Lesson 2

Software Engineering Best Practices: *Action in Accord with All the Laws of Nature*

Topics: Principles of Software Engineering

- The inherent challenge of software engineering
- The importance of analysis and design
- Best practices

Software Development Historical Analysis

- No Silver Bullet – Brooks (1986)
- Essential nature of SW engineering vs. the non-essential
- Non-essential -- E.g., syntax of programming languages
- Major advances already made for non-essential aspects
  - High level languages, IDEs, Application platforms,
  - Software and GUI tools, builds, source control, etc.

Essential Difficulties

- Complexity
  - Combinatorial complexity of states and processes
  - Sheer size
  - Interactive, distributed, networked
- Conformity
  - Expect software to conform to needs of hardware and business domains, even if a more "logical" approach would involve changing these external interfaces

Essential Difficulties (cont)

- Changeability
  - Expect to modify incredibly complex systems
  - Many pressures to modify software: reality changes, functionality extensions, easier than hardware, lasts longer
- Invisibility
  - Software is abstract compared to to buildings, bridges, …
  - Difficult for visual diagrams to capture

Software Development Historical Analysis

- 1994 – W.W. Gibbs, "Software's Chronic Crisis", *Scientific American*, Sept., 1994
  - 25% of large scale SW projects cancelled
  - Average delivery is late by 50% (worse for large scale projects)
  - 75% of large scale SW projects – do not function as intended or are not even used

The Need for a Software Engineering “Process”

1999 - (Jacobson, *The Unified Software Development Process,* 1999, pp.3-4)

"The software problem boils down to the difficulty developers face in pulling together the many strands of a large software undertaking. The software development community needs a controlled way of working. It needs a process that integrates the many facets of software development. It needs a common approach, a process that :

- Provides guidance to the order of a team's activities.
- Directs the tasks of individual developers and the team as a whole.
- Specifies what artifacts should be developed.
- Offers criteria for monitoring and measuring a project's products and activities.

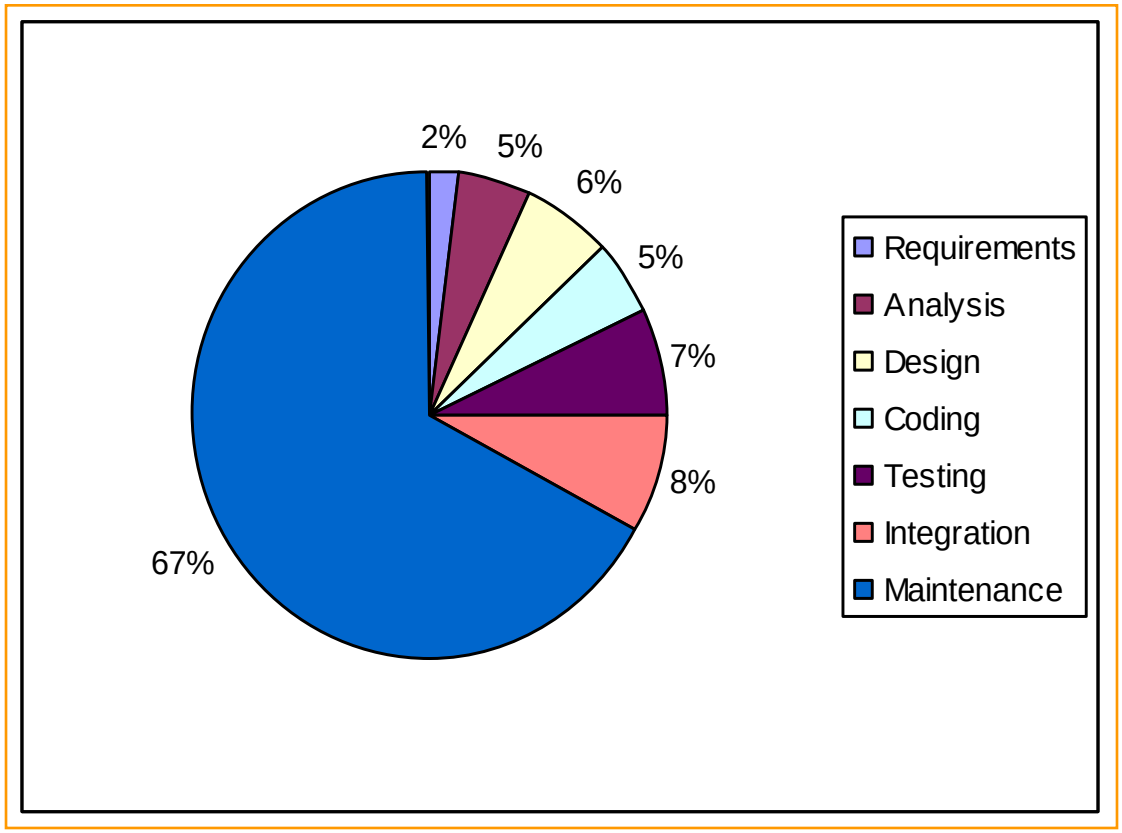
"The presence of a well-defined and well-managed process is a key discriminator between hyperproductive projects and unsuccessful ones."

Topics: Principles of Software Engineering

- The inherent challenge of software engineering
- The importance of analysis and design
- Best practices

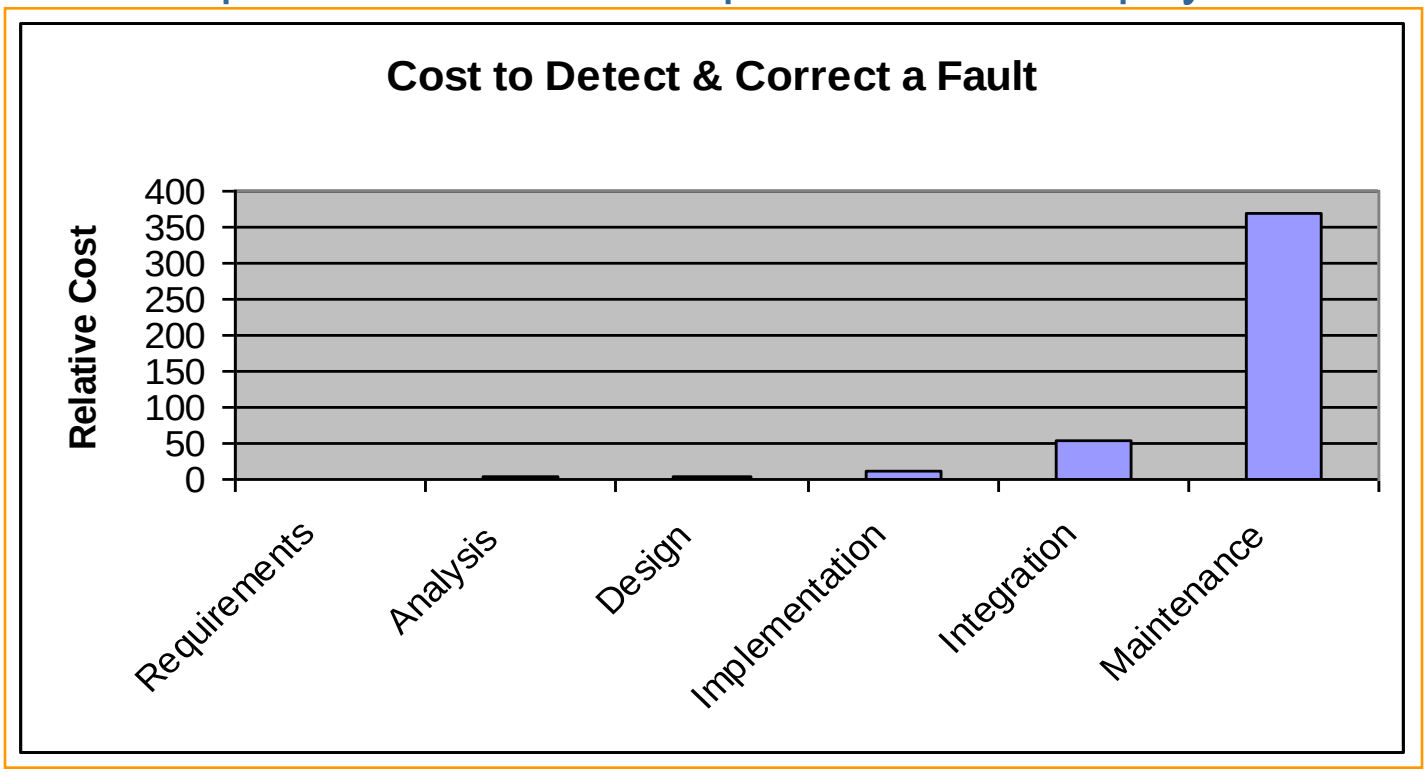
Importance of Maintainability

- 2/3 of life-cycle cost is in system maintenance (for successful projects)
- Only about 5% total costs for coding

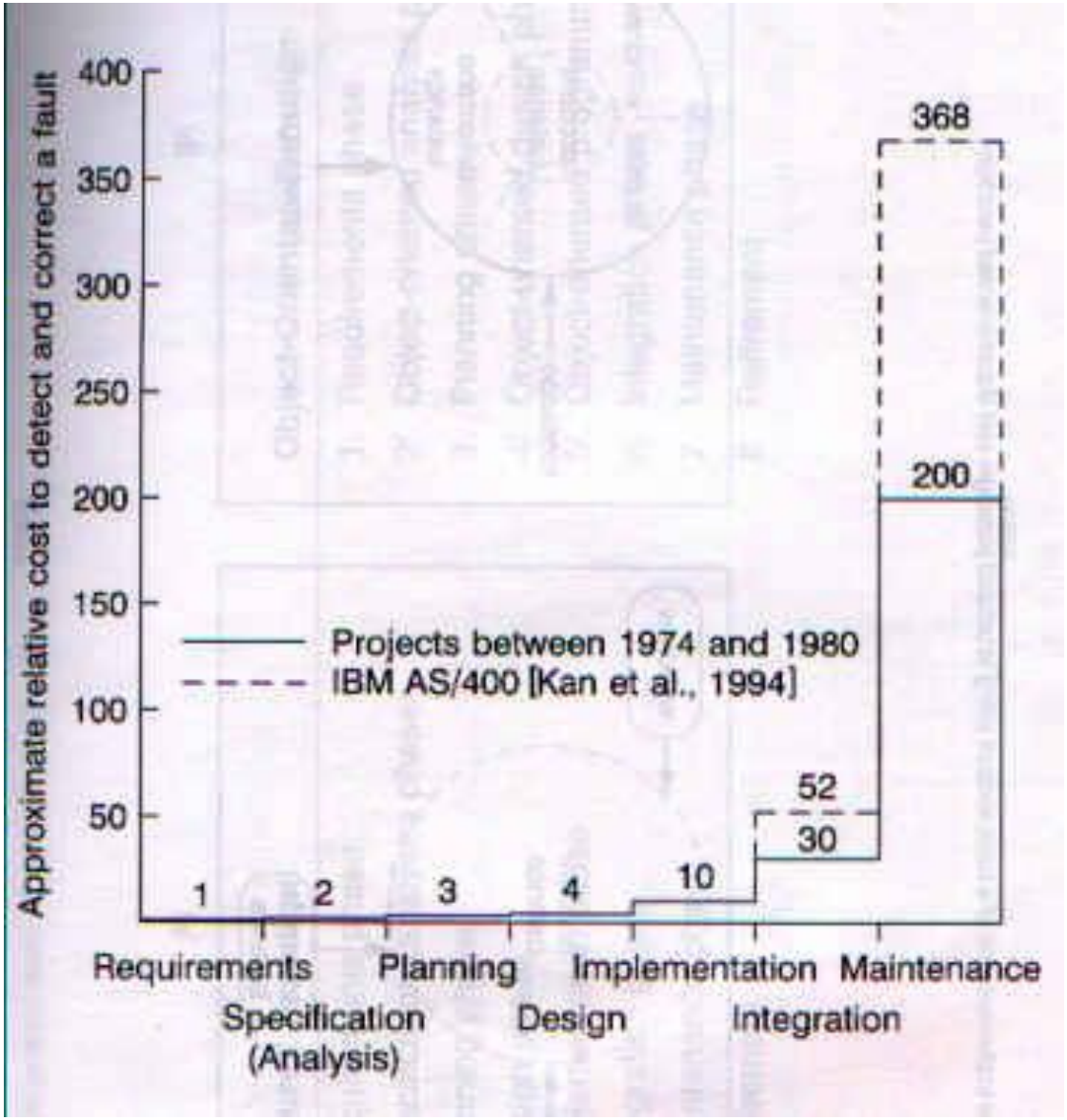


Importance of a Good Start

- Fixing faults is much cheaper earlier in life-cycle than later
- 60-70% faults in large projects specification/design faults.
  - Good A&D facilitate integration and maintenance
  - Greatest potential area for improvement and payback



Increasing Costs to Detect and Correct Faults



Summary Points

- Efficient development of quality software in a timely manner is inherently difficult
- Two-thirds of all the faults in large scale projects have been observed to be specification or design faults. Two-thirds of the life-cycle costs of a software system are incurred during the maintenance phase.

Spec and design faults

Maintenance costs

Topics: Principles of Software Engineering

- The inherent challenge of software engineering
- The importance of analysis and design
- Best practices

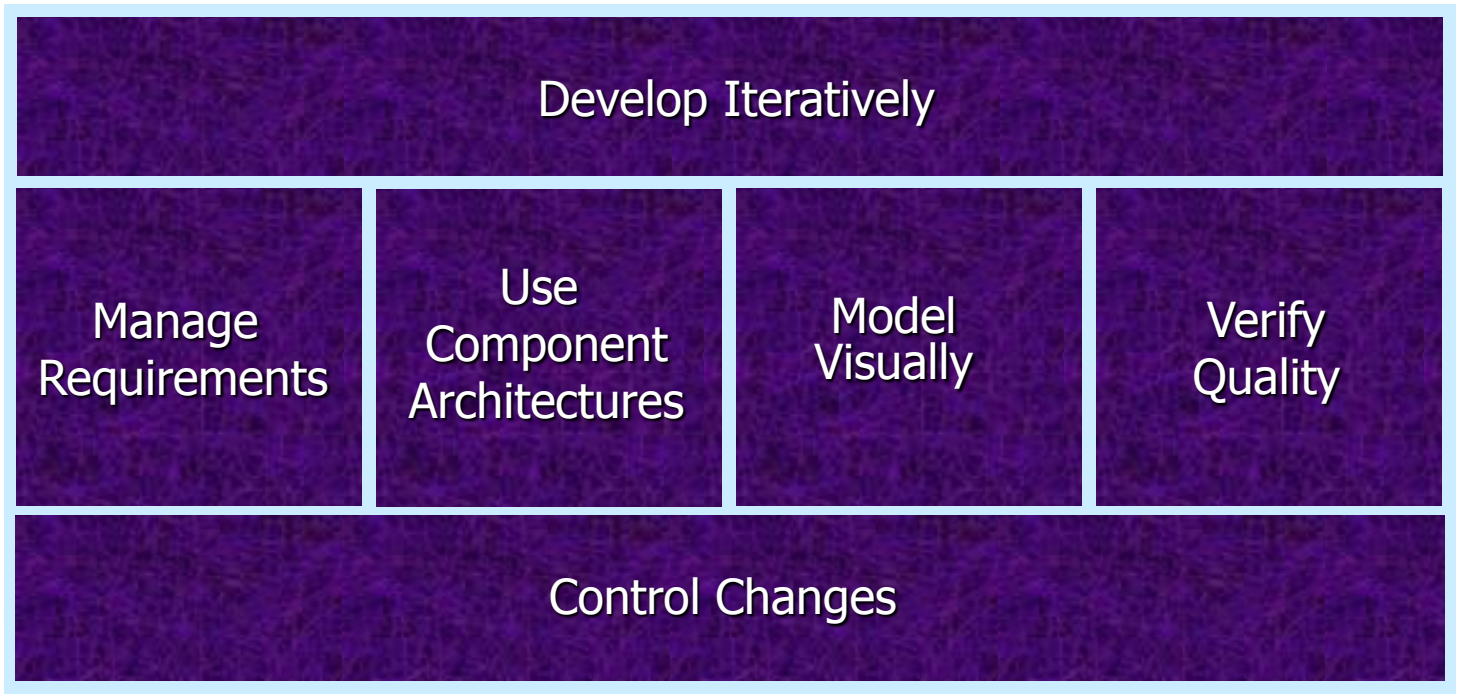
Greatest Potential (Brooks)

- Rapid prototyping
- Incremental development
- Develop great designers

Value of OO Techniques

- Widely accepted in the field as supporting good software engineering principles and practices
- Analysis and design closely integrated with implementation
  - Directly addresses 60-70% faults in specs and design
- Objects support greater integrity of software components
  - Encapsulate implementation details
  - Leads to reusability, extensibility, and maintainability
  - Addresses other area of maximum potential payback (maintenance)

Unified Process Delivers Best Practices



Visually Model Software

- Addresses one of Brooks' "essential" difficulties (inherent abstractness of software)
- Helps to visualize, specify, construct and document structure and behavior of a system
- Using a standard modeling language (like UML) helps to maintain consistency among system's artifacts and facilitates communication among team members

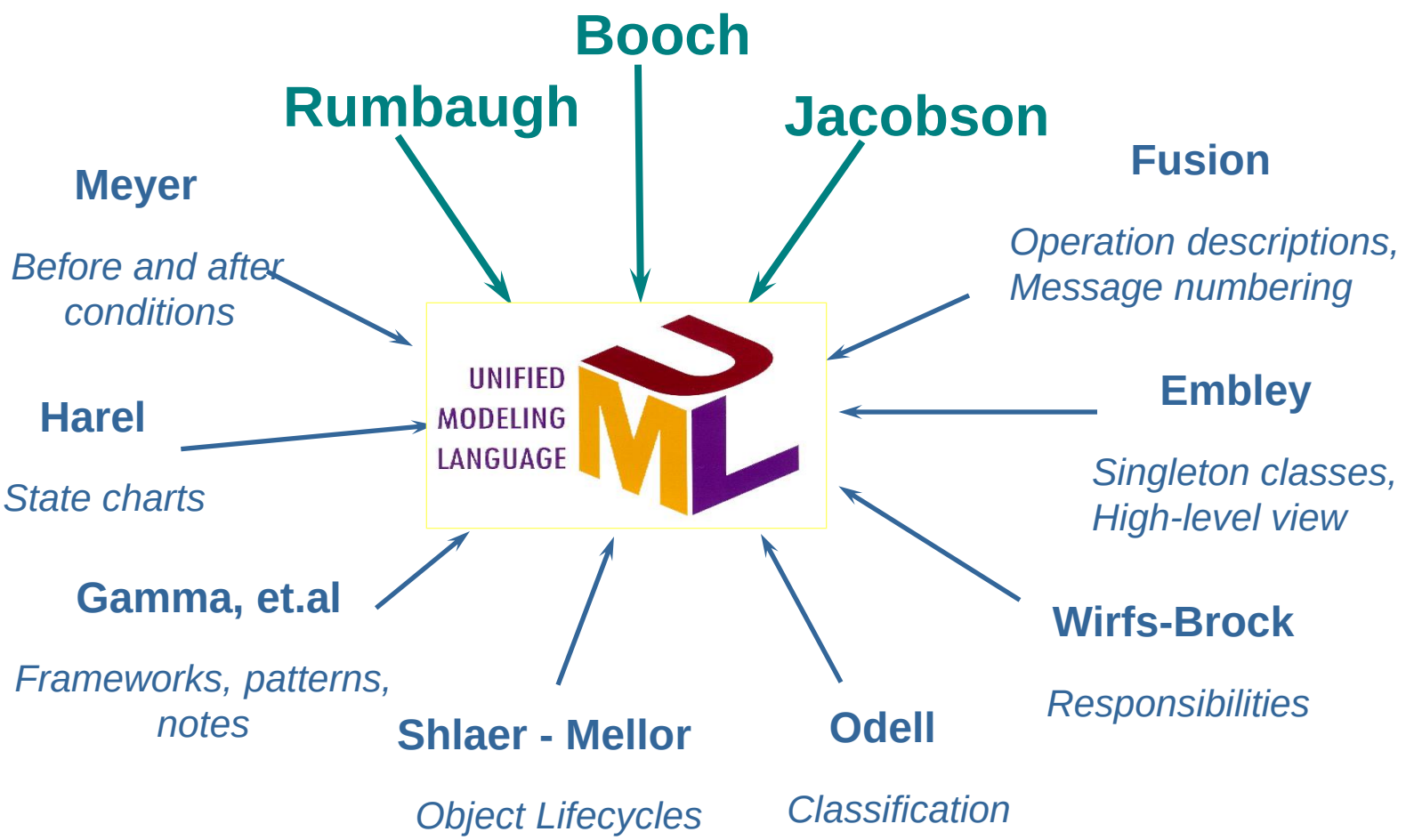
K12-13

What Is the UML?

- The Unified Modeling Language (UML) is a language for
  - Specifying
  - Visualizing
  - Constructing
  - Documenting

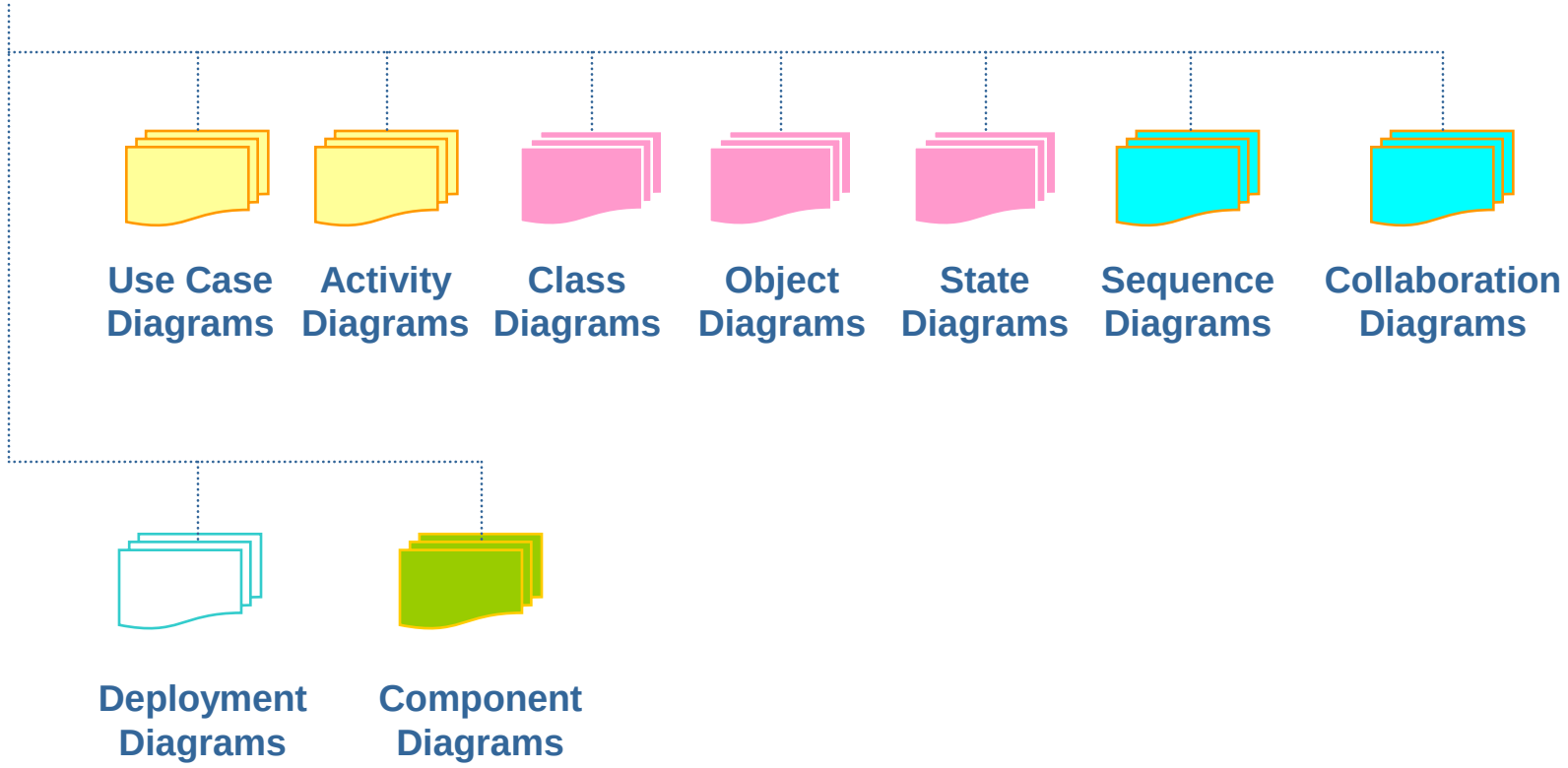
  the artifacts of a software-intensive system

Inputs to UML



The UML Provides Standardized Diagrams

Models



Best Practices Address Root Causes

- Visually model software
- Develop software iteratively
- Use component-based architectures
- Manage requirements
- Verify software quality
- Control changes to software

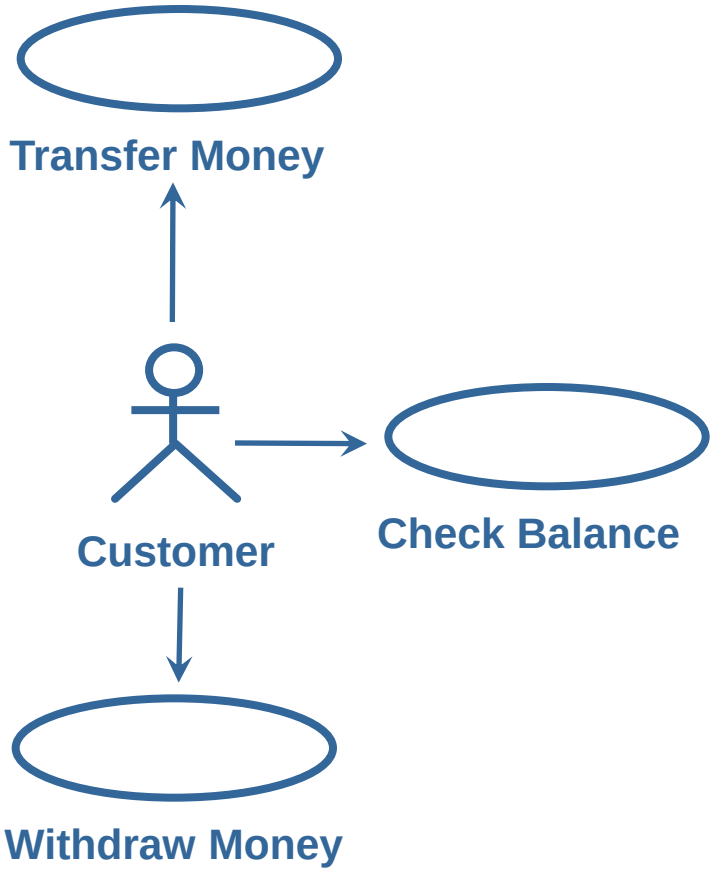
What Is a Software Development Process?

A software development process is the set of activities needed to transform a user's requirements into a software system. The process defines *Who* is doing *What*, *When*, as well as *How* to reach a certain goal.

RUP Key Features

- Use case driven
- Iterative and incremental
  - Models, workflows, phases, and iterations
- Architecture centric

Rational Unified Process Is Use-Case Driven



Use-Cases for an Automated Teller Machine

An **actor** is someone or something outside the system that interacts with the system

A **Use-Case** is a sequence of actions a system performs that yields an observable result of value to a particular actor

Use-Cases Include a Flow of Events

Flow of events for the Withdraw Money Use-Case

1. The Use-Case begins when the client inserts an ATM card. The system reads and validates information on the card.
2. The system prompts for the PIN. Client enters PIN. The system validates the PIN.
3. The system asks which operation the client wishes to perform. Client selects “Cash withdrawal.” System requests amount.
4. Client enters amount. System requests the account type.
5. Client selects account type (checking, savings, credit). The system communicates with the ATM network . . .

Benefits of a Use-Case Driven Process

- Use-Cases are concise, simple, and understandable by a wide range of stakeholders
  - End users, developers and acquirers understand functional requirements of the system
- Use-Cases drive numerous activities in the process:
  - Creation and validation of the design model
  - Definition of test cases and procedures of the test model
  - Planning of iterations
  - Creation of user documentation
- Use-Cases help synchronize the content of different models

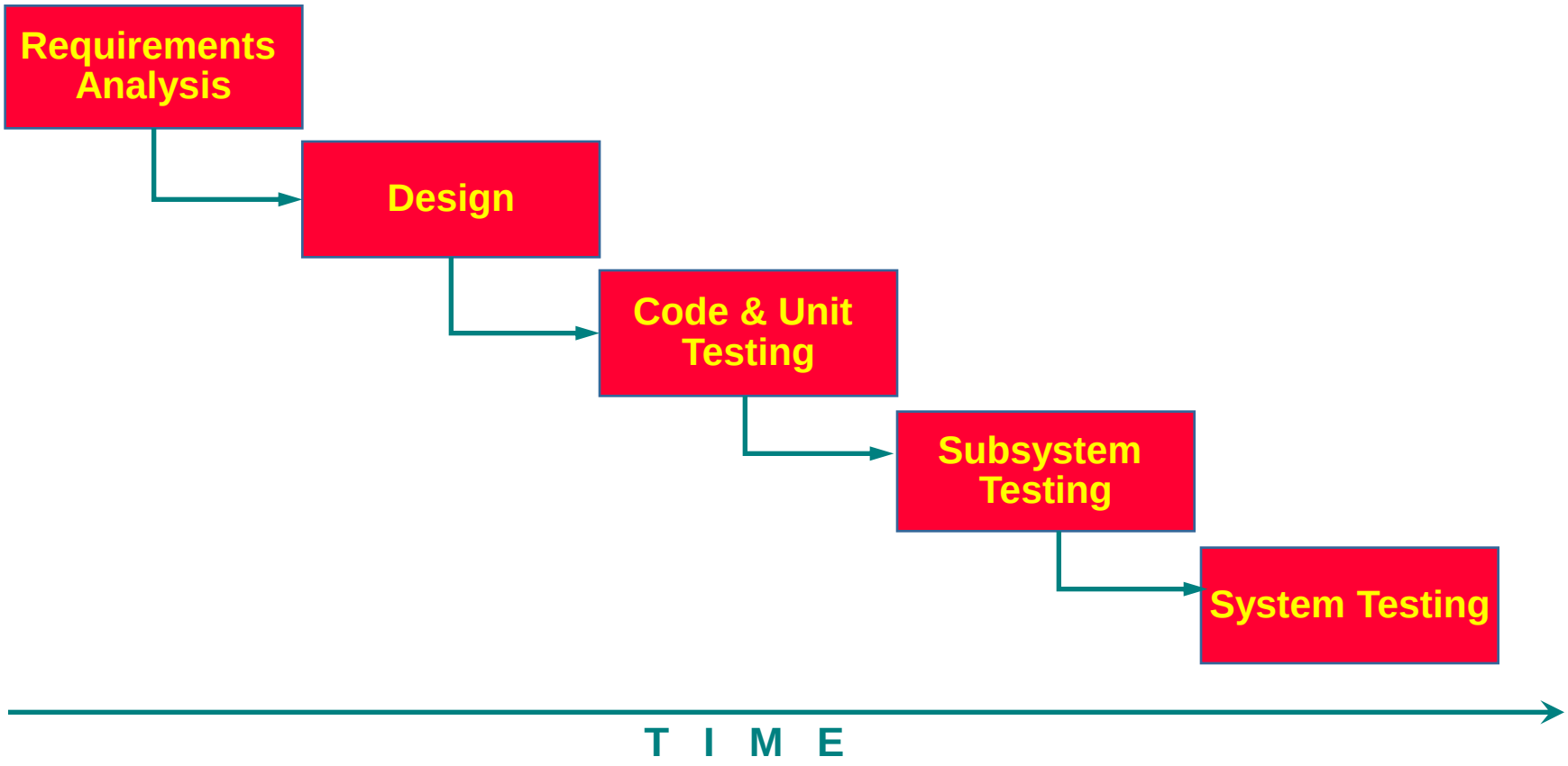
RUP Key Features

- Use case driven
- Iterative and incremental
  - phases, iterations, and workflows
- Architecture-centric

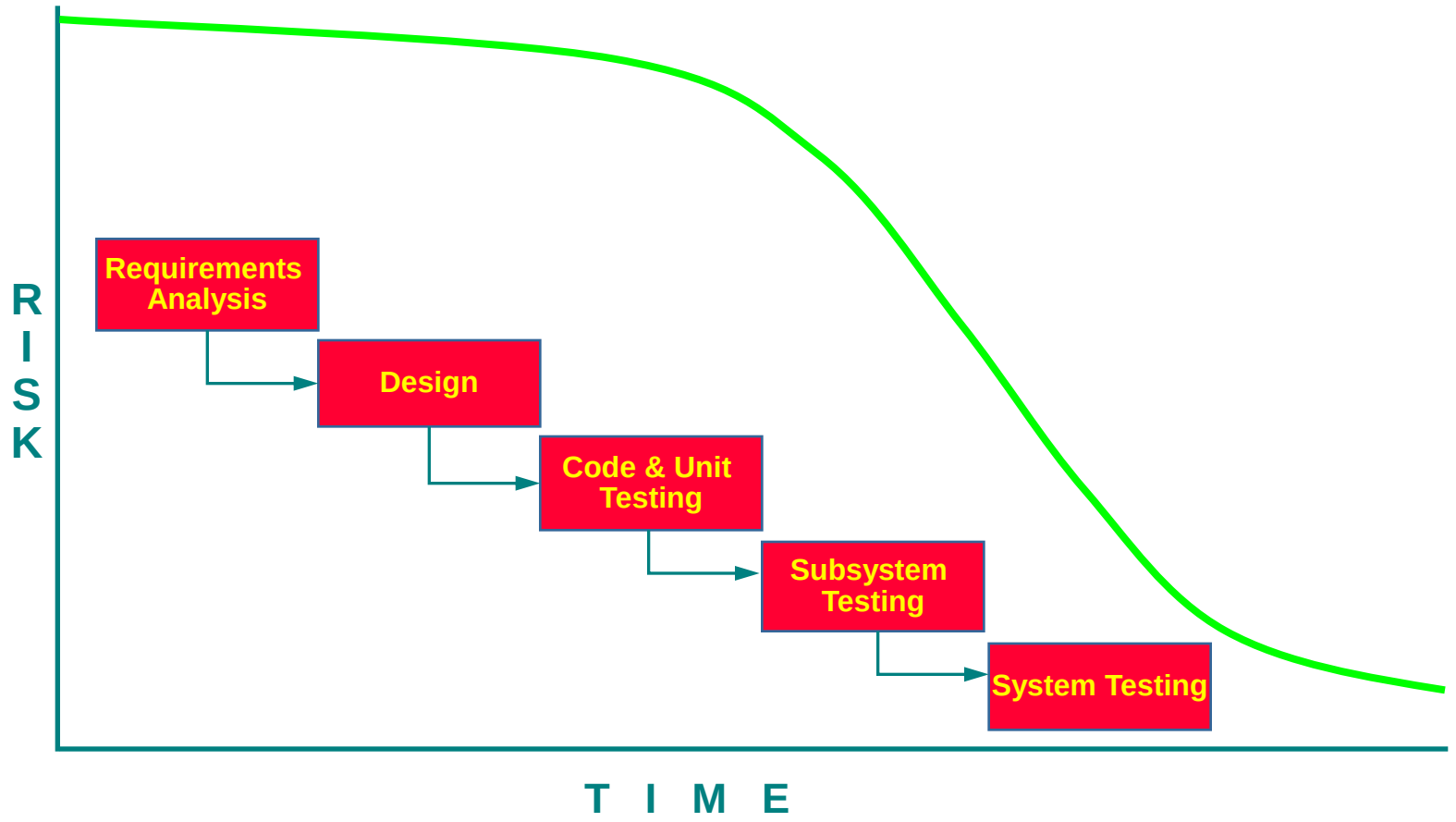
Develop Software Iteratively

- An initial design will likely be flawed with respect to its key requirements
- Late-phase discovery of design defects results in costly over-runs and/or project cancellation

Traditional Waterfall Development

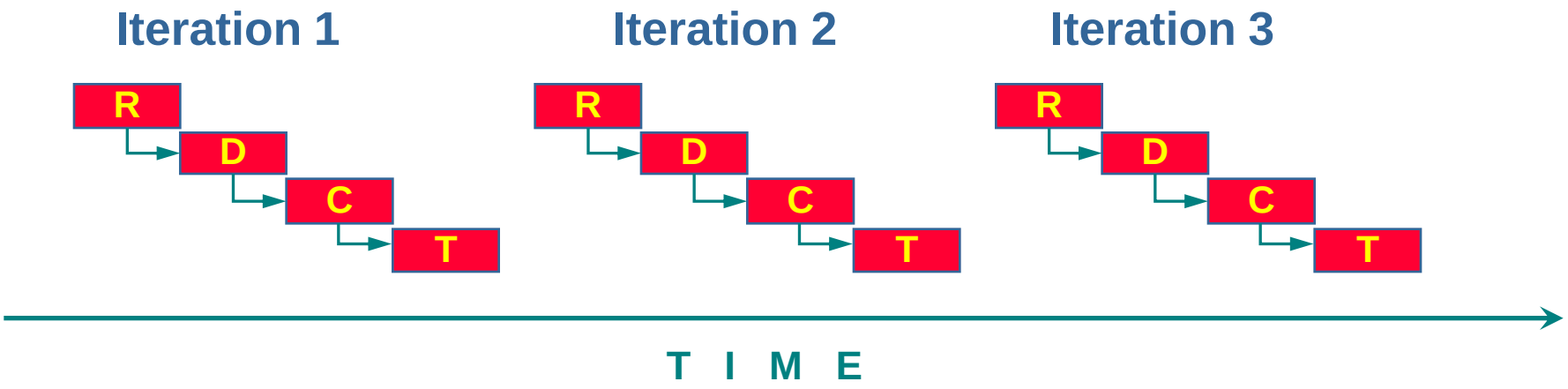


Waterfall Development Delays Reduction of Risk



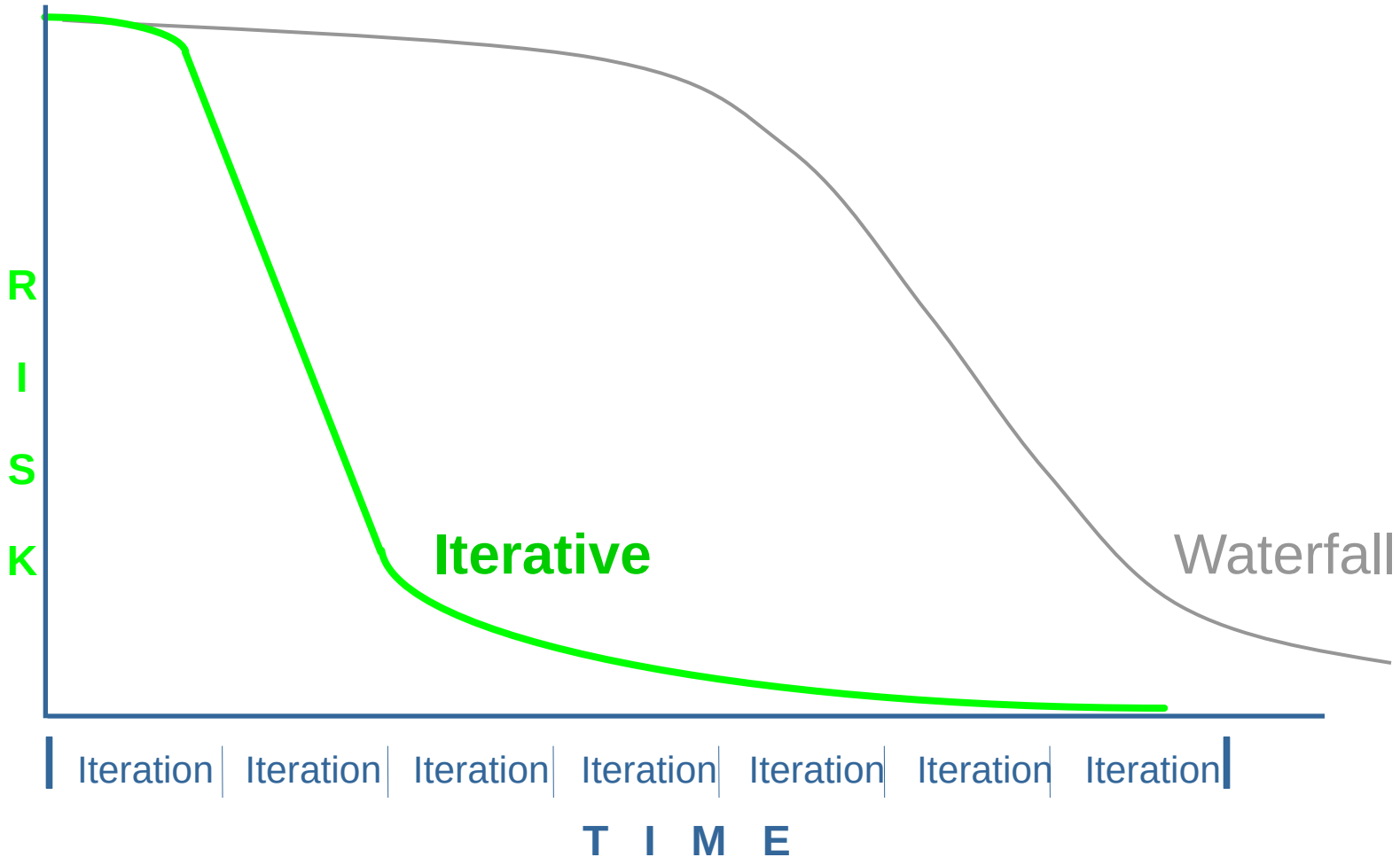
K6-7

Apply the Waterfall Iteratively to System Increments

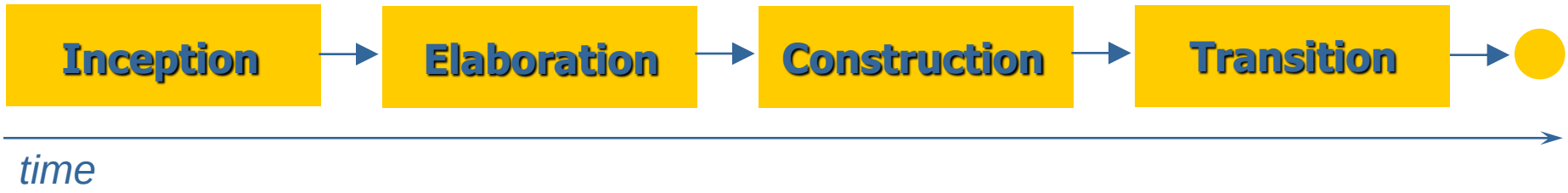


- Earliest iterations address greatest risks
- Each iteration produces an executable release, an additional increment of the system
- Each iteration includes integration and test

Iterative Development Accelerates Risk Reduction



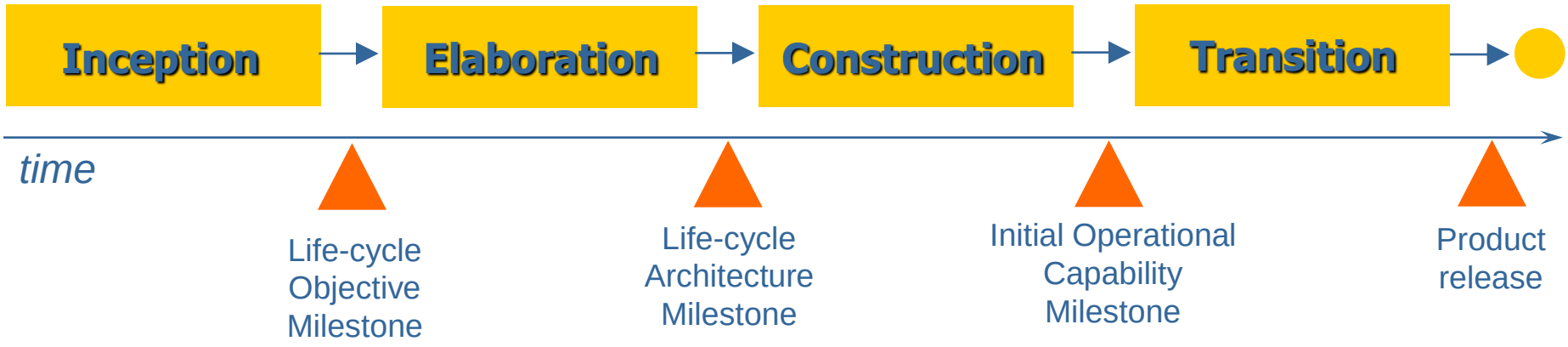
Lifecycle Phases



The Rational Unified Process has four phases:

- **Inception** - Define the vision and scope of project
- **Elaboration** - Plan project, specify features, baseline architecture
- **Construction** - Build product
- **Transition** - Transition product to users

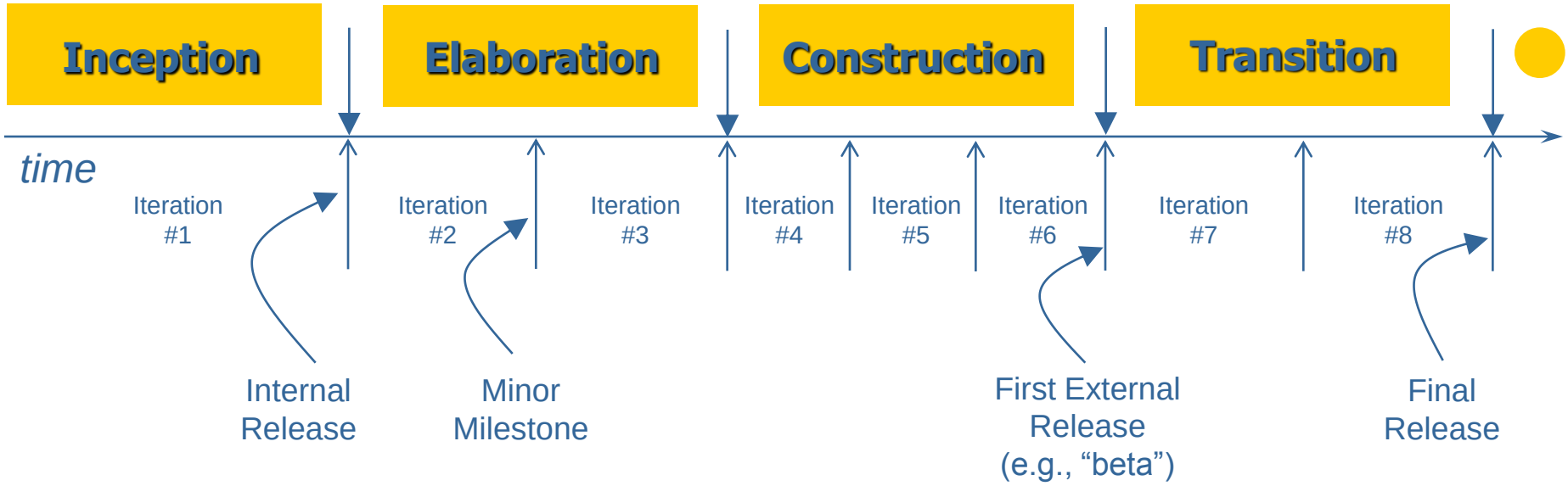
Phase Milestones



K60

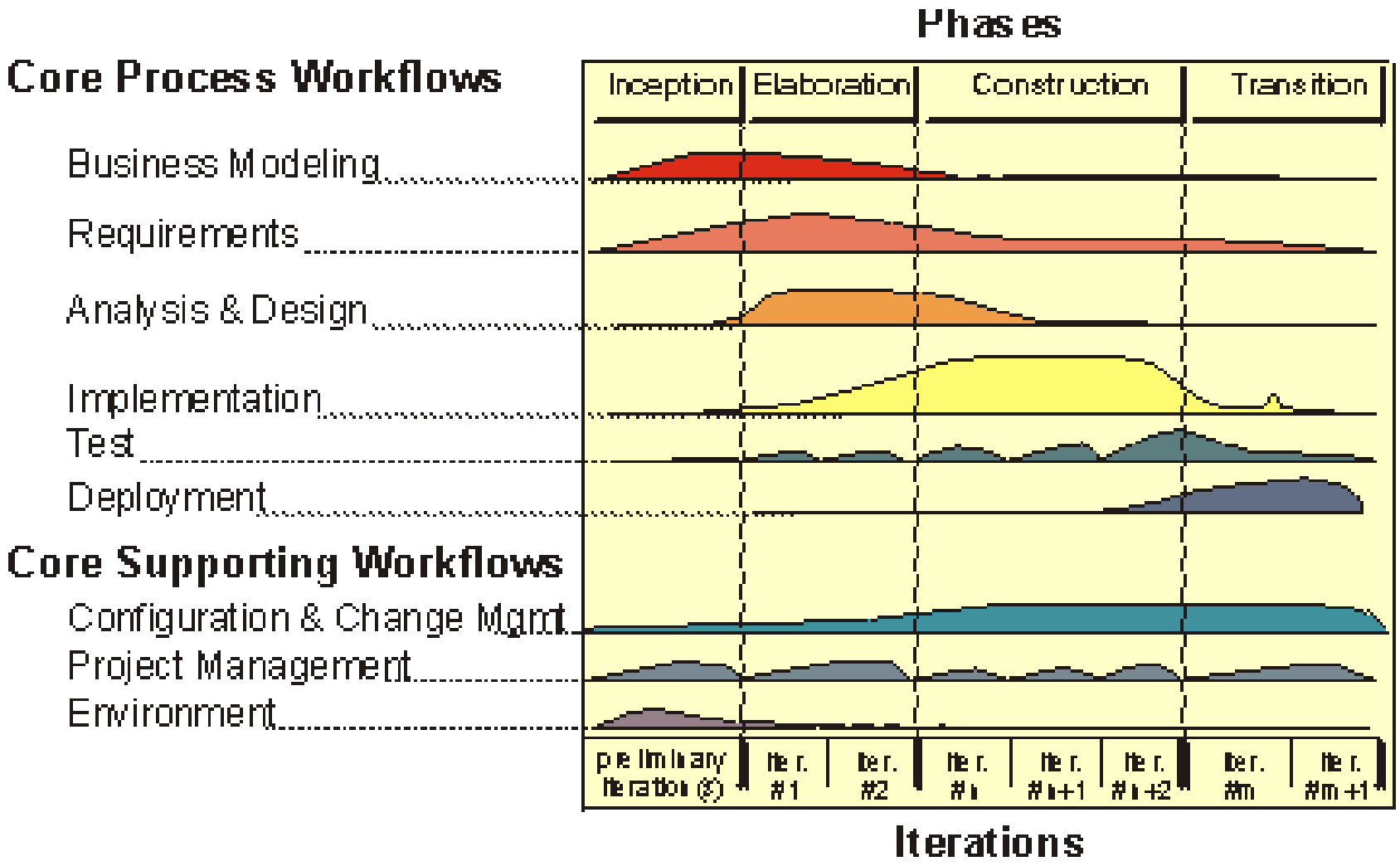
Iterations and Phases

- Each phase consists of one or several iterations.



K62, J9

Overall Architecture of the Rational Unified Process



K45, J11

Benefits of Iterative Development

- Serious misunderstandings become evident early in the life cycle
- Enables and encourages user feedback
- Development focuses on critical issues
- Objective assessment thru testing
- Inconsistencies detected early
- Workload of teams is spread out
- Leverage lessons learned earlier
- Stakeholders are kept up to date on project's status

K8

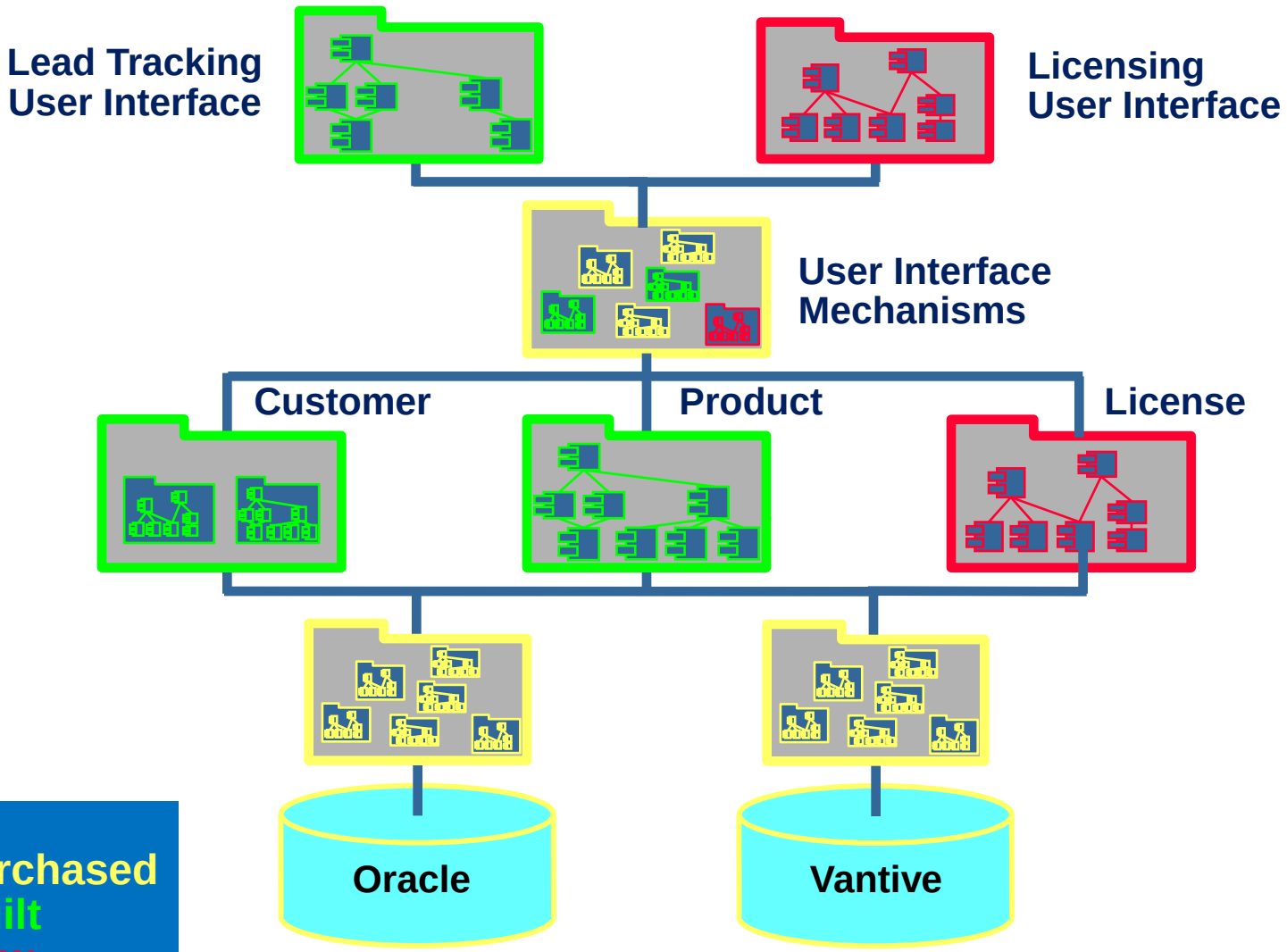
RUP Key Features

- Use case driven
- Iterative and incremental
  - Models, workflows, phases, and iterations
- Architecture-centric

Software Architecture Encompasses

- Organization of a software system
- Selection of the structural elements and their interfaces by which a system is composed
- Their behavior, as specified in collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger subsystems

Example: Component-Based Architecture



K11

Benefits of using Component Architectures

- Components facilitate resilient architectures
- Modularity enables a clear separation of concerns among elements of a system that are subject to change.
- Reuse is facilitated by leveraging standardized frameworks and commercially available components

K10-11

Review Questions

- What are 3 “essential” difficulties in software development?
- What phase of the lifecycle has the highest costs?
- What phase introduces the most significant errors?

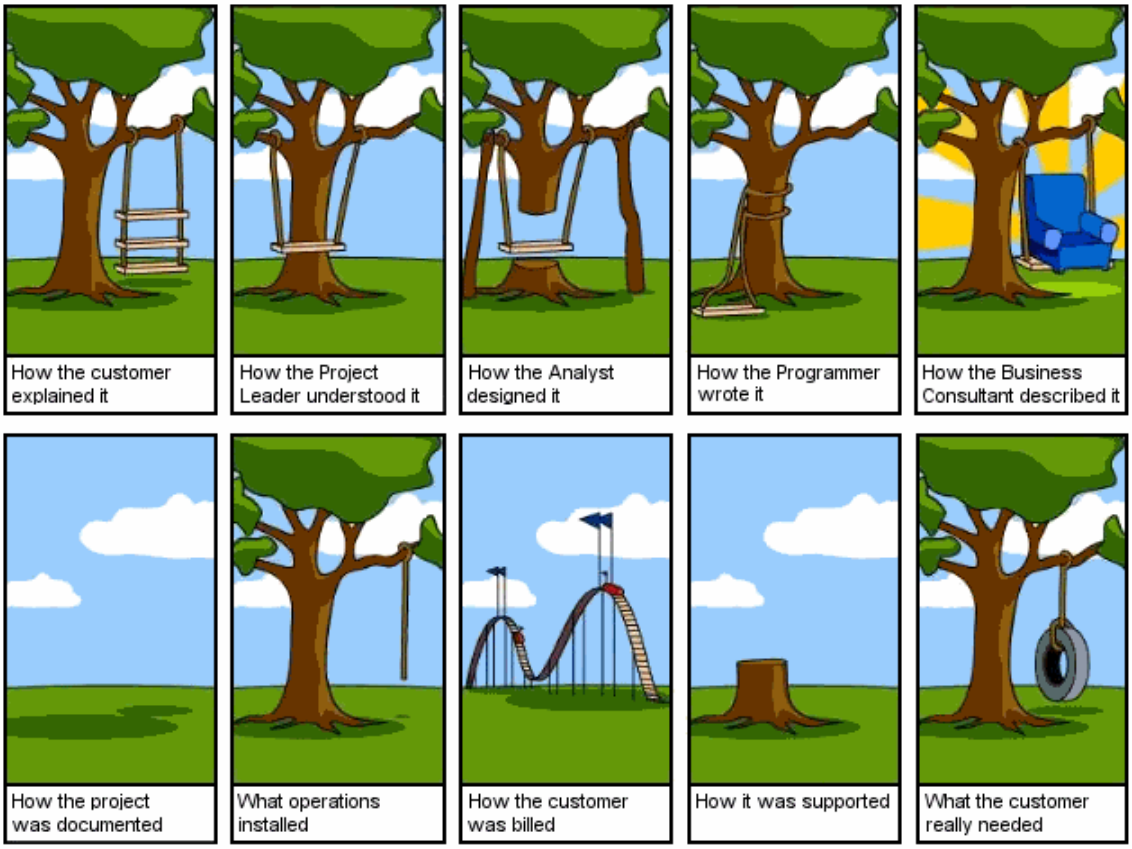
K13-14

Lesson 3

Systems Analysis and Requirements:

Self-Referral Dynamics Present At Every Point Of Creation

We rarely have consistent requirements when we start a project, that is why there are different forms of the following cartoon.



If you don't do all the following:

- Interview all stakeholders for requirements
- Get end-users to articulate their real needs by product management
- Synthesize consistent requirements

Then you will fail to build the correct software. So if you skip any of this work then you are guaranteed to get the response, " It doesn't do what we need ".

Problem Analysis

1. First step in a project is to understand the problem to be solved or the opportunity to be realized. This is accomplished by System Analysts in conjunction with stakeholders

2. This step is known as Problem Analysis. Performing this analysis requires sufficient knowledge of the domain. The problems to be solved must be agreed upon by all stakeholders of the system.

Who Are The Stakeholders?

1. A Stakeholder is anyone who represents an interest group whose needs must be satisfied by the project. The role may be played by anyone who is (or potentially will be) materially affected by the outcome of the project.

2. Different projects may have widely varying stakeholders, and stakeholders may belong to seemingly unrelated groups. Determination of the stakeholders for a project requires careful consideration for each project.

3. Typical examples of interest groups that might be considered Stakeholders for a project:

- Customer or customer representative,
- User or user representative,
- Investor,
- Shareholder,
- Owner,
- Board member,
- Production manager,
- Buyer,
- Designer,
- Tester,
- Documentation writer

Problems → Needs → Features

1. Problems to be solved are typically reformulated as *user needs*, and from these, an initial set of *features* for the new system are listed.

2. Each of these steps of refinement moves in the direction of a concrete specification of requirements for the system. In fact, the goal of System Analysis is ultimately a Software Requirements Specification.

3. The difference between Problems, Needs, and Features is a matter of both detail and orientation. Every problem should be mapped to one or more needs, and every need should be mapped to one or more features that would meet the need.

4. Example:

Problem	Need	Feature
Paper catalogs are clumsy and unsatisfactory to customers	Online catalog	a) System should support online browsing of catalog b) System should support online purchases
System response time is too slow	The part of order processing that depends on paper processing needs to be automated	System should permit orders to be submitted online. Order fulfillment staff should be able to access orders that are stored via a handheld device.

The Vision Document

1. The main artifact emerging from Problem Analysis is the Vision Document.

2. The Vision Document documents the problem, the key needs and features, the business case for the project, and clarifies the scope of the project. It also provides a list of the stakeholders.

3. From RUP:

The Vision Document defines the stakeholders' view of the product to be developed, specified in terms of the stakeholders' ***key needs and features***. Containing an outline of the envisioned core requirements, it provides the contractual basis for the more detailed technical requirements.

4. The Vision document is a primary goal of the Inception phase of development.

Needs And Features → Requirements

1. The Needs and Features discussed in the Vision document are further refined into a list of detailed requirements.

2. Requirements differ from needs and features: Requirements are ***testable.***

3. Example:

Needs/Features from Vision Doc	Software Requirements
Feature 6.1 The defect-tracking system will provide trending information to help the user assess project status	SR6 3.1 Trending information will be provided in a histogram report shoing time on the x-axis and the number of defects found on the y-axis SR6 3.2 The user can enter the trending period in units of days, weeks or months SR6 3.3 An example trend report is shown in attached Figure 1.

Requirements Elicitation

1. Requirements elicitation has inherent challenges because stakeholders are not typically accustomed to thinking about or discussing the kinds of ideas that lead to a software solution. Therefore, requirements (as well as needs and features) often need to be *elicited.*

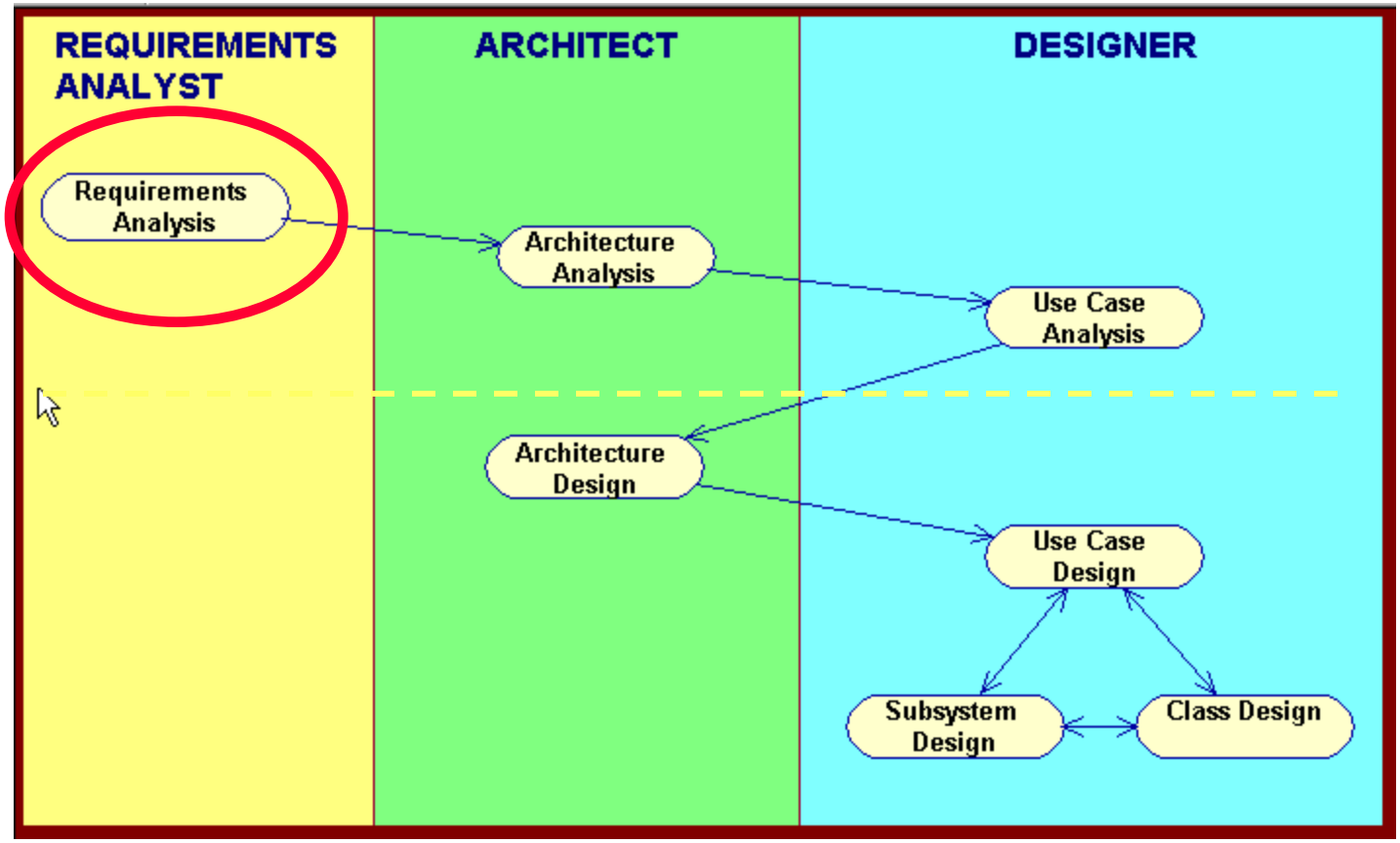
2. Techniques for eliciting requirements:
    a. Interviewing
    b. Requirements workshops
    c. Brainstorming
    d. Storyboarding
    e. Use Cases
    f. Role playing (become the user for a while)
    g. Prototyping

Lesson 4

Rational Unified Process Overview:

Harnessing Nature's Self-Referral Dynamics For Success

Basic RUP OOAD Activities



Overview of Requirements Analysis

Use cases are a widely adopted approach to capturing functional system requirements. Use cases are the integrating view in the RUP model of system architecture, and drive the development process throughout the entire lifecycle.

Roles:

1. System Analyst role leads and coordinates requirements elicitation and use-case modeling by outlining the system's functionality and delimiting the system; for example, identifying what actors exist and what use cases they will require when interacting with the system.
2. The Requirements Specifier role specifies the details of one or more parts of the system's functionality by describing different aspects of the requirements

Library System Problem Statement

You have been hired by Prince University to update their library record keeping. Currently the library has an electronic card catalog that contains information such as author, title, publisher, description, and location of all of the books in the library. All the library member information and book check-in and checkout information, however, is still kept on paper. This system was previously workable, because Prince University had only a few hundred students enrolled. Due to the increasing enrollment, the library now needs to automate the check-in/checkout system.

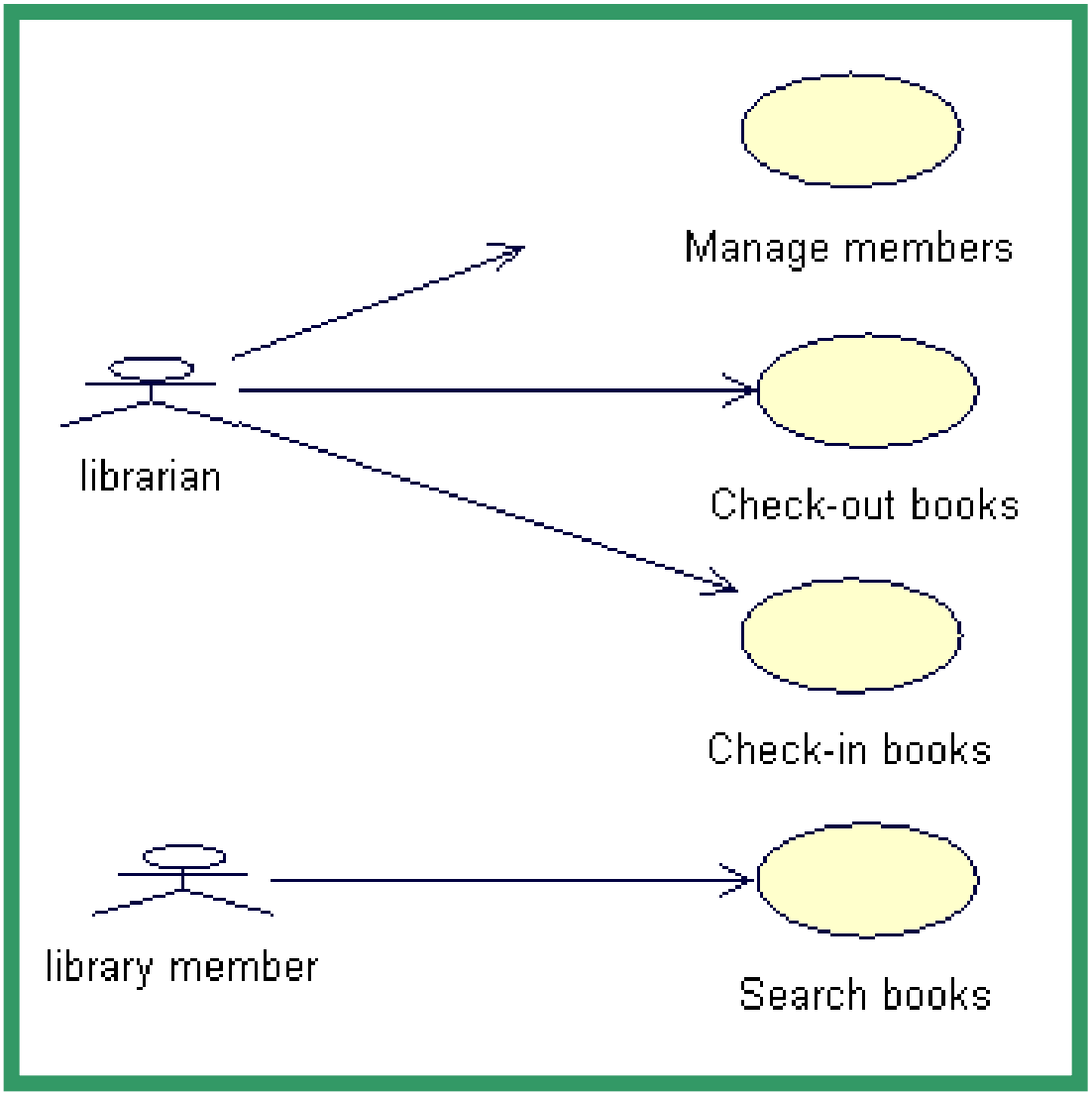
The new system will have a windows-based desktop interface to allow librarians to check-in and checkout books.

All books in the library have a unique bookid.The books in the library are ordered on the shelves by their bookid. The new system must allow library members to search through the electronic card catalog to find the bookid of the desired book.

The system will run on a number of individual desktops throughout the library. Librarians will have their own desktop computers that are not accessible by library members. Only librarians are able to check-in and checkout books.

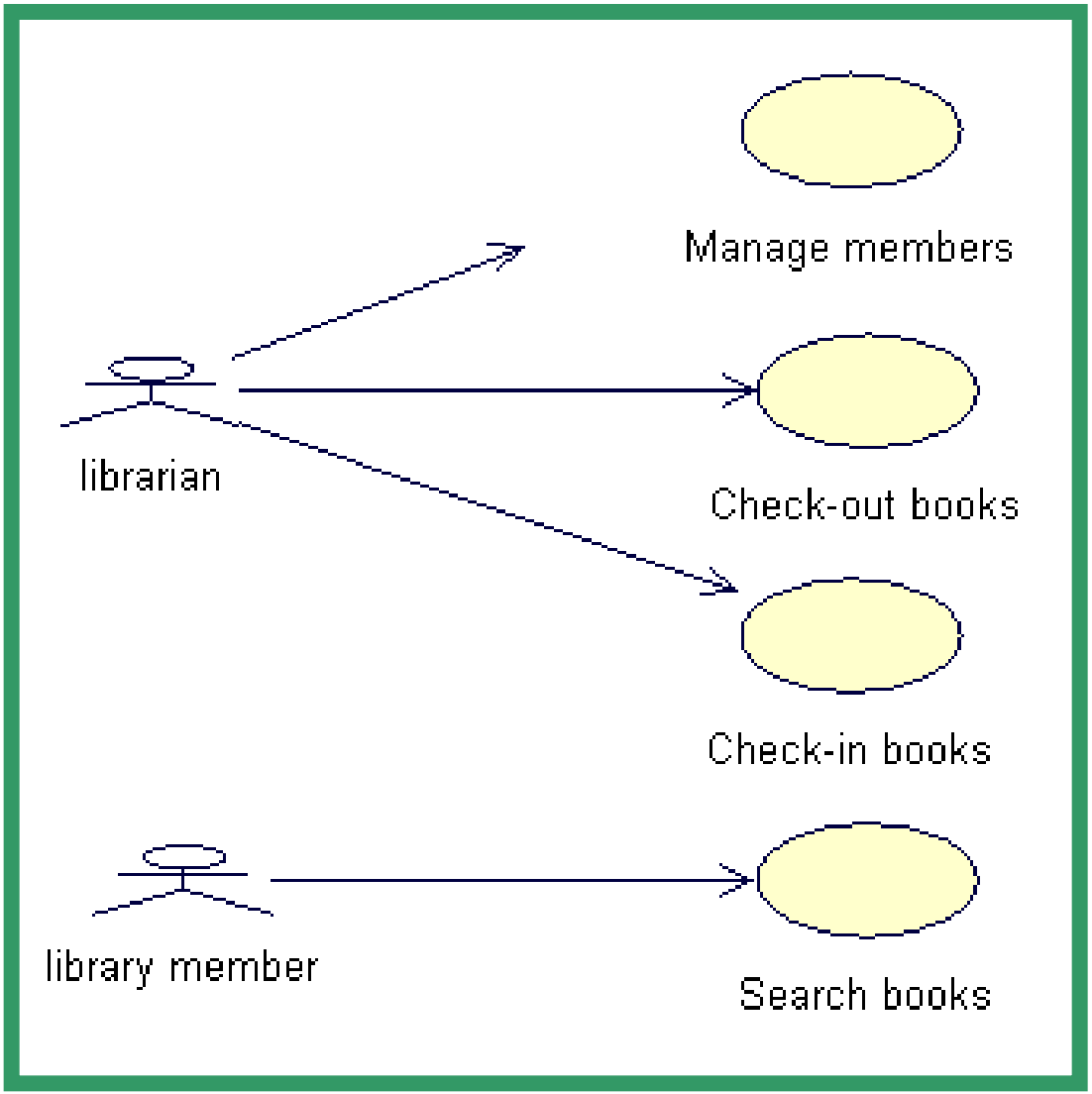
The system will retain information on all library members. Only university students, faculty and staff can become library members. Students can check-out books for a maximum of 21 days. If a student returns a book later than 21 days, then he/she has to pay an overdue fee of 25 cents per day. University staff can also checkout books for a maximum of 21 days, but pay an overdue fee of 10 cents per day. Faculty can checkout books for a maximum of 100 days, and pay only 5 cents per day for every book returned late. The system will keep track of the amount of money that library members owe the library.

Requirements: Use-case diagram



Anything missing?

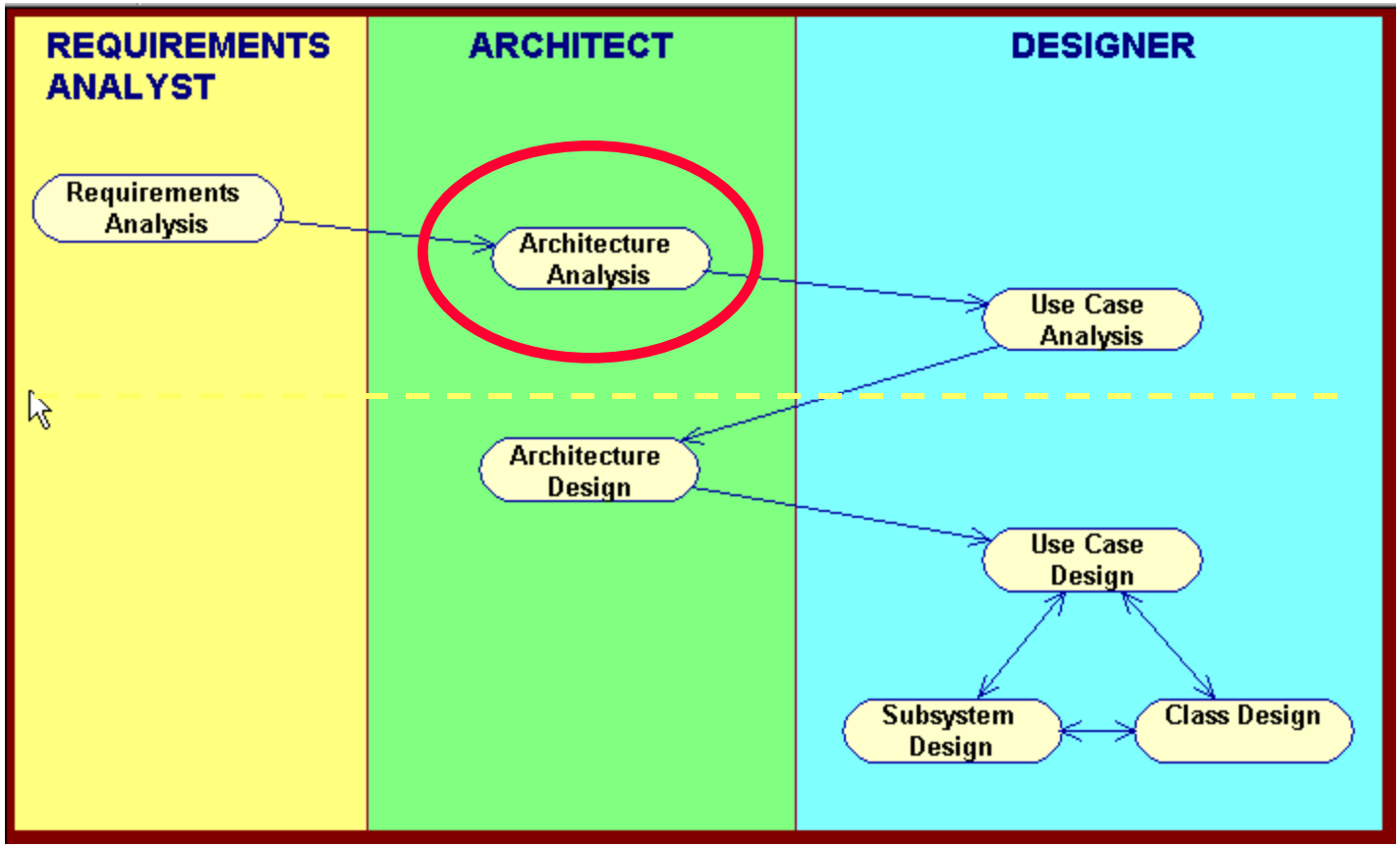
Requirements: Use-case diagram



Members check out books?

Librarians search books?

Basic RUP OOAD Activities



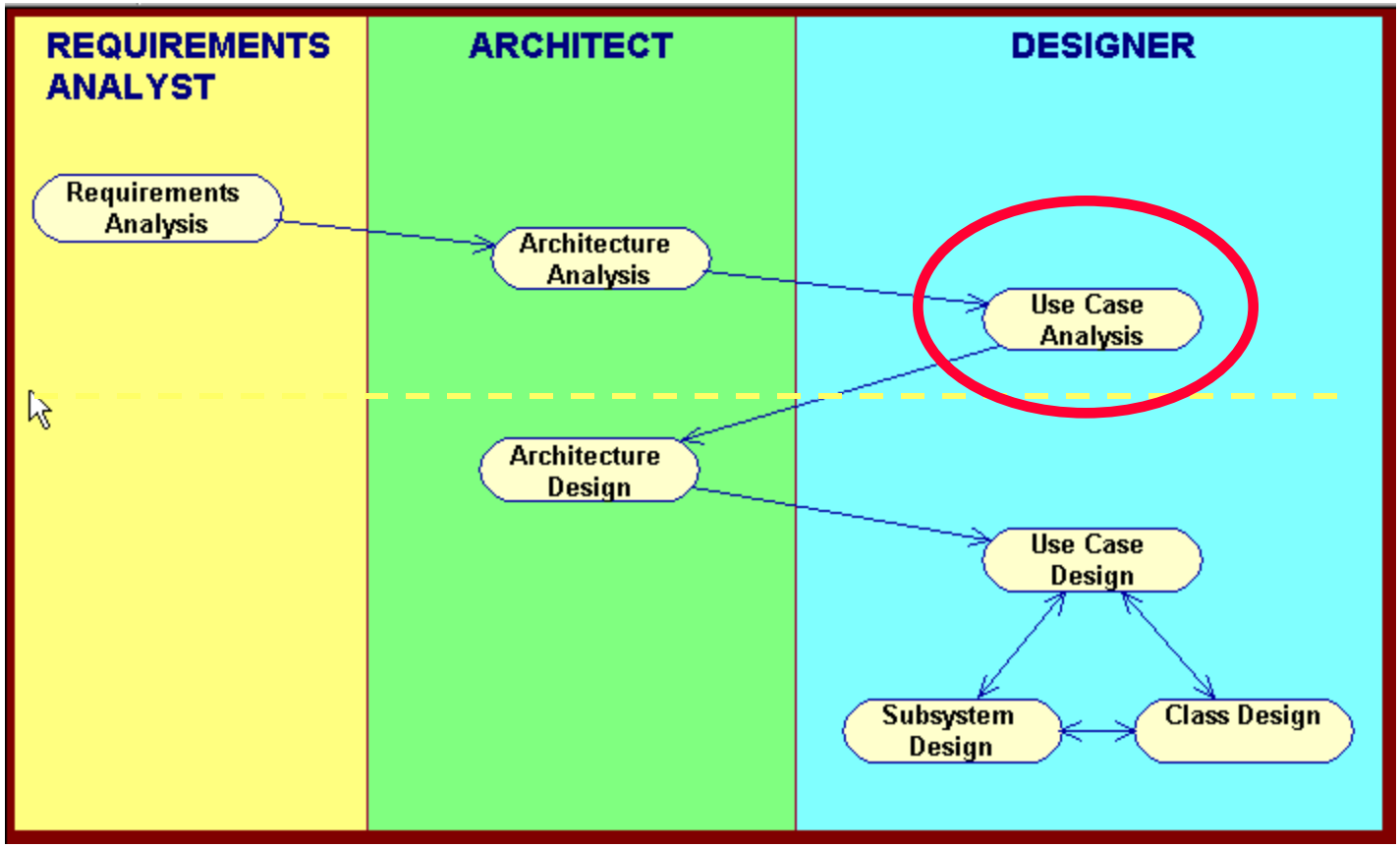
Overview of Arch Analysis

Architectural analysis is a preparation for UC analysis. It provides a comprehensive view of the problem domain and high level system design that will be used across all UC realizations.

Role:

The software architect role is responsible for the software architecture, which includes the key technical decisions that constrain the overall design and implementation for the project.

Basic RUP OOAD Activities



Overview of Use Case Analysis

Analysis is the process of understanding in detail

- the business domain (Systems Analysis)
- the user requirements (Use Case Analysis)

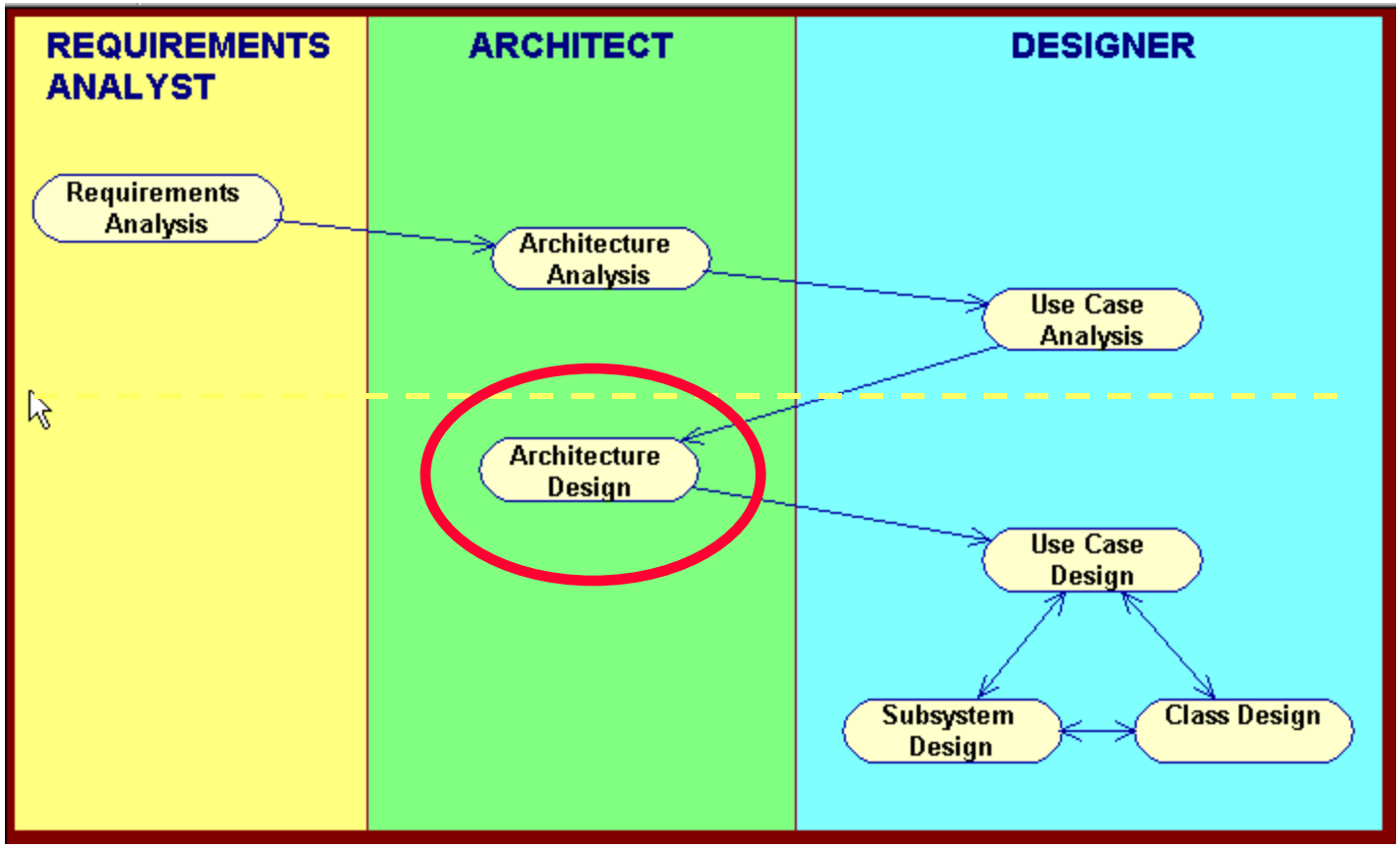
The defining feature of use case analysis is focusing analysis activities on individual use cases. Typically, this phase of analysis makes use of several types of UML diagrams:

1. Use sequence diagrams to determine the responsibilities required for participating classes to carry out each use case.
2. Collaboration diagrams (called Communication Diagrams in UML 2.0) reflect required associations among classes.
3. The VOPC (View of Participating Classes) diagram captures the structural relationships among classes. This is a special kind of *class diagram*

Role:

The designer role is responsible for designing a part of the system, within the constraints of the requirements, architecture, and development process for the project.

Basic RUP OOAD Activities

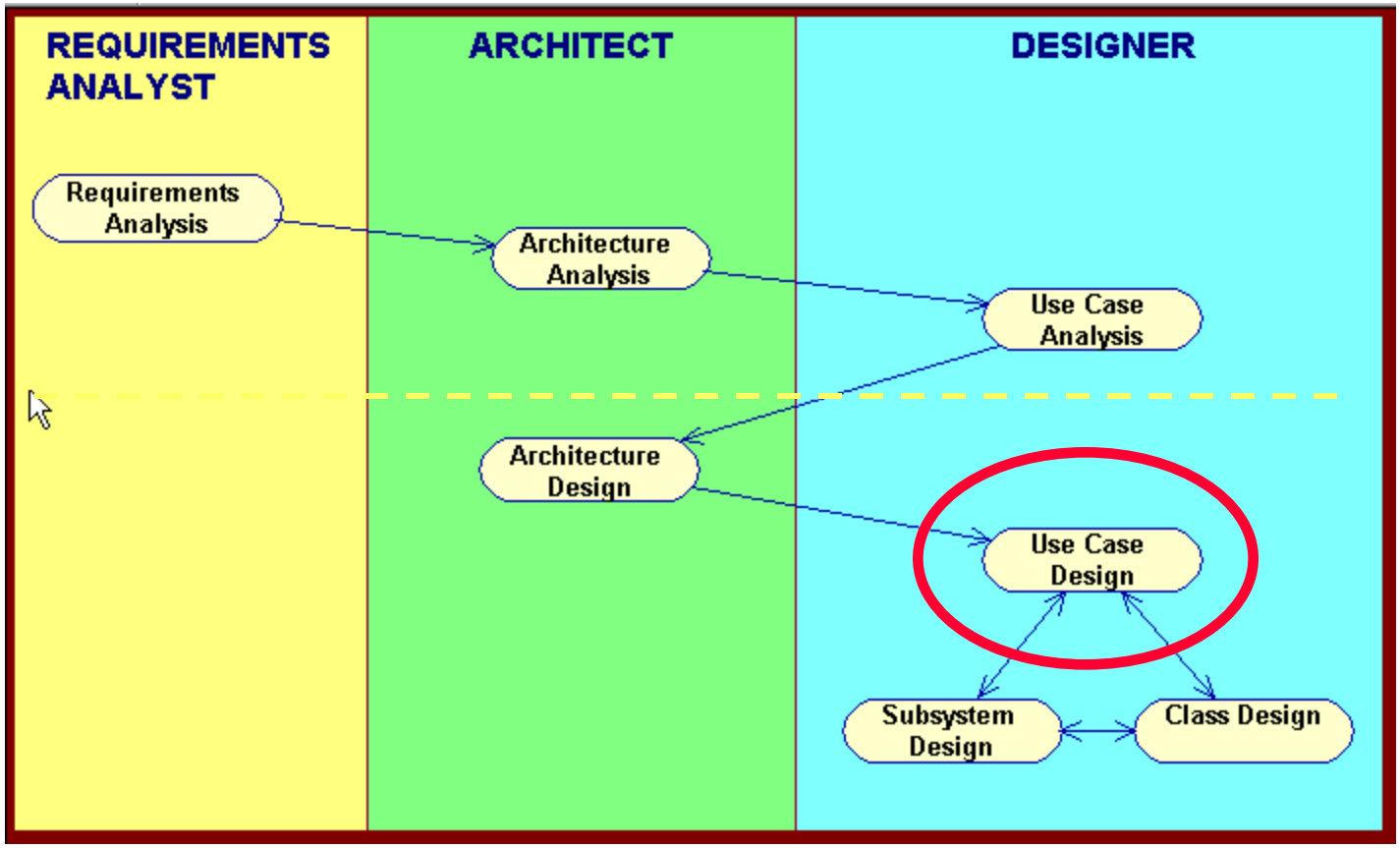


Overview of Arch Design

The architect makes system-wide preparations that guide and coordinate the design activities of individual designers.

- Identify design classes
- Identify packages
- Identify subsystems and their interfaces
- Identify the system layering strategy
- Identify reusable parts at the system level
- Identify components/tools and technology used for the whole project.

Basic RUP OOAD Activities

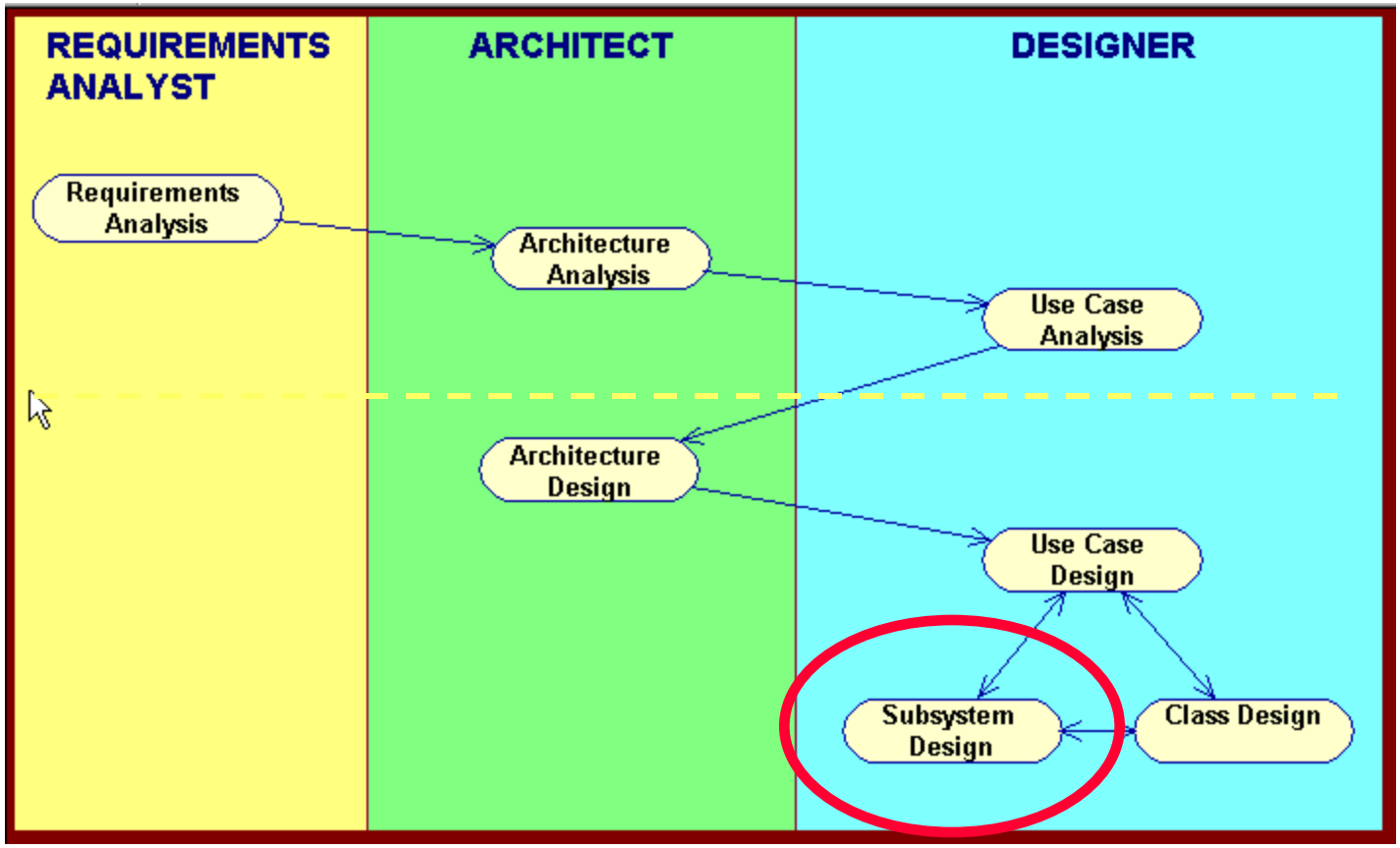


Overview of Use Case Design

During use case design designers inspect and elaborate architectural design results from the perspective of the use cases.

We update our use case diagrams with the architect's design classes, subsystems, layers, components, etc., as needed.

Basic RUP OOAD Activities

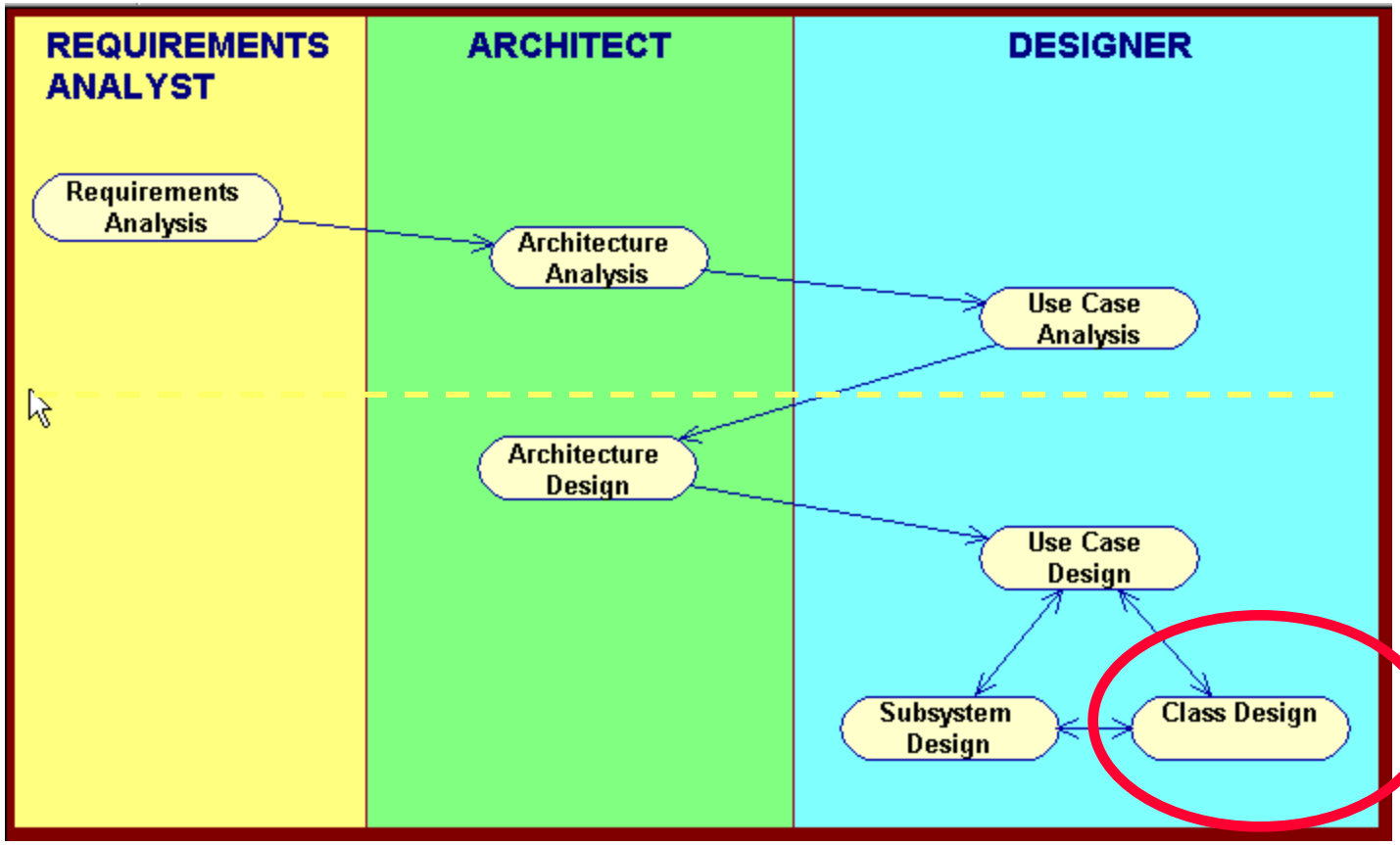


Overview of Subsystem Design

Subsystem design involves working out the details of subsystem operations. The steps are basically the same as those of use case analysis and use case design, except now the work is restricted to a single subsystem (at a time) and there is correspondingly more design detail.

As in use case design, main outputs are interaction (dynamic) and class (static) diagrams of the subsystem elements.

Basic RUP OOAD Activities



Overview of Class Design

Class design is a final preparation for implementation. The overall architecture as well as the functional structure is already set. Class design is making final design refinements.

We focus on refining the operations, attributes, and the relationships for all our classes.

Construction and Transition Roles:

1. Implementer:

The implementer role is responsible for developing and testing components, in accordance with the project's adopted standards, for integration into larger subsystems. When test components, such as drivers or stubs, must be created to support testing, the implementer is also responsible for developing and testing the test components and corresponding subsystems.

2. Tester:

The Tester role is responsible for the core activities of the test effort, which involves conducting the necessary tests and logging the outcomes of that testing.

RUP Artifacts

Requirements

1. Use-case diagram
2. Use-case descriptions

Architectural Analysis

1. Key abstractions
2. Upper level architectural layers

Use-case Analysis

1. Sequence diagrams
2. Collaboration diagrams
3. VOPC diagrams
4. Analysis class to analysis mechanism map

Architectural Design

1. Subsystem context diagram
2. Analysis class to design elements map
3. Design elements to “owning” package map

Use-case Design

1. Updated interaction diagrams including design elements
2. Updated VOPC class diagrams including design elements

Subsystem Design

1. Interaction diagram for each interface definition
2. Class diagram for subsystem

Class Design

1. Refined and updated VOPC class diagrams

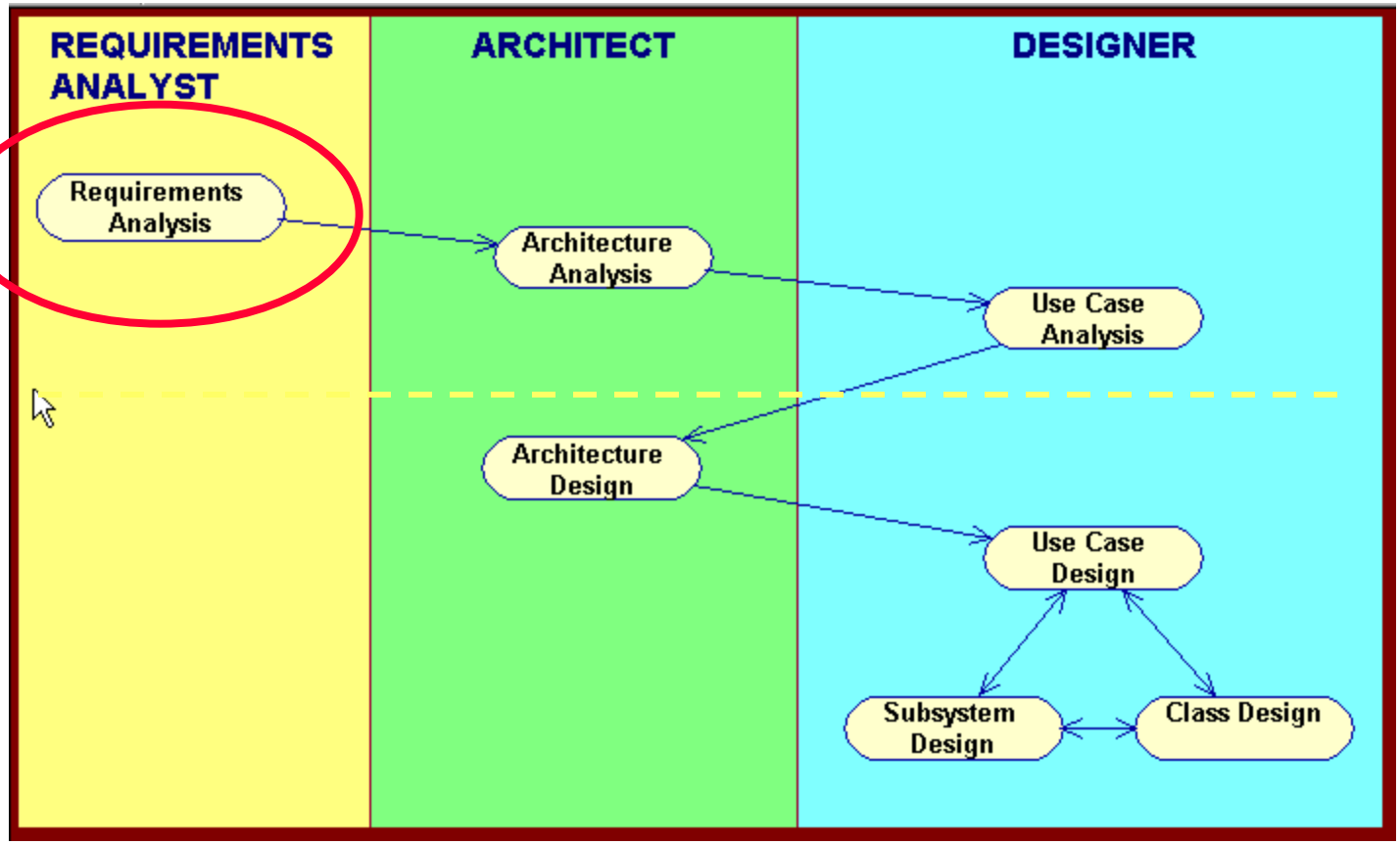
Review of Key Points

- What are the analysis activities?
- What are the design activities?
- What activities are the primary responsibility of the architect?
- What activities are the primary responsibility of the designer?
- What are the 3 diagram artifacts produced during use case analysis?
- What is the purpose of use case analysis?

Lesson 5

The Use Case Approach To Software Requirements:
Unity At The Basis Of Diversity

Basic RUP OOAD Activities



Module Main Points

Theme: Use cases are a widely adopted approach to capturing functional system requirements.

1. The main elements of a RUP Software Requirements Specification:
   - use case models,
   - supplementary specifications,
   - glossary.

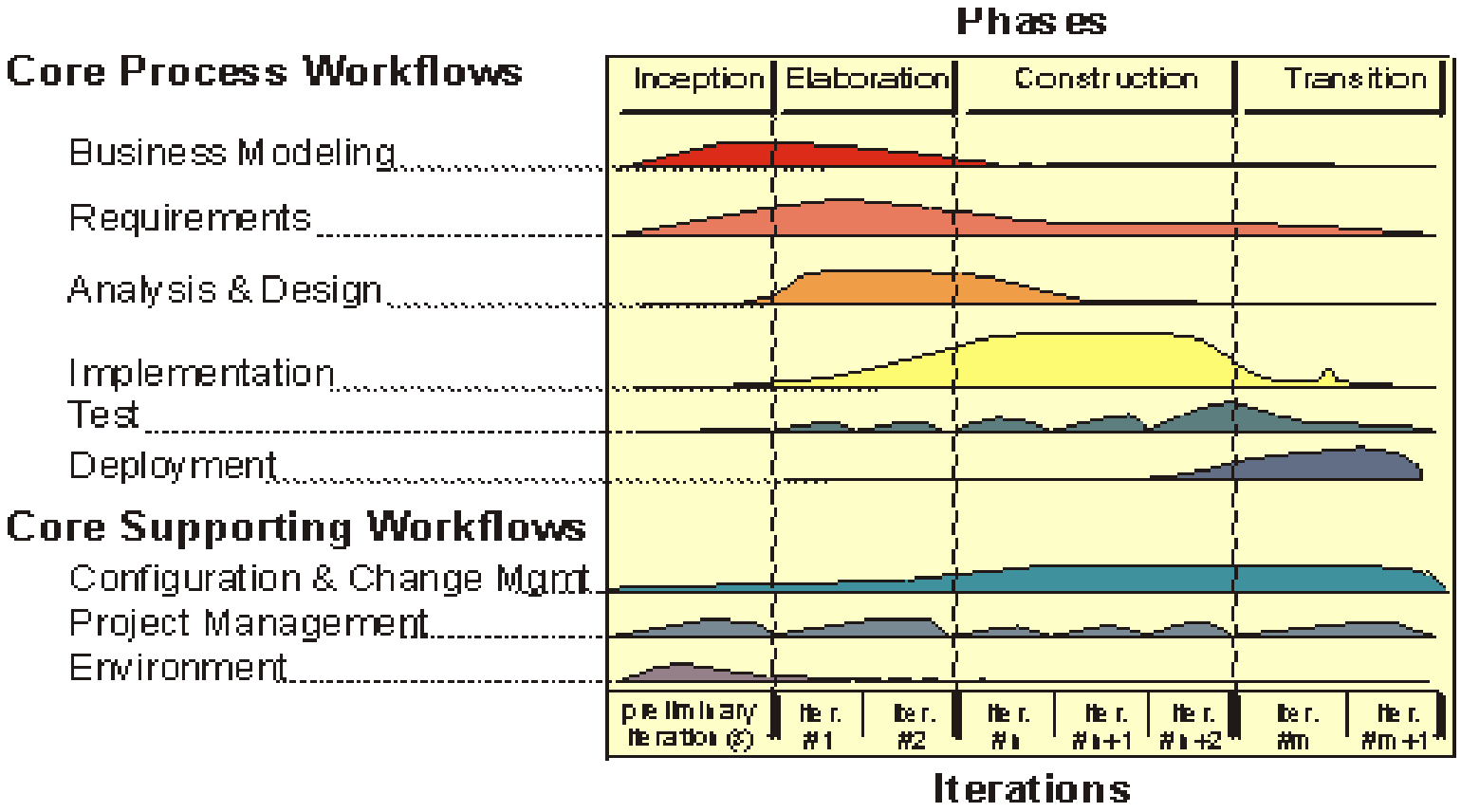
2. A use case is a sequence of actions performed by an actor interacting with the system to achieve a goal, showing how the goal might succeed or fail to be reached

3. Use cases are described in terms of flows and scenarios. A scenario is a single path through a use case. A flow is a set of scenarios that result in the same sort of outcome (e.g., success vs failure flows).

Requirements Analysis Objectives

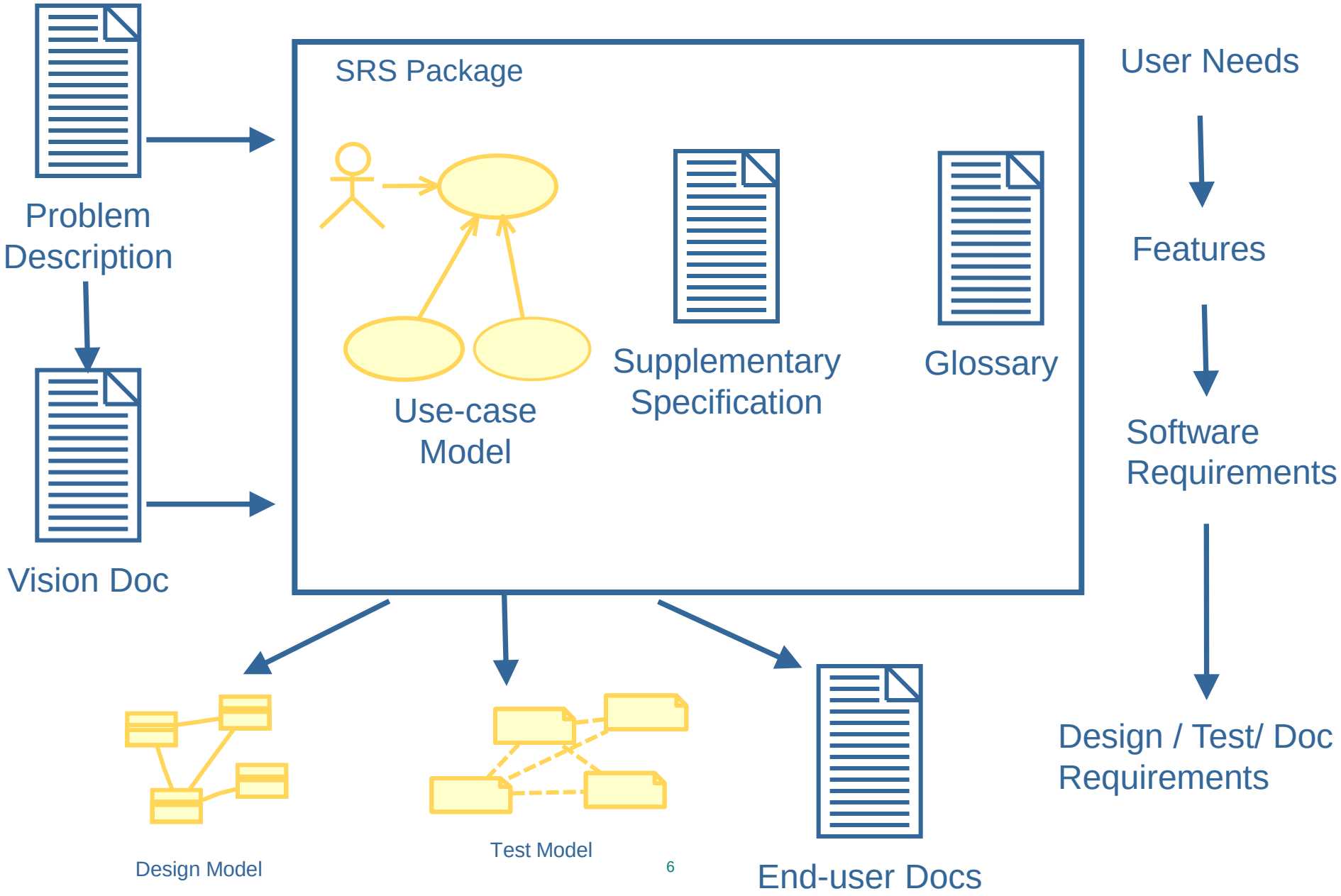
- Understand the content and purpose of the components of a modern Software Requirements Specification
  - Use cases
  - Supplementary Specifications
  - Requirements Glossary
- Understand how use cases are developed, structured, and used
- Understand how use cases integrate and drive the entire software lifecycle

Role of Requirements Analysis in Life Cycle



- Inception: identify most use cases, detail critical ones
- Elaboration: complete about 80% of use cases

Software Requirements Specification



Better Approach to Requirements

- Previous requirements documents often arbitrary collection of paragraphs
  - Poor fit with both business reengineering and implementation
- State requirements in terms of how end-users use the system
  - Easier for end-users to understand and relate to

Requirements Analysis Topics

- A modern software requirements specification (SRS)
- Use-Case Model
  - Use cases drive entire development lifecycle
  - Use case diagrams
  - Use case descriptions
  - Guidelines for developing use cases
    - Business rules in use cases
    - Structuring complex use cases
- Supplementary Specs
- Glossary
- Review

What Is a Use Case?

- A way in which a user interacts with a system in order to achieve some goal
- Goal and the use case can often be used interchangeably
- Example: Withdraw money use case
  - Bank Customer identifies himself or herself, and system verifies
  - Bank Customer chooses from which account to withdraw money and specifies how much to withdraw, and system deducts the amount from the account and dispenses the money

Summary: Use Cases Drive Overall Development

- Use cases form the basis of a modern SRS document. Organizing the requirements document around user goals and actions facilitates user understanding of the requirements and the eventual ***validity and acceptance testing*** of the finished system.

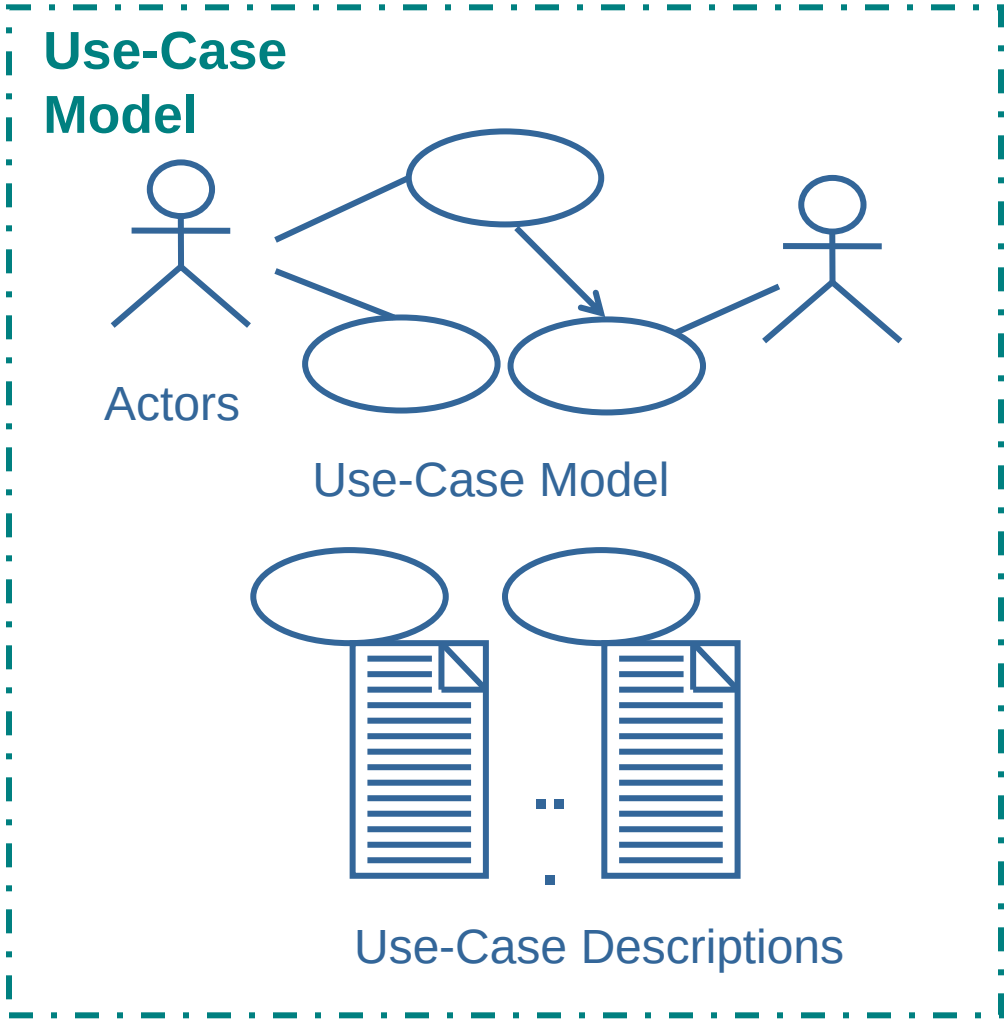
Requirements Analysis Topics

- A modern software requirements specification (SRS)
- Use-Case Model
  - Use cases drive entire development lifecycle
  - Use case diagrams
  - Use case descriptions
  - Guidelines for developing use cases
    - Business rules in use cases
    - Structuring complex use cases
- Supplementary Specs
- Glossary
- Review

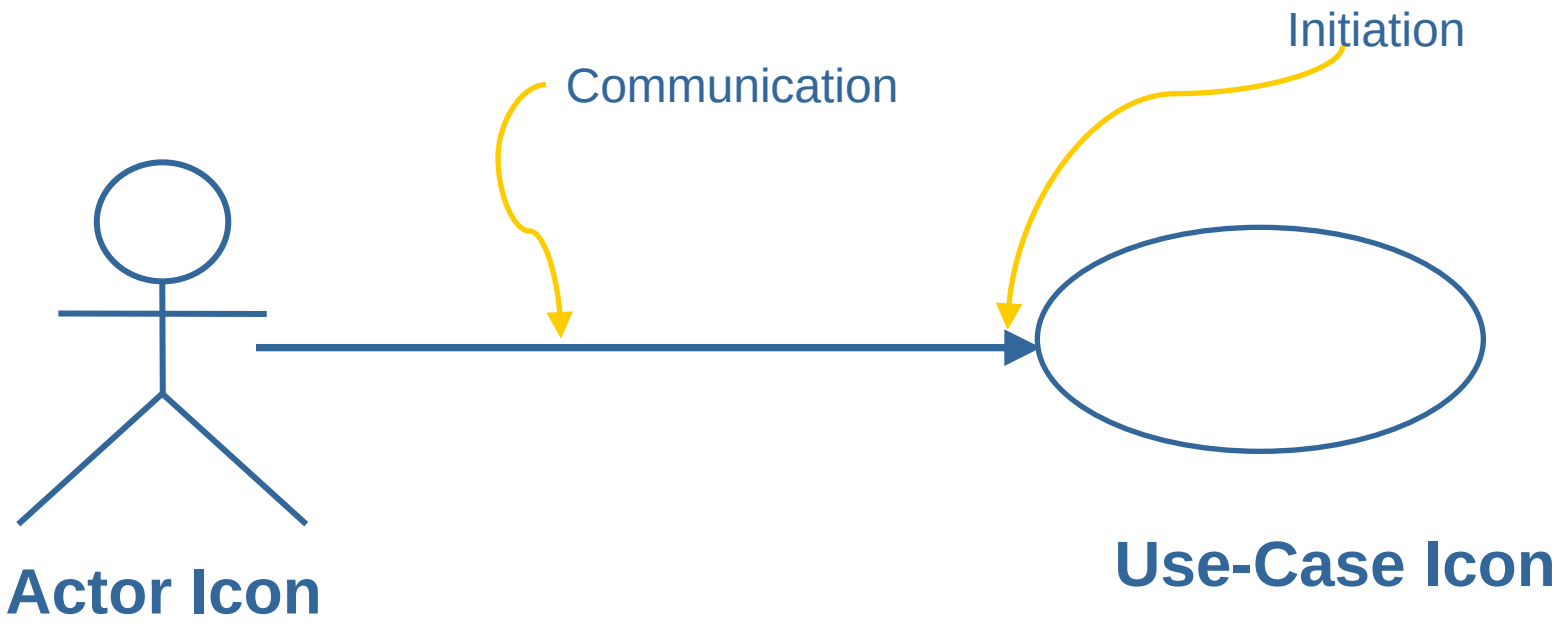
Use-Case Model = Diagrams and Descriptions

- The use-case model is composed of :

1. use-case diagram(s)
2. use-case descriptions

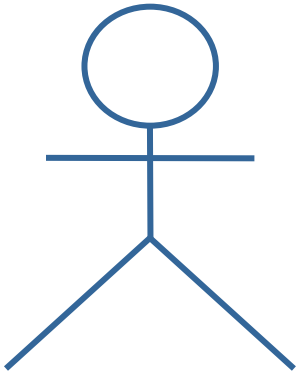


Use Case Diagram Basic Components



Use-Case Modeling: Actors

- The use case model starts in the Inception Phase with the identification of actors and principal use-cases.
- Actors are not part of the system--they represent anyone or anything that must interact with the system

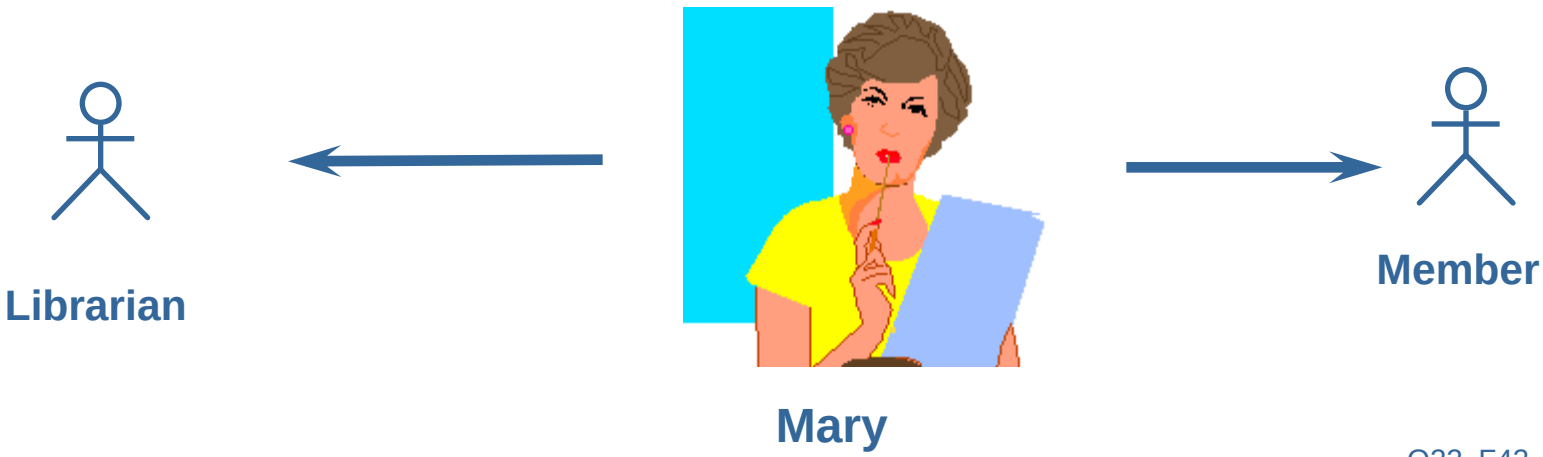


Actor Icon

Q21

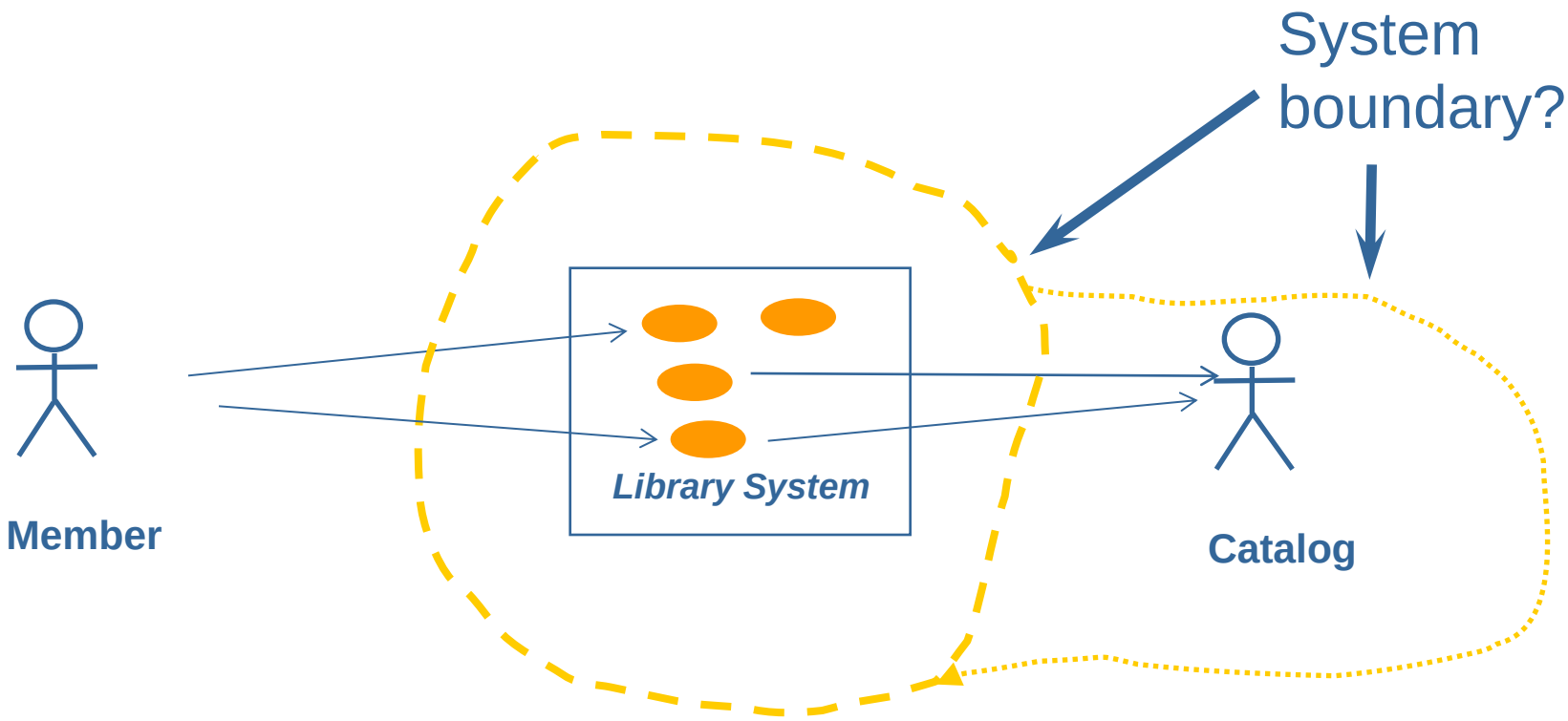
A User May Have Different Roles

- The same person might appear as different actors
- Does the person use the system in a significantly different manner in the different roles?



Q22, F42-43

Actors are External to the System

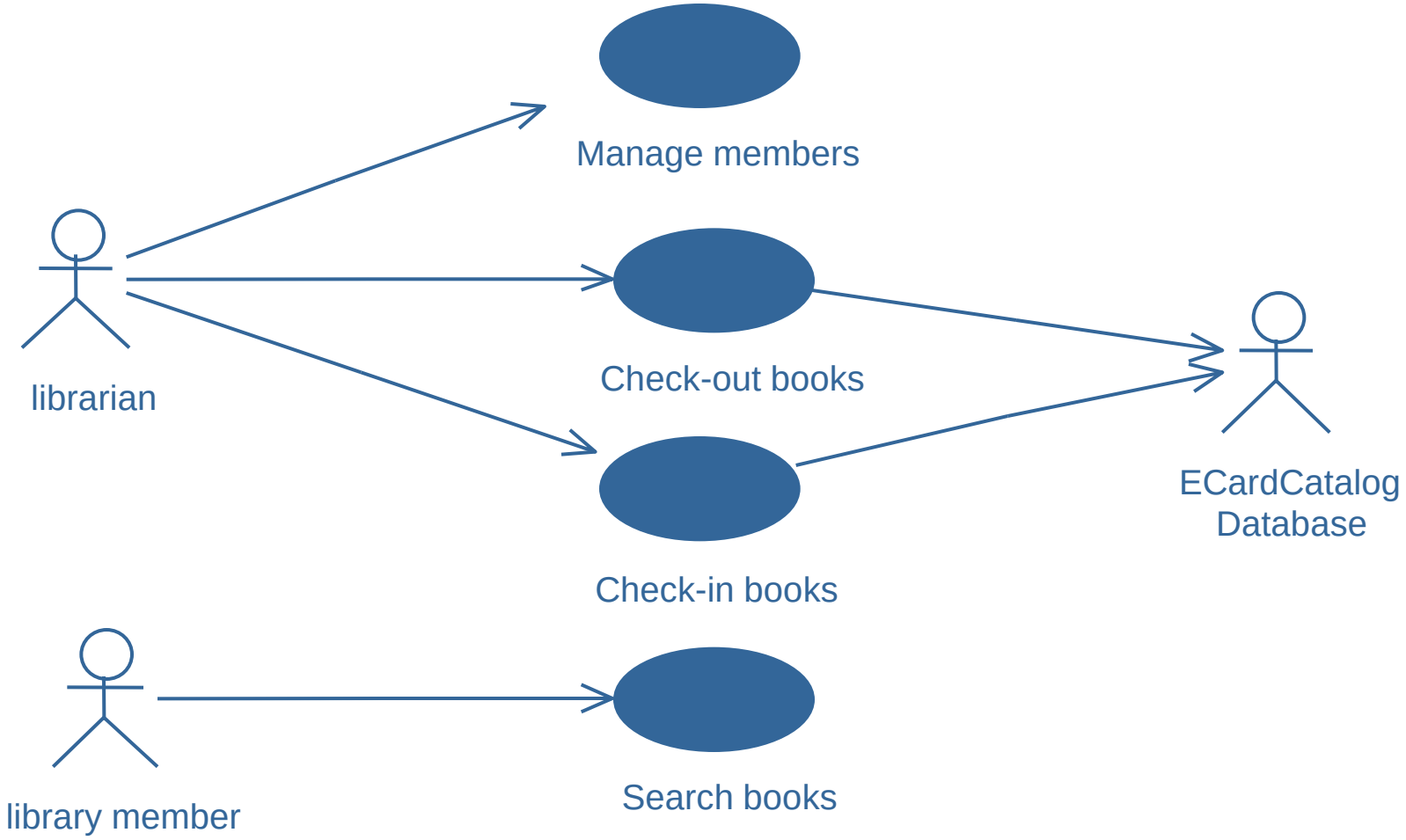


Example: List of Actors and Use-Cases

- Actors
  - Library Member
  - ECardCatalog
  - Librarian
- Use cases
  - Search book
  - Check-in books
  - Check-out books
  - Manage members

Q21

Example: Use-Case Model: Use-Case Diagram



Group Exercise

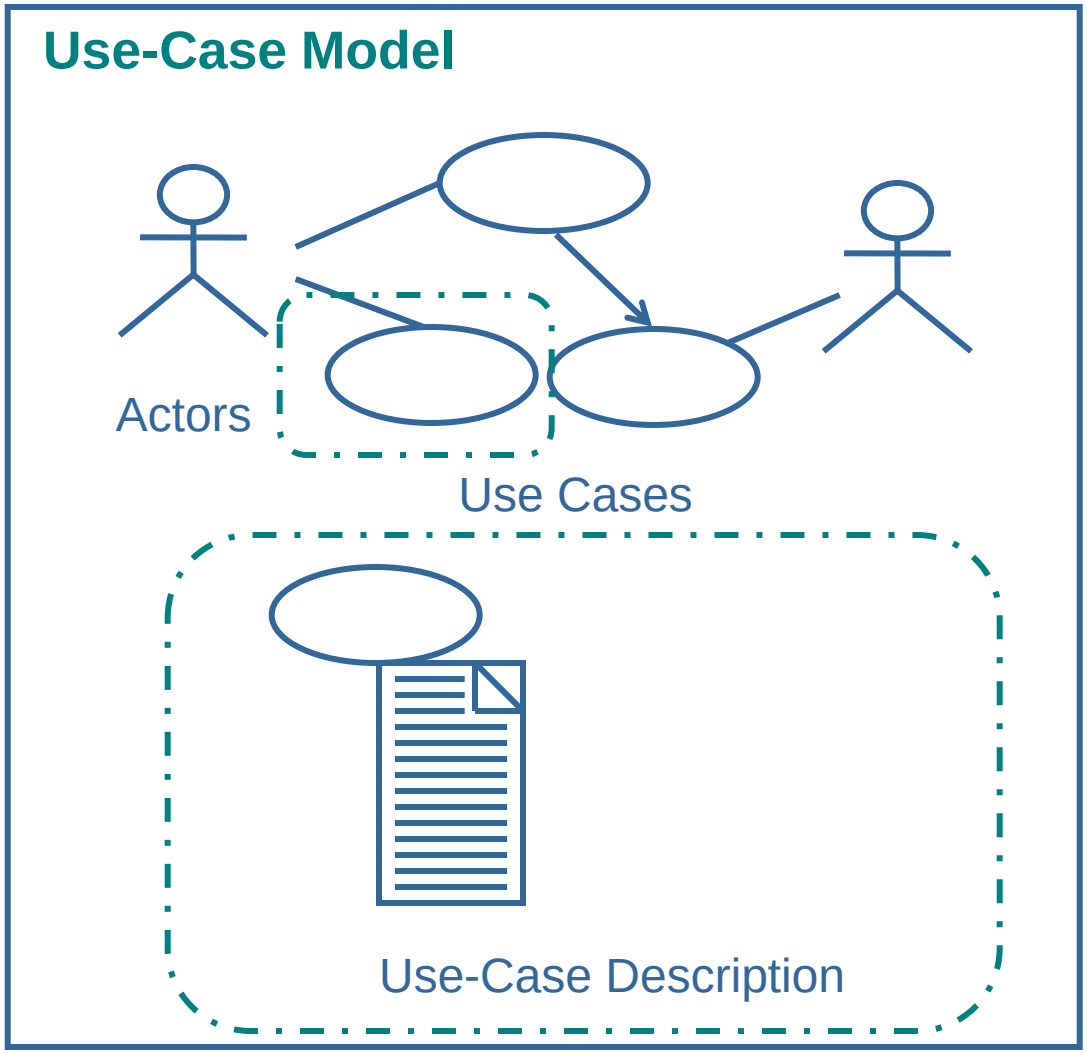
- Each group create a first draft use case diagram for our MUMScrum project
- Think about actors and possible use cases

Requirements Analysis Topics

- A modern software requirements specification (SRS)
- Use-Case Model
  - Use cases drive entire development lifecycle
  - Use case diagrams
  - Use case descriptions
  - Guidelines for developing use cases
    - Business rules in use cases
    - Structuring complex use cases
- Supplementary Specs
- Glossary
- Review

Use Case Descriptions

- Name
- Brief description
- Flows of Events
- Relationships
- Activity and State diagrams
- Use-Case diagrams
- Special requirements
- Preconditions
- Postconditions
- Other diagrams



What Are Scenarios ?

- A scenario is an instance of a use case
- One path
- Like one boat on the river

Use-Case Flows of Events

- A use case “flow” is a collection of scenarios that have a common structure
- Like several boats floating together on the current
- A use case has one normal, *basic flow* and possibly several other *alternative flows* to handle:
  - Regular variants (business logic alternatives)
  - Odd cases
- **Exceptional flows** handle error situations (like system, DB errors, etc.) Things we might catch with exceptions of some type.

Use-Case Description Example (1/4)

Checkout Book

This use case allows the librarian to check-out books that library members want to borrow.

Actors

Librarian

Flow of Events

4.1 Basic Flow

User Action	System Response
1. The librarian enters the library member's ID and requests the system to retrieve the member information	1. The system retrieves the library member's name, address and phone number together with the list of books he/she has borrowed, and shows this information on the Checkout form. The system also retrieves the current balance that the library member owes the library for overdue books.
2. The librarian enters the ID of the book being checked out, and requests the system to retrieve information on the book.	2. The system retrieves the book title, author list and ISBN number and shows this information on the screen.
3. The librarian requests the system to checkout the book.	3. The checked-out book is added to the members checked-out book list, and fields that show the book information are cleared.

Use-Case Description Example (2/4)

Alternate Paths

Library Member Not Found

If the member id does not exist, the system displays a message that the member id cannot be found.

Book Not Found

If the book id does not exist, the system displays a message that the book id cannot be found.

Library member pays some amount towards his negative balance.

User Action	System Response
1. The librarian enters the library member's ID and requests the system to retrieve the member information	1. The system retrieves the library member's name, address and phone number together with the list of books he/she has borrowed, and shows this information on the Checkout form. The system also retrieves the current balance that the library member owes the library for overdue books.
2. After receiving payment, the librarian enters the paid amount information and selects the Update balance button.	2. The system computes the library member's new balance.

Use-Case Description Example (3/4)

Pre-Conditions

Use is logged in to the system

Post-Conditions

New checkout record is added to the Library DB whenever checkout is successful.

Business Rules

Only members with a valid memberID can checkout books.

Only books that are found in the system an be checked out.

Students cannot checkout books if outstanding fines > $10.00.

Use-Case Description Example (4/4)

Note that ***alternate flows*** are used to explain the system response to Business Rule checks:

Business Rules

Only members with a valid memberID can checkout books.

Only books that are found in the system an be checked out.

Students cannot checkout books if outstanding fines > $10.00.

What would be some examples of exceptional flows?

Do we need to create use-case descriptions for those exceptional flows?

Recording Design Issues

- As Use Case descriptions are being developed, questions will arise about how to design or implement aspects of the Use Case
- These issues cannot be resolved completely during analysis, but should be recorded in a Design Issues document that grows over time.

Design Issues For E-Bazaar

Number	Name	Issue	Notes	Design Decision
1	DATA_ACCESS	Making repeated use of JDBC calls to read and write data requires the same code over and over. Better to generalize, but how?	A "data access subsystem" that encapsulates the repetitive steps for JDBC access.	
2	LOGIN	When a user logs in, how do we keep track of the fact that this has happened so that we don't require him to log in again?		
3	CUST_ID	After a user logs in, how do we store in memory the customer ID?		

Recording Design Issues

Have you hit any design issues with MUMScrum?

Requirements Analysis Topics

- A modern software requirements specification (SRS)
- Use-Case Model
  - Use cases drive entire development lifecycle
  - Use case diagrams
  - Use case descriptions
  - Guidelines for developing use cases
    - Business rules in use cases
    - Structuring complex use cases
- Supplementary Specs
- Glossary
- Review

Use Cases DO's

- Use Cases should be named in a way that suggests the goal of the use case -- that is, the goal of the actor in its interaction with the system for this use case.
- A Use Case should include a set of successful paths from a trigger event to the goal (success scenarios)
- Should also record a set of paths from a trigger event that fall short of the goal (failure scenarios)
- Evolve to more detail over the lifecycle

Use Cases DON'Ts

- A Use Case should not specify user interface design
  - Usually it is too early to commit to interface details
  - Instead, the UI is typically designed on the basis of the requirements.
  - However, Use Cases often offer suggestions about the look of the UI as an aid to understand the flows of the Use Case. These suggestions may be adopted, modified, or rejected completely when the UI is designed later on, but the principles they illustrate will survive in one form or another.
- A Use Case should not attempt to specify implementation detail beyond any technical constraints that may have been imposed on the project.

J160-6

Recommended Steps for Creating Use Cases

- Identify actors
- Name use cases
  - Start with verb, reflect goal
- Brief description
- Main success scenario
- Pre and post conditions
- Alternate flows
- Exceptional flows—what might go wrong
- Business rules
- Associated non-functional requirements

Requirements Analysis Topics

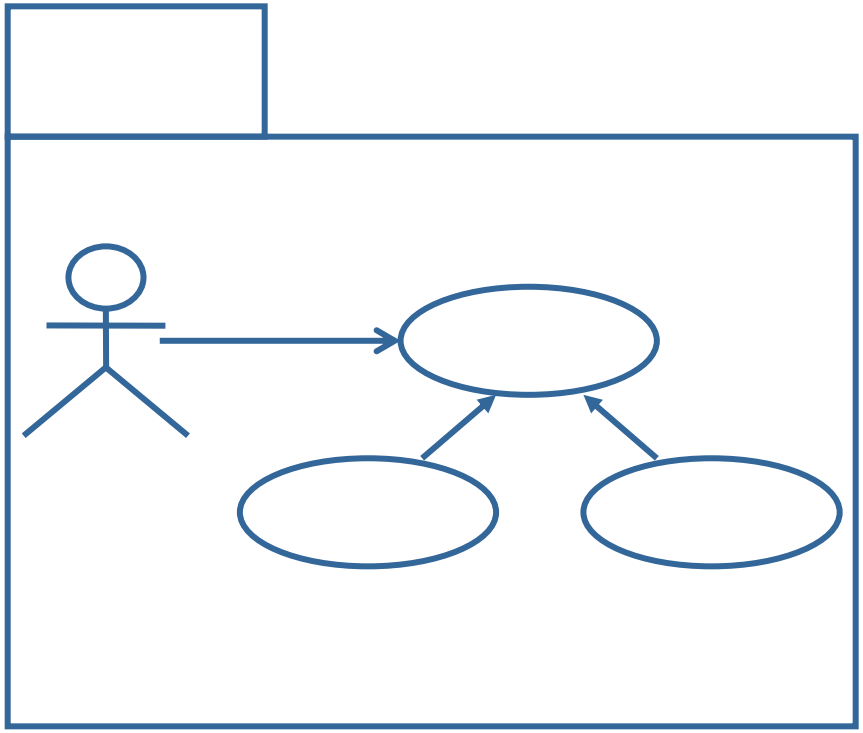
- A modern software requirements specification (SRS)
- Use-Case Model
  - Use cases drive entire development lifecycle
  - Use case diagrams
  - Use case descriptions
  - Guidelines for developing use cases
    - Business rules in use cases
    - Structuring complex use cases
- Supplementary Specs
- Glossary
- Review

Structuring Complex Use Cases

- Packages
- Use case relationships

Organize Related Use Cases in Packages

- Semantically related groups
- Development groups
- Delivery groups



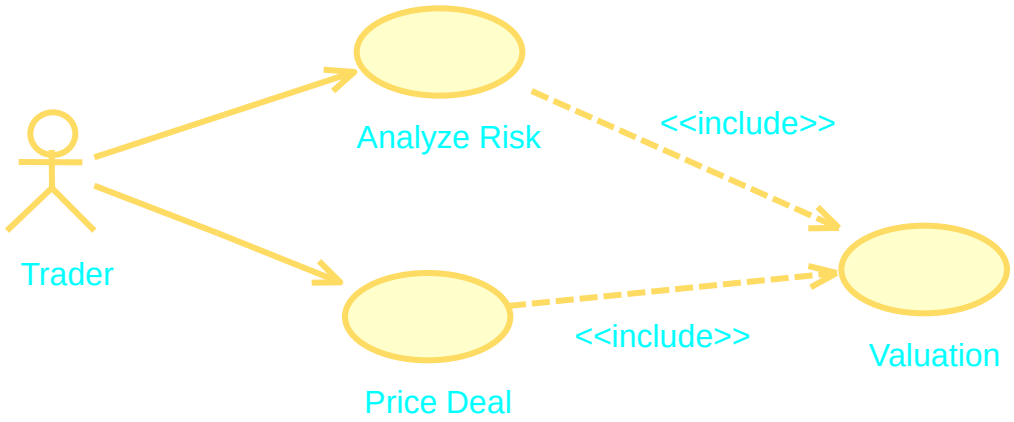
Use Case Relationships

- Decompose complex use cases
- **Include**
  - used sometimes
- Extend -- we ignore
- Generalize – we ignore

Q32-34, F44-45

Use Case Relationships: Include

- Common steps used in several use cases
- Subgoals/subflows
- E.g., login, print, valuation



F42-45

Requirements Analysis Topics

- A modern software requirements specification (SRS)
- Use-Case Model
  - Use cases drive entire development lifecycle
  - Use case diagrams
  - Use case descriptions
  - Guidelines for developing use cases
    - Business rules in use cases
    - Structuring complex use cases
- Supplementary Specs
- Glossary
- Review

Nonfunctional Requirements

- E.g., performance, security, user sophistication, hardware, software, …
- Attach use-case specific ones to use cases
- Others in supplementary specifications list

Supplementary Specification

- Functionality
- Usability
- Reliability
- Performance
- Supportability
- Design constraints

Supplementary Specification

Supplementary Spec Example (1/2)

Objectives

The purpose of this document is to define non-functional requirements of the Library System. This Supplementary Specification lists the requirements that are not readily captured in the use cases of the use-case model. The Supplementary Specifications and the use-case model together capture a complete set of requirements on the system.

Scope

This Supplementary Specification applies to the Library System .
This specification defines the non-functional requirements of the system; such as reliability, usability, performance, and supportability as well as functional requirements that are common across a number of use cases. (The functional requirements are defined in the Use Case Specifications.).

Supplementary Spec Example (2/2)

Reliability
The main system must be running 95% of the time.

Performance
The system shall support up to 15 simultaneous users against the central database at any given time.

Security
None

Design Constraints
The system shall integrate with an existing legacy system, the ECardCatalog Database, which is a MS Access database running on an Windows NT workstation.

Requirements Analysis Topics

- A modern software requirements specification (SRS)
- Use-Case Model
  - Use cases drive entire development lifecycle
  - Use case diagrams
  - Use case descriptions
  - Guidelines for developing use cases
    - Business rules in use cases
    - Structuring complex use cases
- Supplementary Specs
- Glossary
- Review

Glossary

- Problem domain terms
- Facilitates common understanding among developers as well as domain experts

Glossary

Glossary Example (1/2)

1. Introduction

This document is used to define terminology specific to the problem domain, explaining terms, which may be unfamiliar to the reader of the use-case descriptions or other project documents. Often, this document can be used as an informal *data dictionary*, capturing data definitions so that use-case descriptions and other project documents can focus on what the system must do with the information.

2. Definitions

The glossary contains the working definitions for the key concepts in the Library System.

2.1 ECardCatalog Database

The legacy database that contains all information regarding books in the library.

2.2 Library Member

Person who checks-out books from the library.

Glossary Example (2/2)

2.3 Librarian

Person who works for the library and performs the check-in and check-out procedure. This person also manages library member information.

2.4 Check-out Book

The procedure done if a library member wants to take a library book out of the library.

2.5 Check-in Book

The procedure done if a library member returns a checked out book to the library.

2.6 Overdue Fee

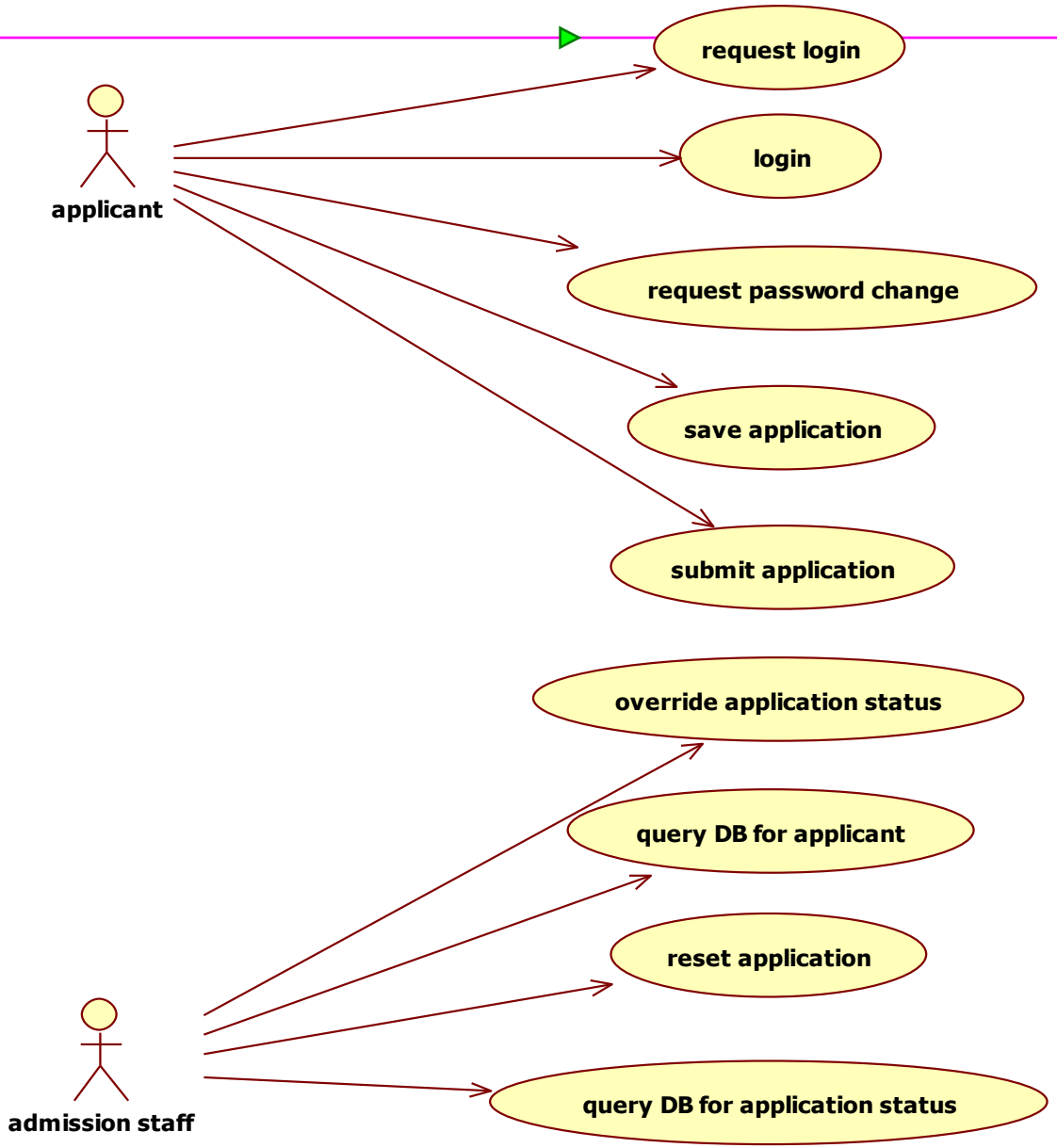
This is the amount that the library member has to pay the library when he/she returns a book later than the maximum allowed check-out days.

Requirements Analysis Topics

- A modern software requirements specification (SRS)
- Use-Case Model
  - Use cases drive entire development lifecycle
  - Use case diagrams
  - Use case descriptions
  - Guidelines for developing use cases
    - Business rules in use cases
    - Structuring complex use cases
- Supplementary Specs
- Glossary
- Review

Review Questions

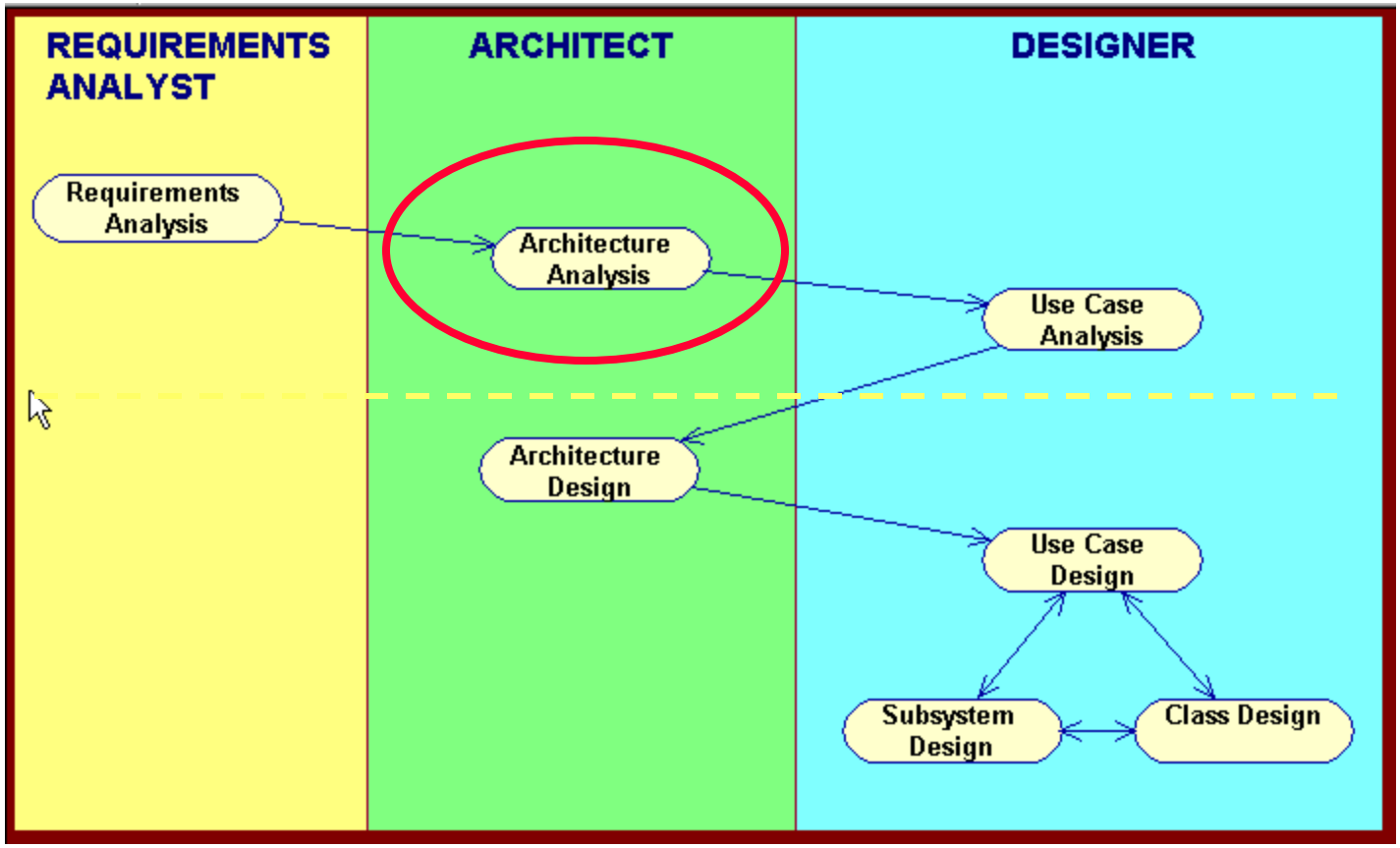
- What are the main artifacts of requirements analysis, and what are their purposes?
- What is a use case? List examples of use case properties (I.e., the parts of a UC description).
- What is a scenario?
- Name 3 elements of a UC diagram.


request login
login
applicant
request password change
save application
submit application
override application status
query DB for applicant
reset application
admission staff
query DB for application status

Architectural Analysis

The whole is contained in every part.

Basic RUP OOAD Activities



Architectural Analysis Topics

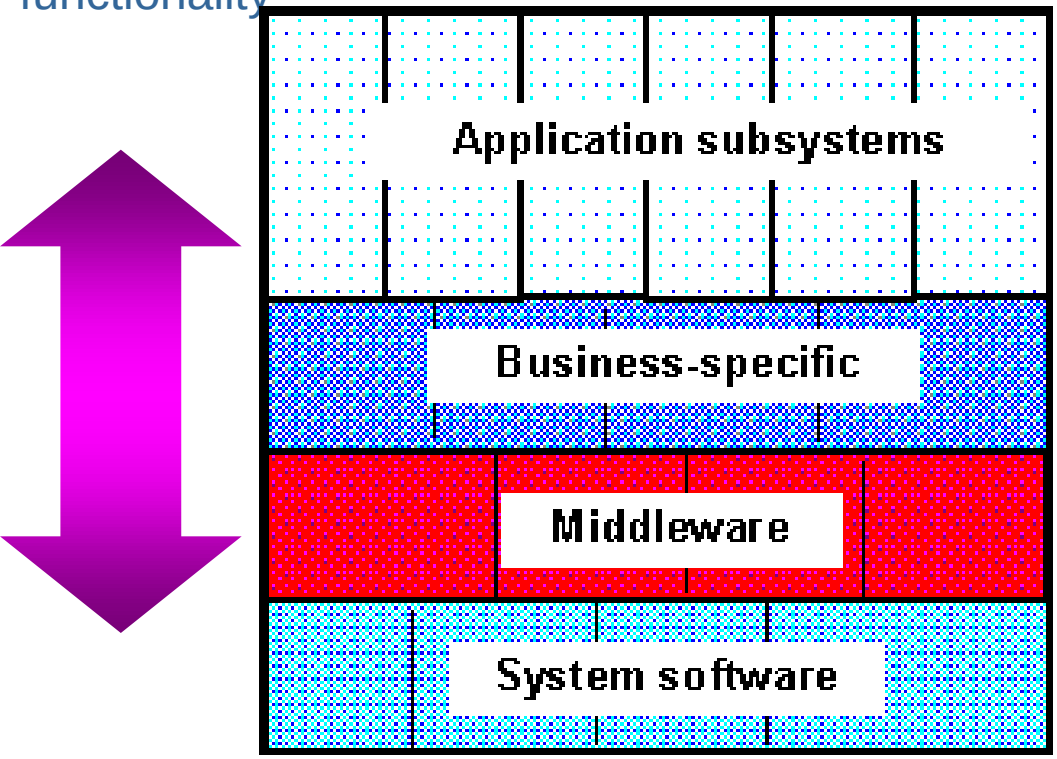
- Hi-Level System Architecture
  1. Layers
  2. Analysis Mechanisms
  3. Key Abstractions

Design Patterns

- A design pattern is a customizable solution to a common design problem
  - Describes a common design problem
  - Describes the solution to the problem
  - Discusses the rationale, results and trade-offs of applying the pattern
- Design patterns provide the capability to
  - reuse successful designs
  - Capture master-class design expertise
- Architectural patterns are design patterns applied to software architecture. Example: the Layering Pattern

Typical Layered Architecture

Specific functionality



General 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 sybsystems 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.

NOTE: During Architectural Analysis, the focus is on the upper layers – application and business layers. The middleware and system software layers will be detailed in Architectural Design.

Part 1 - The Notion of Architectural Layers

Layering Guidelines

Layering provides a logical partitioning of subsystems into a number of sets, with certain rules as to how relationships can be formed between layers. The layering provides a way to restrict inter-subsystem dependencies, with the result that the system is more loosely coupled and therefore more easily maintained.

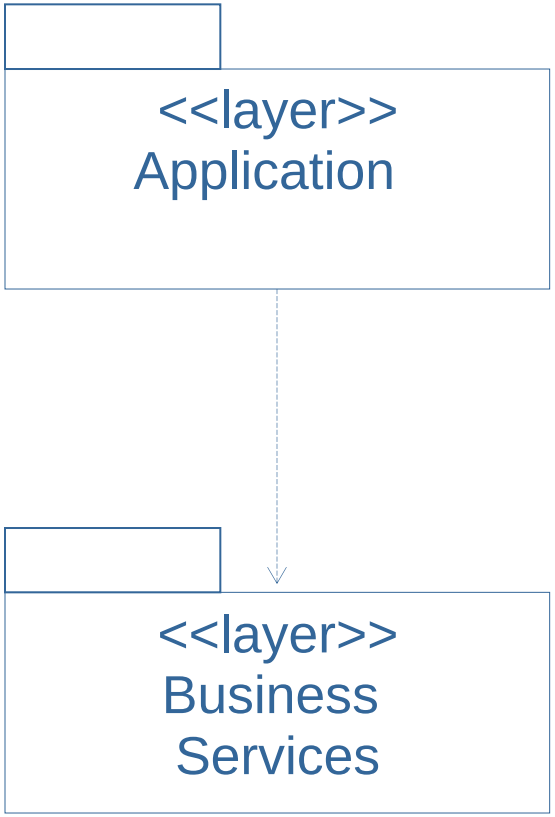
The criteria for grouping subsystems follow a few patterns:

- **Visibility**. Subsystems may only depend on subsystems in the same layer and the next lower layer.
- **Volatility**.
  - **In the highest layers**, put elements which vary when user requirements change.
  - **In the lowest layers**, put elements that vary when the implementation platform (hardware, language, operating system, database, etc.) changes.
  - Sandwiched in the middle, put elements which are generally applicable across wide ranges of systems and implementation environments.
  - Add layers when additional partitions within these broad categories helps to organize the model.
- **Generality**. Abstract model elements tend to be placed lower in the model. If not implementation-specific, they tend to gravitate toward the middle layers.

UML for Modeling Architectural Layers

- Use stereotyped packages to show Architectural layers
- Use the “standard” <<layer>> stereotype

Model Organization : High-Level Example



MUMScrum Layers

- What are the simple layers for or our MUMScrum project?
- At the analysis stage the layers should be the same for all implementations:
  - .NET
  - Java EE
  - Spring Hibernate
  - Android
  - Etc.

MUMScrum -- Layers or MVC or Both?

- MVC is a design pattern for separating the model (data), view (user interface) and controller (business logic) concerns and creating associations between them.
- Often MVC is considered to be strictly for user interface: e.g., Spring MVC, or .NET MVC.
- Developing a layers architecture can be used with MVC. In the layers we may make decisions on things like Operating Systems, Databases, Networks, etc.
- We call these ***architectural mechanisms*** at the analysis stage.

Part 2 - The Notion of Architectural Mechanisms

- A project's architectural mechanisms are standards for how common design problems are to be solved on this project
  - Promotes uniformity of solutions and increases possibility of reuse, ease of maintenance and minimizes entropy of multiple solution strategies
  - An architectural mechanism is really an architectural pattern for the project.
- Whenever possible, represent in diagrams as a placeholder
  - Short-hand representation for complex behavior
- Example: Plan to use a legacy database (since it is a known constraint) and represent the database as a placeholder in diagrams. Don't need to design the database or the infrastructure for connecting since company already knows how this should be done – mechanism is known.

Identifying Analysis Mechanisms

Two approaches:

- Top Down -- use well-known approaches in the context of your new application (e.g. persistence)
- Bottom Up -- identify some pattern in emerging application that is relevant for the current application and possibly others -- e.g., a rules engine or other useful framework

Sample Analysis Mechanisms

- Distribution of components on multiple tiers or via web services – e.g. SOAP or RESTful
- Transaction management
- Concurrency
- Persistence -- e.g. Object Relational Mapping - ORM
- Security -- e.g. Java Authentication and Authorization
- Error and Exception detection / handling / reporting
- Rules engine
- Wrapping of legacy systems
- Web Application Framework
- User Interface Framework

MUMScrum Analysis Mechanisms

- What are some analysis mechanisms for our MUMScrum project?

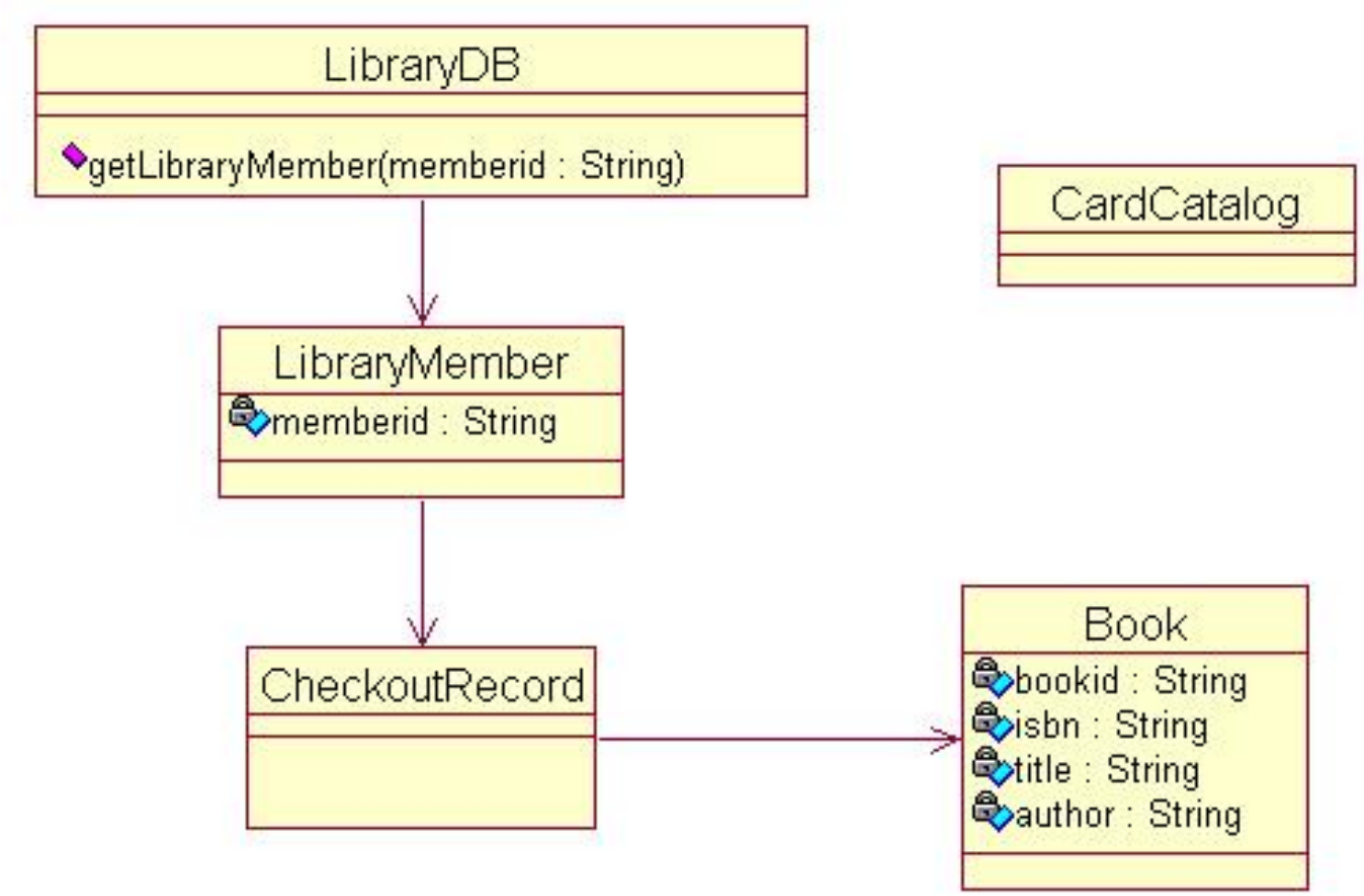
Part 3 -- Key Abstractions Guidelines

- Key abstractions are concepts identified during requirements analysis. Examples: Customer, Product, LegacyDB, Course, Section, Faculty, etc.
- Goal is "analysis classes" common across the use cases; these will be refined and fleshed out as the project evolves
- Architect will come up with these based on
  - Statement of the Problem
  - List of Requirements
  - Past experience with similar systems
  - Understanding of business domain

Key Abstractions Guidelines (cont)

- Include obvious entities acted upon by the use cases, e.g.,
  - Actors for which system has internal data (e.g., customer, student, etc; not librarian, not product manager, etc)
  - External systems which require system interaction (e.g. databases, messaging service) and a GUI (if there is one)
  - Entities that are important parts of the business operation
    - Example: CheckoutRecord in the library example
- Include any obvious attributes and operations. Key abstraction diagrams are class diagrams.

Example: Key Abstractions



Compro Admissions Forms Key Abstractions

- Each group list some key abstractions for our MUMScrum project.

Architecture Analysis Summary

1. Activity performed by system architect in preparation for use case analysis.
2. Identify an initial system architecture--preferably one that separates the project into nonchanging and changing subsets, and the changing elements should be grouped such that those grouped together tend to change together.
3. Architectural mechanisms are identified which are known solution patterns to complex problems. We might specify ORM and Web Application but not the specific frameworks.
4. Key abstractions are identified which will provide a common set of key domain elements across use case realizations.

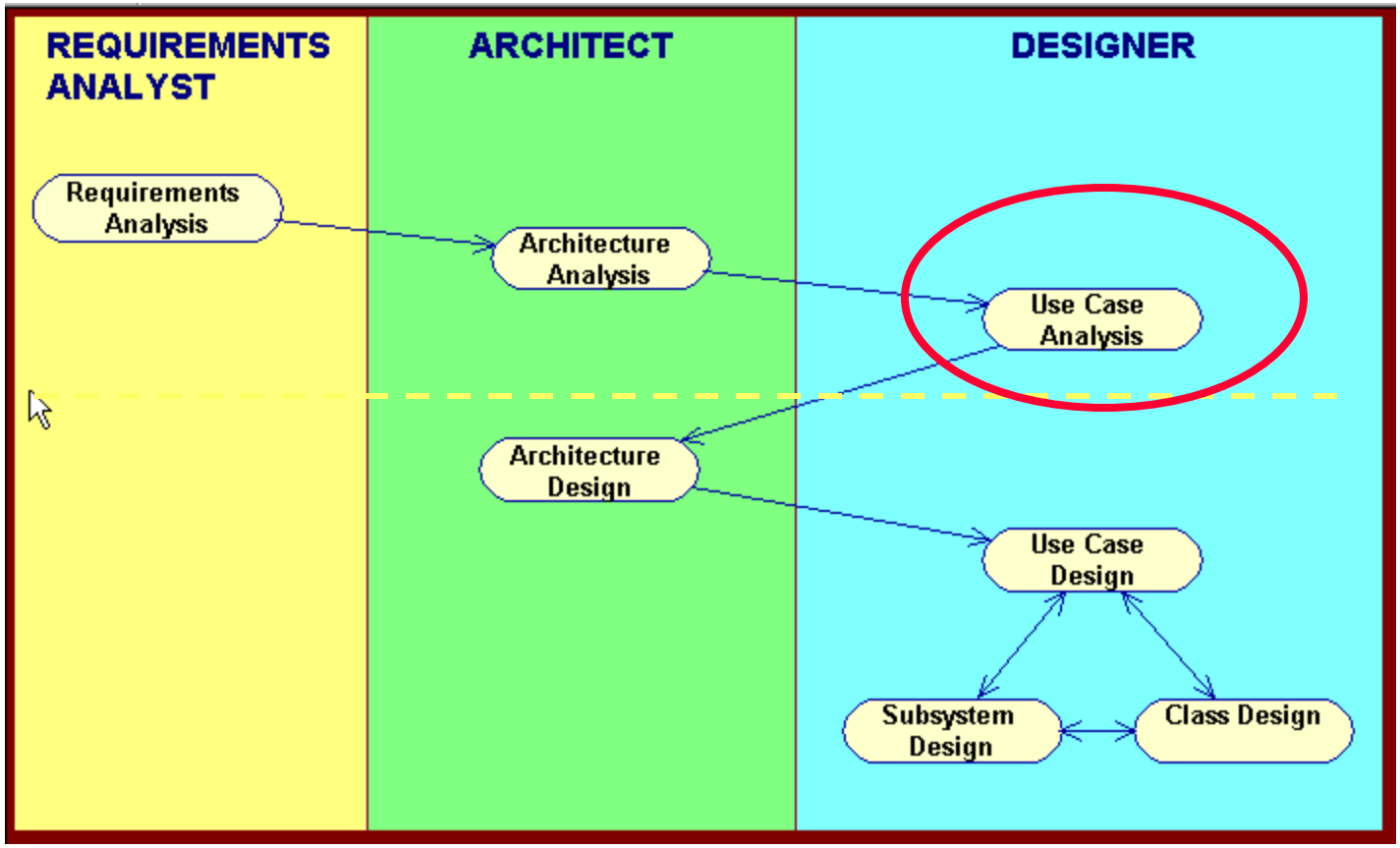
Review of Architectural Analysis

- Describe the purpose of Architectural Analysis.
- Describe the rationale for defining analysis mechanisms.
- How are key abstractions identified during architectural analysis? Why are they identified?
- Describe the advantages of a layered architecture with an example of typical layers.

Use Case Analysis

Yoga Is Skill In Action

Basic RUP OOAD Activities



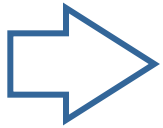
Use-Case Analysis Steps

- Refine the use-case description
- Model behavior using a sequence diagram
  - Identify analysis classes for the use-case
  - Model object collaborations
- Model structure in VOPC diagram
  - Capture responsibilities from sequence diagram
  - Add analysis-level attributes and associations
  - Note analysis mechanisms
- Document business rules

Refine Use-Case Description

Original Use Case

...
...
The system displays a list of course offerings
...



Refined Use Case

...
...
The system retrieves a list of the selected semester's course offerings from the database
...

Use-Case Analysis Steps

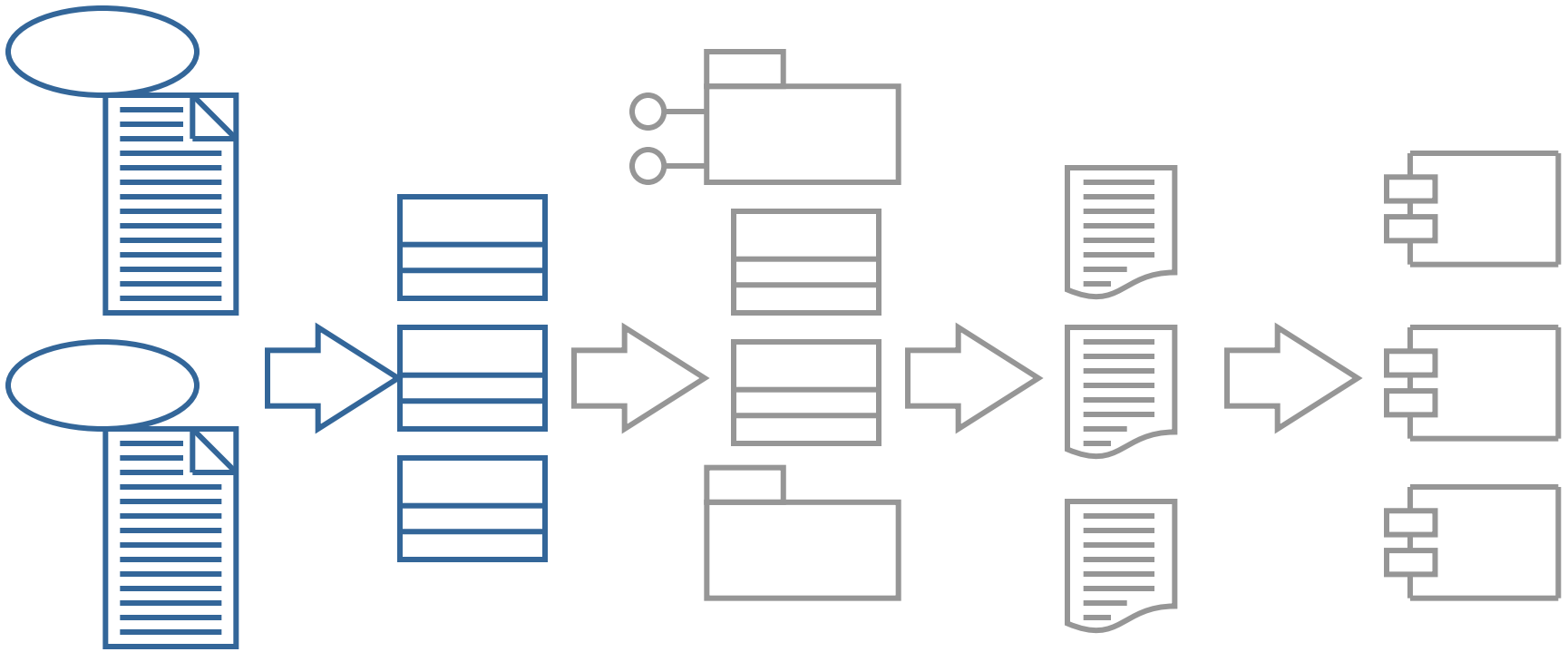
- Refine the use-case description
- Model behavior using a sequence diagram
  - Identify analysis classes for the use-case
  - Model object collaborations
- Model structure in VOPC diagram
  - Capture responsibilities from sequence diagram
  - Add analysis-level attributes and associations
  - Note analysis mechanisms
- Integrate analysis classes
- Document business rules

Analysis Classes

- Represent high level abstractions of one or more classes and/or subsystems
- Emphasize functional requirements
- Capture behavior in terms of conceptual responsibilities
- Model attributes at high level
- Use 3 “stereotypes”: entity, boundary, control

Q57-9, J181-5, J204-5

Analysis Classes: A First Step Towards Executables

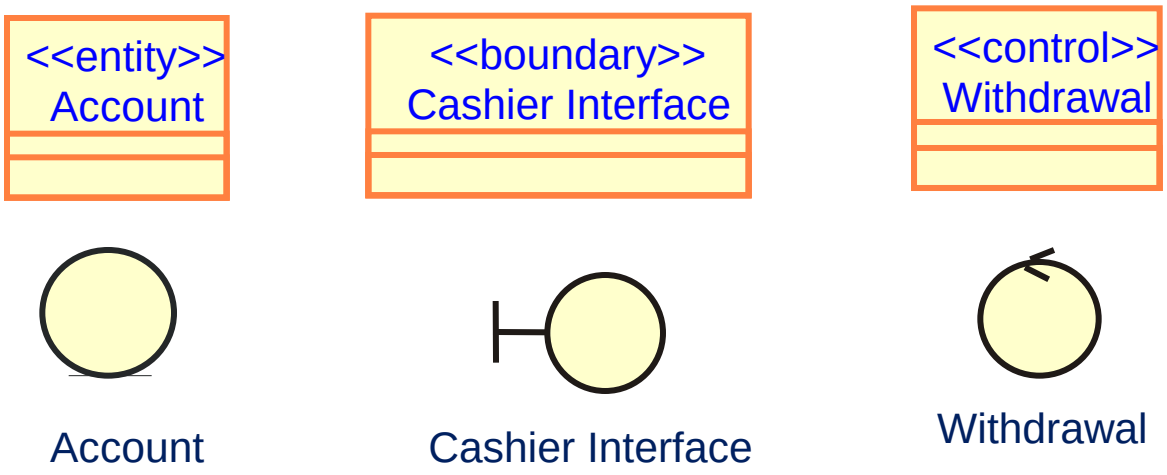


3 Analysis Class "Stereotypes"

System information

System boundary

Use-case behavior coordination

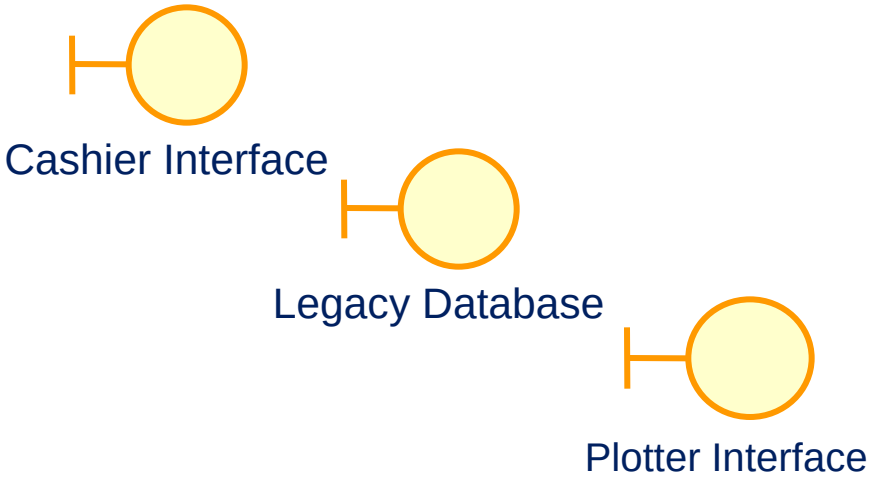


1. **Analysis classes -- 'things in the system which have responsibilities and behavior'**
2. **Analysis classes are used to capture a 'first-draft', rough-cut of the object model of the system**
3. **Analysis classes model objects from the problem domain.**
4. **Analysis classes can be used to represent "the objects we want the system to support"**

J183

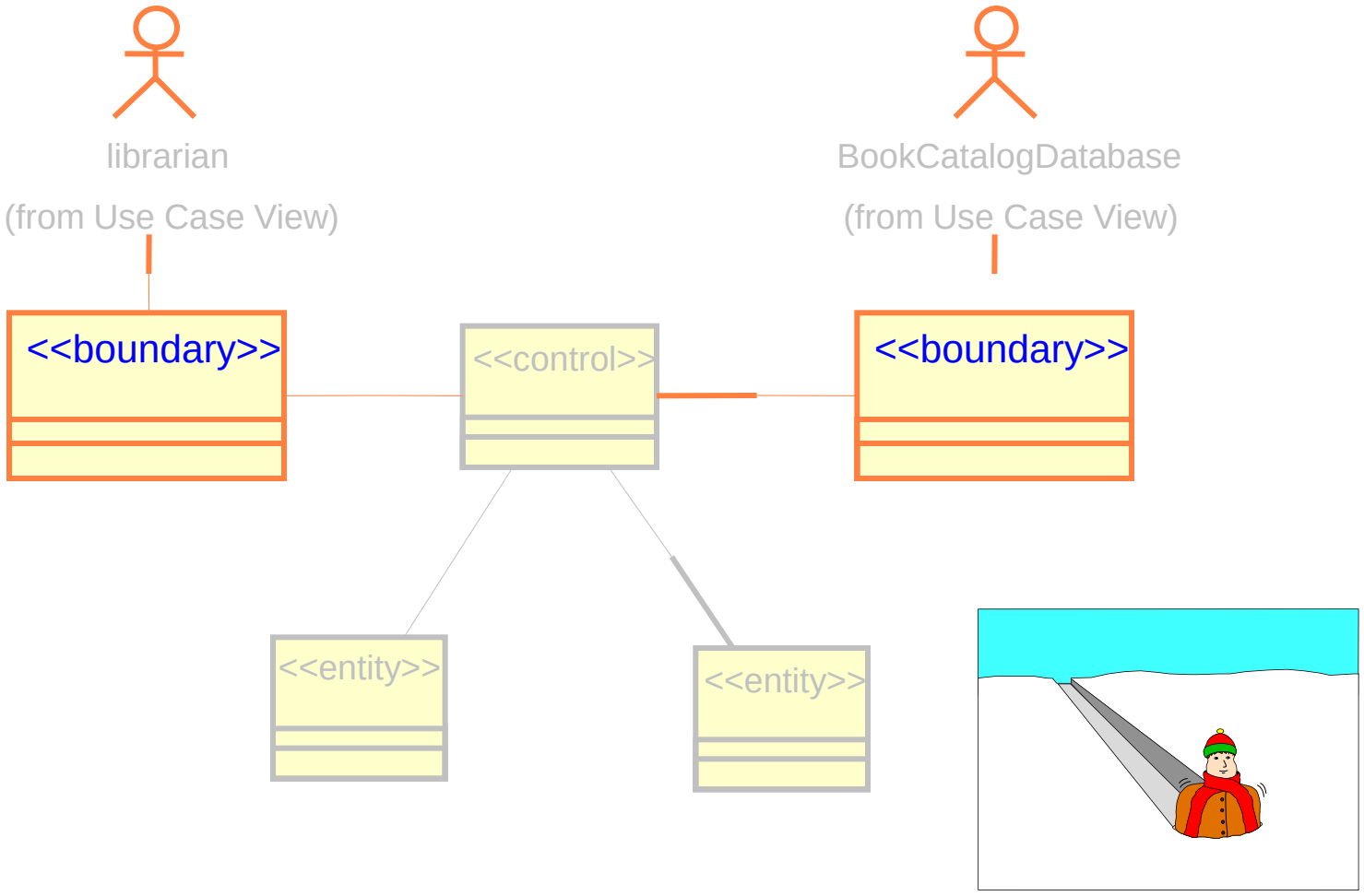
Boundary Classes

- Capture interaction between system and actors (external systems)
- Help isolate changes to external systems
- Typical examples
  - User interfaces
  - System interfaces
  - Device interfaces



- *Guideline: One boundary class per actor/use-case pair*

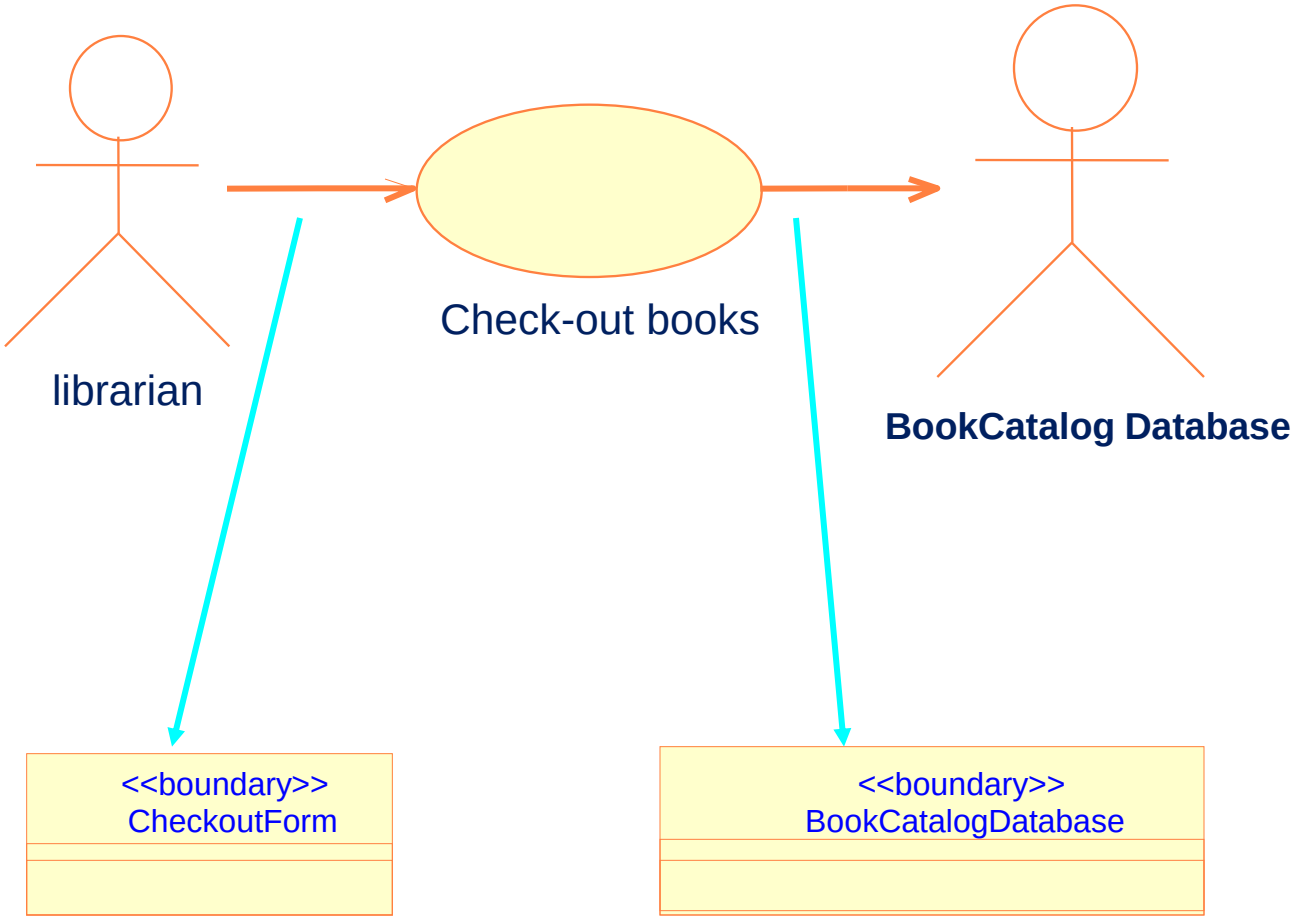
Boundary Classes Isolate System/Actor Interactions



J183-4

Identify Boundary Classes

- For each actor/use case pair



J183, J204-5

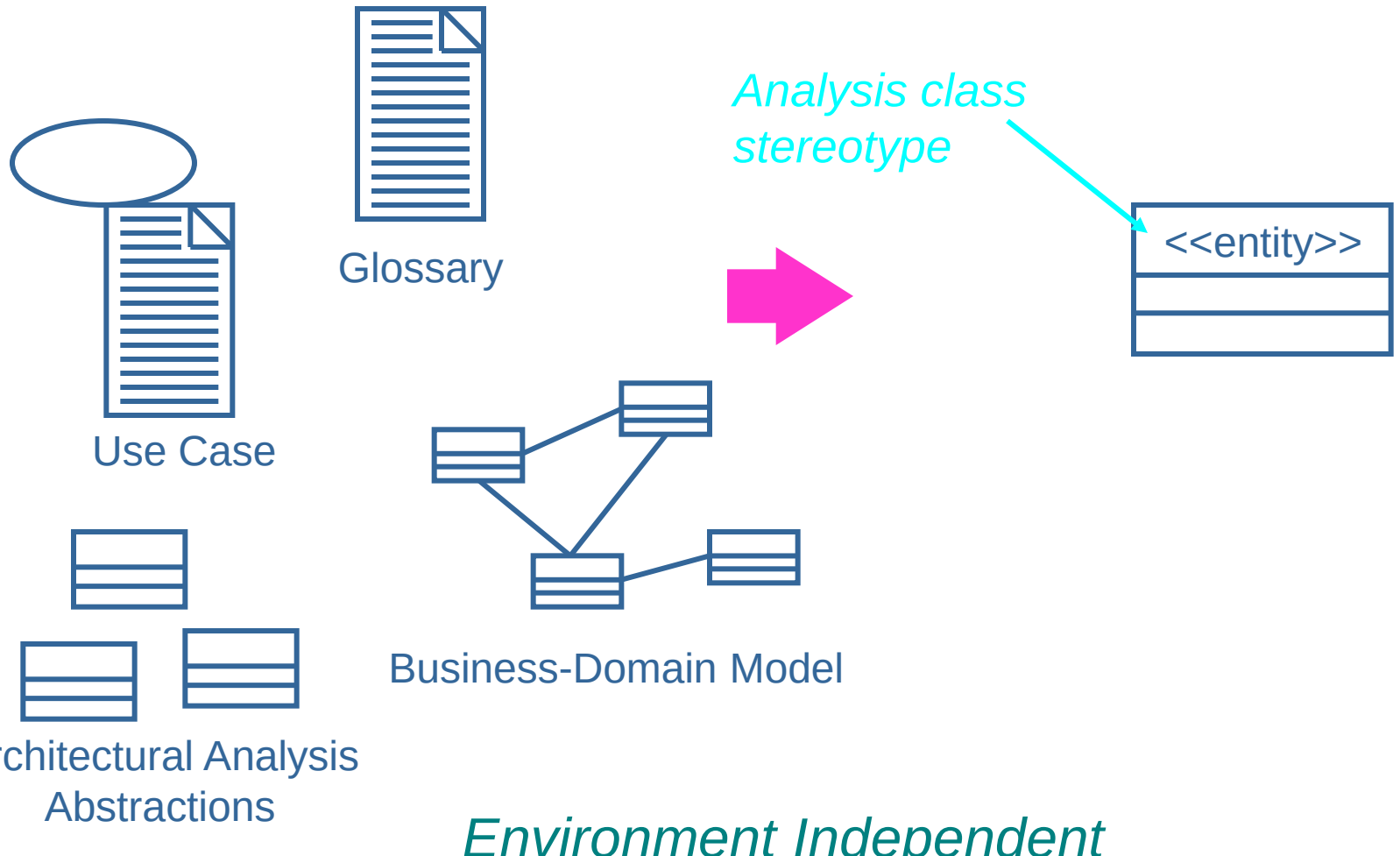
Boundary Class Guidelines

- User Interface
  - Focus on what information is presented
  - UI details get worked out in design and implementation
- System and Device Interface
  - Focus on what is needed to facilitate communication with external systems
  - How to implement is worked out later

Describe "what" is to be achieved, not "how" to achieve it

What is an Entity Class?

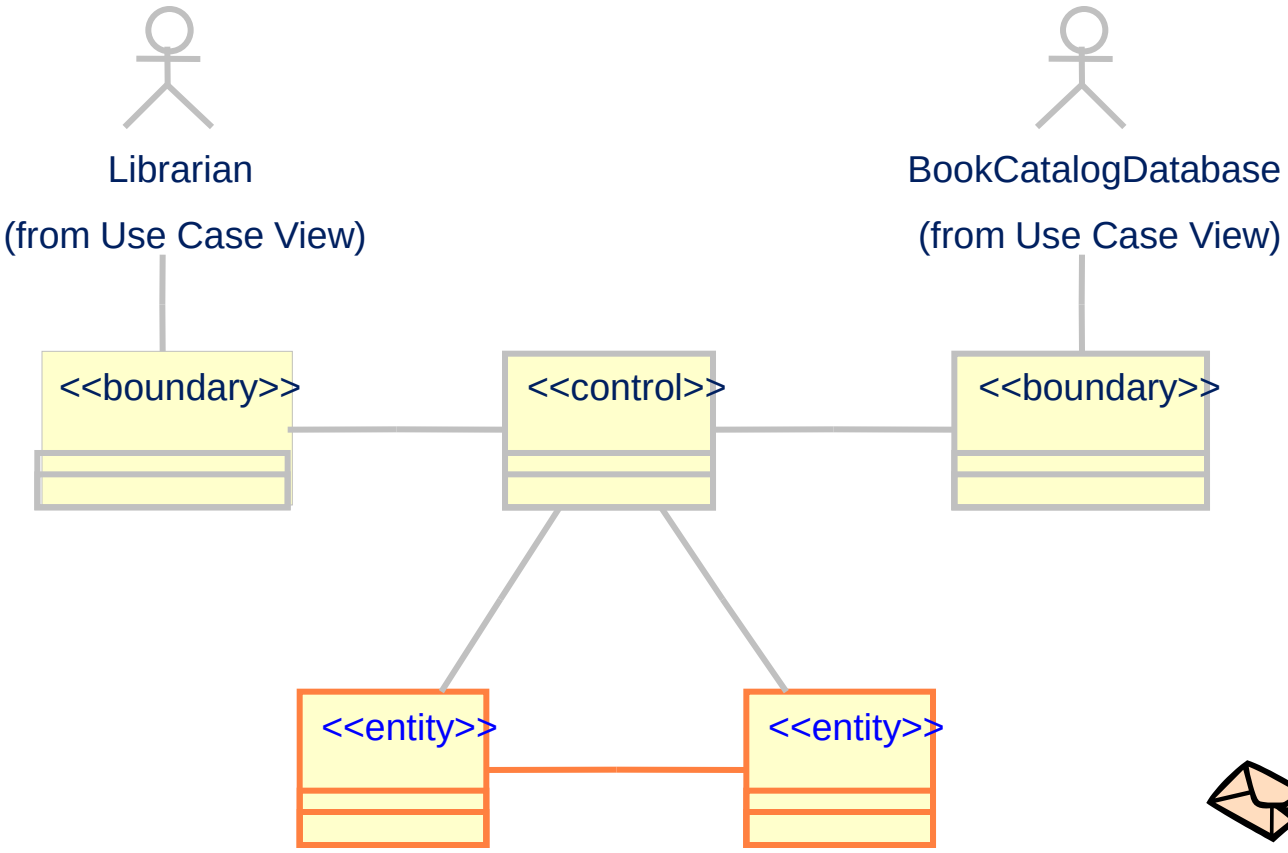
- Key abstractions of the system

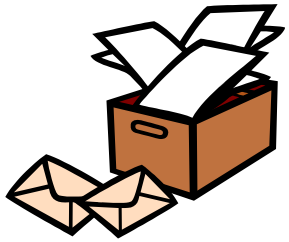


Environment Independent

J184-5, Q57

Entity Classes Model Persistent Information



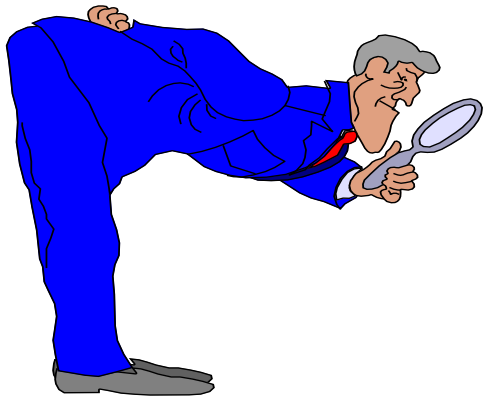


- **Responsibilities: to store and manage information in the system**
- **Hold and update information about events or a real-life object**
- **Usually persistent**

J184-5, Q57

Finding Entity Classes by Filtering Use-Case Nouns

- Start with key abstractions
- Noun clauses
  - Ignore redundant or vague candidates
  - Ignore actors (out of scope)
  - Ignore implementation constructs
  - Look for things acted on by business rules



J184-5, J204, Q57

Library System Problem Statement

You have been hired by Prince University to update their library record keeping. Currently the library has an electronic card catalog that contains information such as author, title, publisher, description, and location of all of the books in the library. All the library member information and book check-in and checkout information, however, is still kept on paper. This system was previously workable, because Prince University had only a few hundred students enrolled. Due to the increasing enrollment, the library now needs to automate the check-in/checkout system.

The new system will have a windows-based desktop interface to allow librarians to check-in and checkout books.

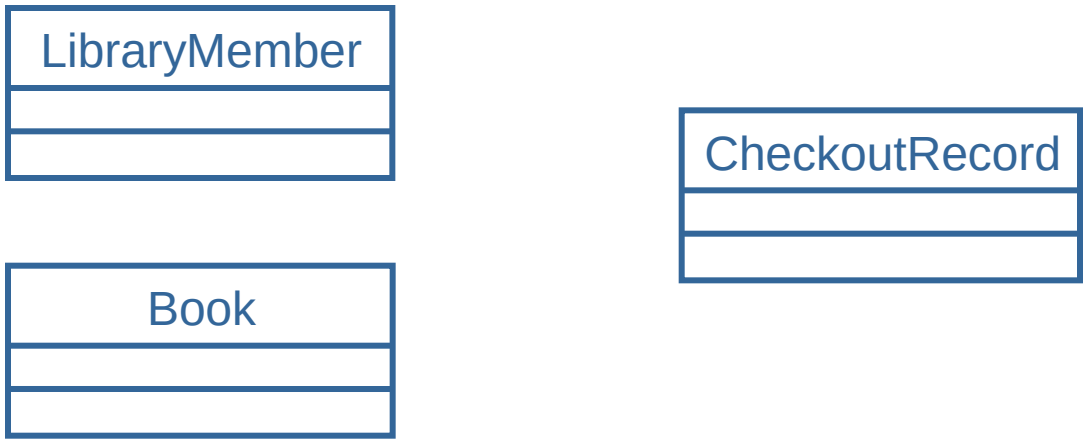
All books in the library have a unique bookid.The books in the library are ordered on the shelves by their bookid. The new system must allow library members to search through the electronic card catalog to find the bookid of the desired book.

The system will run on a number of individual desktops throughout the library. Librarians will have their own desktop computers that are not accessible by library members. Only librarians are able to check-in and checkout books.

The system will retain information on all library members. Only university students, faculty and staff can become library members. Students can check-out books for a maximum of 21 days. If a student returns a book later than 21 days, then he/she has to pay an overdue fee of 25 cents per day. University staff can also checkout books for a maximum of 21 days, but pay an overdue fee of 10 cents per day. Faculty can checkout books for a maximum of 100 days, and pay only 5 cents per day for every book returned late. The system will keep track of the amount of money that library members owe the library.

Example: Find Entity Classes

- Check-out book



Question: Why is LibraryMember considered an Entity class but not CardCatalog?

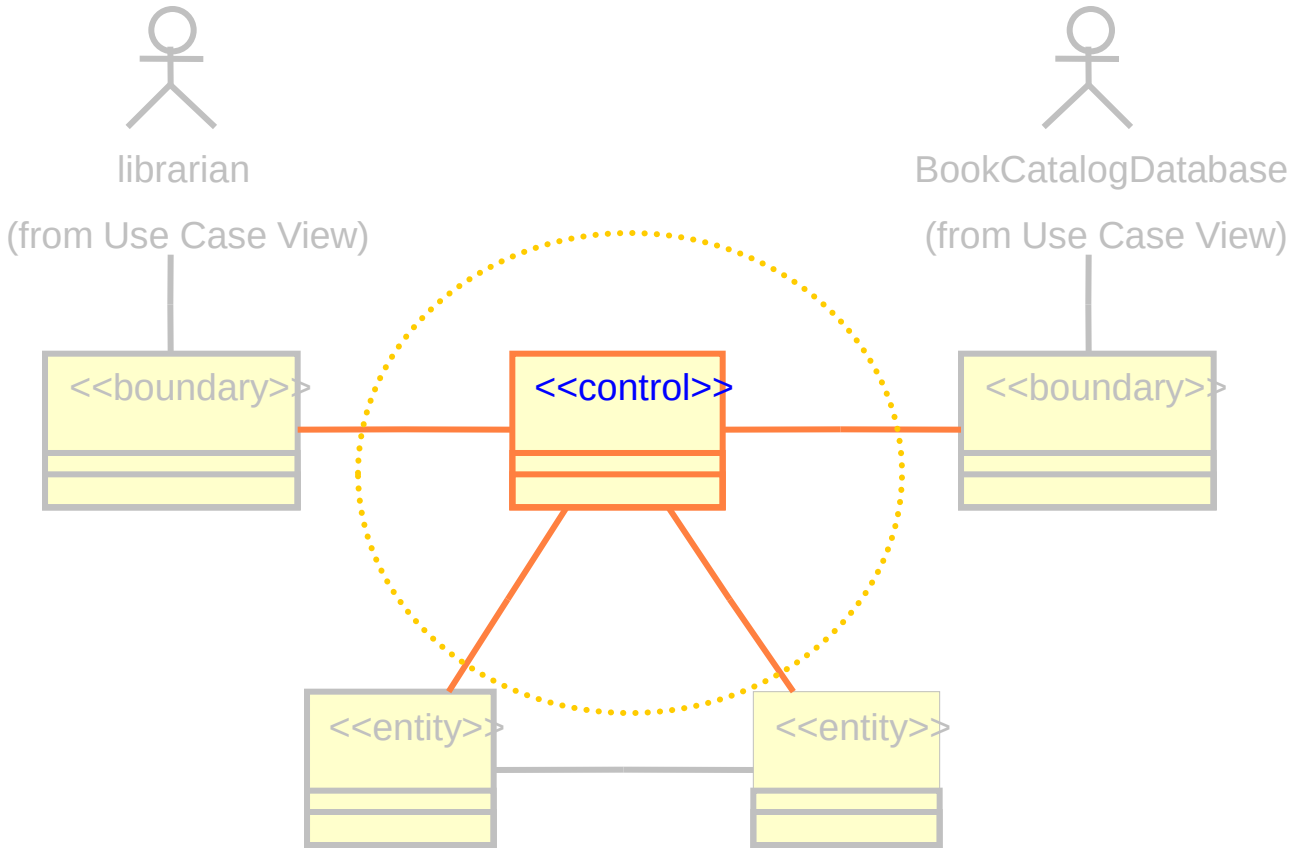
J184-5, J204, Q57

Control Classes

- Represent coordination, sequencing, transactions among groups of objects
- Encapsulate control of individual use-cases
- Guideline: One control class per use case
- Control classes provide behavior that:
  - Defines control logic (order between events) and transactions within a use case. Changes little if the internal structure or behavior of the entity classes changes
  - Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes
  - Is not performed in the same way every time it is activated (flow of events features several states)

J185, J205, Q58-9

Control Class as Use-Case Coordinator

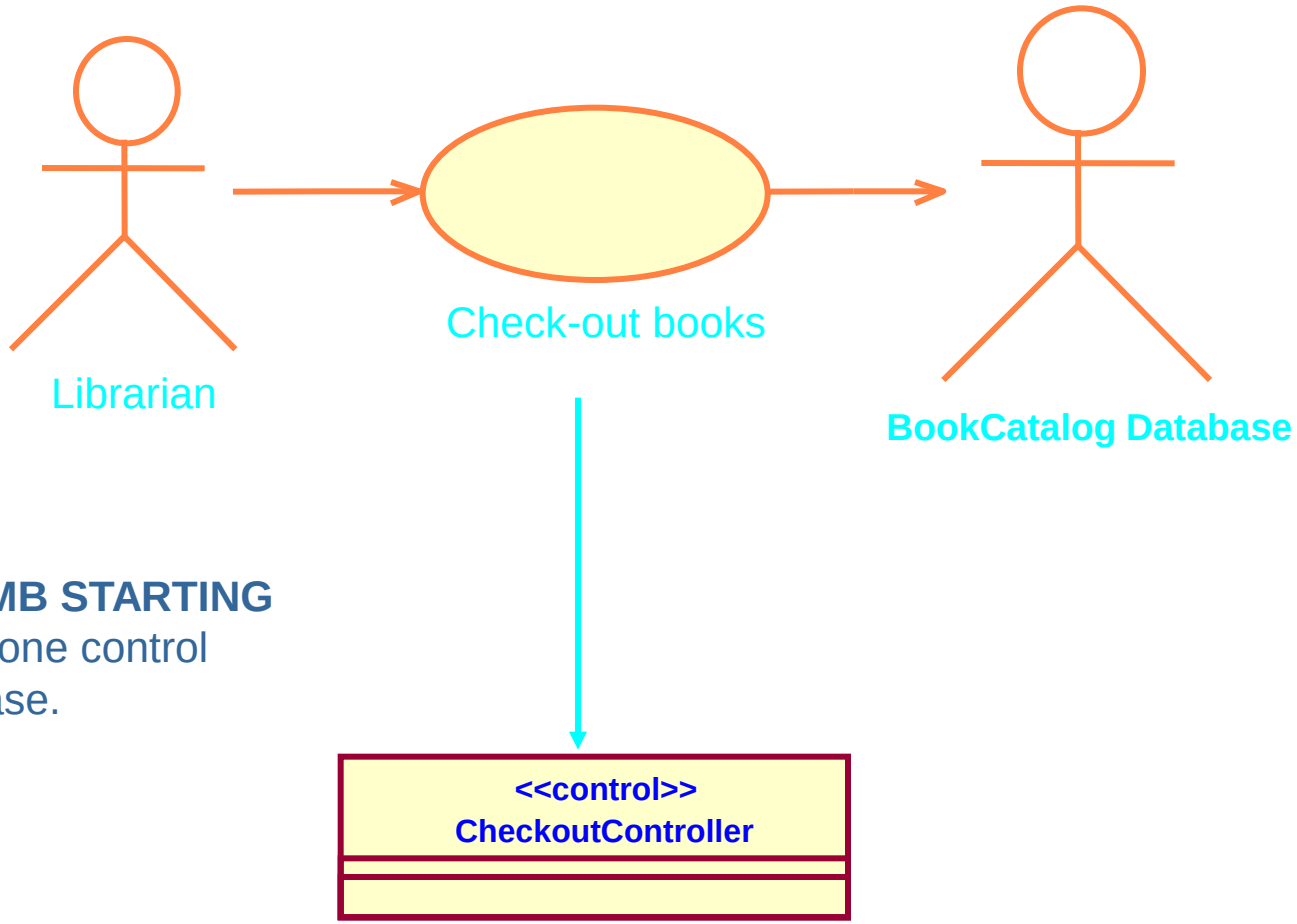


Note: Not always necessary for a use-case realization to contain a control class. Sometimes control is better encapsulated in a boundary class.

J185, J205, Q58-9

Introduce Control Classes

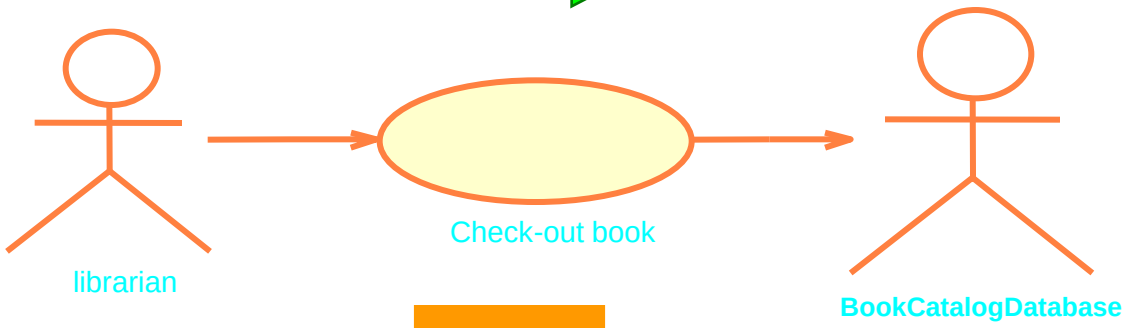
- For each use case



RULE OF THUMB STARTING OUT: Introduce one control class per use case.

J185, J205, Q58-9

Example: Analysis Classes



<<boundary>> CheckoutForm	<<control>> CheckoutController	<<boundary>> BookCatalogDatabase

<<entity>> LibraryMember	<<entity>> Book	<<entity>> CheckoutRecord

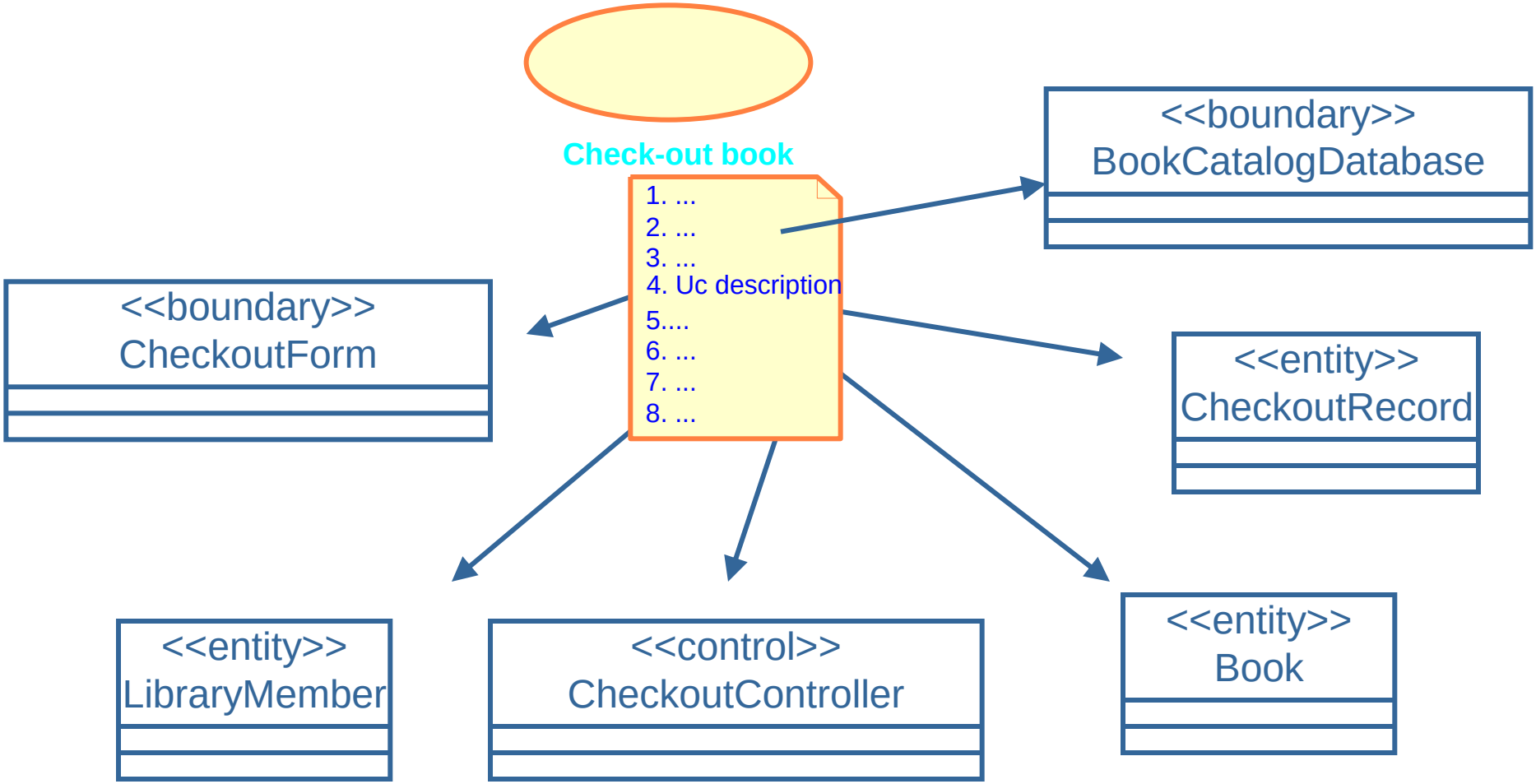
J181-5, J204-5, Q57-9

Use-Case Analysis Steps

- Refine the use-case description
- Identify analysis classes for the use-case
- Model behavior using a sequence diagram
- Model object collaborations
- Model structure in VOPC diagram
  - Capture responsibilities from sequence diagram
  - Add analysis-level attributes and associations
  - Note analysis mechanisms
- Integrate analysis classes
- Document business rules

Assign Use-Case Behavior to Classes

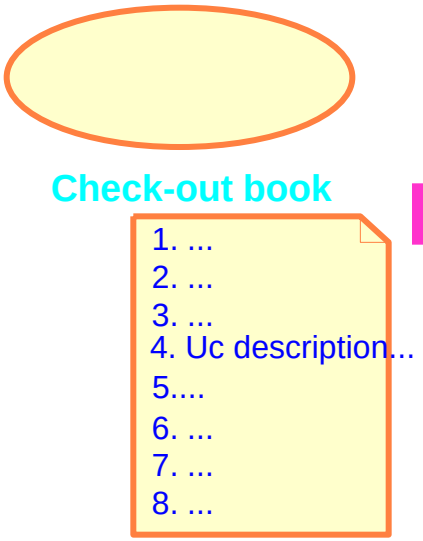
- Assign all use case steps to the analysis classes.

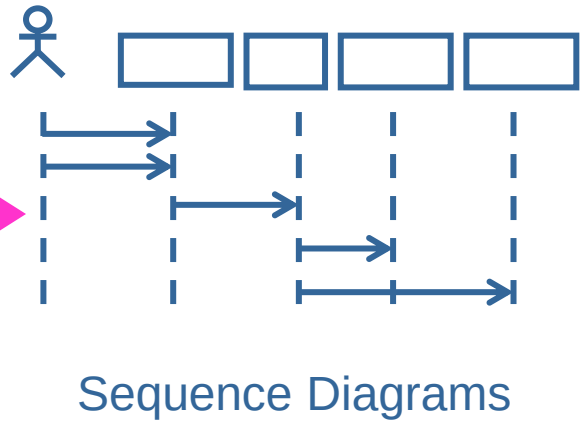


J203-6, Q73-4

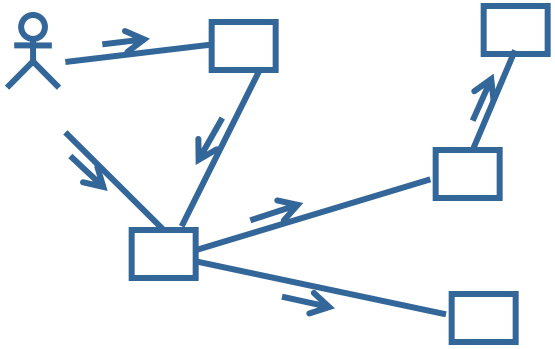
Assign Use-Case Behavior to Classes

- For each flow of events
  - Identify analysis classes
  - Model use-case behavior in interaction diagrams
  - Don't model interactions between actors





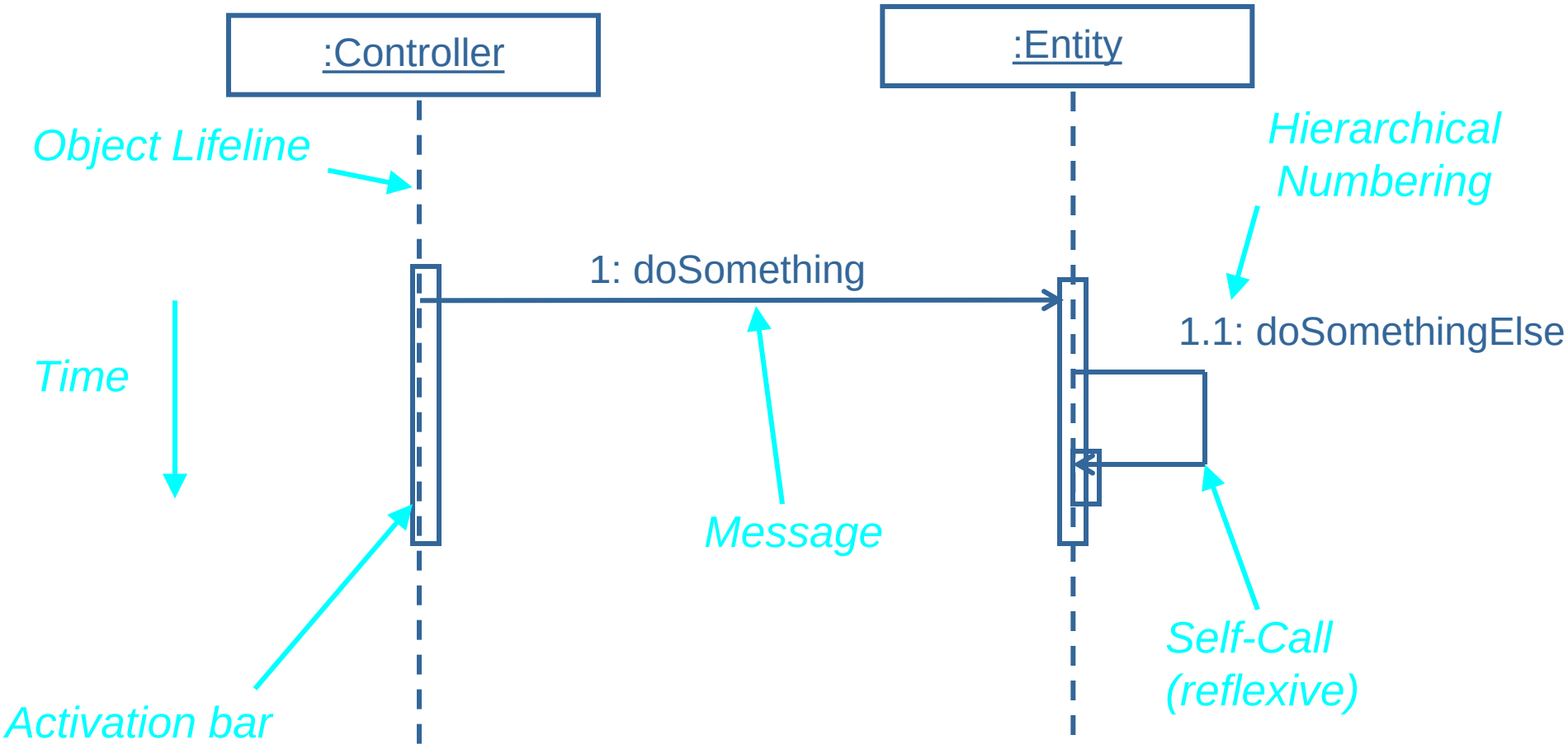
Sequence Diagrams



Communication Diagrams

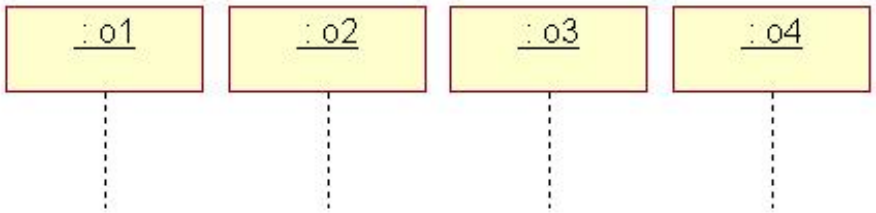
J203-6, Q73-4

Sequence Diagram Features



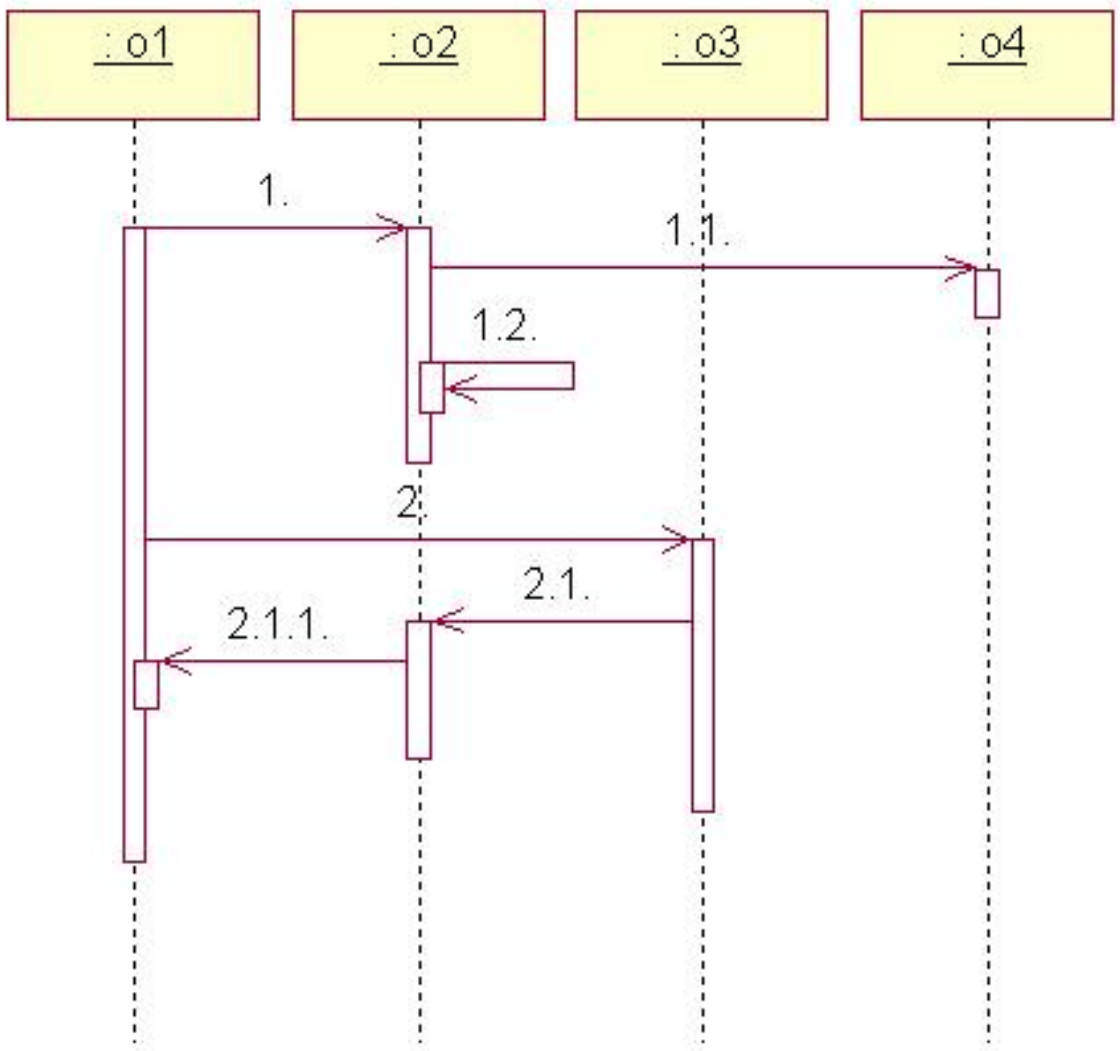
F68-71

Example

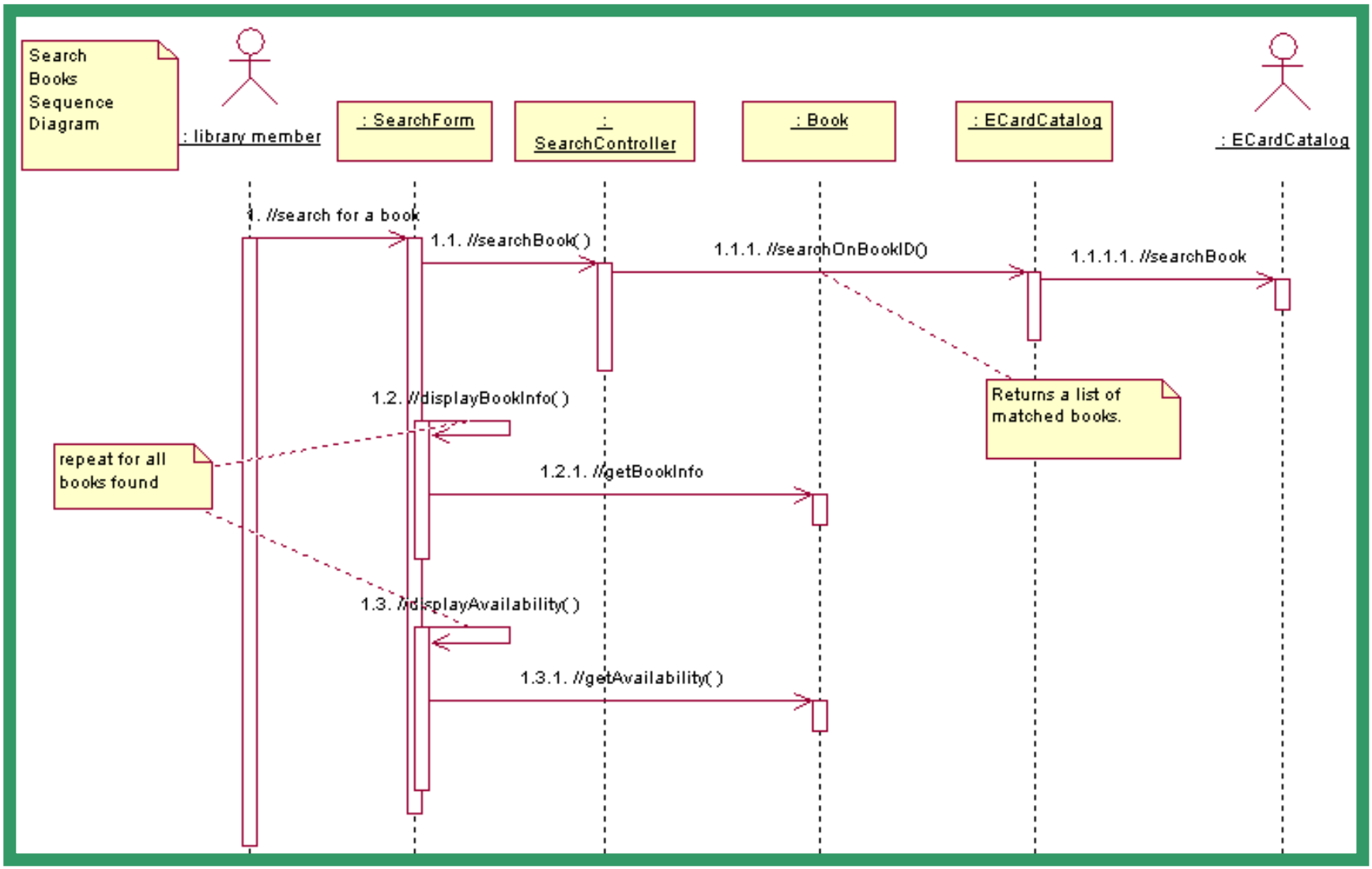


1. o1 sends to o2
1.1 sends to o4
1.2 sends to o2
2. o1 sends to o3
2.1 sends to o2
2.1.1 sends to o1

Example (cont)



Example: Sequence Diagram
(what is missing in this diagram?)



Example: Sequence Diagram
(need labels for enity, control, and boundary)

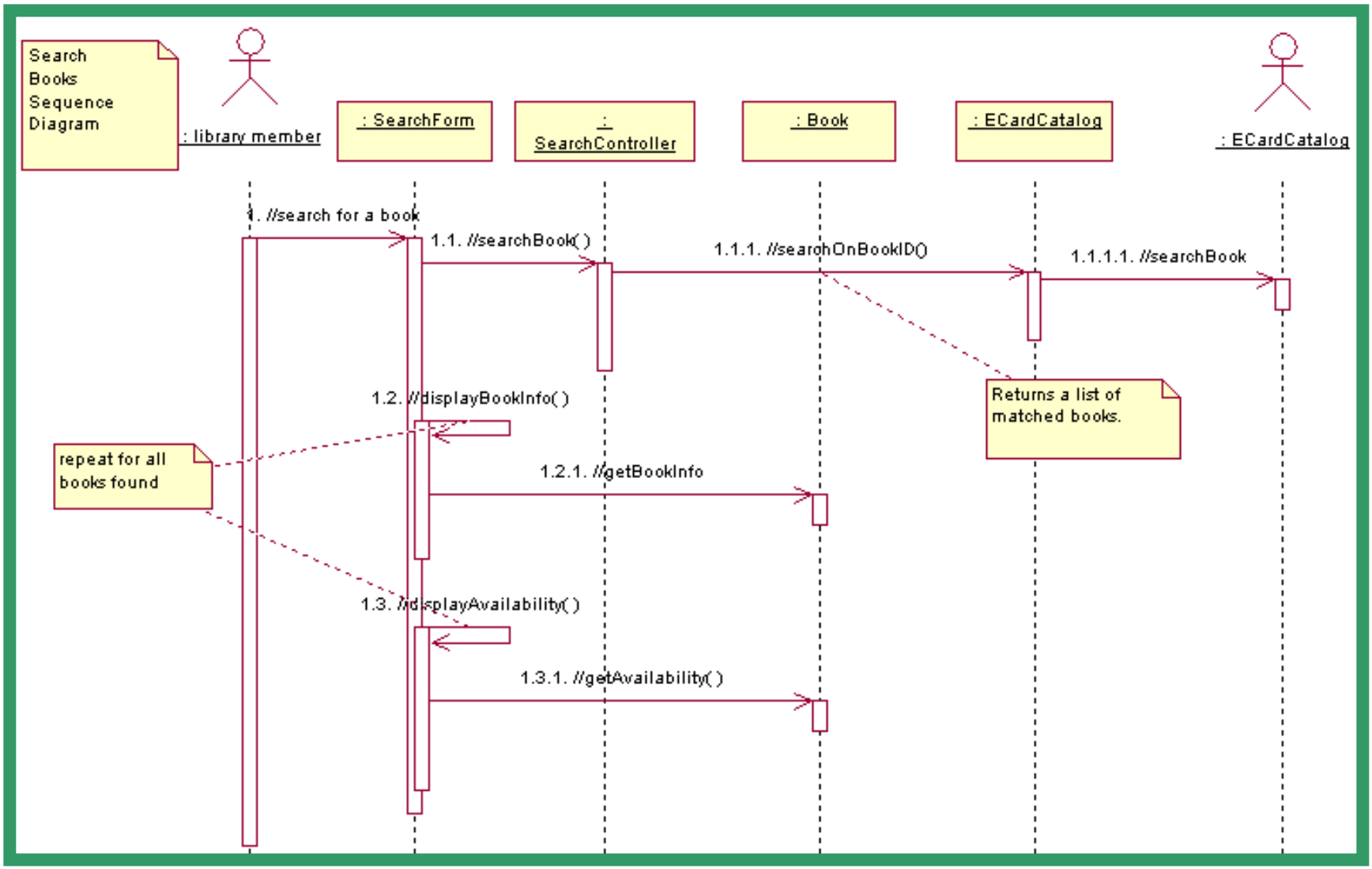


Diagram Scenarios vs Flows vs Use Case?

- A scenario is a single pass
  - Straightforward to create and read
- A flow is a set of similar scenarios
  - Might involve conditions or loops
  - Separate diagrams for significant flows
- A use-case is the set of all flows
  - In general, too complex for interaction diagrams
  - Full coverage infeasible
- Sequence diagrams are most useful to show what classes we will need and their interactions
- Sequence diagram are ***not*** very good at modeling complex logic flow

Our Sequence Diagram Conventions

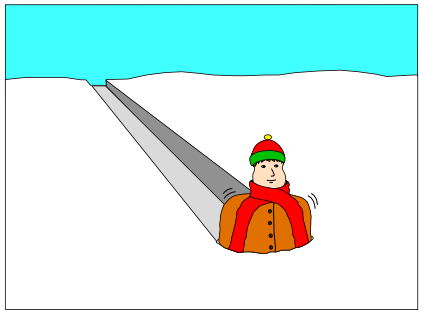
- Label our analysis classes with <<Boundary>>, <<Control>>, or <<Entity>>
- We show communication with our Data Base thru a data base boundary class. ORM is part of our OO Analysis and Design.
  - If we are ***adding*** an entity class to the DB we show the controller "new" of the entity class and then later the DB boundary class will show the ***create*** to the DB.
  - If we are ***updating***, ***reading***, or ***deleting*** an entity class that already exists in the DB, we show ***first*** the controller reading the entity class from the DB.
  - We show the return of the entity class with a comment.
  - We show the entity class is active when an object (will usually be the controller) is getting or setting data in the entity class.
  - We show the **update** or ***delete*** operations from the DB boundary class to the DB at the time the save or delete occurs.

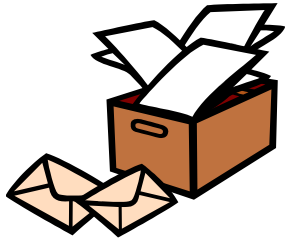
Group Exercise for MUMScrum Sequence Diagram

- As a group create/review one of your Use Case Descriptions.
- As a group create a ***first draft*** sequence diagram for *that use case.*
- Identify boundary, control and entity classes.
- Show the basic success flow.
- Any preconditions for this use case?
- How would you show your alternative paths and business rules?

Analysis Class Stereotypes Guide Assignments

- Boundary classes responsible for actor communications
- Entity classes responsible for processing persistent information
- Control classes responsible for use-case specific interactions or mediating other event flows



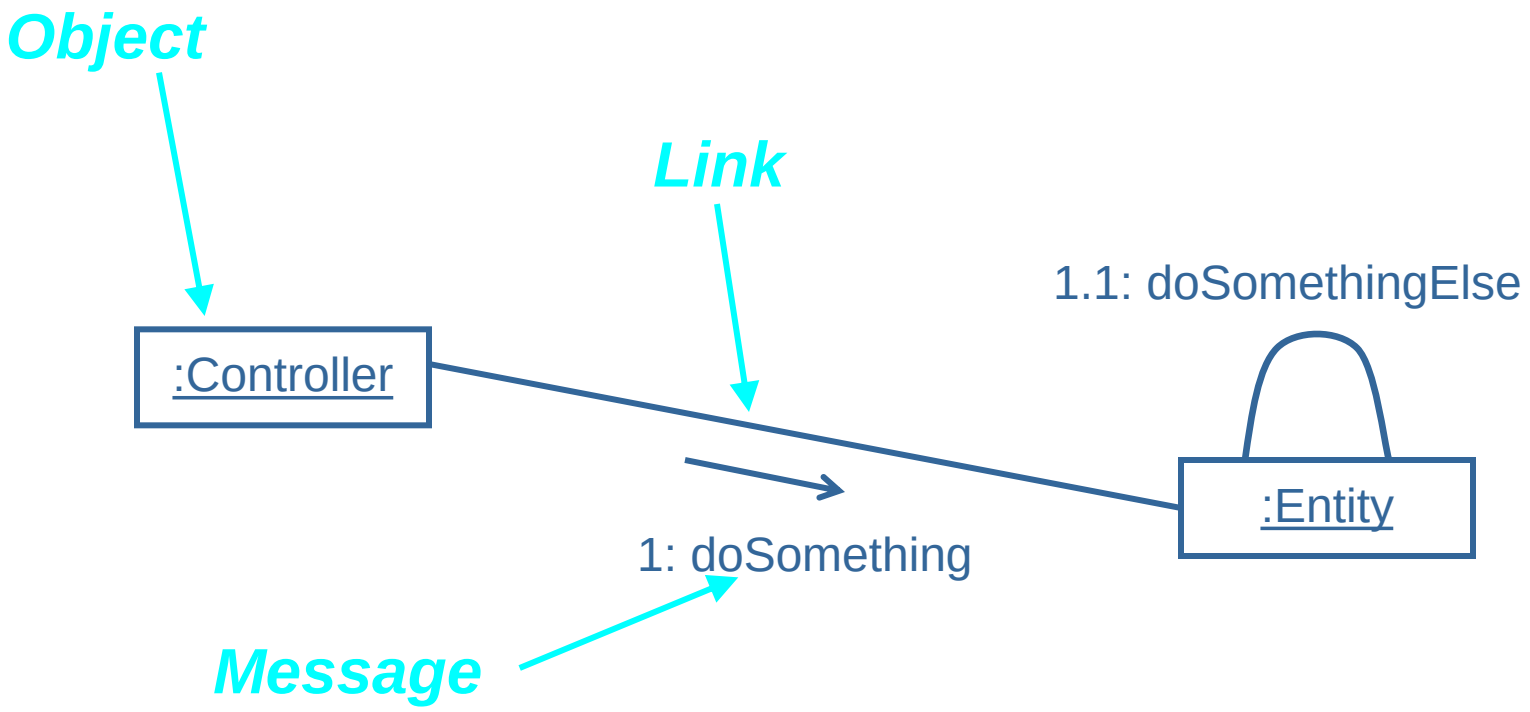


Q57-9, J181-5, J204-5

Further Responsibility Assignment Guidelines

- In general, class with the data should have the responsibility
- We favor entity classes having operations on *their own data*
- If data is spread across classes we can:
  - Put the responsibility with one class and add a relationship to the other
  - Create a new class, put the responsibility in the new class, and add relationships to classes needed to perform the responsibility
  - Put the responsibility in the control class, and add relationships to classes needed to perform the responsibility

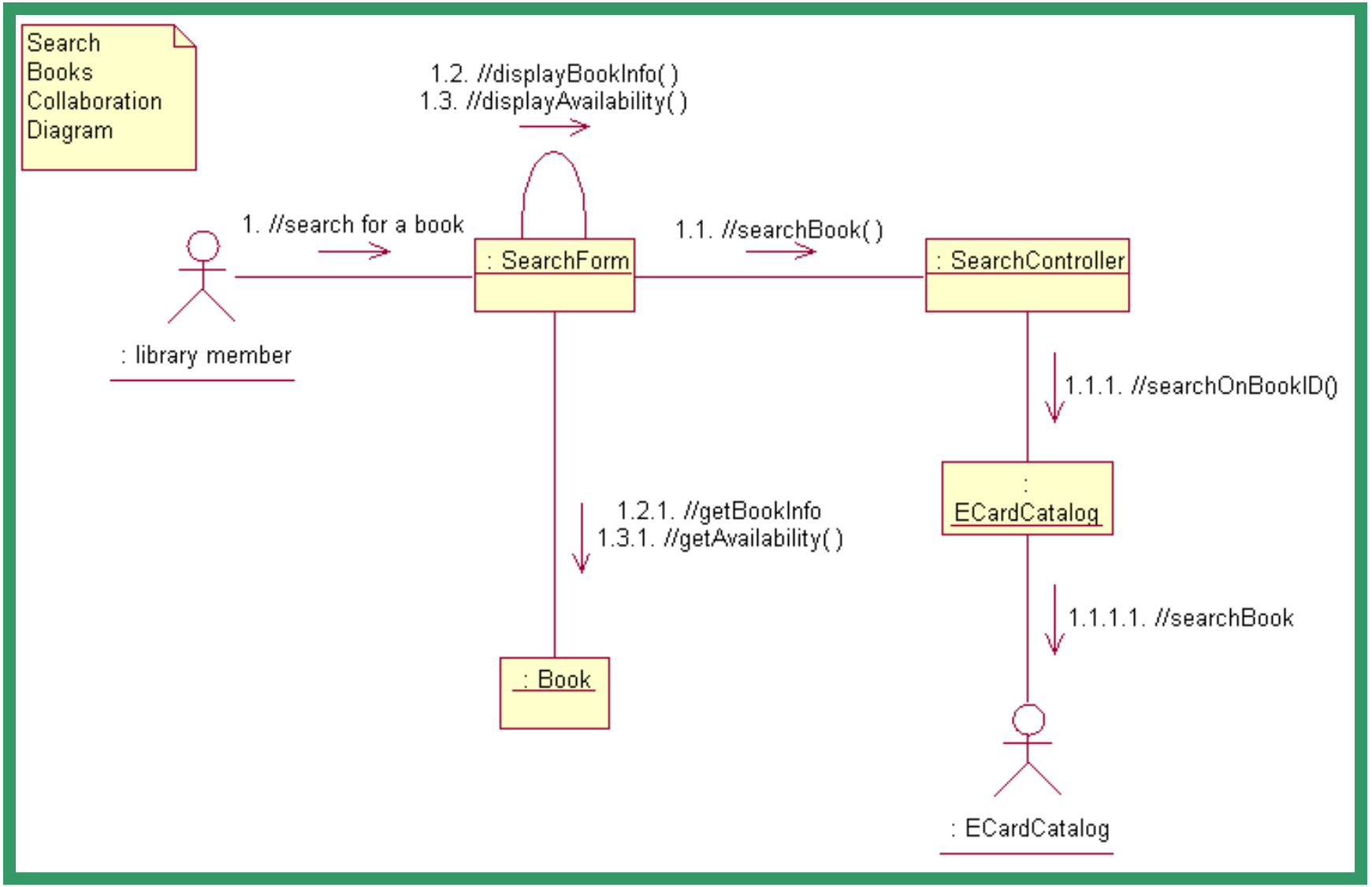
Collaboration Diagrams



1. A *link* is a relationship among objects across which a message can be sent
2. A *message* is a communication between objects that conveys information resulting in some activity – shown with a labeled arrow
3. Often use sequence numbers to label messages to keep track of the flow of events

F:72-4

Example: Collaboration Diagram



Sequence vs Collaboration Diagrams

- Sequence Diagrams
  - Show the explicit sequence of messages
  - Better for visualizing overall flow
  - Better for real-time specifications
- Collaboration Diagrams
  - Show relationships in addition to interactions (use for VOPC diagrams)
  - Better for visualizing patterns of collaboration
  - Better for visualizing all of the effects on a given object

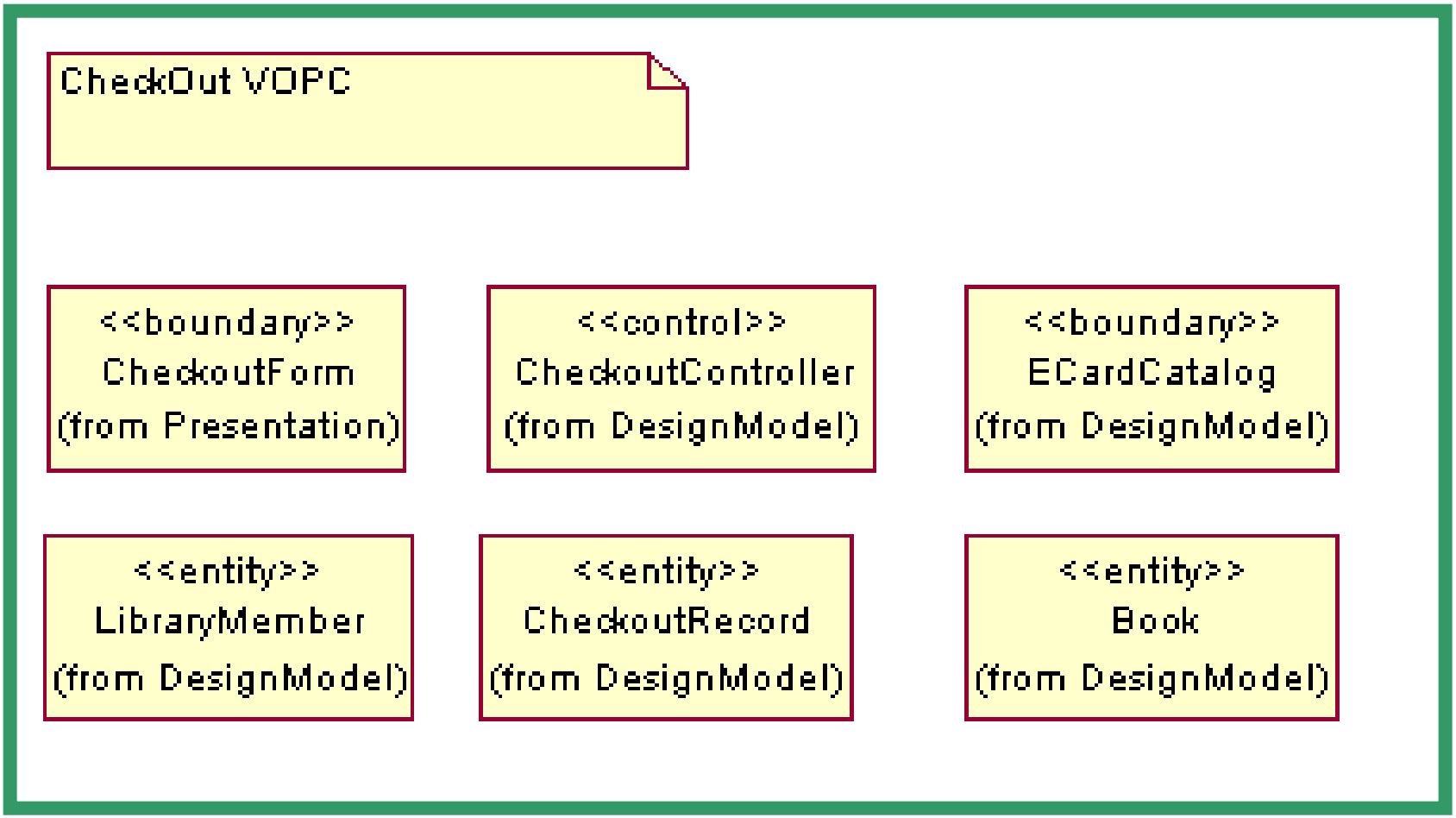
F75, Q84

Use-Case Analysis Steps

- Refine the use-case description
- Model behavior using a sequence diagram
  - Identify analysis classes for the use-case
  - Model object collaborations
- Model structure in VOPC diagram
  - Capture responsibilities from sequence diagram
  - Add analysis-level attributes and associations
  - Note analysis mechanisms
- Integrate analysis classes
- Document business rules

Building the VOPC Diagram

May use the Key Abstractions diagram as a starting point.



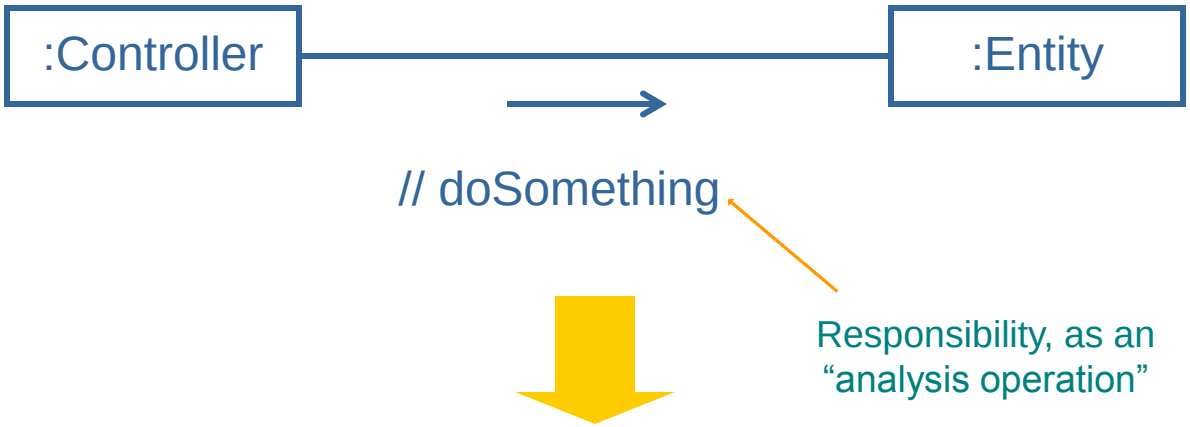
Use-Case Analysis Steps

- Refine the use-case description
- Model behavior using a sequence diagram
  - Identify analysis classes for the use-case
  - Model object collaborations
- Model structure in VOPC diagram
  - Capture responsibilities from interaction diagram
  - Add analysis-level attributes and relationships
  - Note analysis mechanisms
- Integrate analysis classes
- Document business rules

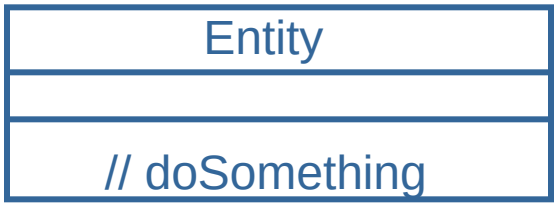
Responsibilities from Collaboration Diagram

- Responsibilities are specifications of object behavior

Collaboration Diagram



Class Diagram



Guidelines for (re)Structuring Responsibilities and Classes

- "Orthogonal" responsibilities within classes
  - ➔separate (e.g. //parseMessage and //displayMessage)
- Redundant responsibilities across classes
  - ➔Integrate (e.g. two classes determining look of gui)
- Each analysis class should have several compatible responsibilities. A class with only one responsibility is probably too simple.

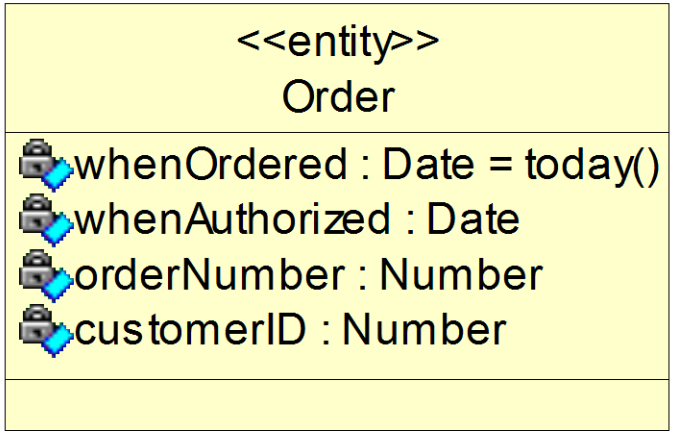
Q105

Use-Case Analysis Steps

- Refine the use-case description
- Model behavior using a sequence diagram
  - Identify analysis classes for the use-case
  - Model object collaborations
- Model structure in VOPC diagram
  - Capture responsibilities from sequence diagram
  - Add analysis-level attributes and relationships
  - Note analysis mechanisms
- Integrate analysis classes
- Document business rules

Attributes

- A named property of a class that describes a range of values that instances of the property may hold.
  - Types can be conceptual during analysis. E.g. ‘amount’ might become ‘integer’ or ‘double’ during design.
- Information retained by identified classes



B50,R208

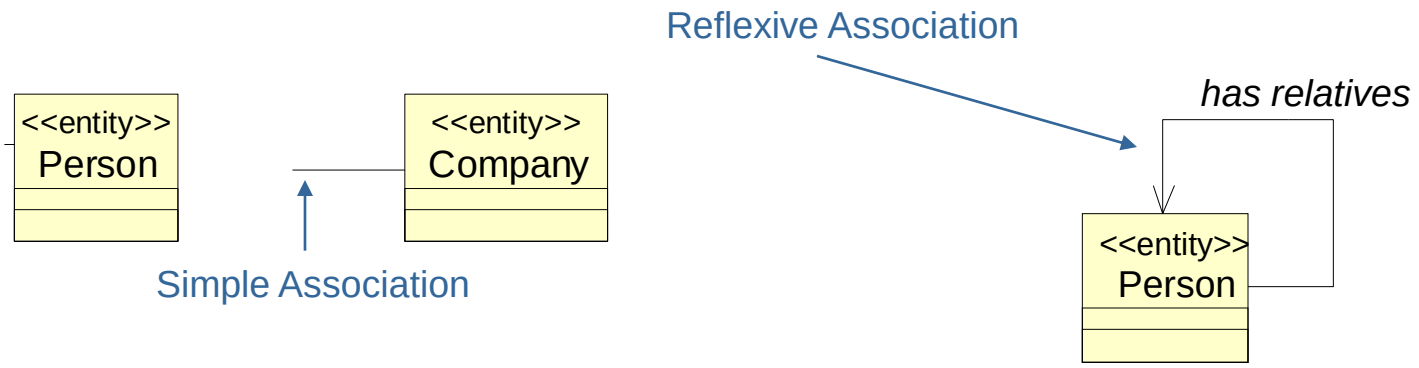
Where To Find Attributes

- Requirements: problem statement, set of system requirements, and flow of events documentation
- Domain expert
- “Nouns” that did not become classes
  - Information whose value is the important thing
  - Information that is uniquely "owned” by an object
  - Information that has no behavior
  - Note: Attributes are often realized as objects like instances of Date or List

Q110, J208-209

Review of Associations

- An association models a structural relationship between objects



B65-66, Q89

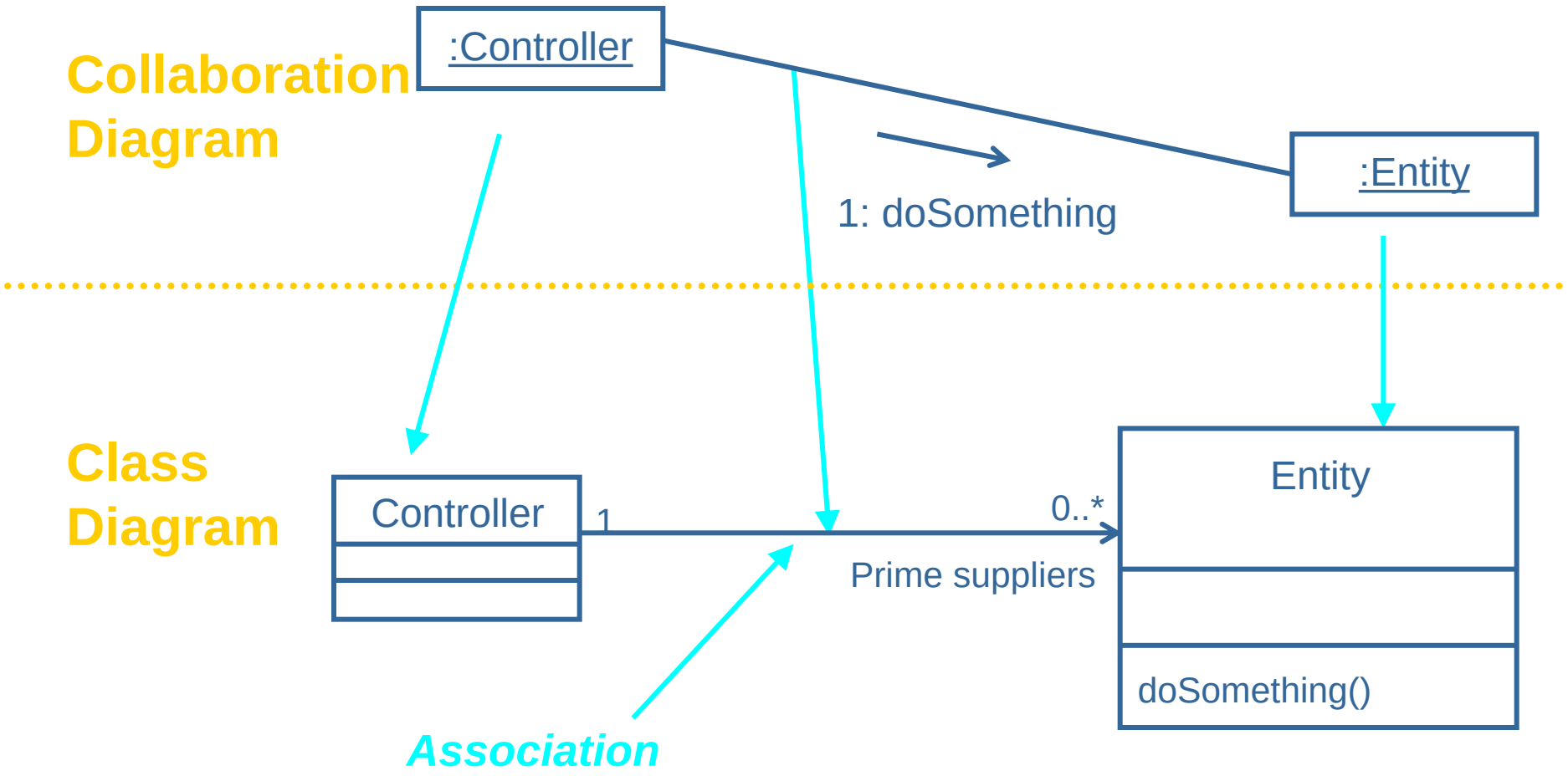
Review of Association Adornments

- Name
- Role
- Multiplicity
- Aggregation

B66

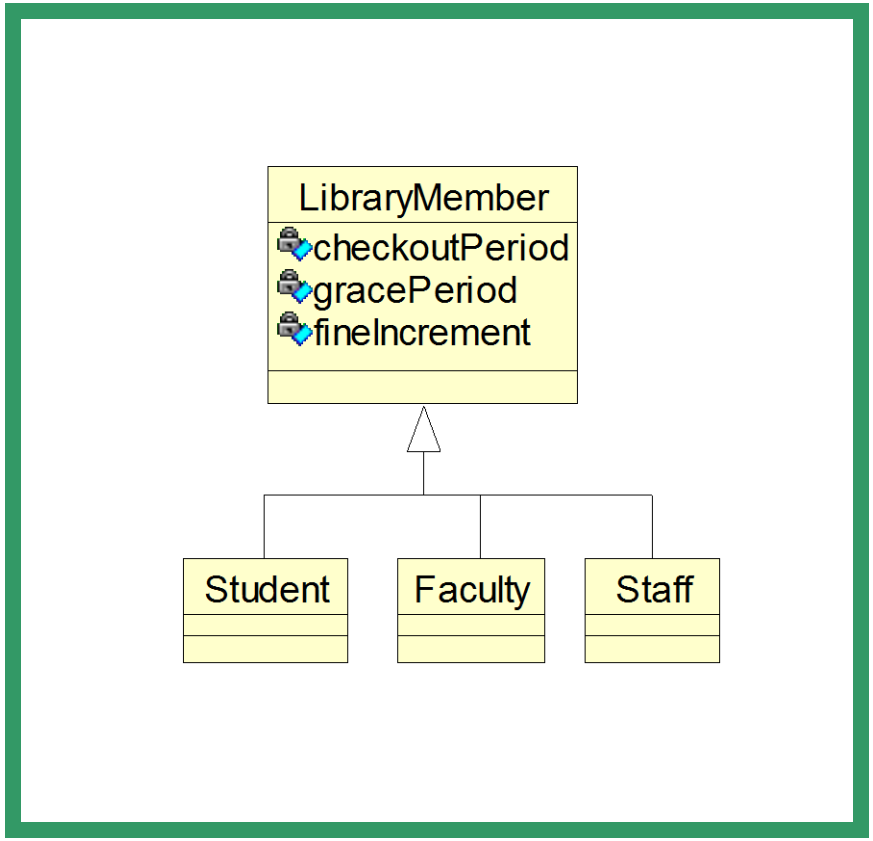
Relationships from Collaboration Diagram

- Create a relationship for each link



Review of Generalization

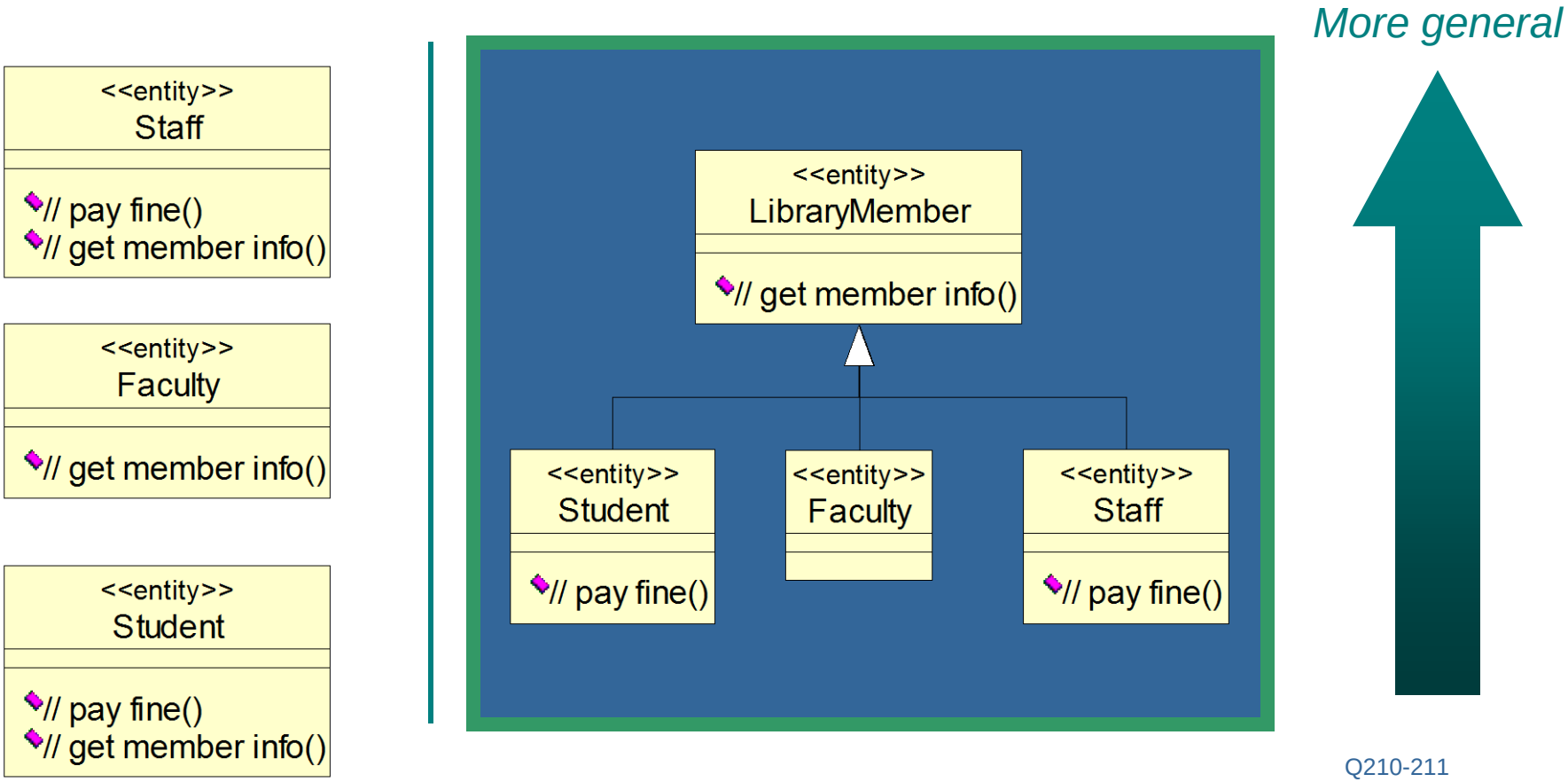
- One class shares the attributes and/or behavior of one or more classes
- "Is-a" relationship
- In analysis, keep on conceptual level just to make model easier to understand
- This is the only reason for our VOPC and our Sequence Diagram to have different classes.
- Sequence diagram can usually stay at the **superclass** level.



J210-211

Finding Generalization: Generalization of Classes

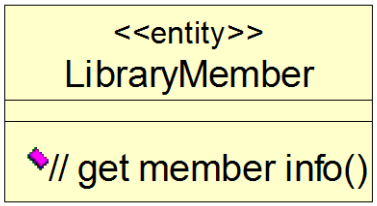
- Create superclasses which encapsulate structure and common behavior of several classes.

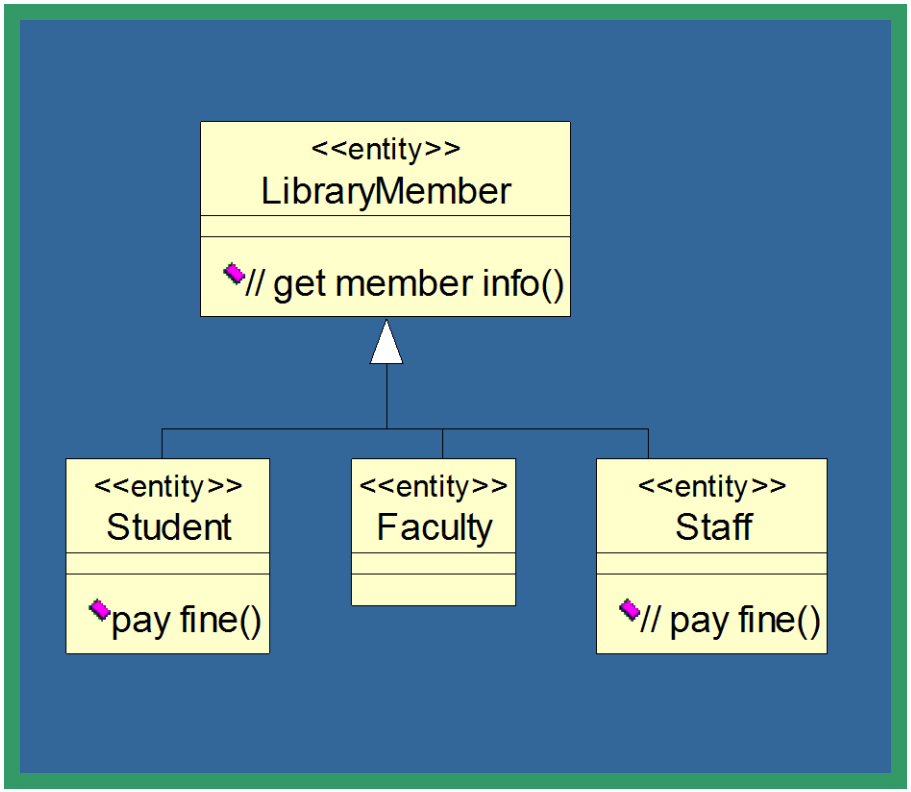


Q210-211

Finding Generalization: Specialization of Classes

- Create subclasses that add refinements of the superclass





More specific

Q210-211

Use-Case Analysis Steps

- Refine the use-case description
- Model behavior using a sequence diagram
  - Identify analysis classes for the use-case
  - Model object collaborations
- Model structure in VOPC diagram
  - Capture responsibilities from sequence diagram
  - Add analysis-level attributes and associations
- Verify business rules are covered

Use Case Analysis steps

1. Supplement the Use Case Descriptions
2. For each use case realization, find classes and distribute use case behavior to classes
3. Model the analysis classes with
   A. Sequence diagrams to show the flow of the application
   B. Collaboration diagrams to suggest relationships
   C. VOPC diagrams to specify static relationships and to elaborate attributes of each class
4. Verify that all your business rules are reflected in your use case diagrams. Your diagrams should show:
   A. Business rules are checked/verified
   B. An alternative path is noted for Business Rule violation.

Use-Case Analysis Review

- What is done during Use Case analysis?
- What is a use-case realization?
- What are the input and output artifacts of Use-Case Analysis?
- What is an analysis class? Describe the three analysis stereotypes.
- What are the dynamic and static aspects of use case analysis?
- How many sequence diagrams should be produced during Use-Case Analysis?