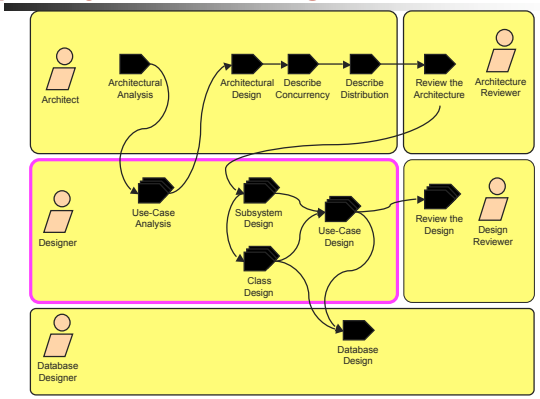
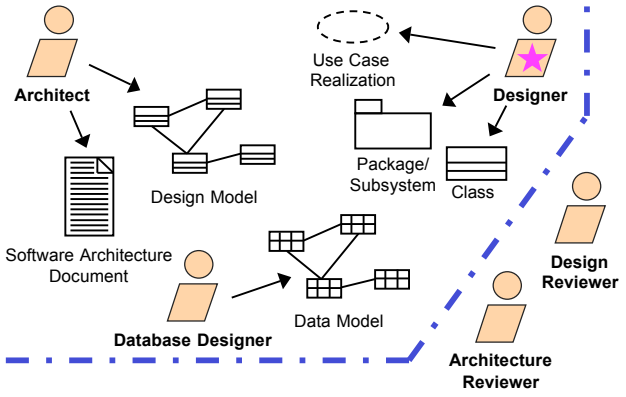
18-Design

- Readings
  - OOAD Using the UML

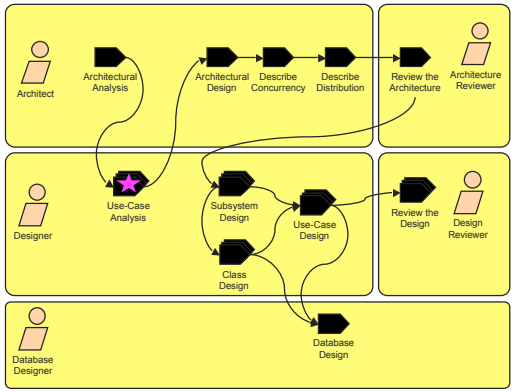
Analysis and Design Workflow



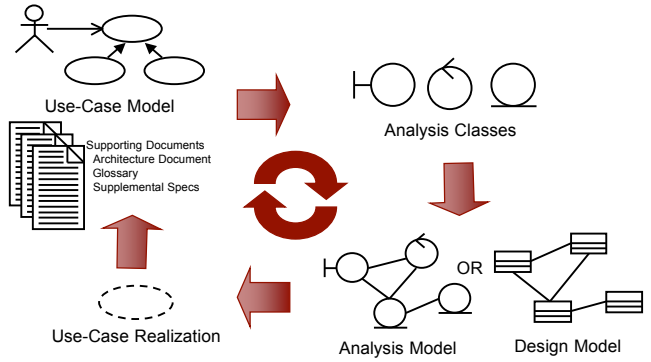
Worker Responsibilities



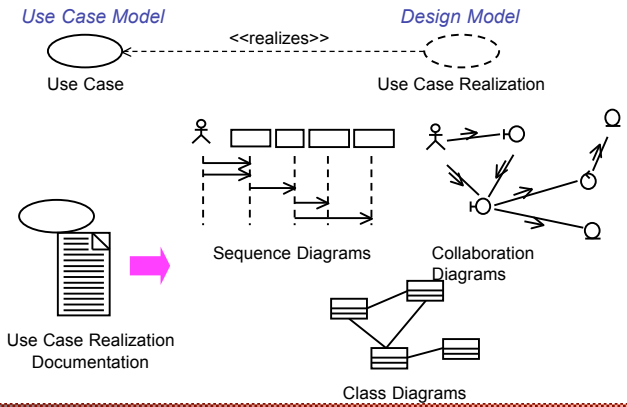
So what do we do next?



Use Case Analysis Overview



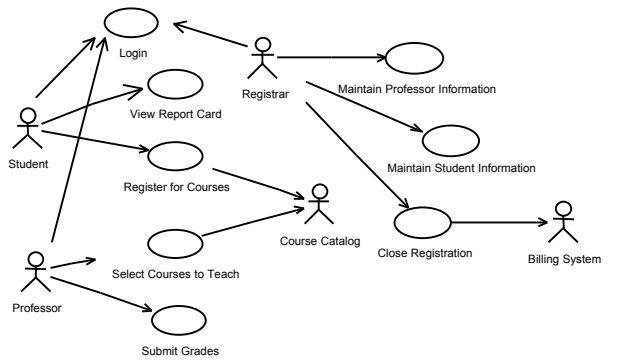
What is a Use-Case Realization?



Use Case Analysis Steps

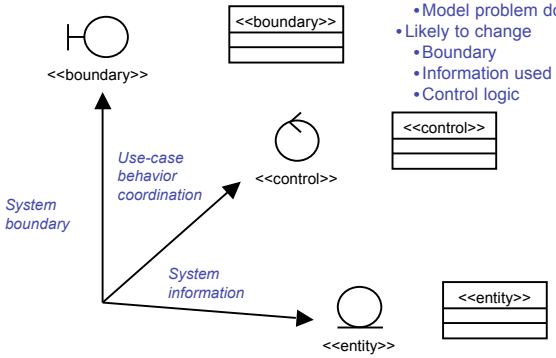
- Supplement the Descriptions of the Use Case
- For each use case realization
  - Find Classes from Use-Case Behavior
  - Distribute Use-Case Behavior to Classes
- For each resulting analysis class
  - Describe Responsibilities
  - Describe Attributes and Associations
  - Qualify Analysis Mechanisms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

University Course Registration System

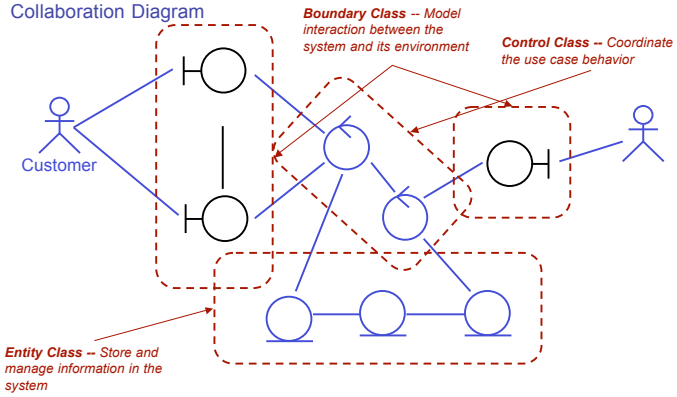


What is an Analysis Class?

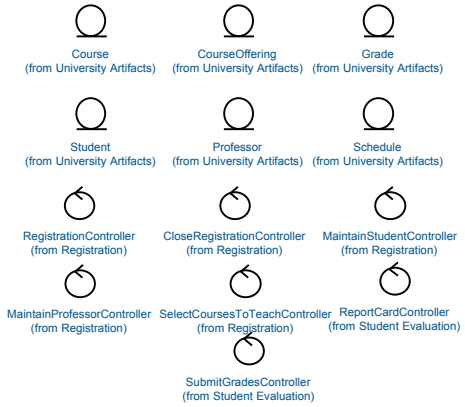
- Early conceptual model
  - Functional requirements
  - Model problem domain
- Likely to change
  - Boundary
  - Information used
  - Control logic



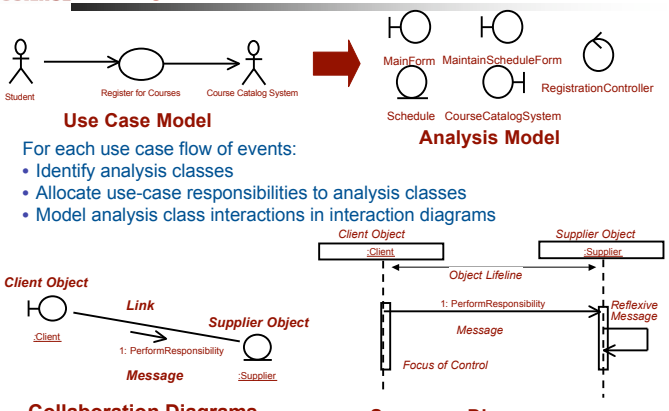
The Roles



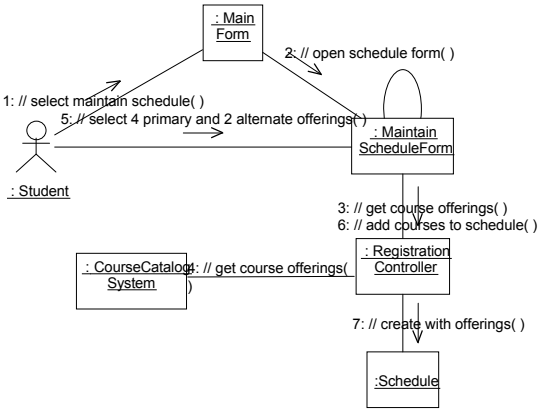
Example: Entity & Control Classes



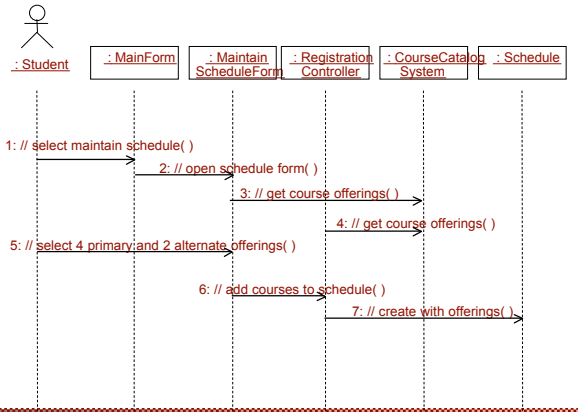
Analysis Classes



Example: Collaboration Diagram



Example: Sequence Diagram

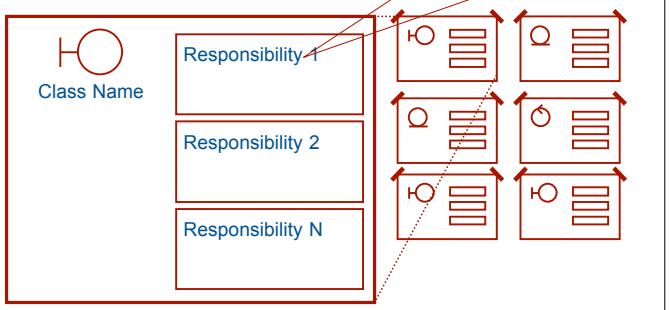


Collaboration Vs Sequence Diagrams

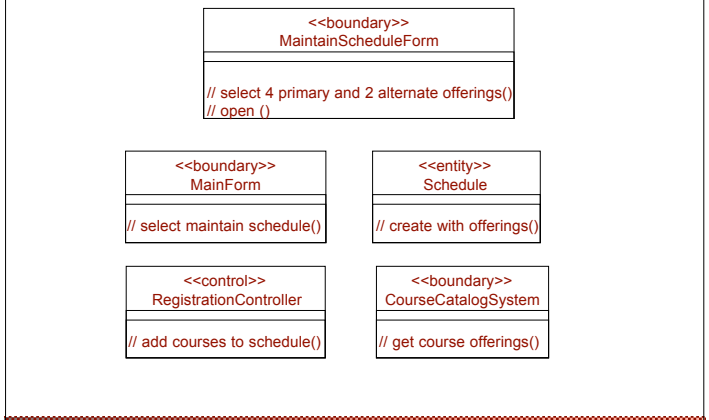
- Collaboration Diagrams
  - Use in Use Case Analysis
  - Show relationships in addition to interactions
  - Better for visualizing patterns of collaboration
  - Better for visualizing all of the effects on a given object
  - Easier to use for brainstorming sessions (reordering of messages easier)
- Sequence Diagrams
  - Use in Use Case Design
  - Show the explicit sequence of messages
  - Better for visualizing overall flow
  - Better for real-time specifications and for complex scenarios

Describe Responsibilities

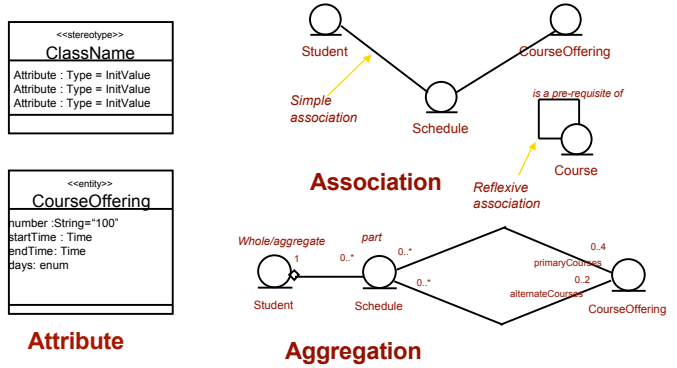
- What are responsibilities?
- How do we find them?
  - First cut at class operations
    - Actions that object can perform
    - Knowledge object maintains
    - Non-functional requirements
  - Class should have **multiple** responsibilities



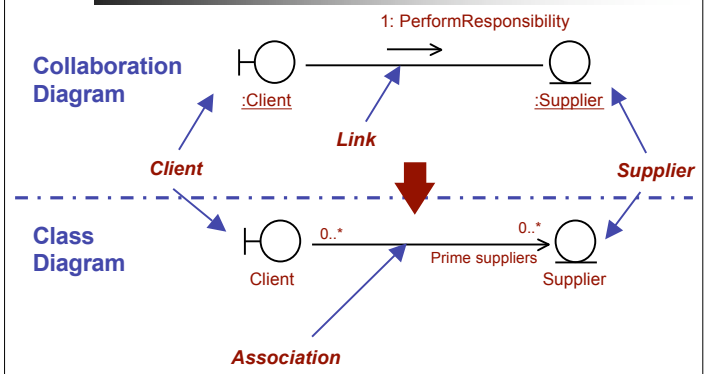
Class Responsibilities



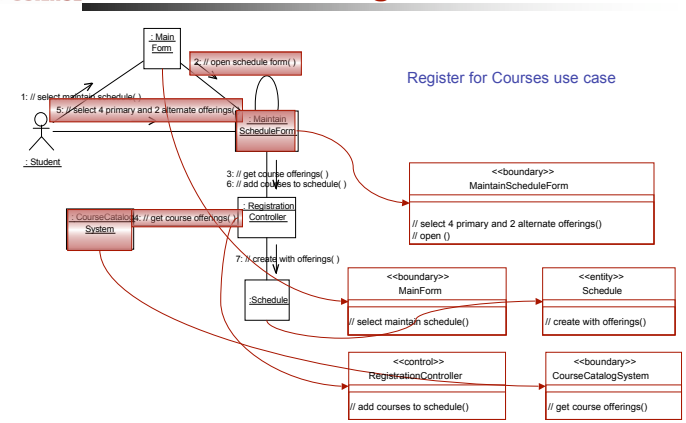
Attributes, Association, Aggregation



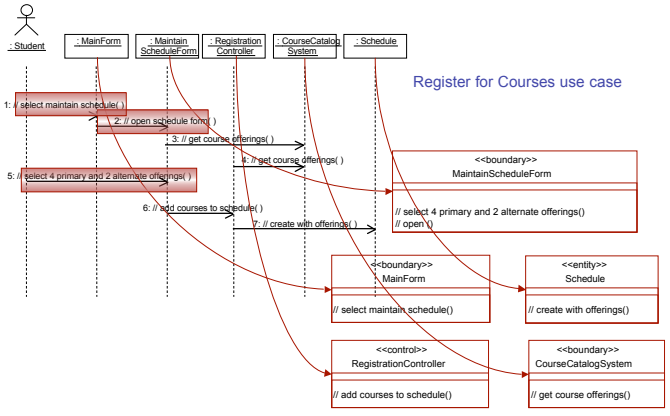
Finding Relationships



Class Responsibilities from a Collaboration Diagram

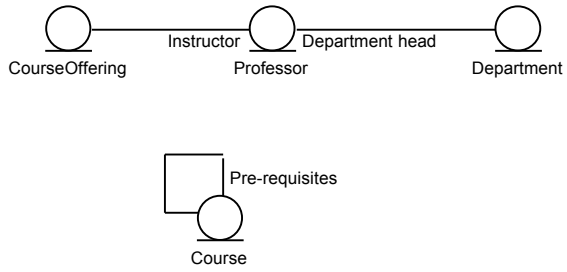


Class Responsibilities from a Sequence Diagram

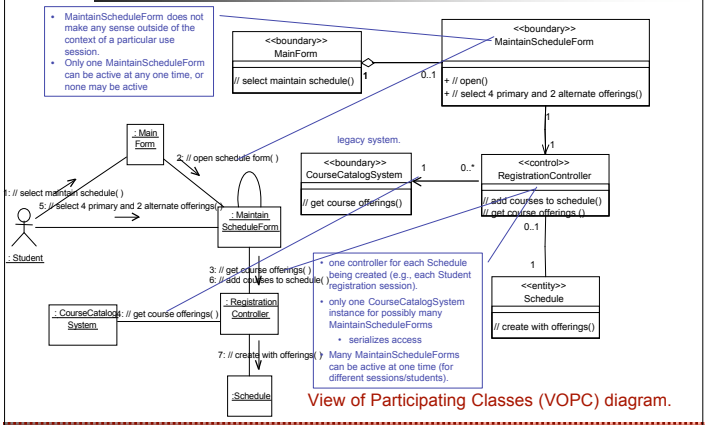


What are Roles?

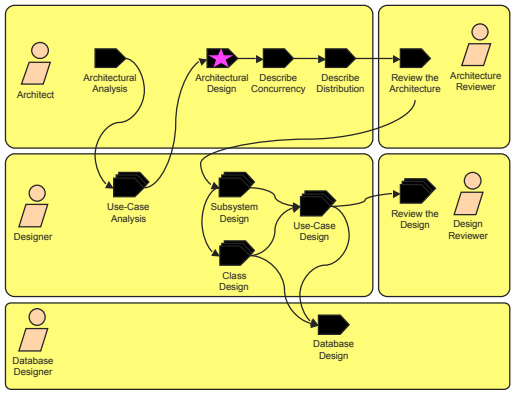
- The "face" that a class plays in the association



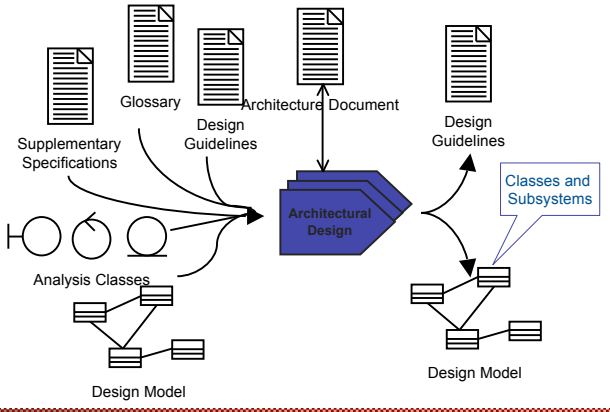
Example: Finding Relationships



So Where Are We?

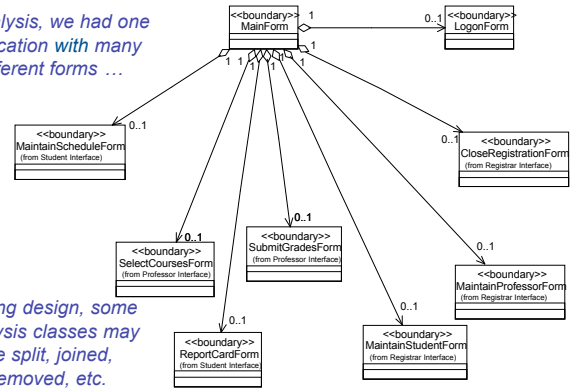


Architectural Design Overview



Design Classes

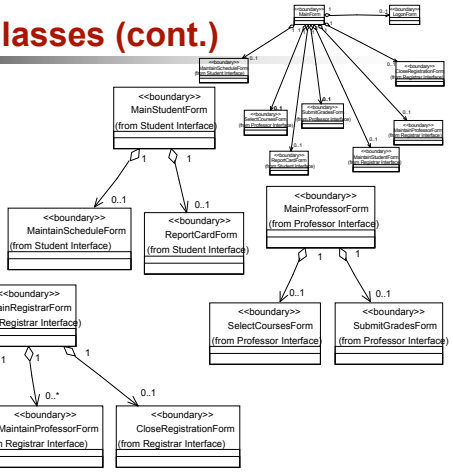
In analysis, we had one application with many different forms …



During design, some analysis classes may be split, joined, removed, etc.

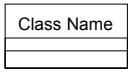
Design Classes (cont.)

In design, the one application becomes three applications, each with it's own forms ...

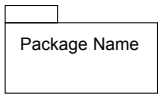


Classes & packages

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



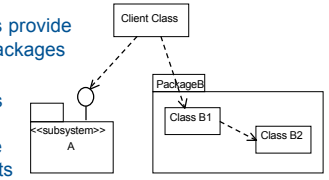
- What is a package?
  - A general purpose mechanism for organizing elements into groups
  - A model element which can contain other model elements



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

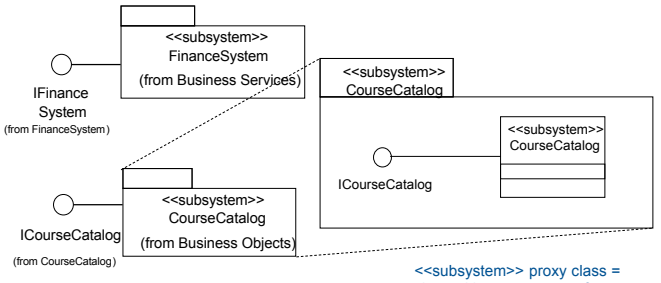
- Subsystems provide behavior, packages do not
- Subsystems completely encapsulate their contents
- Dependencies are on the interface of the subsystem
- Subsystems are easily replaceable



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

Modeling Design Subsystems

Note: Rose does not fully support subsystems

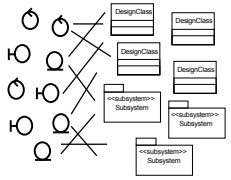


<<subsystem>> package = package with a stereotype of <<subsystem>>

<<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.



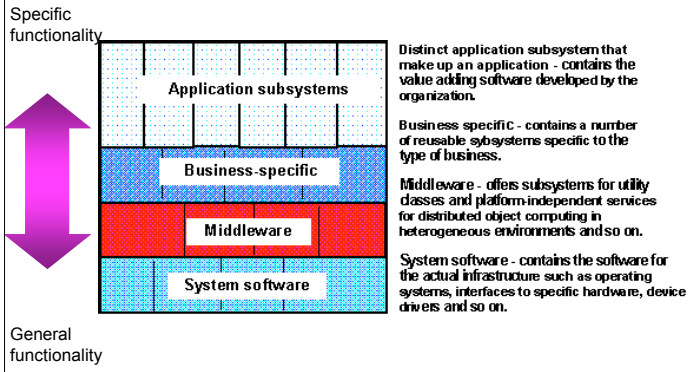
Design goals

- 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

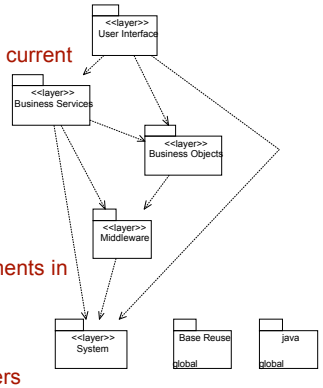
- Coupling and cohesion
  - design elements with tight coupling/cohesion (e.g., lots of relationships and communication) should be should be placed in the same partition
  - design elements with loose coupling/cohesion should be placed in separate partitions.
- User organization
  - not a good long-term 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

Typical Layering Approach



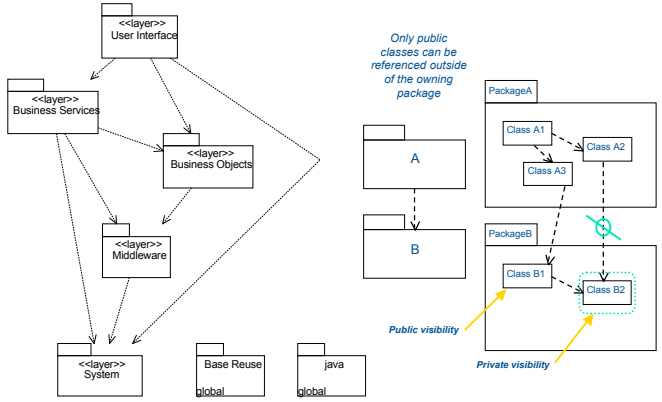
Layering Guidelines

- 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

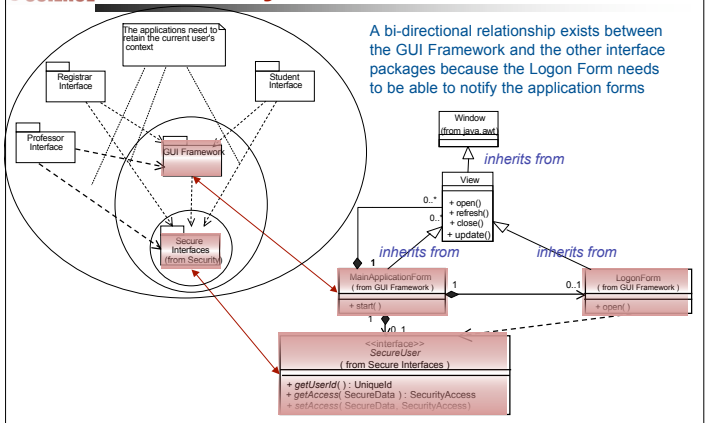
Layers & Visibility



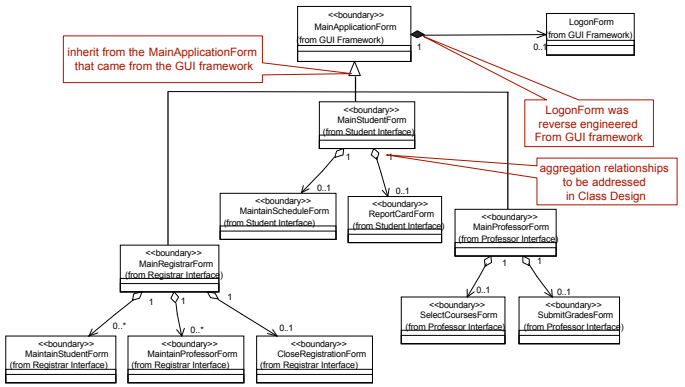
Layering

- 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

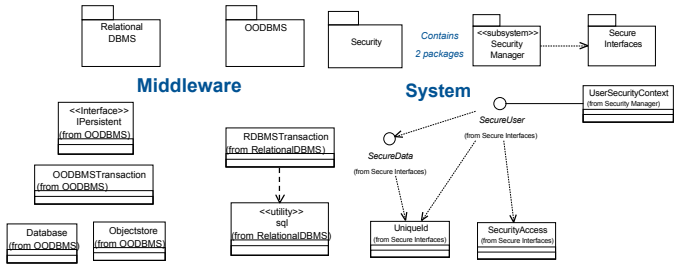
Back to layers



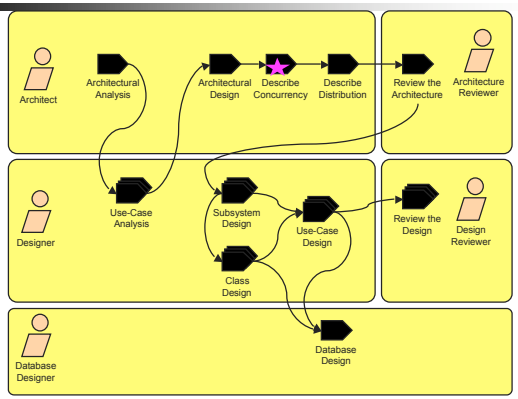
User Interface Layer: Main Forms



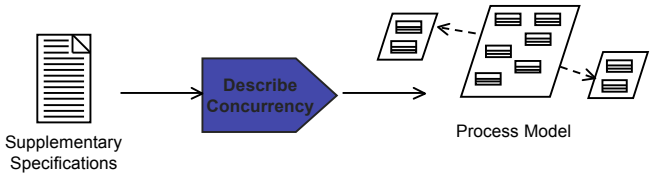
More Layers



So Where Are We?



Describe Concurrency Overview



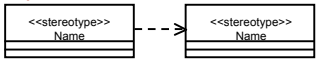
- the independent threads of control are identified and the identified model elements (subsystems and classes) are mapped to these threads of control.

Modeling Processes

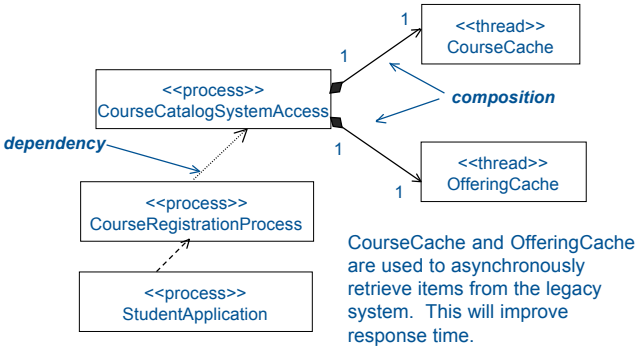
- Processes may be modeled using
  - Active classes (Class Diagrams) and Objects (Interaction Diagrams)
    - "owns" it's own thread of execution and can initiate control activity
    - may execute in parallel (i.e., concurrently) with other active classes.

  - Components (Component Diagrams)

  - Relationships



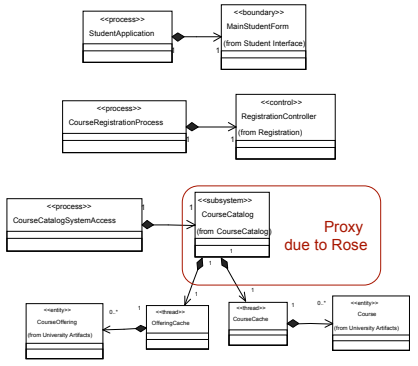
Modeling Processes: Class Diagram



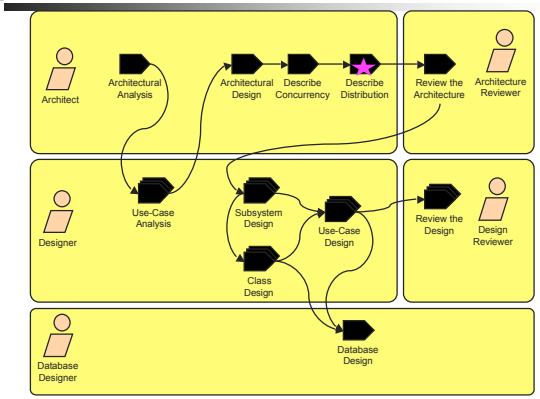
CourseCache and OfferingCache are used to asynchronously retrieve items from the legacy system. This will improve response time.

Mapping Design Elements to Processes

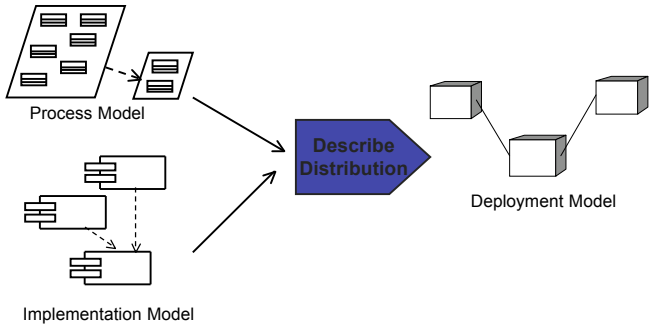
- The classes associated with the individual user interfaces were mapped to the application processes.
- The classes associated with the individual business services were mapped to the controller processes.
- The classes associated with access to the external systems were mapped to the access processes
- 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?

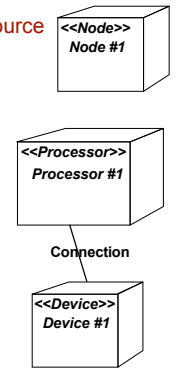
- Reduce processor load
- Special processing requirements
- Scaling concerns
- Economic concerns
- Distribution Patterns
  - Client/Server
    - 3-tier
    - Fat-Client
    - Web Application
    - Distributed Client/Server
  - Peer-to-peer

Distribution patterns

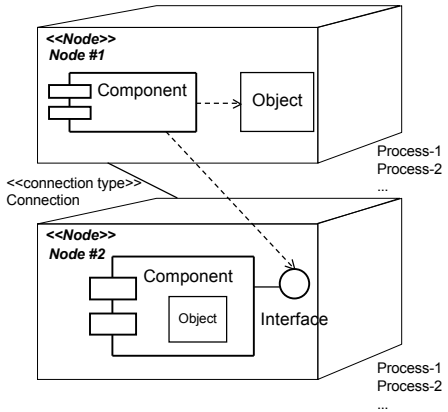
- Common services
  - Presentation Services
    - UI including, the visual appearance of output and how user input is handled
  - Business Services
    - Business rules and logic
  - Data Services
    - Data relationships, efficiency of storage, and data integrity
- Patterns
  - One-Tier
  - Two-Tier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

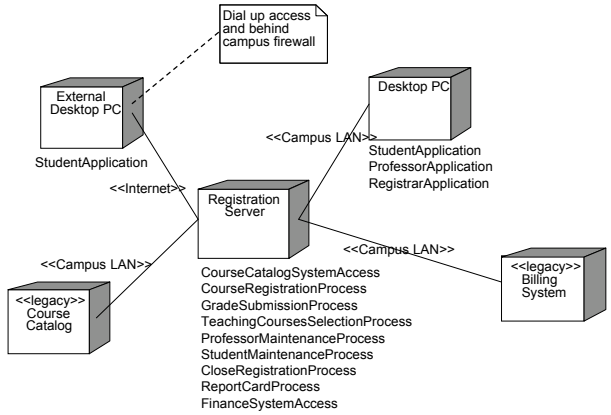
- Node
  - Physical run-time computational resource
- Processor
  - Execute system software
- Device
  - Support devices
  - Typically controlled by a Processor
- Connection
  - Communication mechanisms
  - Physical medium
  - Software protocol



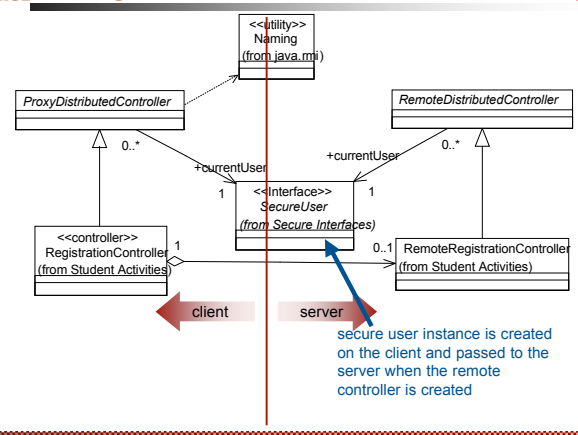
Deployment Diagrams



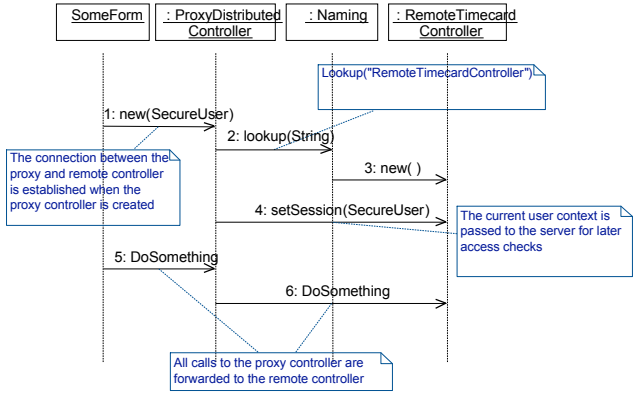
Process-to-Node Allocation



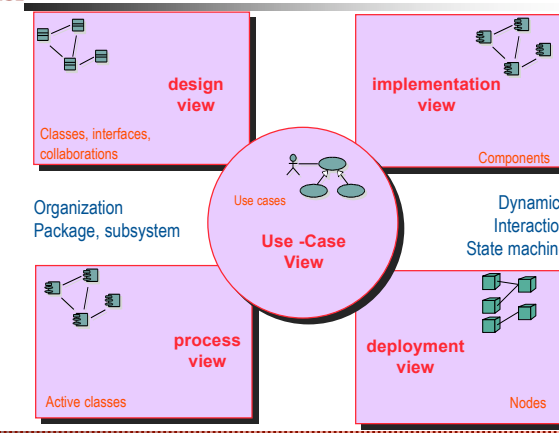
Design Distribution Pattern: Proxy



Design Distribution Pattern: Proxy



The Rational 4+1 Views



Affect on Process Model Associations