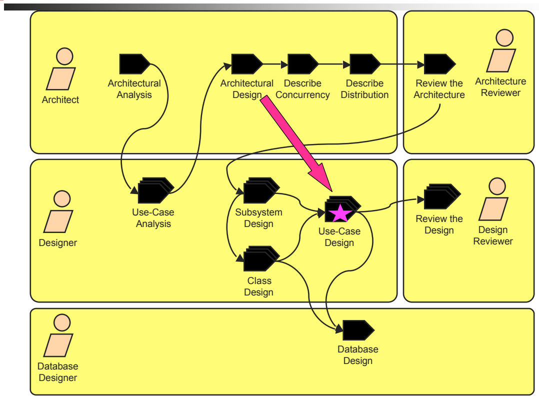
User Interface Layer: Main Forms



More Layers



So Where Are We?

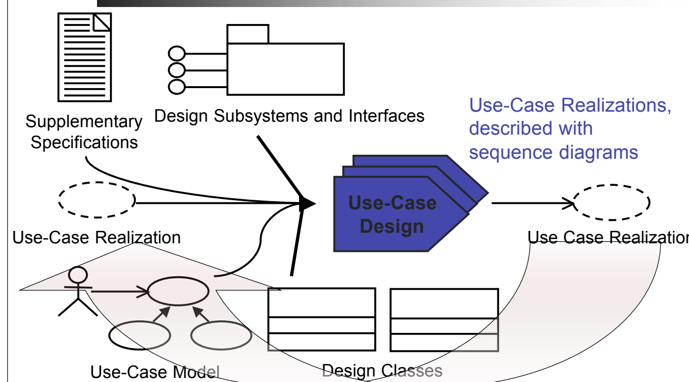


A look ahead to Use Case Design

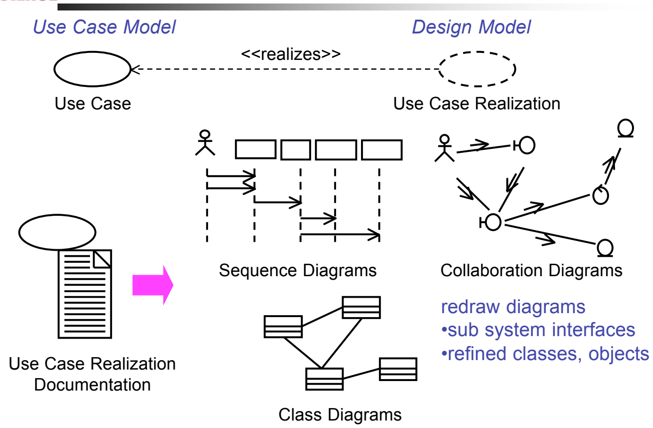
- Use-Case design vs. analysis
  - in analysis, the classes we discovered are “large” to keep the model “small” so we can understand the interactions (and diagrams)
  - in design, flesh out the class structure (“look inside”) to add design elements to implement the publicly visible behaviors, **but** defer subsystem design to the subsystem designers
- We have:
  - an initial architectural definition
  - defined the major elements of our system (e.g., the subsystems, their interfaces, the design classes, the processes and threads) and their relationships, and we have an understanding of how these elements map into the hardware on which the system will run.
- In Use Case Design, concentrate on how a use case has been implemented and make sure that there is consistency from beginning to end, and that nothing has been missed

tend to alternate between Subsystem Design, Class Design and Use Case Design

Use Case Design Overview

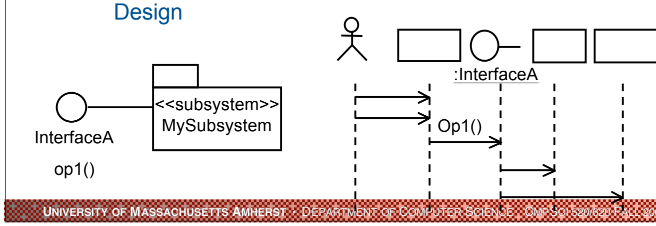


Use Case Realization



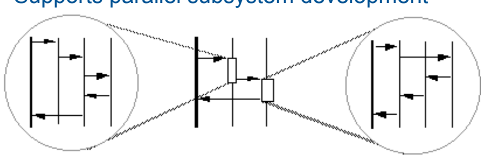
Encapsulating Subsystem Interactions

- 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



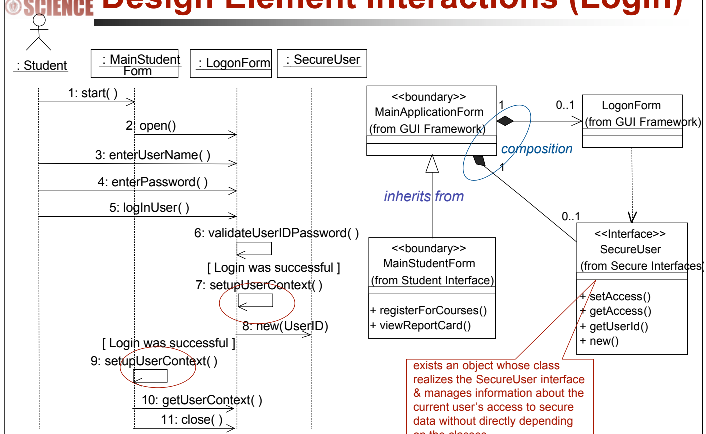
Advantages of Encapsulation

- Use-case realizations are less cluttered
- Use-case realizations can be created before the internal designs of subsystems are created
- Use-case realizations are more generic and easy to change
- Supports parallel subsystem development

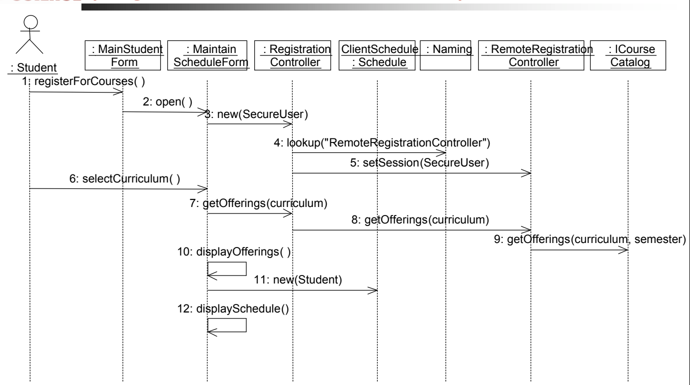


Raises the level of abstraction

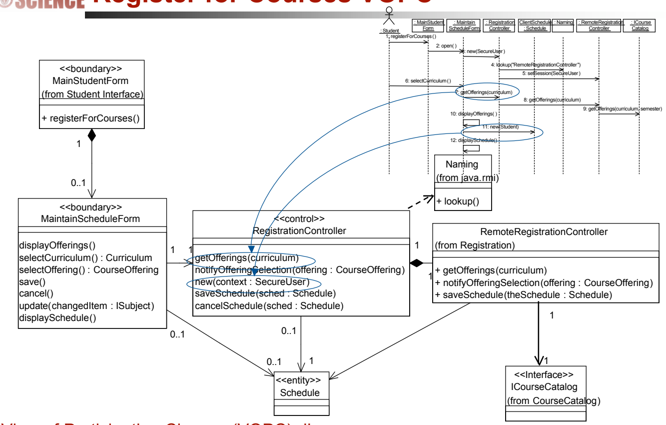
Design Element Interactions (Login)



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

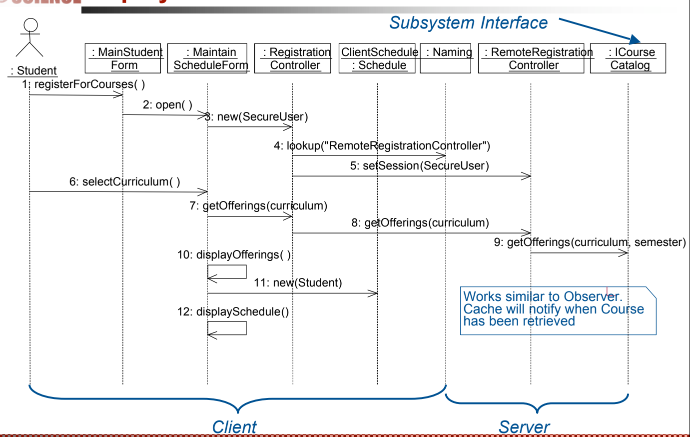


Example: Design Classes Relationships: Register for Courses VOPC



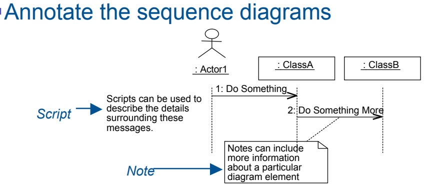
View of Participating Classes (VOPC) diagram.

Deployment

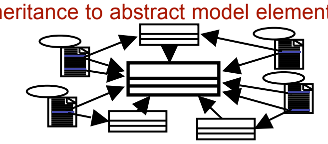


More steps

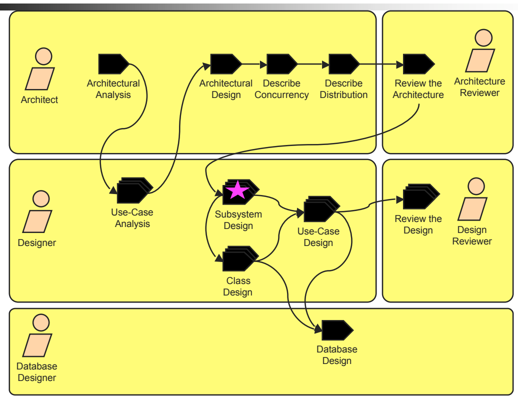
- Annotate the sequence diagrams



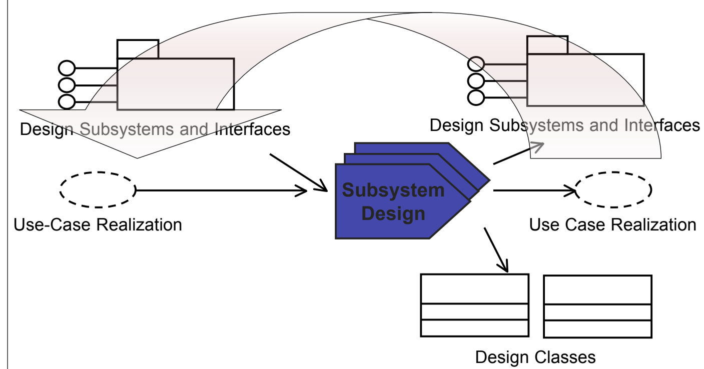
- Unify classes & subsystems
  - merge similar model elements
  - use inheritance to abstract model elements



So Where Are We?



Subsystem Design Overview



UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

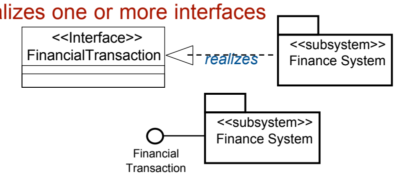
Subsystem design

- 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

UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

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



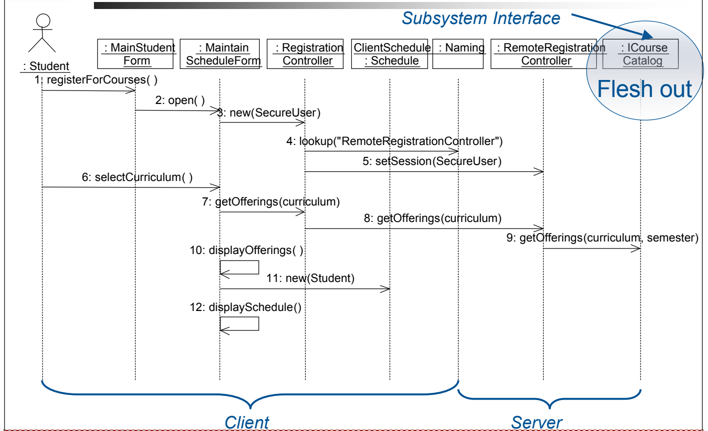
Recall: packages provide no behavior; they are simply containers of things which provide behavior

UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

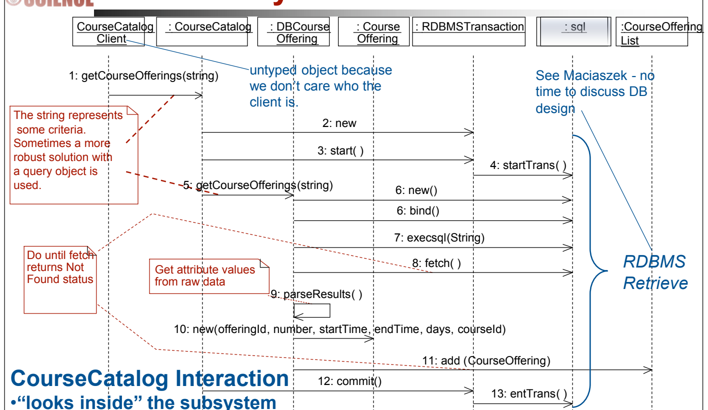
Distribute Subsystem Responsibilities

- 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

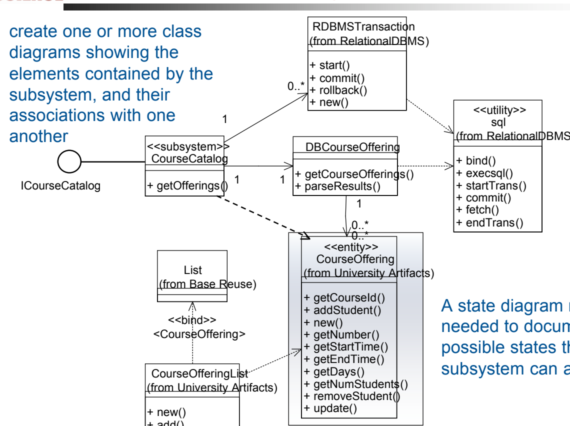


CourseCatalog Interaction

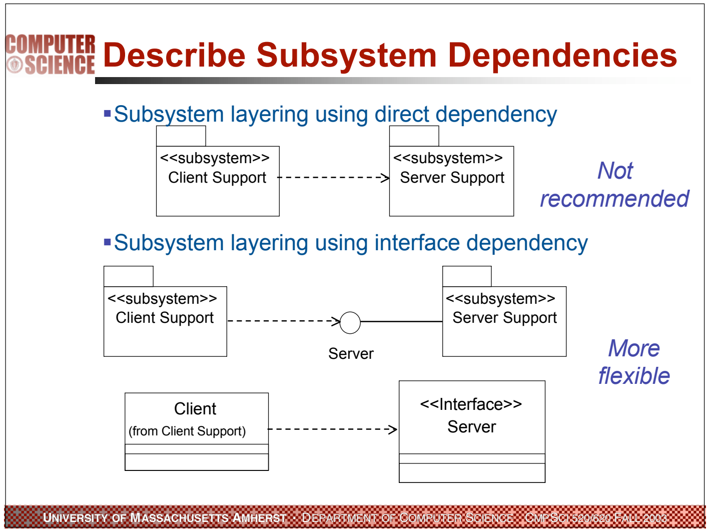
- **"looks inside" the subsystem**
- **one or more per subsystem**

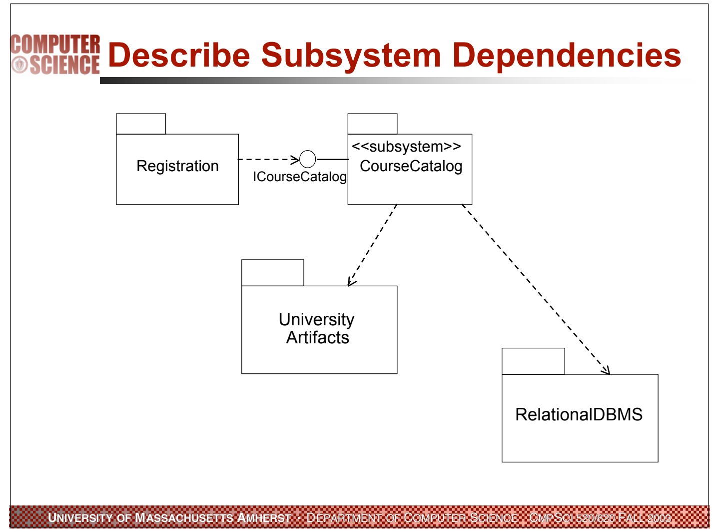
Document Subsystem Elements

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


COMPUTER SCIENCE
Describe Subsystem Dependencies
Subsystem layering using direct dependency
<<subsystem>> Client Support
<<subsystem>> Server Support
Not recommended
Subsystem layering using interface dependency
<<subsystem>> Client Support
Server
<<subsystem>> Server Support
More flexible
Client
(from Client Support)
<<Interface>> Server
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003


COMPUTER SCIENCE
Describe Subsystem Dependencies
Registration
ICourseCatalog
<<subsystem>> CourseCatalog
University Artifacts
RelationalDBMS
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003