

Introduction to UML: The Unified Modeling Language

What is the UML?

- UML stands for Unified Modeling Language
- The UML combines the best of the best from
 - Data modeling concepts (entity relationship diagrams)
 - Business modeling (work flow diagrams)
 - Object modeling
 - Component modeling
- The UML is a language standard for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system
- It can be used with all processes, throughout the development life cycle, and across different implementation technologies

UML Concepts

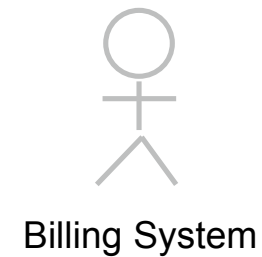
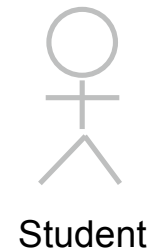
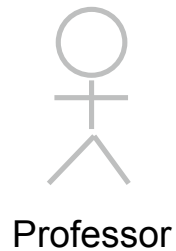
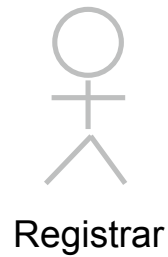
- The UML may be used to:
 - Display the boundary of a system & its major functions using use cases and actors
 - Illustrate use case realizations with interaction diagrams
 - Represent a static structure of a system using class diagrams
 - Model the behavior of objects with state transition diagrams
 - Reveal the physical implementation architecture with component & deployment diagrams

Putting the UML to Work

- A University wants to computerize its registration system
 - The Registrar sets up the curriculum for a semester
 - One course may have multiple course offerings
 - Students select 4 primary courses and 2 alternate courses
 - Once a student registers for a semester, the billing system is notified so the student may be billed for the semester
 - Students may use the system to add/drop courses for a period of time after registration
 - Professors use the system to receive their course offering rosters
 - Users of the registration system are assigned passwords which are used at logon validation

Actors

- An actor is someone or something that must interact with the system under development



Use Cases

- A use case is a pattern of behavior the system exhibits
 - Each use case is a sequence of related transactions performed by an actor and the system in a dialogue towards achieving a set of related services
- Actors are examined to determine their needs
 - Registrar -- maintain the curriculum
 - Professor -- request roster
 - Student -- maintain schedule
 - Billing System -- receive billing information from registration



Maintain Curriculum



Request Course Roster



Maintain Schedule

Documenting Use Cases

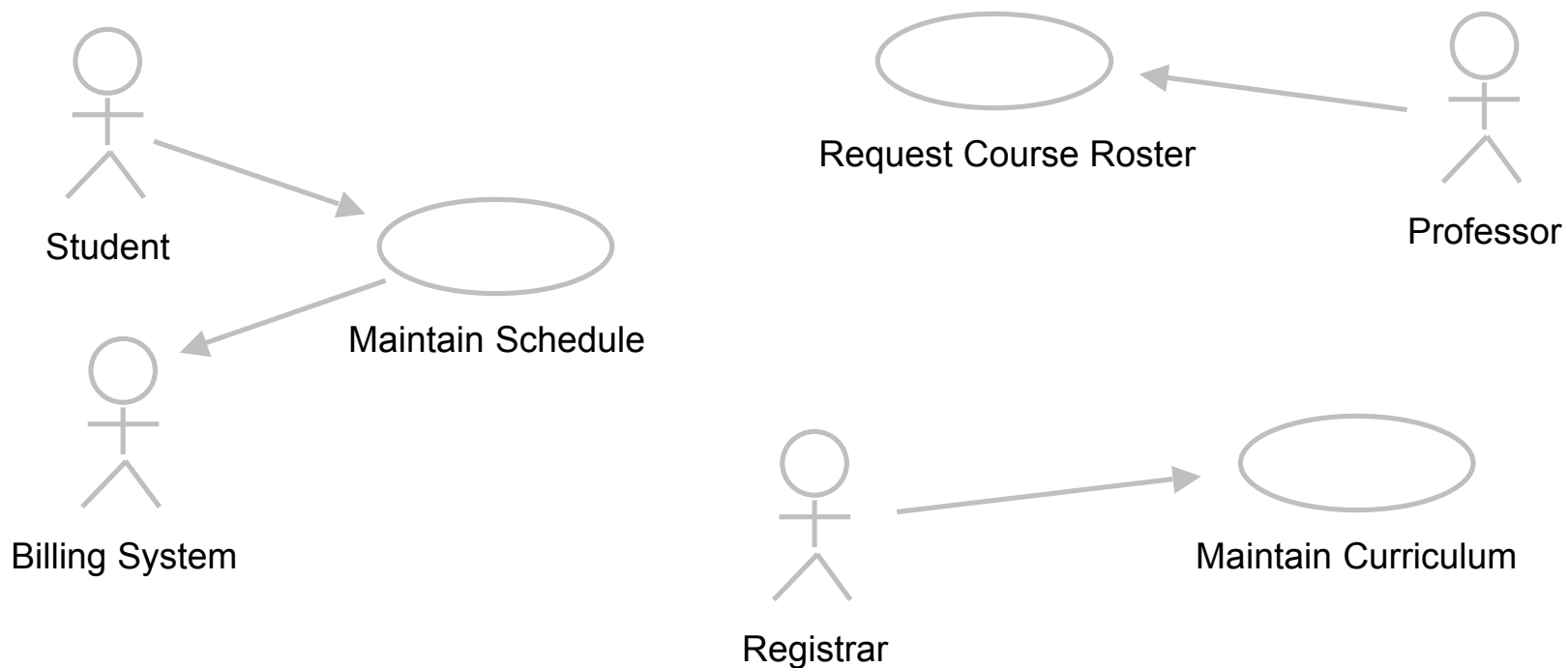
- A flow of events document is created for each use case
 - Written from an actor point of view
- Details what the system must provide to the actor when the use case is used
- Typical contents
 - How the use case starts and ends
 - Normal flow of events
 - Alternate or exceptional flows of events

Flow of Events

- Maintain Curriculum
 - This use case begins when the Registrar logs onto the Registration System and enters his/her password. The system verifies that the password is valid (E-1) and prompts the Registrar to select the current semester or a future semester (E-2). The Registrar enters the desired semester. The system prompts the professor to select the desired activity: ADD, DELETE, REVIEW, or QUIT.
 - If the activity selected is ADD, the S-1: Add a Course subflow is performed.
 - If the activity selected is DELETE, the S-2: Delete a Course subflow is performed.
 - If the activity selected is REVIEW, the S-3: Review Curriculum subflow is performed.
 - If the activity selected is QUIT, the use case ends.
 - ...

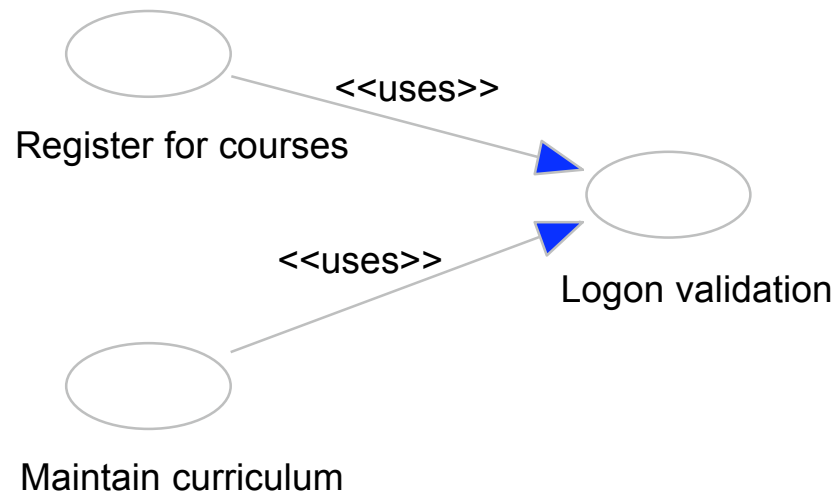
Use Case Diagram

- Use case diagrams are created to visualize the relationships between actors and use cases



Use Case Relationships

- As the use cases are documented, other use case relationships may be discovered
 - A <<uses>> relationship shows behavior that is common to one or more use cases
 - An <<extends>> relationship shows optional behavior

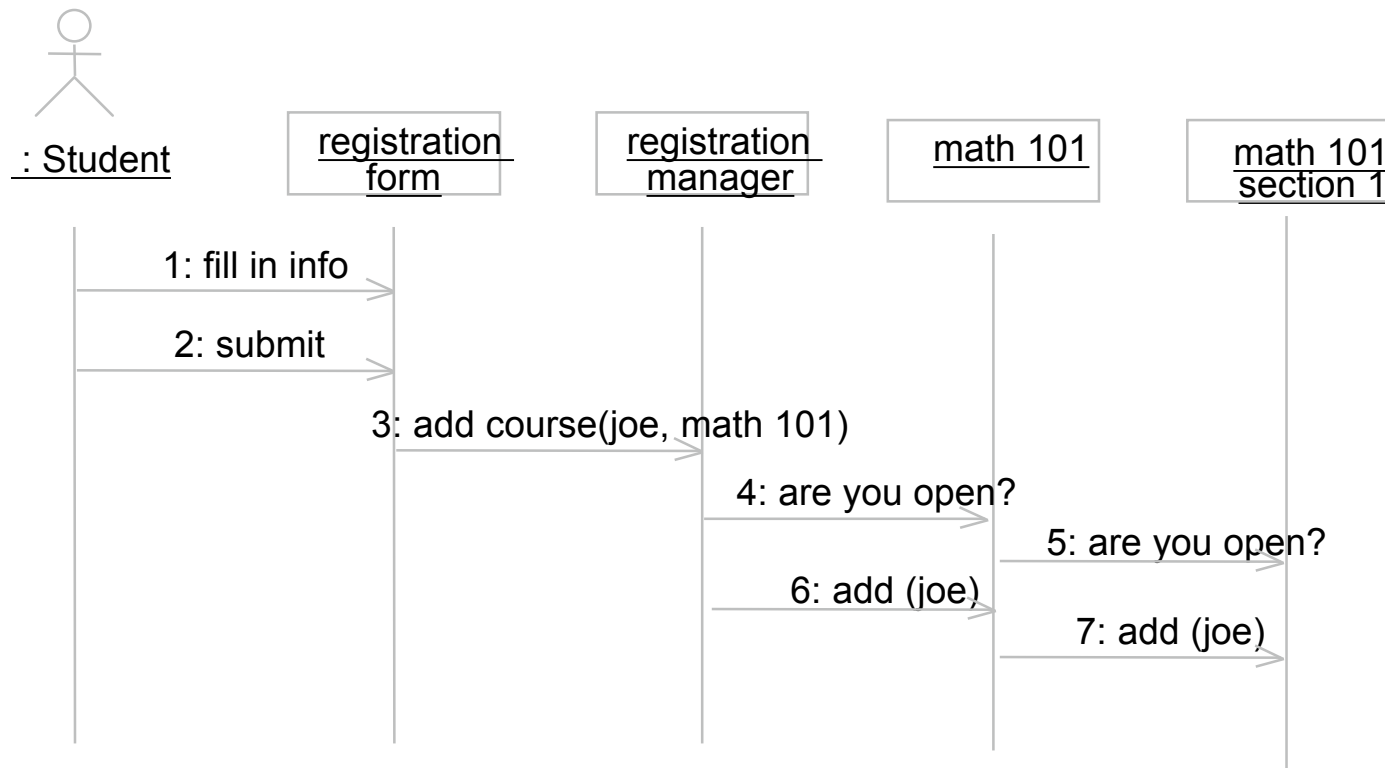


Use Case Realizations

- The use case diagram presents an outside view of the system
- Interaction diagrams describe how use cases are realized as interactions among societies of objects
- Two types of interaction diagrams
 - Sequence diagrams
 - Collaboration diagrams

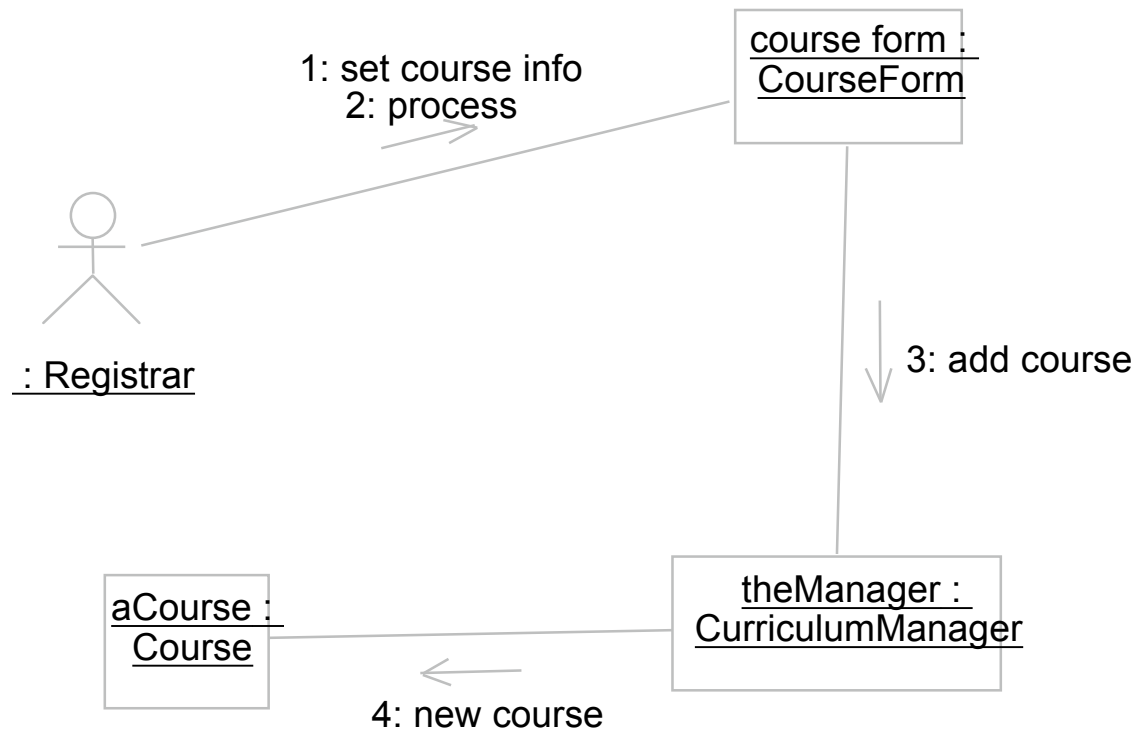
Sequence Diagram

- A sequence diagram displays object interactions arranged in a time sequence



Collaboration Diagram

- A collaboration diagram displays object interactions organized around objects and their links to one another



Class Diagrams

- A class diagram shows the existence of classes and their relationships in the logical view of a system
- UML modeling elements in class diagrams
 - Classes and their structure and behavior
 - Association, aggregation, dependency, and inheritance relationships
 - Multiplicity and navigation indicators
 - Role names

Classes

- A class is a collection of objects with common structure, common behavior, common relationships and common semantics
- Classes are found by examining the objects in sequence and collaboration diagram
- A class is drawn as a rectangle with three compartments
- Classes should be named using the vocabulary of the domain
 - Naming standards should be created
 - e.g., all classes are singular nouns starting with a capital letter

Classes

RegistrationForm

ScheduleAlgorithm

RegistrationManager

Course

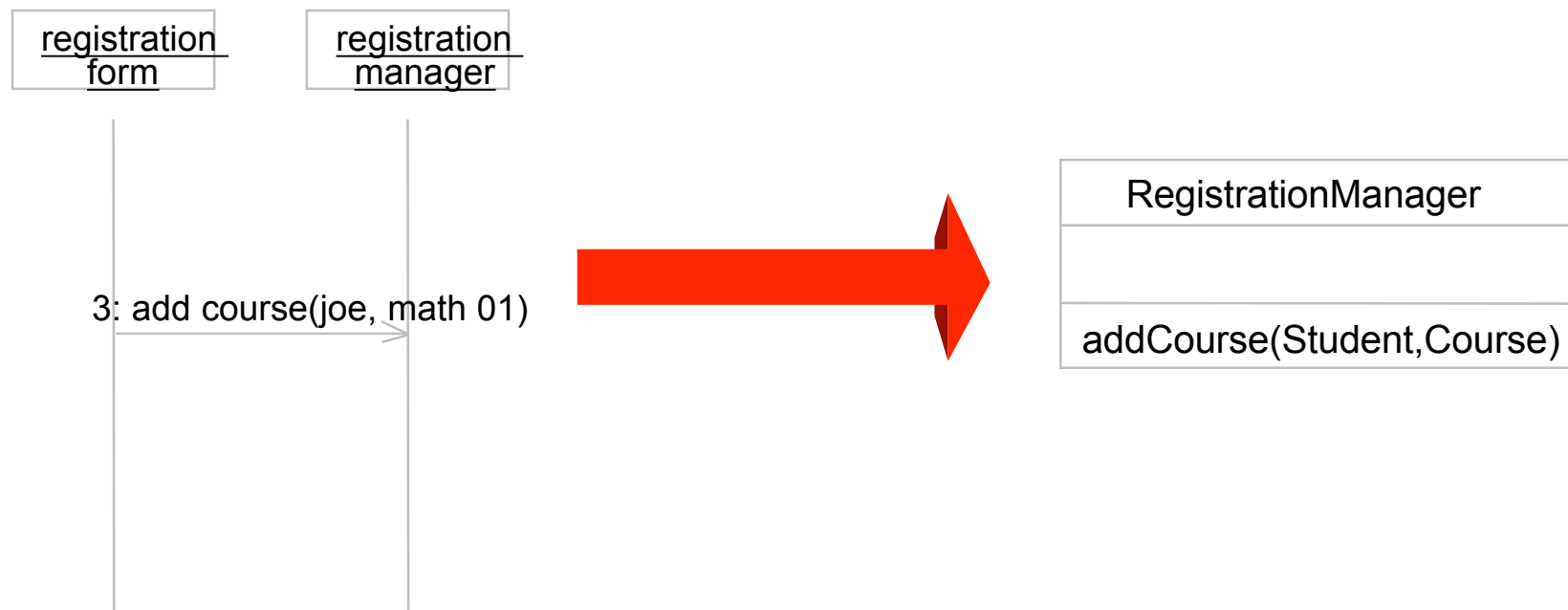
Student

Professor

CourseOffering

Operations

- The behavior of a class is represented by its operations
- Operations may be found by examining interaction diagrams



Attributes

- The structure of a class is represented by its attributes
- Attributes may be found by examining class definitions, the problem requirements, and by applying domain knowledge

Each course offering
has a number, location
and time



CourseOffering
number
location
time

Classes

RegistrationForm

ScheduleAlgorithm

RegistrationManager

addStudent(Course, StudentInfo)

Course

name
numberCredits

open()
addStudent(StudentInfo)

Student

name
major

Professor

name
tenureStatus

CourseOffering

location

open()
addStudent(StudentInfo)

Relationships

- Relationships provide a pathway for communication between objects
- Sequence and/or collaboration diagrams are examined to determine what links between objects need to exist to accomplish the behavior -- if two objects need to communicate there must be a link between them
- Three types of relationships are:
 - Association
 - Aggregation
 - Dependency

Relationships

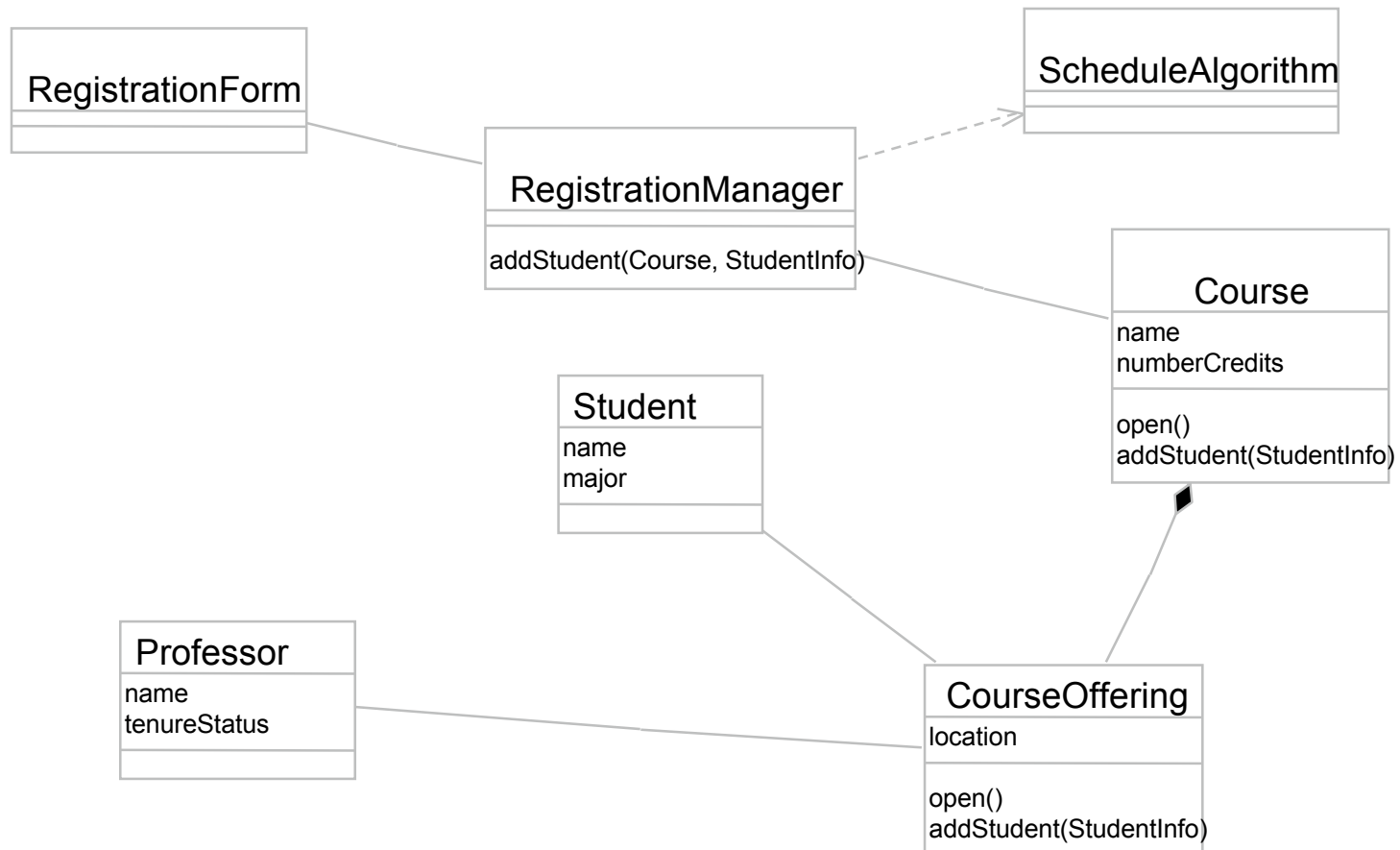
- An **association** is a bidirectional connection between classes
 - An association is shown as a line connecting the related classes
- An **aggregation** is a stronger form of relationship where the relationship is between a whole and its parts
 - An aggregation is shown as a line connecting the related classes with a diamond next to the class representing the whole
- A **dependency** relationship is a weaker form of relationship between a client and a supplier where the client does not have semantic knowledge of the supplier
 - A dependency is shown as a dashed line pointing from the client to the supplier

Finding Relationships

- Relationships are discovered by examining interaction diagrams
 - If two objects must “talk” there must be a pathway for communication



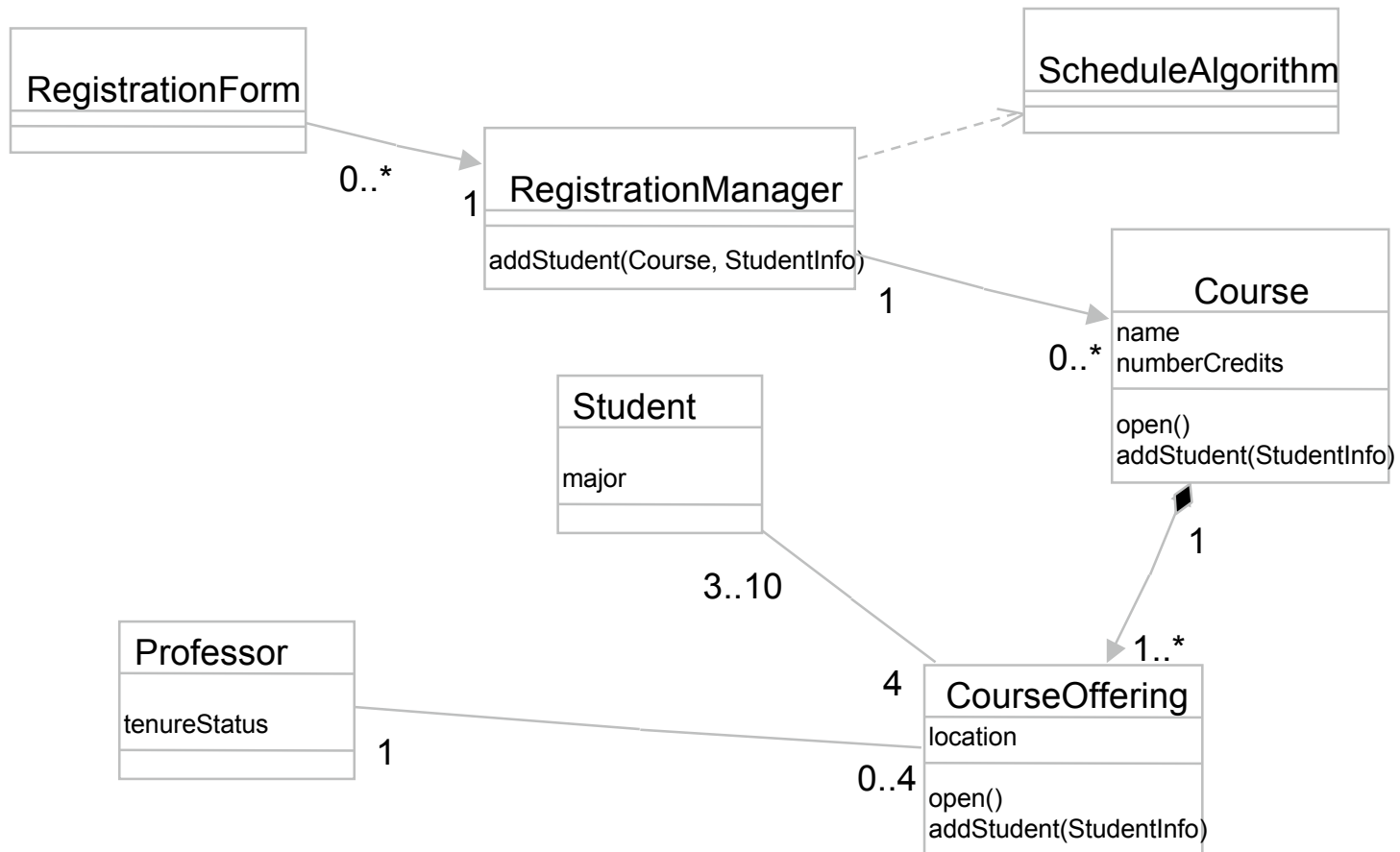
Relationships



Multiplicity and Navigation

- Multiplicity defines how many objects participate in a relationship
 - Multiplicity is the number of instances of one class related to ONE instance of the other class
 - For each association and aggregation, there are two multiplicity decisions to make: one for each end of the relationship
- Although associations and aggregations are bidirectional by default, it is often desirable to restrict navigation to one direction
- If navigation is restricted, an arrowhead is added to indicate the direction of the navigation

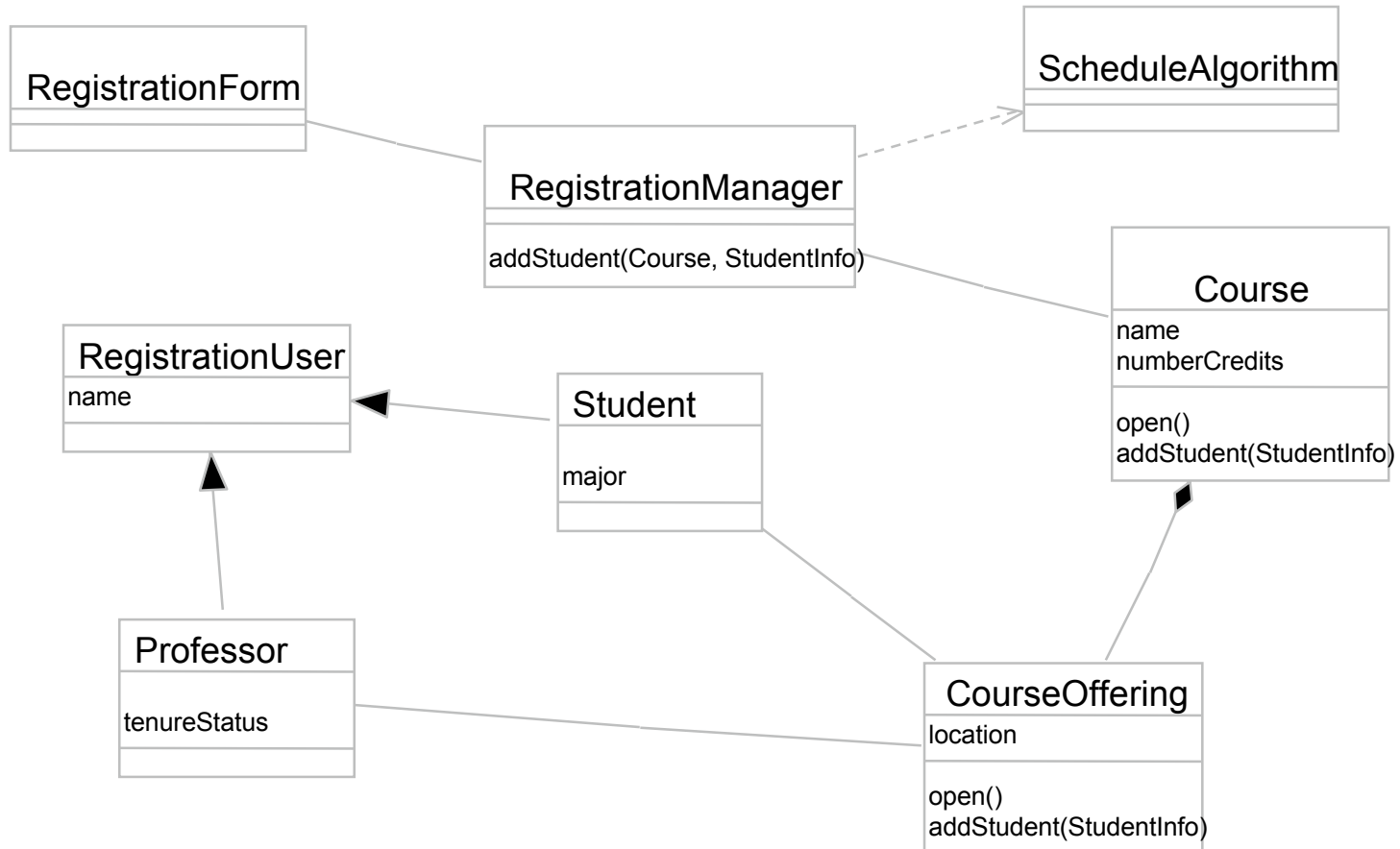
Multiplicity and Navigation



Inheritance

- Inheritance is a relationship between a superclass and its subclasses
- There are two ways to find inheritance:
 - Generalization
 - Specialization
- Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy

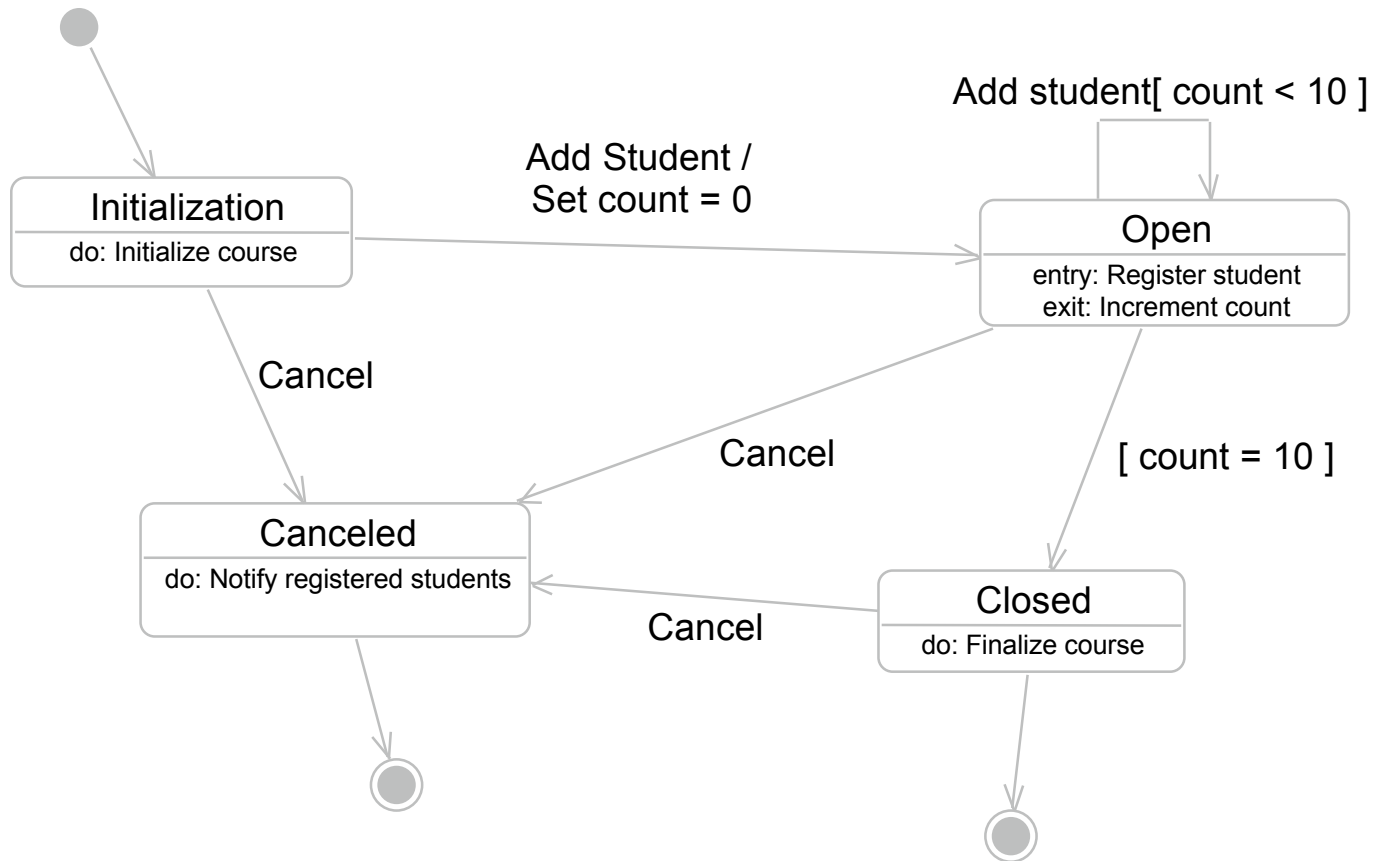
Inheritance



The State of an Object

- A state transition diagram shows
 - The life history of a given class
 - The events that cause a transition from one state to another
 - The actions that result from a state change
- State transition diagrams are created for objects with significant dynamic behavior

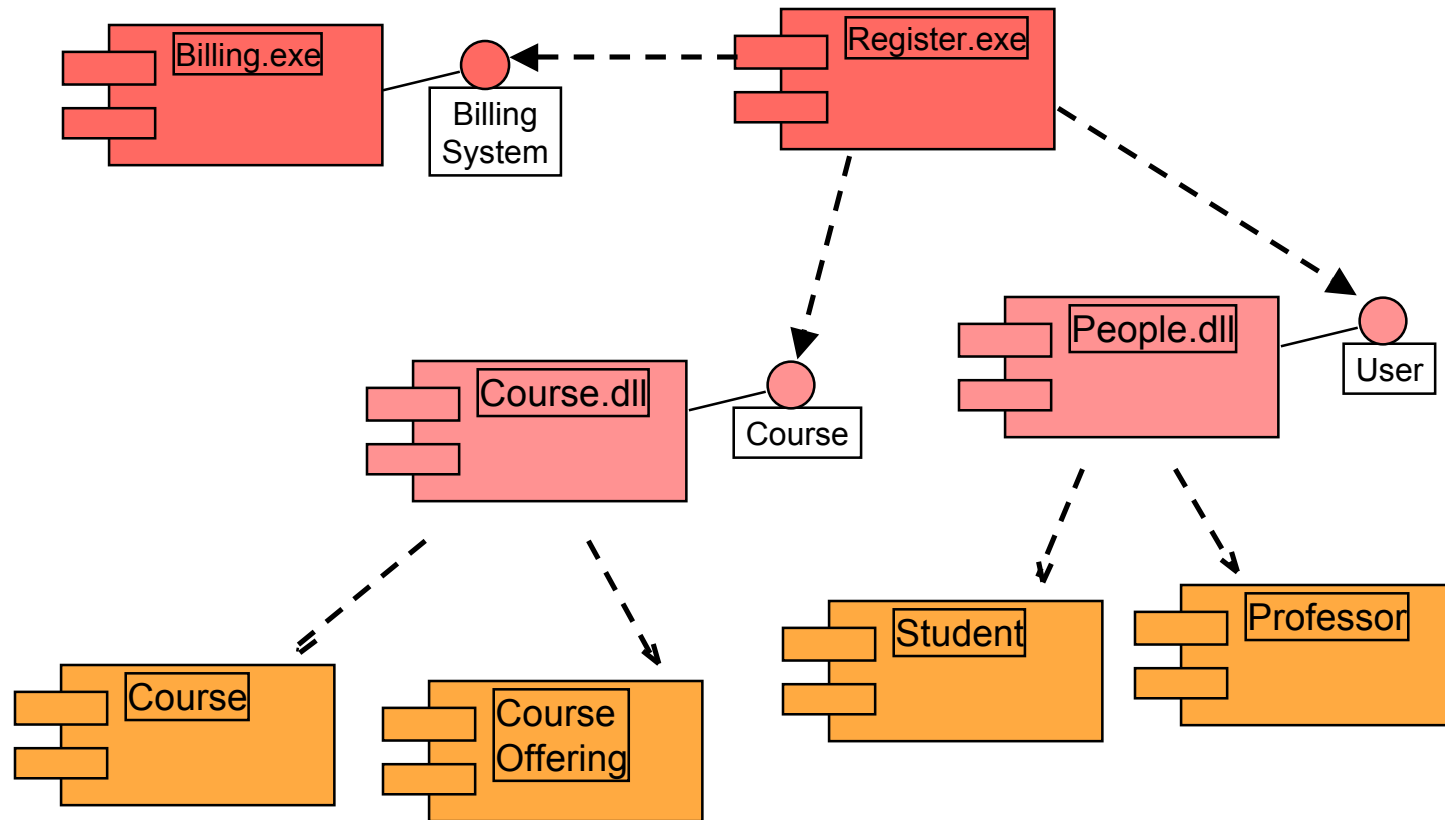
State Transition Diagram



The Physical World

- Component diagrams illustrate the organizations and dependencies among software components
- A component may be
 - A source code component
 - A run time component or
 - An executable component

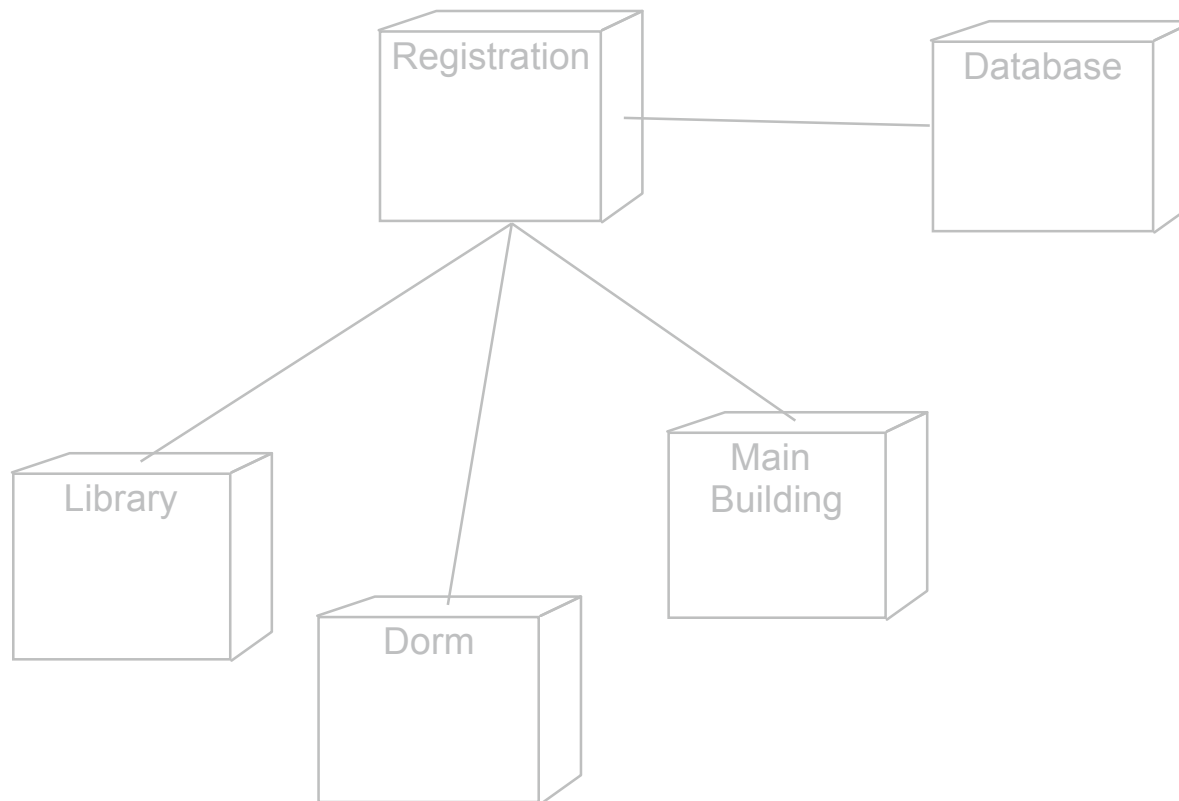
Component Diagram



Deploying the System

- The deployment diagram shows the configuration of run-time processing elements and the software processes living on them
- The deployment diagram visualizes the distribution of components across the enterprise.

Deployment Diagram



Part III

The Iterative Approach to Software Design

The Iterative Approach

- Continuous integration
 - Not done in one lump near the delivery date
- Frequent, executable releases
 - Some internal; some delivered
- Attack risks through demonstrable progress
 - Progress measured in products, not documentation or engineering estimates

Resulting Benefits

- Releases are a forcing function that drives the development team to closure at regular intervals
 - Cannot have the “90% done with 90% remaining” phenomenon
- Can incorporate problems/issues/changes into future iterations rather than disrupting ongoing production
- The project’s supporting elements (testers, writers, toolsmiths, etc.) can better schedule their work

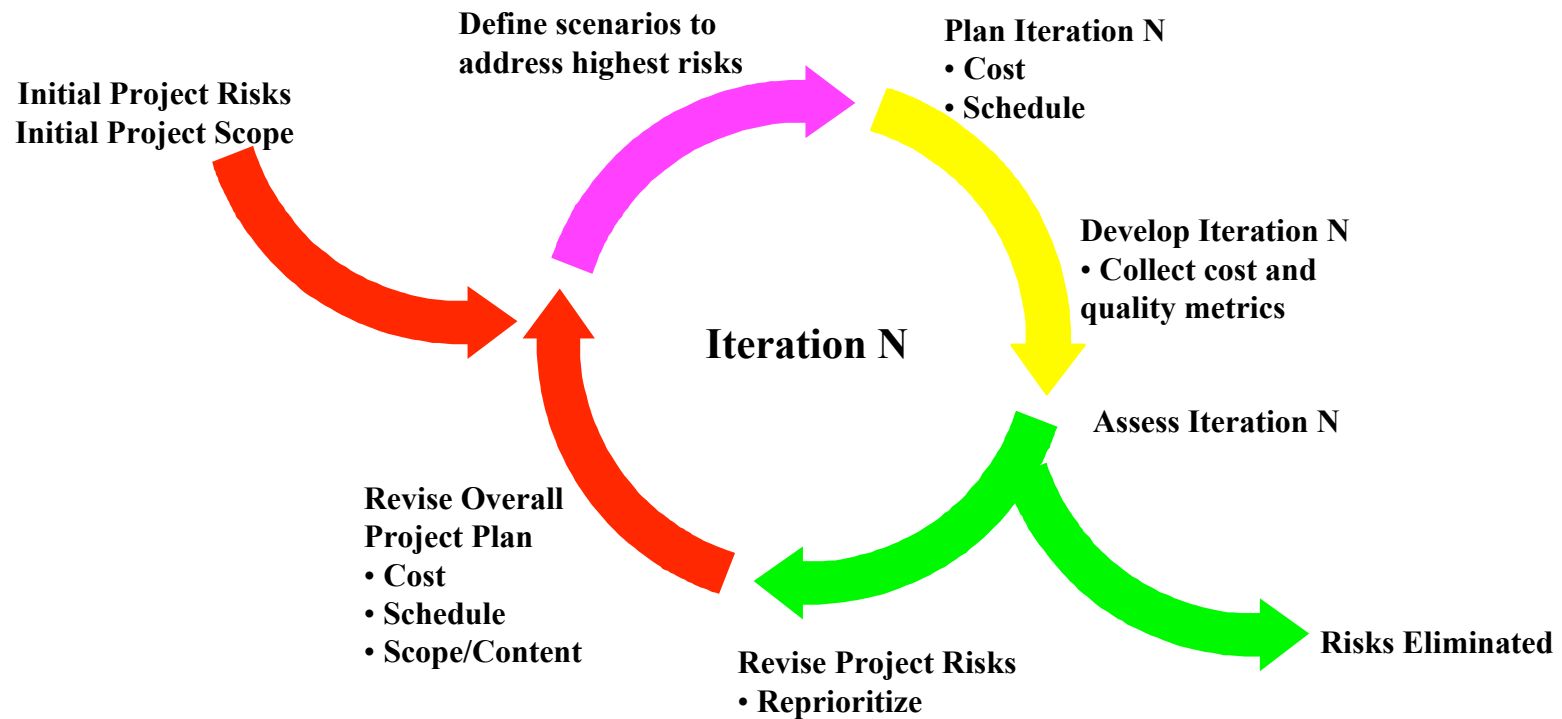
Managing Risk Phase-by-Phase

- Inception
 - Gather information about the project
 - Bracket the project's risks
- Elaboration
 - Develop a common understanding of the system's scope and desired behavior by exploring scenarios with end users and domain experts
 - Establish the system's architecture
 - Design common mechanisms to address system-wide issues

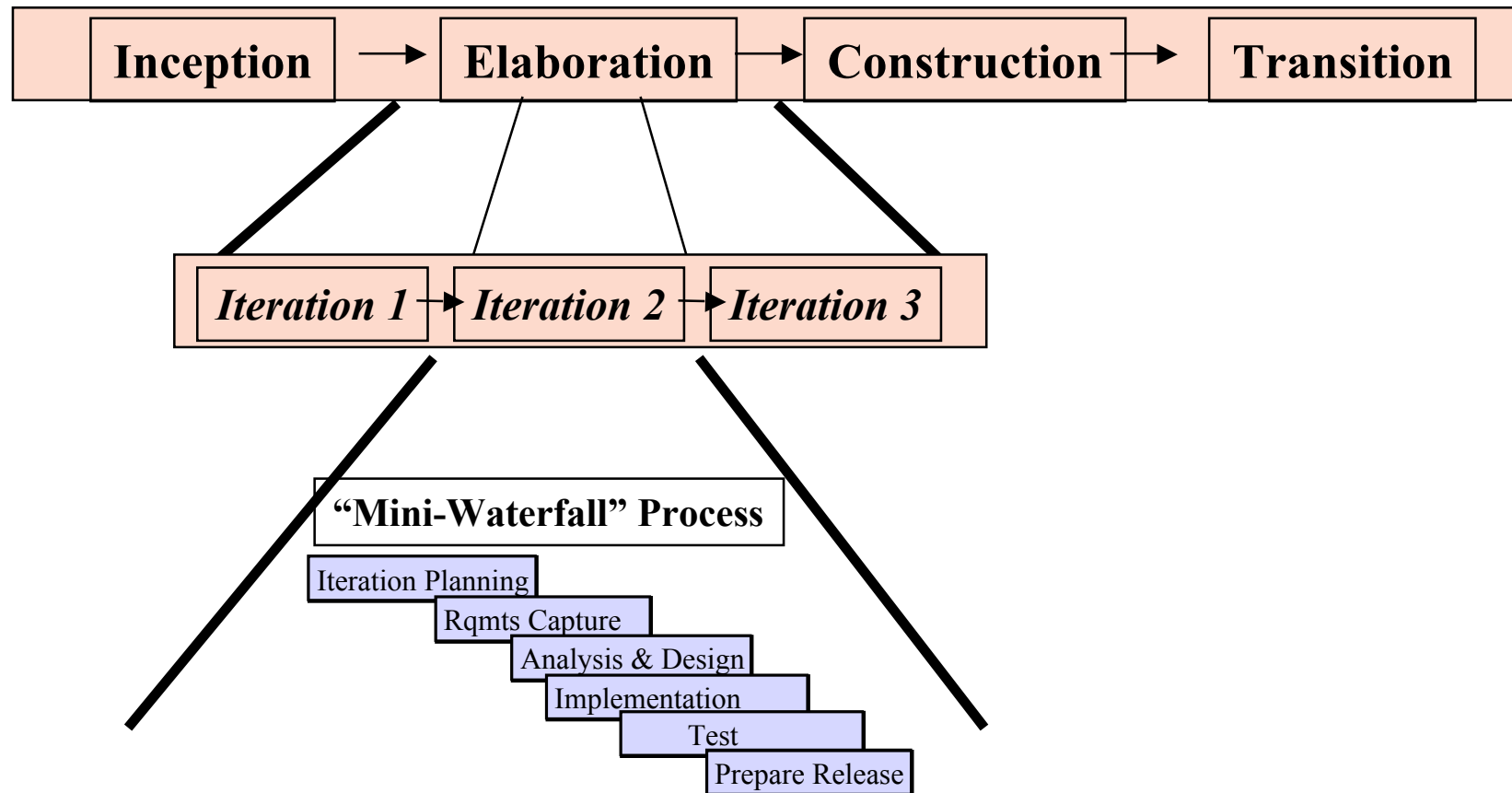
Managing Risk Phase-by-Phase

- Construction
 - Refine the architecture
 - Risk-driven iterations
 - Continuous integration
- Transition
 - Facilitate user acceptance
 - Measure user satisfaction
- Post-deployment cycles
 - Continue evolutionary approach
 - Preserve architectural integrity

Risk Reduction Drives Iterations

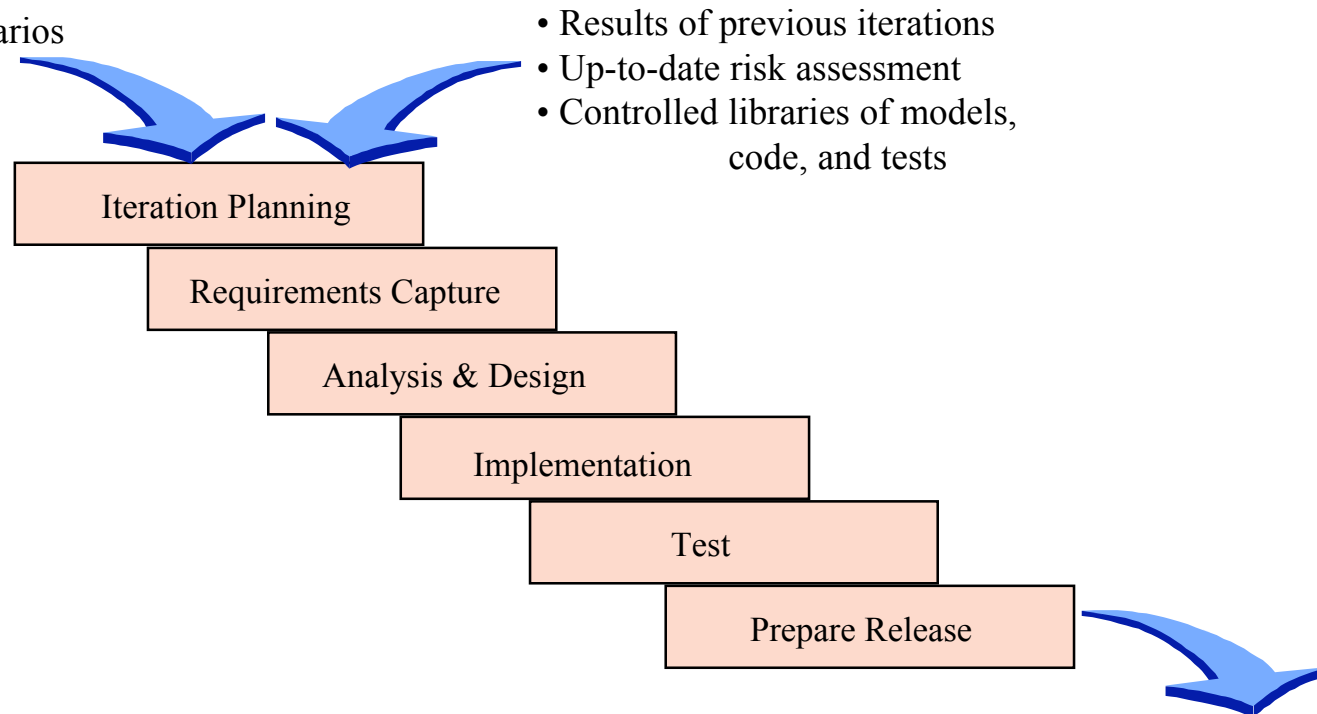


Use Cases Drive the Iteration Process



The Iteration Life Cycle: A Mini-Waterfall Model

Selected scenarios



- Results of previous iterations
- Up-to-date risk assessment
- Controlled libraries of models, code, and tests

Release description
Updated risk assessment
Controlled libraries

Iteration Life Cycle Activities

- Iteration planning
 - Before the iteration begins, the general objectives of the iteration should be established based on
 - Results of previous iterations (if any)
 - Up-to-date risk assessment for the project
 - Determine the evaluation criteria for this iteration
 - Prepare detailed iteration plan for inclusion in the development plan
 - Include intermediate milestones to monitor progress
 - Include walkthroughs and reviews

Iteration Life Cycle Activities

- Requirements Capture
 - Select/define the use cases to be implemented in this iteration
 - Update the object model to reflect additional domain classes and associations discovered
 - Develop a test plan for the iteration

Iteration Life Cycle Activities

- Analysis & Design
 - Determine the classes to be developed or updated in this iteration
 - Update the object model to reflect additional design classes and associations discovered
 - Update the architecture document if needed
 - Begin development of test procedures
- Implementation
 - Automatically generate code from the design model
 - Manually generate code for operations
 - Complete test procedures
 - Conduct unit and integration tests

Iteration Life Cycle Activities

- Test
 - Integrate and test the developed code with the rest of the system (previous releases)
 - Capture and review test results
 - Evaluate test results relative to the evaluation criteria
 - Conduct an iteration assessment
- Prepare the release description
 - Synchronize code and design models
 - Place products of the iteration in controlled libraries

Work Allocation in each Iteration

- Work to be accomplished in an iteration is determined by
 - The (new) use cases to be implemented
 - The rework to be done
- Packages are convenient for developers
 - High-level packages can be assigned to teams
 - Lower-level packages can be assigned to individual developers
- Use Cases make convenient work packages for test and assessment teams

Iteration Assessment

- Assess iteration results relative to the evaluation criteria established during iteration planning:
 - Functionality
 - Performance
 - Quality measures
- Consider external changes that have occurred during this iteration
 - For example, changes to requirements, user needs, etc.
- Determine what rework, if any, is required and assign it to the remaining iterations

Selecting Iterations

- How many iterations do I need?
 - On projects taking 18 months or less, 3 to 6 iterations are typical
- Are all iterations on a project the same length?
 - Iteration length may vary by phase. For example, elaboration iterations may be shorter than construction iterations

The First Iteration

- The first iteration is usually the hardest
 - Requires the entire development environment and most of the development team to be in place
 - Many tool integration issues, team-building issues, staffing issues, etc. must be resolved
- Teams new to an iterative approach are usually overly-optimistic
- Be modest regarding the amount of functionality that can be achieved in the first iteration
 - Otherwise, completion of the first iteration will be delayed,
 - The total number of iterations reduced, and
 - The benefits of an iterative approach reduced

Remember That

- The main reason for using the iterative life cycle:
 - You do not have all the information you need up front
 - Things will change during the development period
- You must expect that
 - Some risks will not be eliminated as planned
 - You will discover new risks along the way
 - Some rework will be required; some lines of code developed for an iteration will be thrown away
 - Requirements will change along the way