Course Review

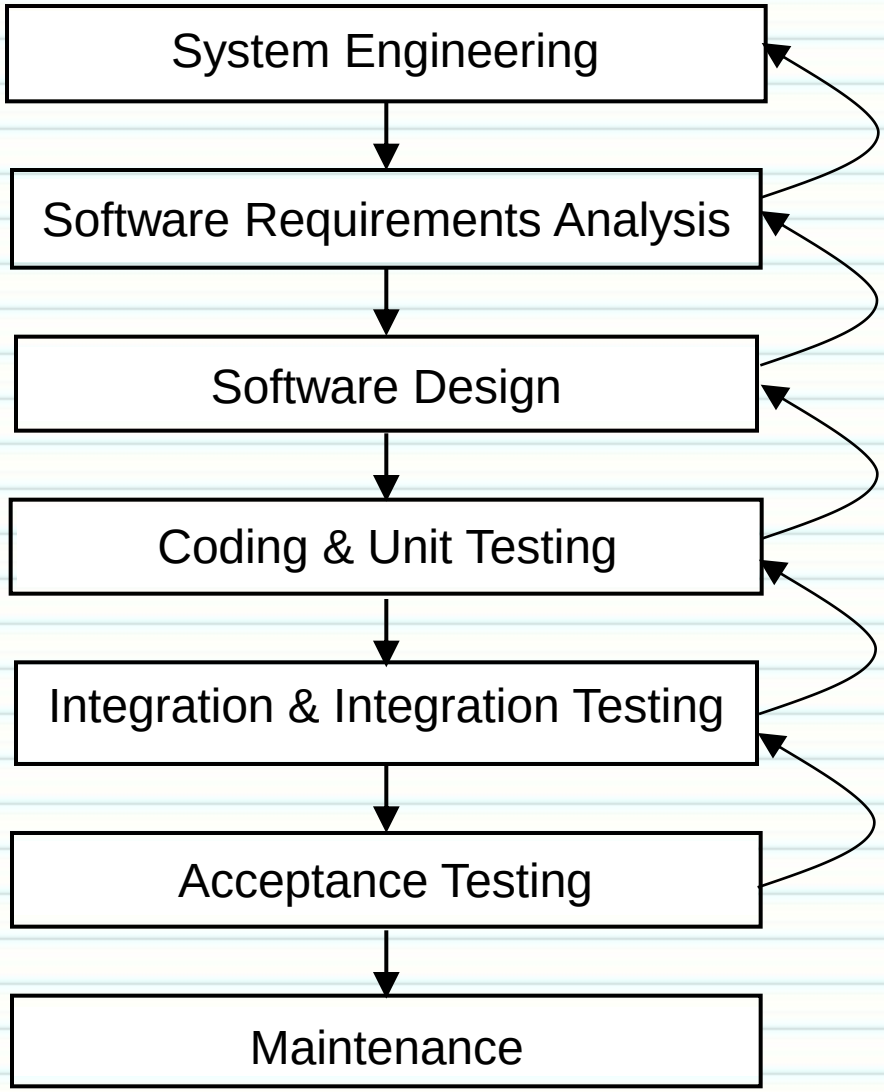
Dr John H Robb, PMP
UT Arlington
Computer Science and Engineering

Software Process

System development challenges call for an engineering approach for software development. A software process is required.

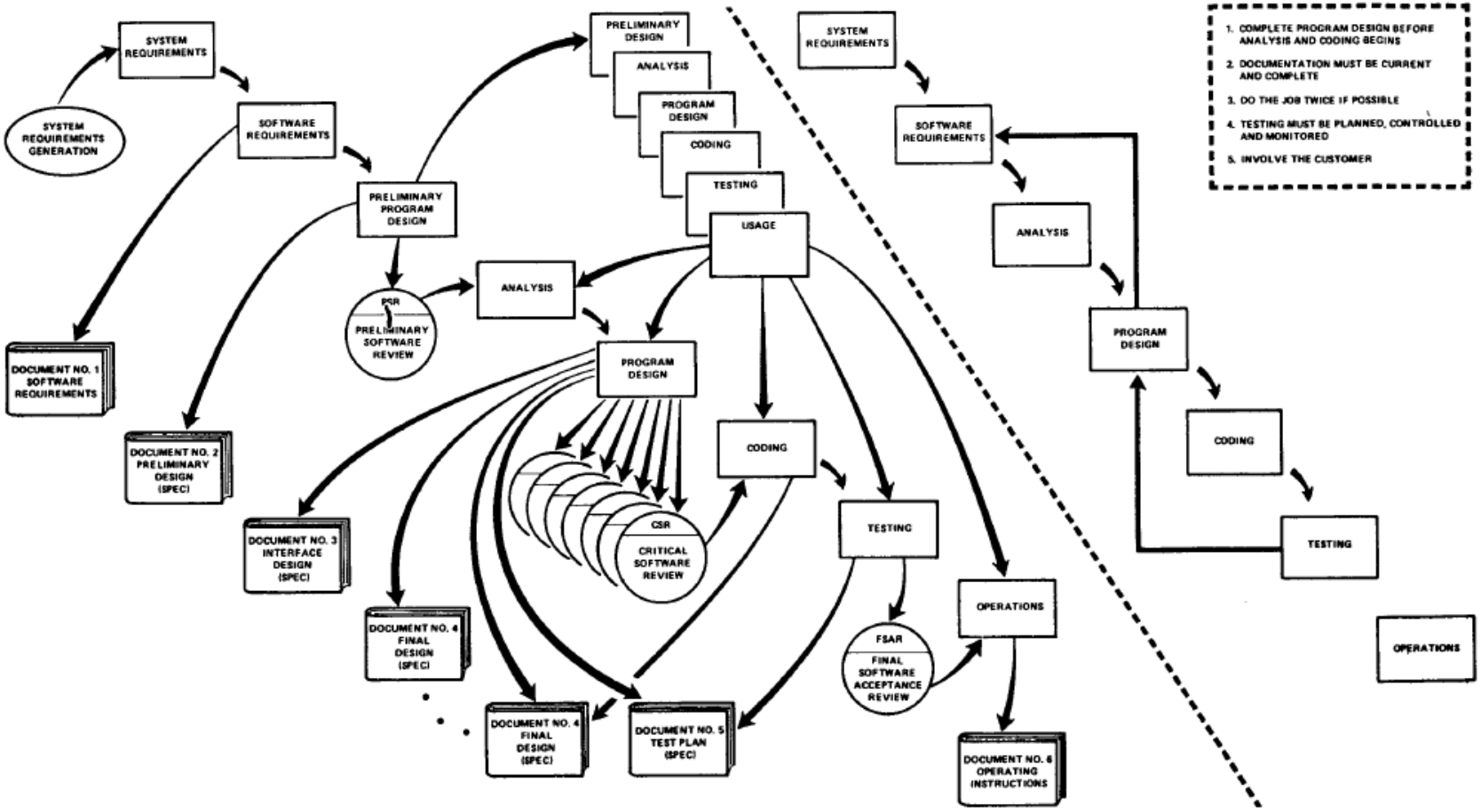
Definition 2.1 A *software process* defines a series of activities performed to construct a software system. Each activity produces some artifacts, which are the input to other phases. Each phase has a set of entrance criteria and a set of exit criteria.

The Waterfall Process



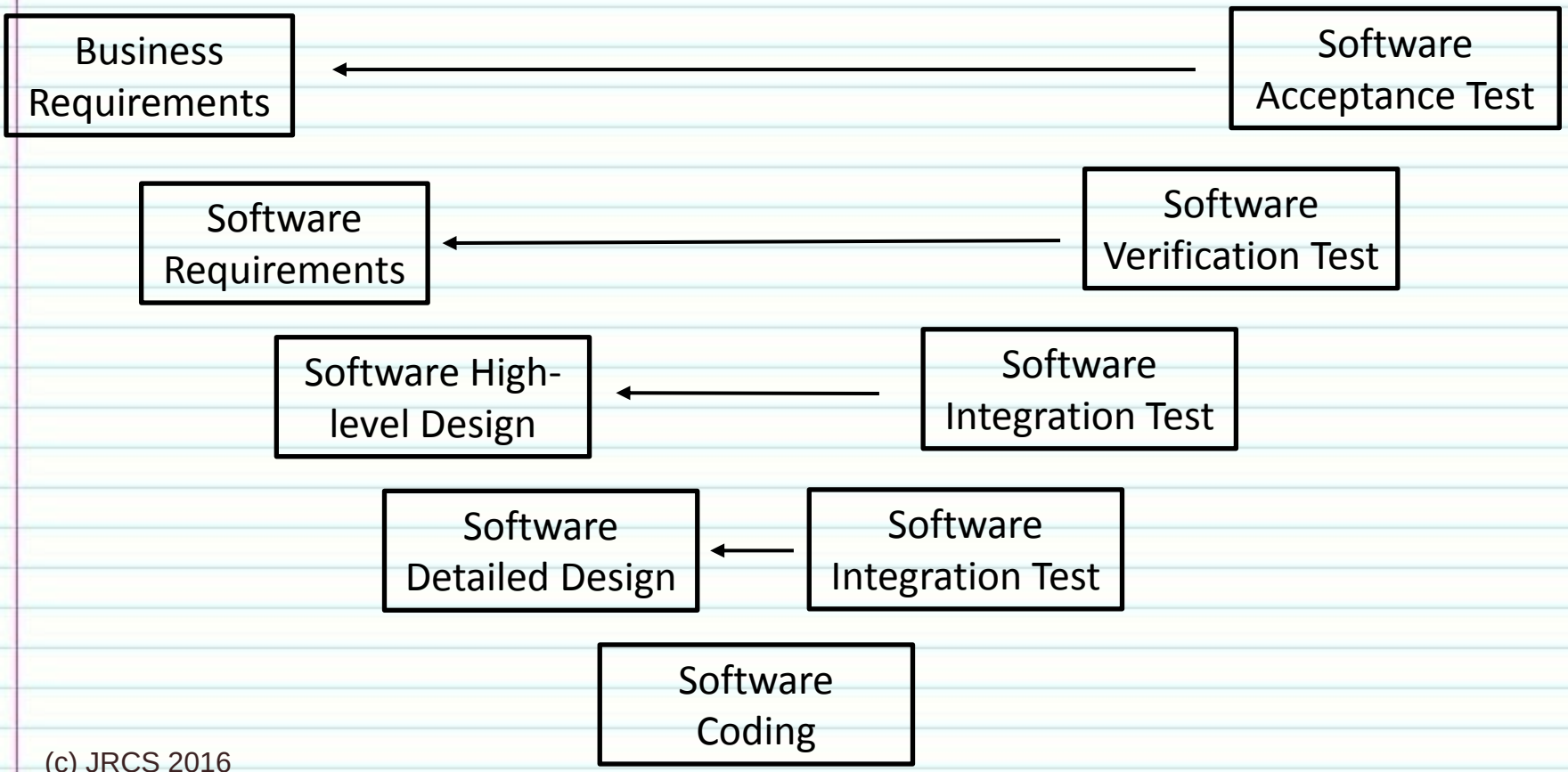
The Waterfall is All Wet! (cont.)

- This is Winston Royce's Waterfall



The V-Model

- The V-model places test activities in parallel to development activities - test and development activities occur at the same time - this is needed!
- Most people use an incremental V-model to avoid the at the end of the life-cycle customer delivery issue



Software Process Models

- V-Model
- Prototyping Process Model
- Evolutionary Process Model
- Spiral Process Model
- Unified Process Model
- Personal Software Process Model
- Team Software Process Model
- Agile Process Models

Spiral Process Model

It begins here.

If risks remains
{plan next phase
conduct prototyping }
else if risks resolved
{proceed as waterfall}
else if prototype works & robust
{proceed as evo. model}

Methodology

- A methodology is a cook-book for performing a task. It describes
  - steps to accomplish a series of subtasks
  - input and output of each step
  - representations of input and output
  - entrance and exit conditions for each step
  - procedures for carrying out each step
  - methods and techniques used by each step
  - relationships, or control flow and data flow between the steps

Process and Methodology

Process

- Defines a framework of phased activities
- Specifies phases of WHAT
- Does not dictate representations of artifacts
- It is paradigm-independent
- A phase can be realized by different methodologies.

Examples

Waterfall, spiral, prototyping, unified, and agile processes

Methodology

- Defines steps to carry out phases of a process
- Describes steps of HOW
- Defines representations of artifacts (e.g., UML)
- It is paradigm-dependent
- Steps describe procedures, techniques & guidelines

Examples

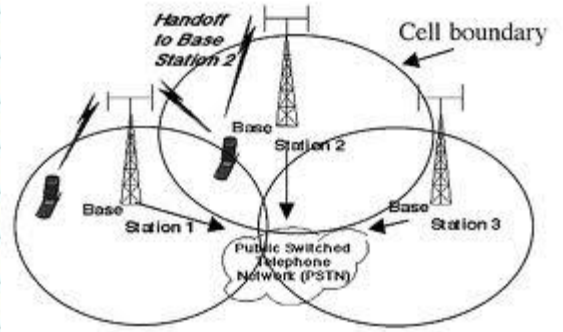
Structured analysis/structured design (SA/SD), Object Modeling Technique (OMT), Scrum, DSDM, FDD, XP, and Crystal Orange

The Methodology Presented in This Book

- It is designed for beginners as well as seasoned developers.
- It is aimed at educating software architects and systems analysts.
- It can be applied to agile as well as plan-driven projects. It has been applied to sponsored as well as industrial projects.
- Team members should work together from project start to completion.
- Many students continue practicing the methodology after graduation.

Main Characteristics of a System

A system consists of interacting



A system exists in a hierarchy of systems – a system may be a subsystem of another system.

Each system exists in an environment and interacts with the environment.

Systems are ever evolving.

The Requirements Process

The Requirements Process consists of the following steps

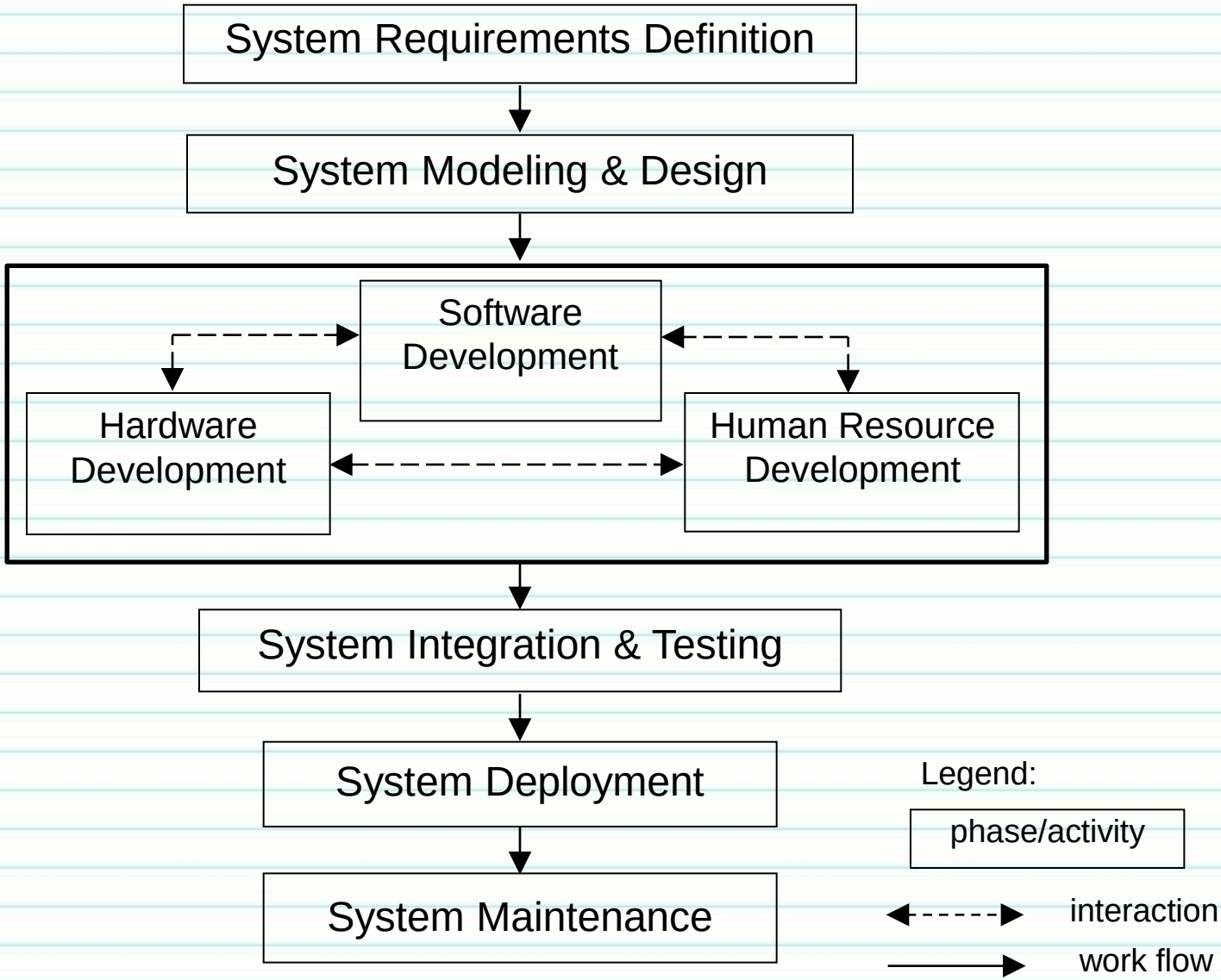
A. Requirements Planning (estimating requirements work)
B. Requirements Elicitation (draw-out the requirements)
C. Requirements Analysis (do they work and work together?)
D. Software Requirements Specification (capture requirements)
E. Requirements Validation
F. Requirements Management (requirements will change - they must be managed)
G. Requirements status reporting

Most of the industry is particularly weak in all but D and E above and many are weak here as well.

Questions to discuss:

1. What are typical software estimation measures and how do they apply to software requirements?

System Engineering Process



System Decomposition Strategies

1. Decompose the system according to system functions.
2. Decompose the system according to engineering disciplines.
3. Decompose the system according to existing architecture.
4. Decompose the system according to the functional units of the organization.
5. Decompose the system according to models of the application.

System Configuration Management

- System configuration management ensures that the system components are updated consistently.
- System configuration management is needed because
  - a system may have different versions and releases to satisfy the needs of different customers,
  - the engineering teams may update the system configuration concurrently.
- It is performed during the development phase as well as the maintenance phase.
- Its functions include configuration identification, configuration change control, configuration auditing, and configuration status reporting.

Key Takeaway Points

- Requirements are capabilities that the system must deliver.
- *The hardest single part of building a software system is deciding precisely what to build—i.e., the requirements*. (Frederick P. Brooks, Jr.)
- Requirements are the main challenge in developing software - this is the problem area!
- Software requirements elicitation is aimed to identify the real requirements for the system – this may not be the same as what the customer asked for!

- This module provides an overview of the requirements process and then focuses on the requirements elicitation - which is the emphasis of the textbook and requirements specification - which is the focus of the project

The Requirements Process

The Requirements Process consists of the following steps

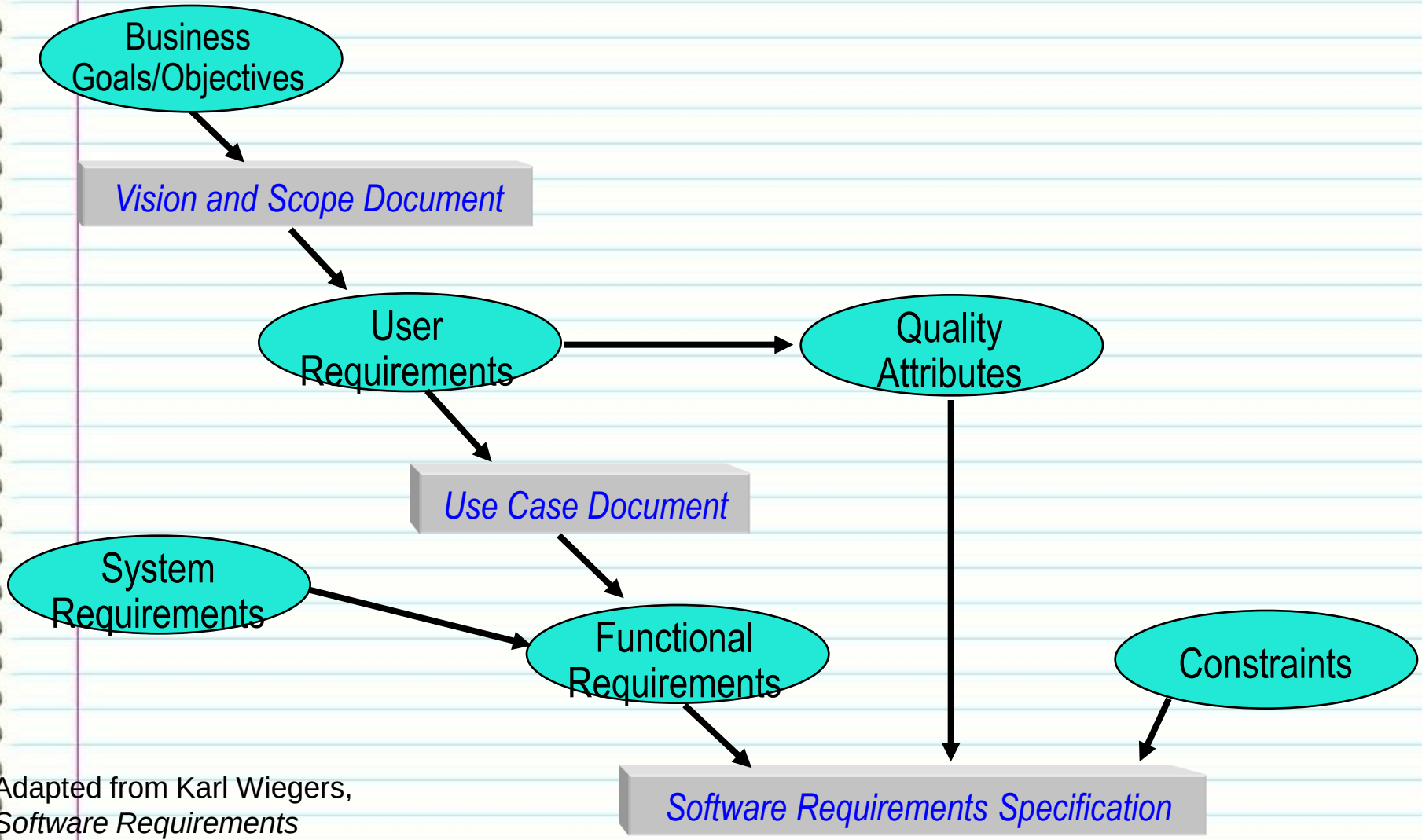
A. Requirements Planning (estimating requirements work)
B. Requirements Elicitation (draw-out the requirements)
C. Requirements Analysis (do they work and work together?)
D. Software Requirements Specification (capture requirements)
E. Requirements Validation
F. Requirements Management (requirements will change - they must be managed)
G. Requirements status reporting

Most of the industry is particularly weak in all but D and E above and many are weak here as well.

Questions to discuss:

1. What are typical software estimation measures and how do they apply to software requirements?

Software Requirements Specification



Adapted from Karl Wiegers, *Software Requirements*

Types of Requirement

- Functional requirements – statements of information processing capabilities that the software system must possess.
- Nonfunctional requirements include
  - Performance requirements
  - Quality requirements
  - Safety requirements
  - Security requirements
  - Interface requirements

Examples of Constraints

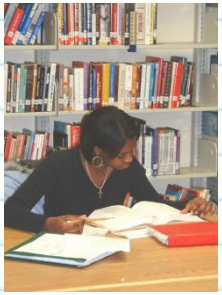
- For this class a constraint is the following example:
  - C1 - The system will work on our existing technical infrastructure - no new technologies will be introduced.
  - C2 - The system will only use the data contained in the existing corporate database.
- Notice that a constraint uses the word only, no, not - words limiting choices - a constraint should cause one or more non-functional requirements
- Notice that these could be captured as the following non-functional requirements (but they are different than the constraints)
  - R1 - The system shall use the existing technical infrastructure, no new technologies shall be introduced
  - R2 - The system shall use only the data contained in the existing corporate database - no other data shall be used
- Constraints spawn non-functional requirements, not vice-versa. The best requirements have both and show the linkage between them.

Information Collection Techniques

Customer presentation

business forms

operating procedures



regulations & standards

Literature survey

Stakeholder survey

User interviewing

Writing user stories

Requirements Specification

1. Introduction to Document
   - 1.1 Purpose of Product
   - 1.2 Scope of Product
   - 1.3 Acronyms, Abbreviations, Definitions
   - 1.4 References
   - 1.5 Outline of the Rest of the SRS
2. General Description of Product
   - 2.1 Context of Product
   - 2.2 Product Function
   - 2.3 User Characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and Dependencies
3. Specific Requirements
   - 3.1 External Interface Requirements
     - 3.1.1 User Interfaces
     - 3.1.2 Hardware Interfaces
     - 3.1.3 Software Interfaces
     - 3.1.4 Communication Interfaces
   - 3.2 Functional Requirements
     - 3.2.1 Class 1
     - 3.2.2 Class 2
     - 3.2.3 ...
   - 3.3 Performance Requirements
   - 3.4 Design Constraints
   - 3.5 Quality Requirements
   - 3.6 Other Requirements
4. Appendices

IEEE SRS Standard by Objects, 1998

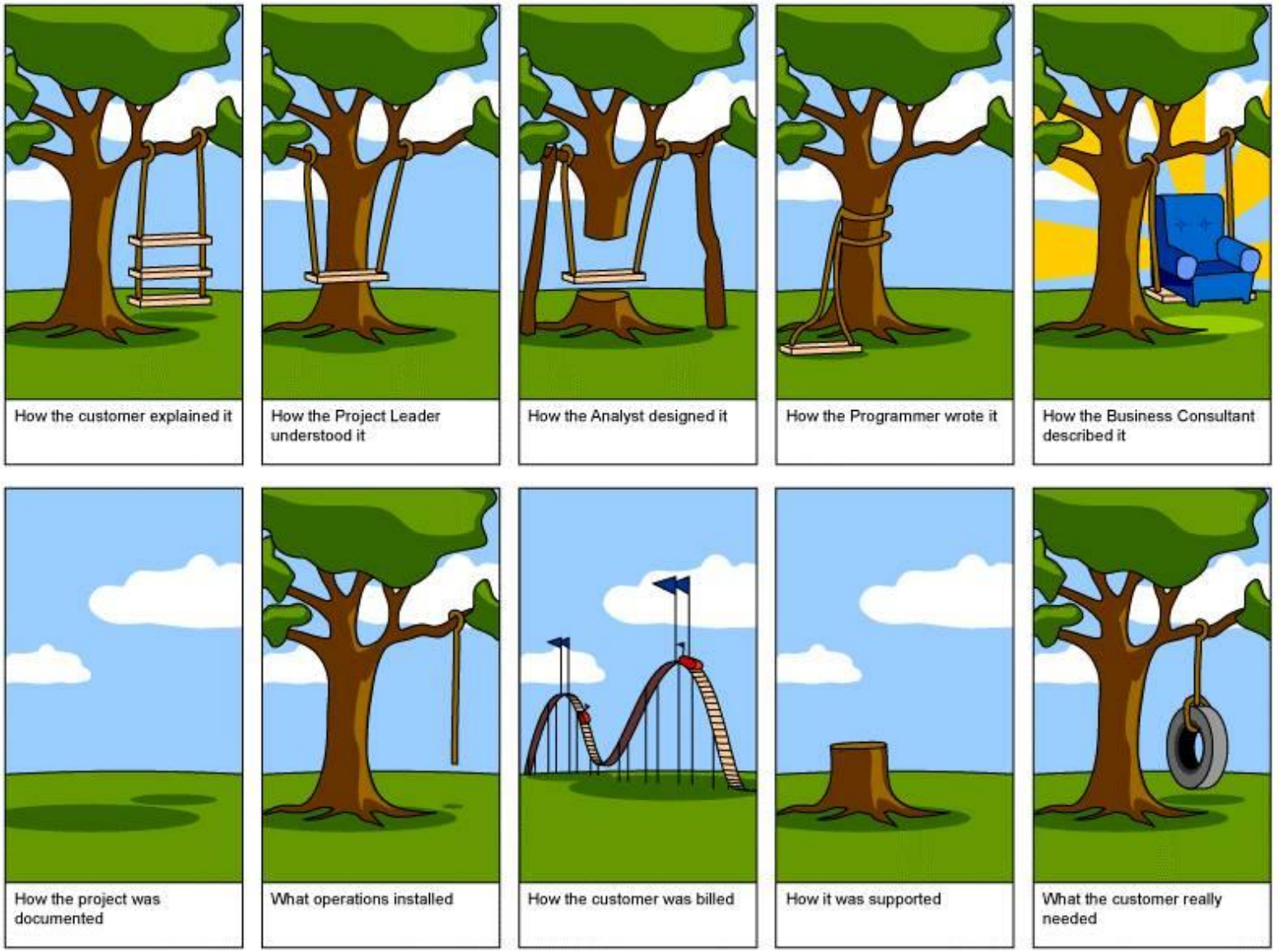
Feasibility Study

- Not all projects are practically doable with technology, time, and resource constraints.
- Feasibility study aims at determining if the project is doable under the given constraints.
- Feasibility study in RE is concerned with
  - the feasibility of the functional, performance, nonfunctional, and quality constraints
  - adequacy of the technology
  - timing and cost constraints
  - constraints imposed by the customer, industry and government agencies

Three Types of Requirements Review

- Technical review is an internal review performed by the technical team. Techniques include:
  - **peer review -** peers perform informal "desktop reviews" sometimes guided by a review questionnaire
  - **walkthrough -** the analyst explains each requirement while the reviewers examine it and raise doubts
  - **inspection** - inspector is guided by a checklist of commonly encountered problems in SRS (e.g., incompleteness, duplicate definition, inconsistency, etc.)
- In every case there is a requirement for period of time between which the review materials are available and the review is held – to allow for the pre-review (this is where most errors are found)

What is the right system to build ?



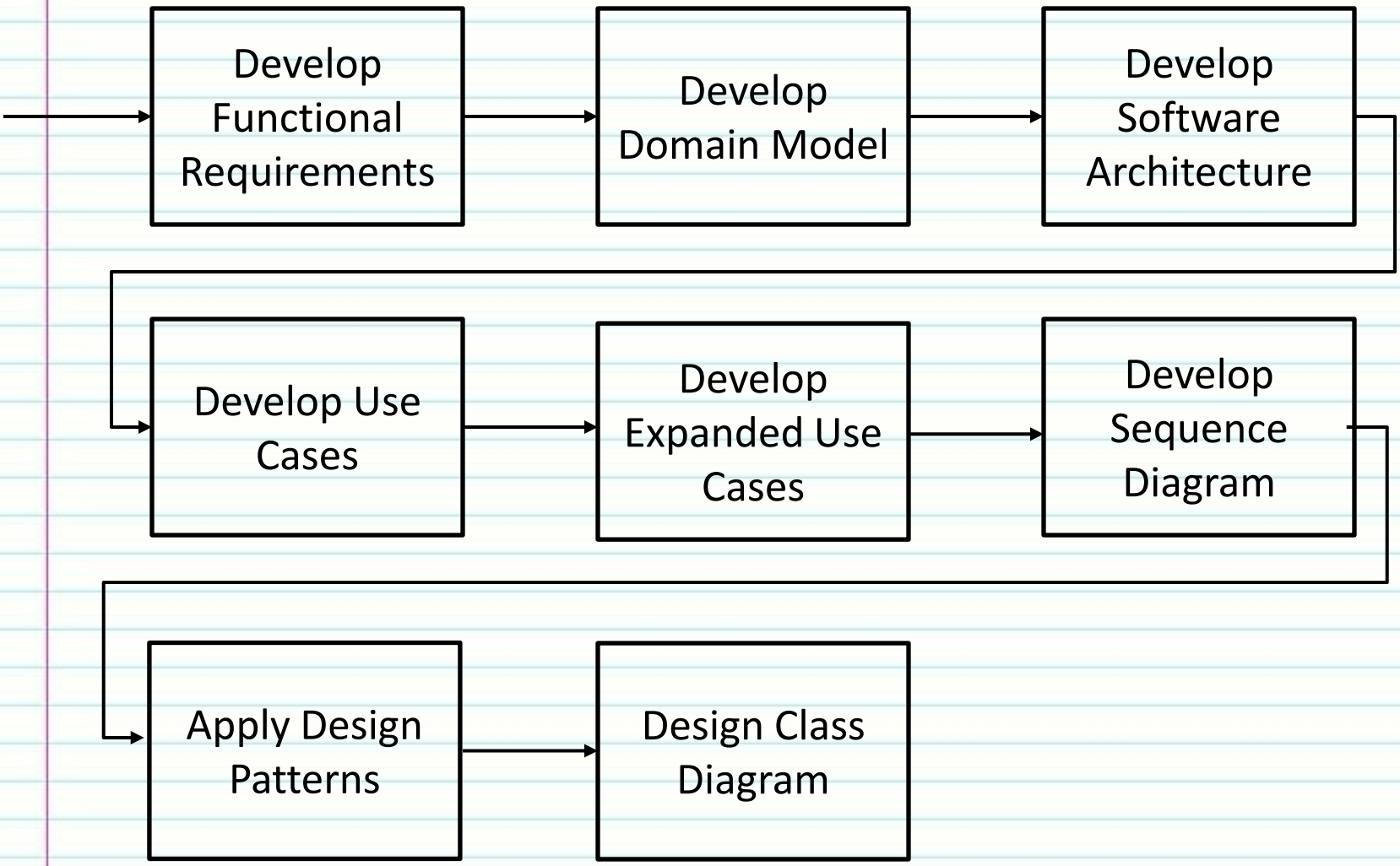
Approach to Use for the Class Project

1. Develop Scenarios
   a) For each project functions - step through how each will be performed by each of the users
      i. example, search for a restaurant - what fields are used in the search? What results and fields are returned from the search?
   b) from these scenarios develop
      i. specific requirements (what the function does)
      ii. the inputs required to perform the function (as a table)
      iii. the outputs provided by the function (as a table)
   c) Capture these requirements and tables in the provided spreadsheet
2. Make sure that each function provides:
   a) A confirmation of actions (e.g., "login successful" message)
   b) If results are returned how are they ordered?
3. Total number of functional requirements should be a minimum of 20-25 and not to exceed 35

Specifying Requirements and Attributes (cont.)

- This works because Android is basically a data packaging device - so it performs simple tasks on input attributes and produces specific output attributes
- Typically the number of inputs and output attributes are very small - but this needs to be nailed down very early.
- Much of Android functionality is performed by providing functions such as a Search function - this should be viewed as a form to be filled out.
- Android functionality is replete with forms - specify these attributes early - use them as associative classes to break up the *:* multiplicities

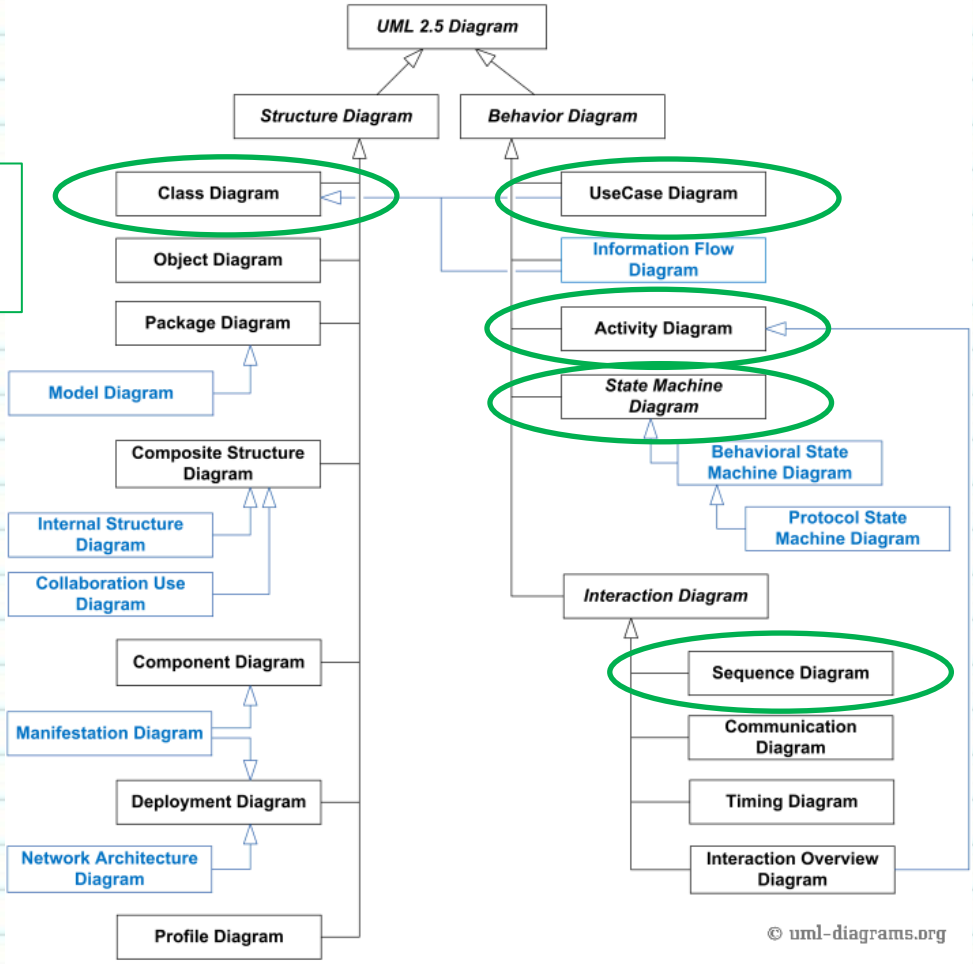
Book Approach to OOSE



Domain Modeling and UML

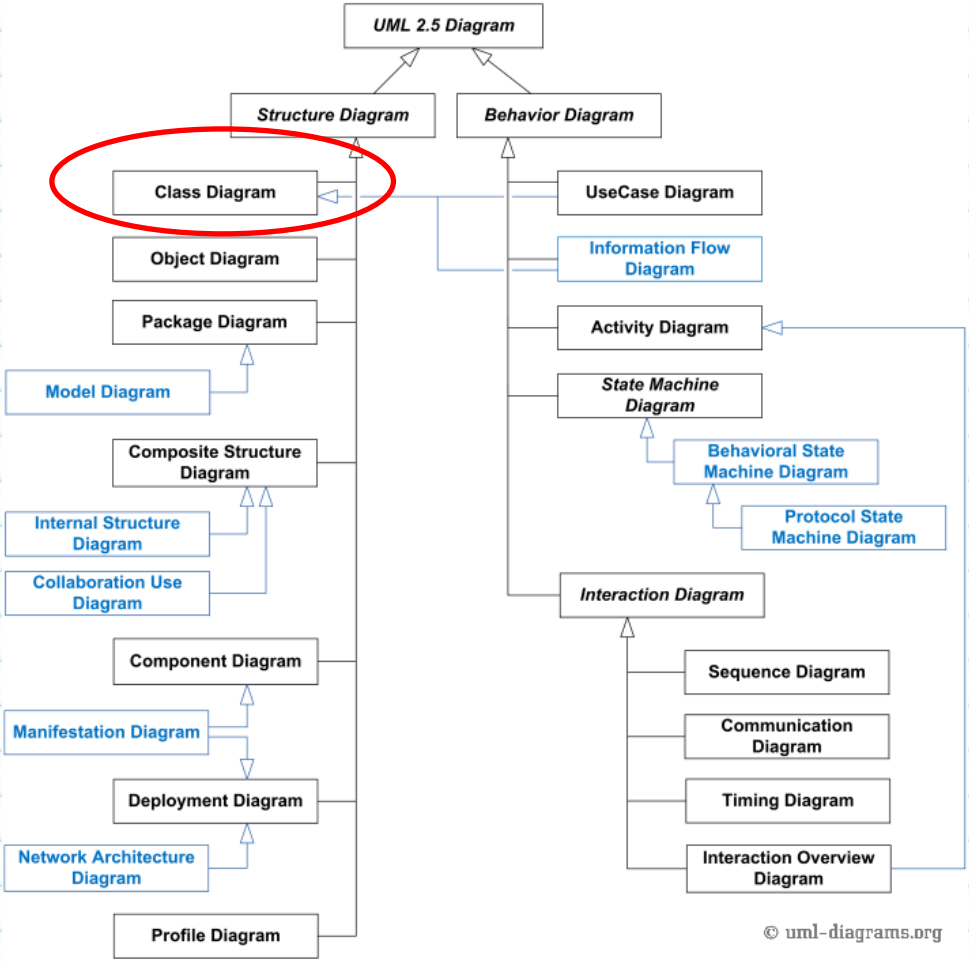
- The Domain model represents our first use of a UML structure in the class so it's important to look at the overall UML strucuture
- UML has two sides - behavior (dynamic) and structure (static)

We will learn these UML diagrams this semester



Domain Modeling and UML (cont.)

- The Domain model and Domain diagram are not officially part of UML 2.5 - the domain model is a top-level Class diagram which is in UML 2.5



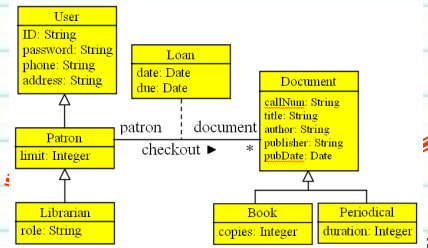
Domain Modeling Steps

1.Collecting application domain information

2. Brainstorming

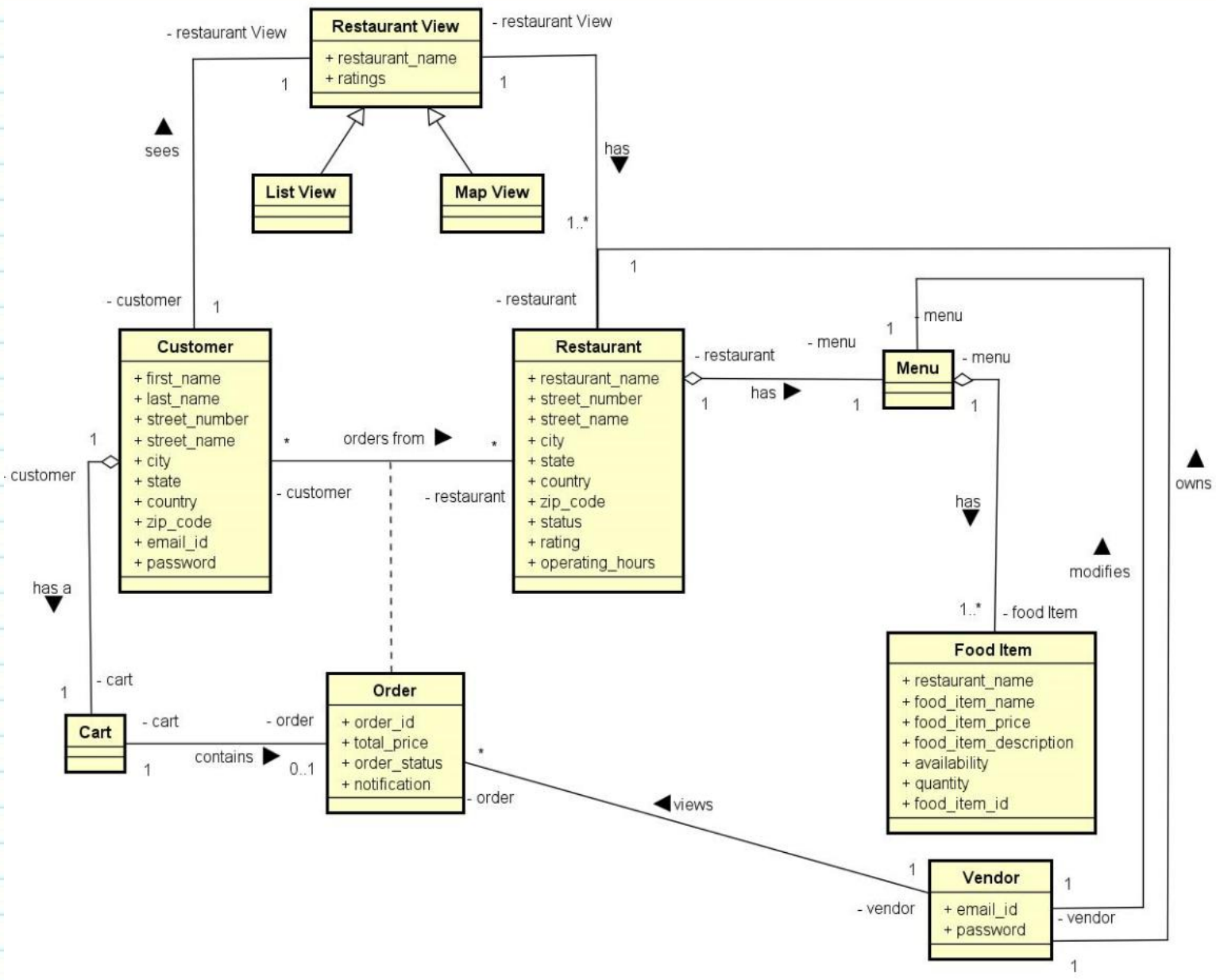
3. Classifying brainstorming result

4. Visualizing the domain model



5. Reviewing the domain model.

Student Project Domain Model



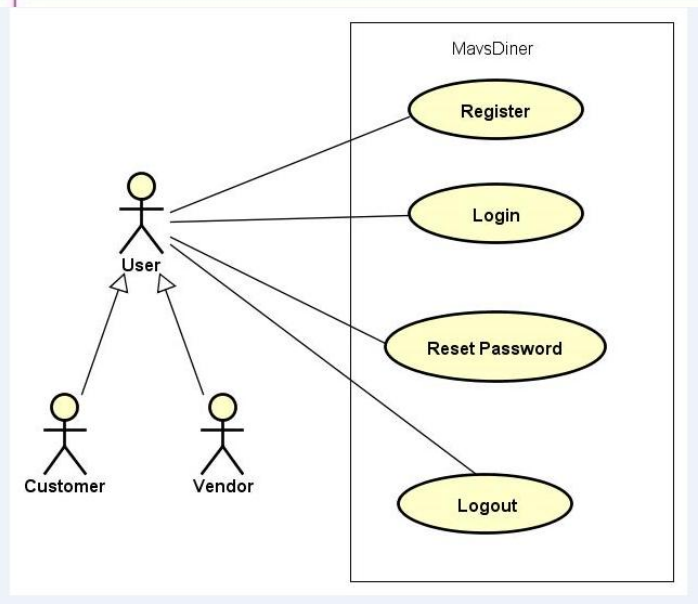
What Is a Use Case?

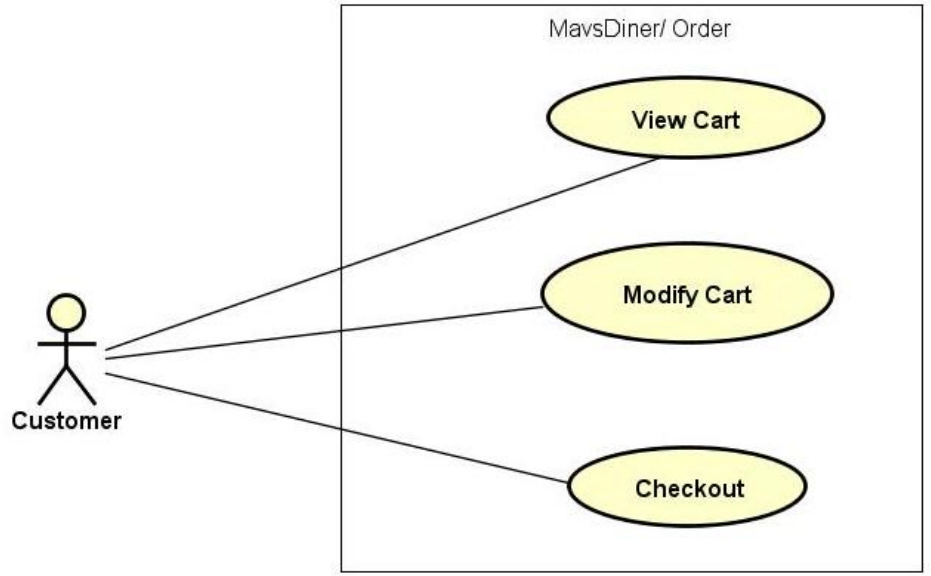
- *A use case is a business process.*
- A use case must be initiated by an actor.
- A use case must end with the actor.
  - The actor explicitly or implicitly acknowledges the accomplishment of the business task.
- A use case must accomplish a business task (for the actor).

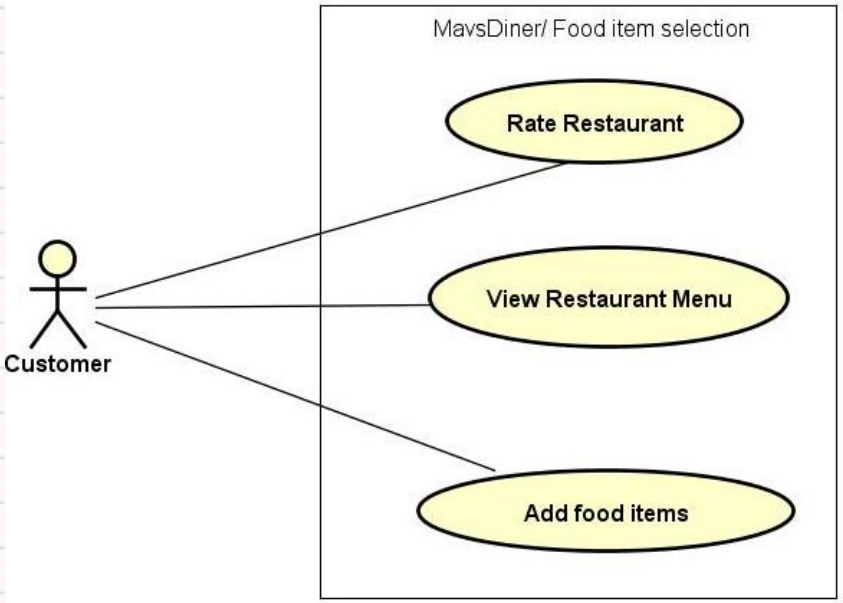
What Is an Actor?

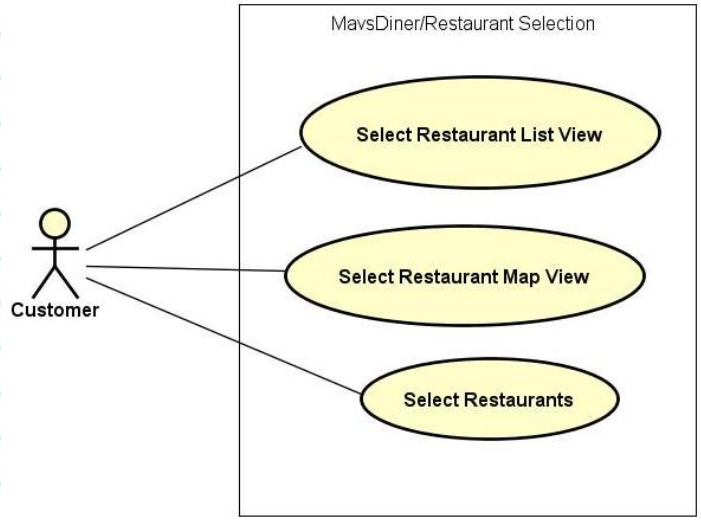
- An actor denotes a *business role* played by (and on behalf of) a set of business entities or stakeholders.
- Actors are not part of the system.
- Actors interact with the system.
- Actors are often human beings but can also be a piece of hardware, a system, or another component of the system.
- Actors initiate use cases, which accomplish business tasks for the respective actors.

Student Project UCD





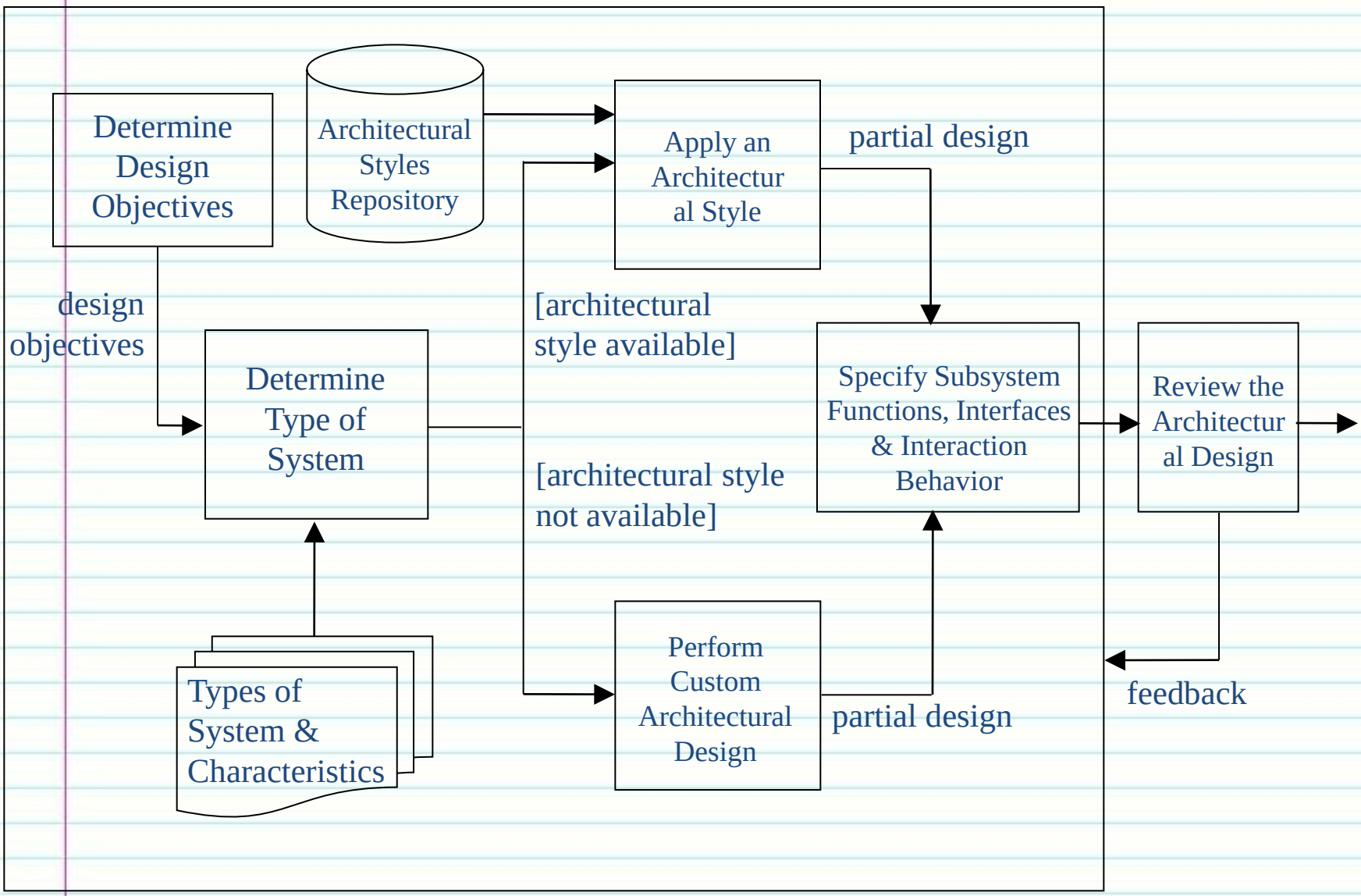




HL UCs

UC 1: Signup	- **TUCBW** the Commuter/rider will be able to click on the Create a new account link on Login Page. -**TUCEW** the Commuter/Rider gets the access and can view the “Login” page.
UC 2: Login	- **TUCBW** the user enters the UTA email id, password and clicks on the “Login” button. -**TUCEW** the user gains access into the system. New Commuter/Rider can view the “Profile” page; existing Commuter/Rider can view “Ride Management” page and the “Admin” will be able to see “Admin Home” page.
UC 2.1: Reset Password	- **TUCBW** the Commuter/Rider clicks on “Forgot Password?” link on the “Login” page. -**TUCEW** the Commuter/Rider sees the new password has been emailed message.
UC 3: Profile Management	- **TUCBW** the Commuter/Rider view create profile or Commuter/Rider clicks on update profile -**TUCEW** the Commuter/Rider sees the successfully created/updated profile message.
UC 3.1: Create User Profile	- **TUCBW** the new Commuter/Rider fills in ‘create profile’ form and clicks on Submit button. -**TUCEW** the Commuter/Rider sees the successfully created profile message.
UC 3.2: Update User Profile	- **TUCBW** the Commuter/Rider clicks on the Profile Management Button on the Home Page. -**TUCEW** the Commuter/Rider sees the successfully updated profile message.

Architectural Design Process



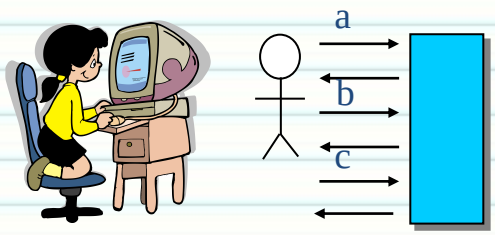
Guidelines for Architectural Design

1. Adapt an architectural style when possible.
2. Apply software design principles.
3. Apply design patterns.
4. Check against design objectives and design principles.
5. Iterate the steps if needed.

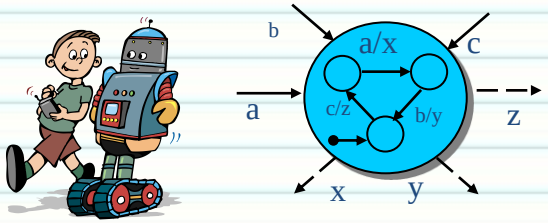
Architectural Design Considerations

- Ease of change and maintenance.
- Use of commercial off-the-shelf (COTS) parts.
- System performance – does the system require to process real-time data or a huge volume of transactions?
- Reliability.
- Security.
- Software fault tolerance.
- Recovery.

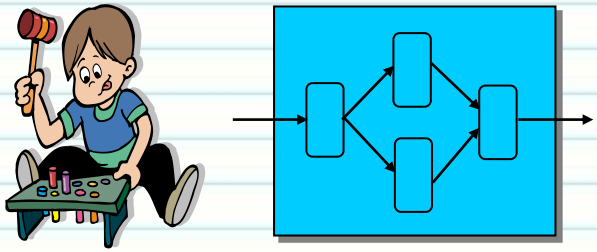
Four Common Types of Systems



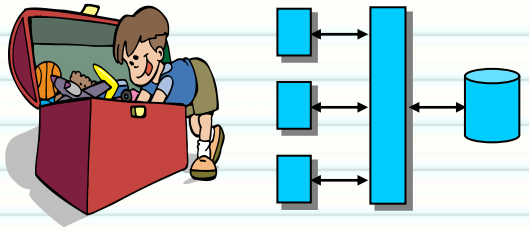
(a) Interactive subsystem



(b) Event-driven subsystem



(c) Transformational subsystem



(d) Database subsystem

System Types and Architectural Styles

Type of System	Architectural Style
Interactive System	N-Tier
Event-Driven System	Event-Driven
Transformational System	Main Program and Subroutines
Object-Persistence Subsystem	Persistence Framework
Client-server	Client-server
Distributed, decentralized	Peer-to-peer
Heuristic problem-solving	Blackboard

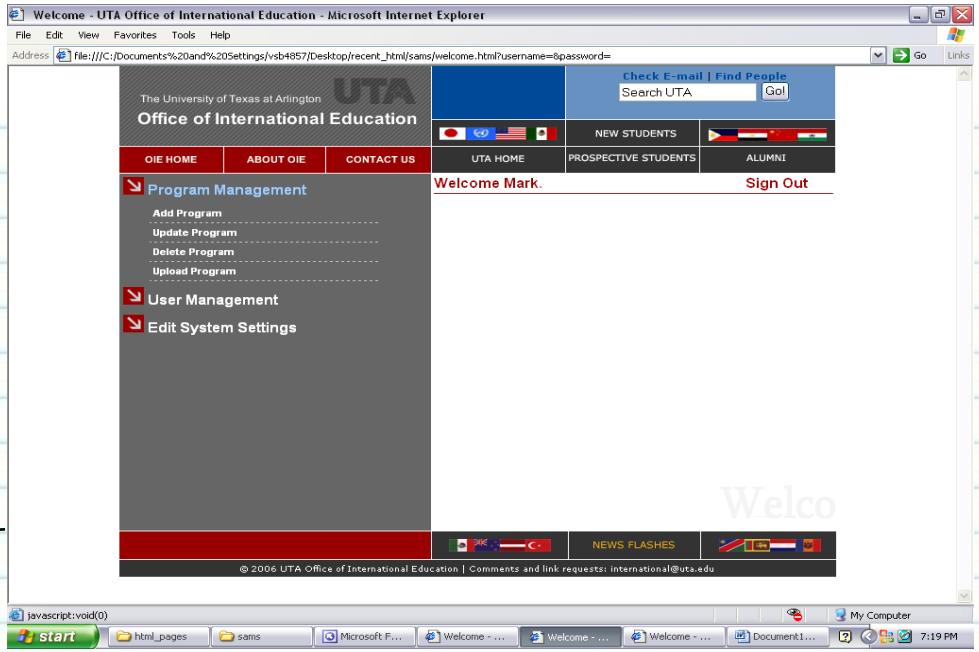
Expanded Use Case with Pre/Post-Conditions

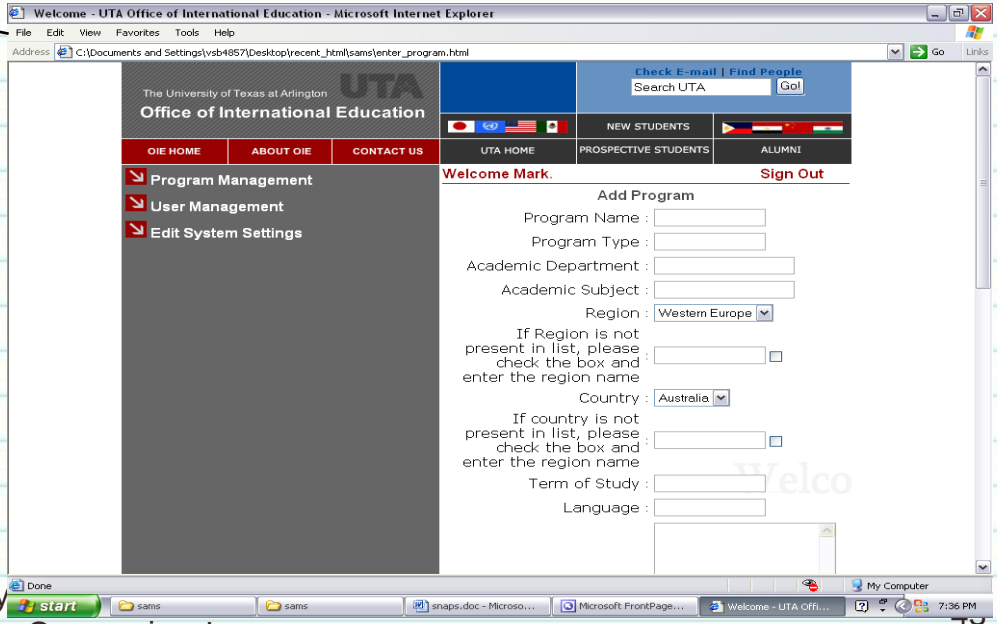
***Precondition**: This use case assumes that the staff user has logged into the system and is seeing the staff main page.*	
Actor: Staff User	System: SAMS
	0. System displays the staff main page.
1. **TUCBW** the staff user clicks the "Add Program" button.	2. System displays the Add Program page.
3. The staff user enters program detail and clicks the "submit" button.	4. System checks the submitted info and shows a confirmation message if no error is found.
5. **TUCEW** the staff user clicks the "OK" button on the confirmation page.	
***Postcondition**: The added program is immediately available for search.*	

Showing UI Prototypes w/ Expanded Use Case

UI Prototype always shown only on the System Side!

Actor : Staff User	System : SAMS
(1) **TUCBW** staff member clicks on Program Management link on welcome page.	(2) System shows the submenu consisting of various operations that a staff user can do under Program Management.
(3) Staff member clicks on Add Program link.	(4) System shows the Add Program Form
(5) User fills the form and clicks on submit button.	(6) The System updates the database with the new program details and displays a success message.
(7) **TUCEW** staff member is shown a message that the program has been successfully added.	





Example Student Project

UC 1: Signup

Precondition: The user does not have a registered account in the UTA Shared Ride System.	
Actor: Commuter/Rider	**System: UTA Shared Ride**
	0. System displays the Login page.
1. **TUCBW** the Commuter/rider will be able to click on the *Create a new account* link on Login Page.	2. System display Signup form in Signup page. (Refer Figure 1)
3. Commuter/Rider fills the form and clicks on *Sign Me Up* button.	4. System displays Signup Successful message and Commuter/Rider is redirected back to Login Page.(Refer Figure 2, Figure 3)
5. TUCEW the Commuter/Rider gets the access and can view the "Login" page.	
Post condition: A new account for Commuter/Rider is created in the system.	

Example Student Project (cont.)

Figure 1:

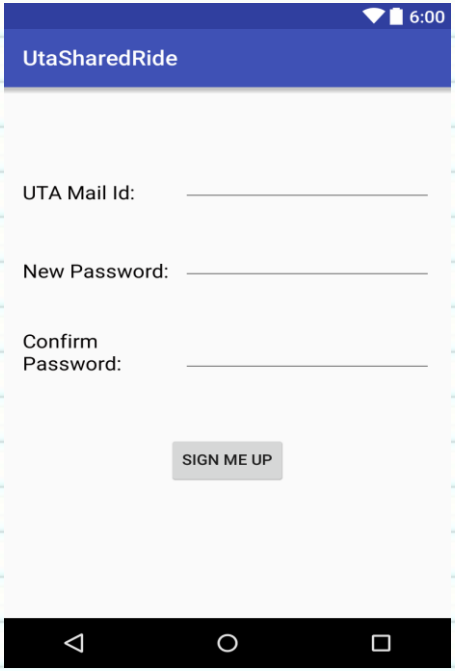
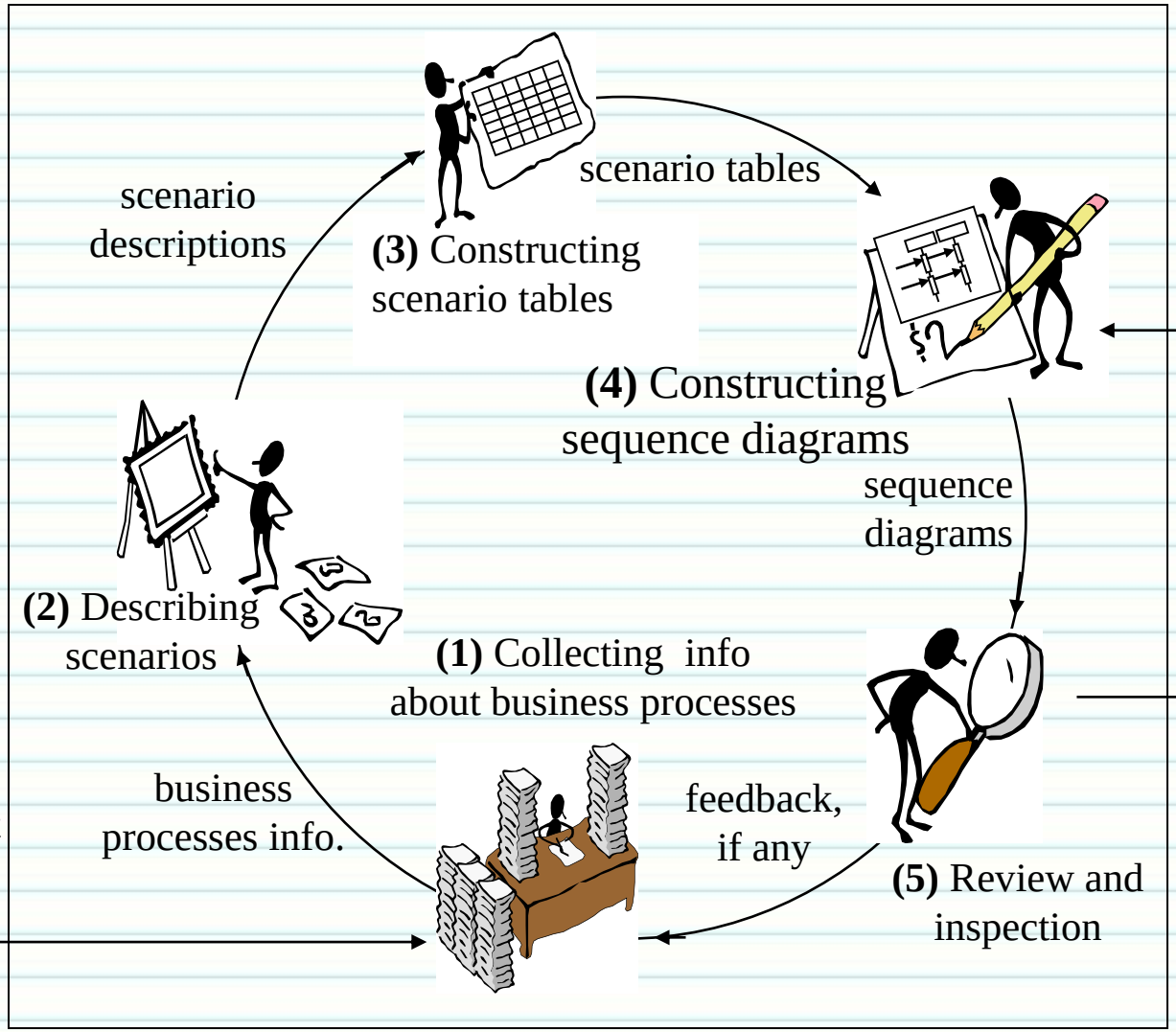
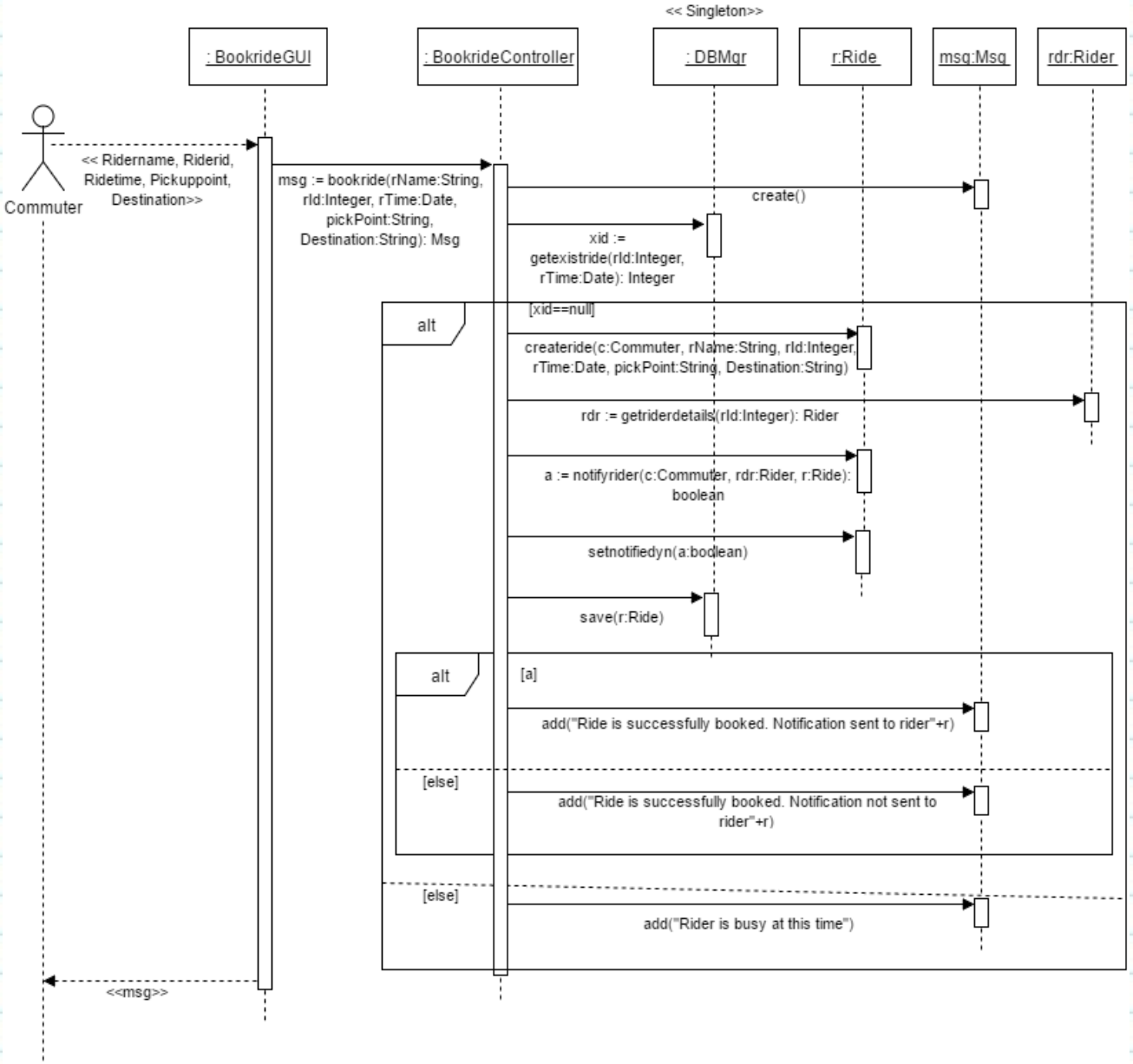


Figure 2:

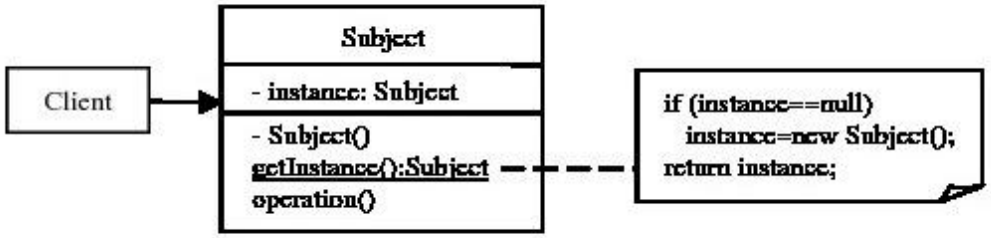
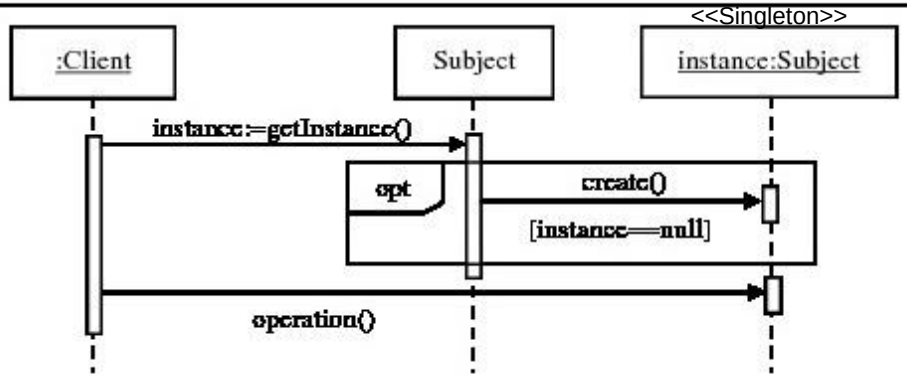
Object Interaction Modeling Steps



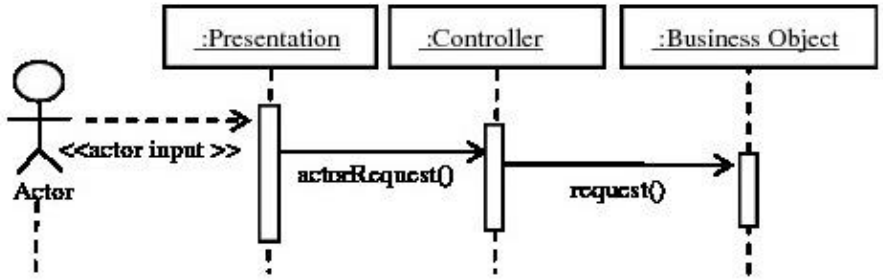
Book Ride


: BookrideGUI
: BookrideController
<< Singleton>>
: DBMgr
r:Ride
msg:Msg
rdr:Rider
Commuter
<< Ridername, Riderid, Ridetime, Pickuppoint, Destination>>
msg := bookride(rName:String, rId:Integer, rTime:Date, pickPoint:String, Destination:String): Msg
create()
xid := getexistride(rId:Integer, rTime:Date): Integer
alt
[xid==null]
createride(c:Commuter, rName:String, rId:Integer, rTime:Date, pickPoint:String, Destination:String)
rdr := getriderdetails(rId:Integer): Rider
a := notifyrider(c:Commuter, rdr:Rider, r:Ride): boolean
setnotifiedyn(a:boolean)
save(r:Ride)
alt
[a]
add("Ride is successfully booked. Notification sent to rider"+r)
[else]
add("Ride is successfully booked. Notification not sent to rider"+r)
[else]
add("Rider is busy at this time")
<<msg>>

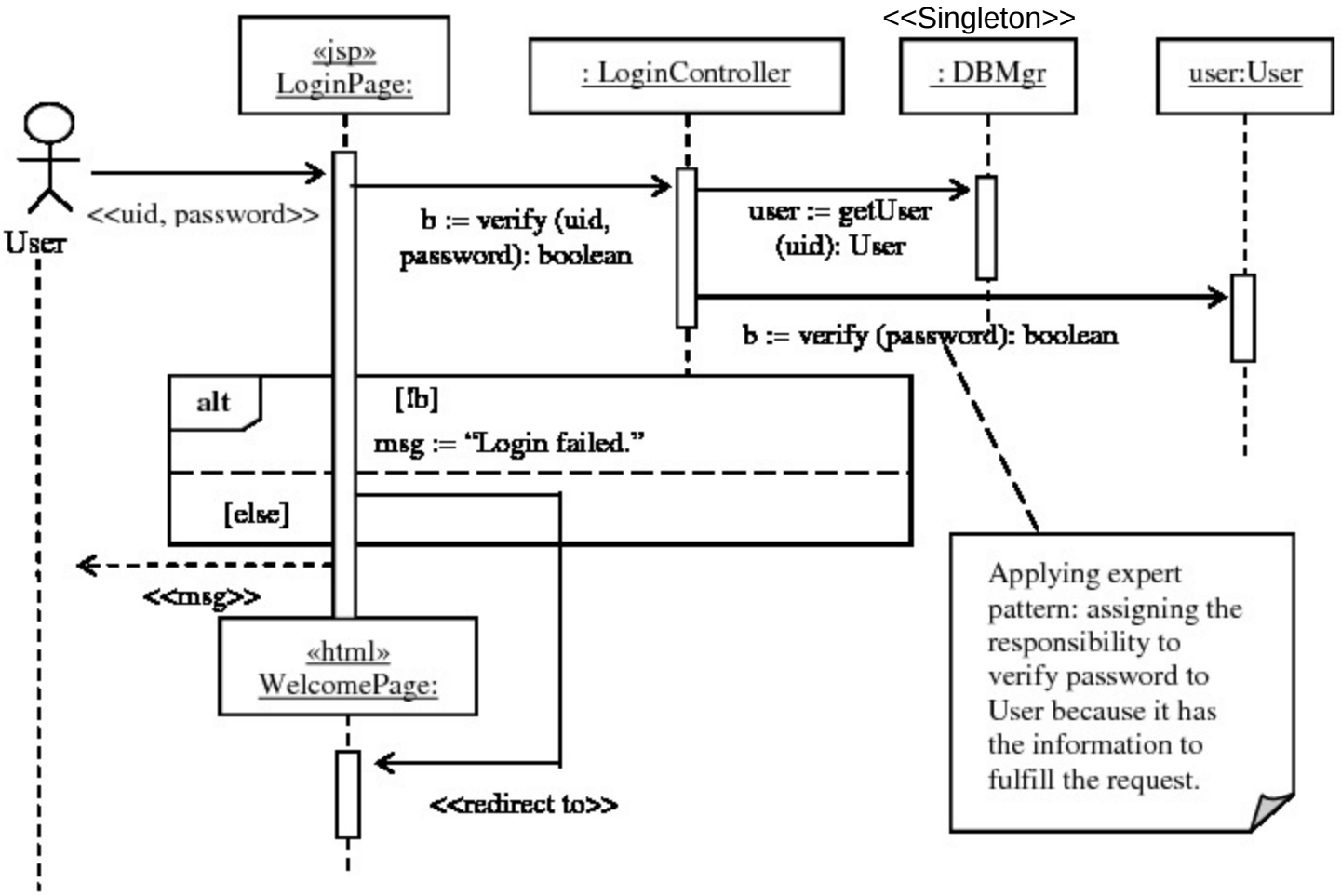
Specification of the Singleton Pattern

Name	**Singleton**
Type	**GoF/Creational**
Specification	
Problem	**How to design a class that has only a limited number of globally accessible instances?**
Solution	**Make the constructor of the class private and define a public static method to control the creation of the instances.**
Design	
Structural	
Behavioral	
Roles and Responsibilities	• **Subject: It provides a public static getInstance() method for the client to retrieve its instance.** • **Client: It calls the getInstance() method of Subject to retrieve the instance and calls its operation().**
Benefits	• **It limits the number of instances.** • **It supports global access to the instance(s).**
Liabilities	• **Concurrent update to the shared instance may cause unwanted effect.**
Guidelines	
Related Patterns	• **Singleton limits the number of instances of a class. Flyweight supports numerous occurrences of an object. Prototype reduces the number of classes.** • **Visitor is often a Singleton.**
Uses	**Singleton is used in many applications.**

Specification of the Controller Pattern

Name	Controller
Type	General responsibility assignment
Specification	
Problem	Who should be responsible for handling an actor request?
Solution	Assign the responsibility to handle the request to a dedicated class called the controller.
Design	
Structural	
Behavioral	
Roles and Responsibilities	• Business Objects: Object classes responsible for the business logic of an application. • Controller: A class dedicated to handle designated actor requests. It takes requests from the Representation and works with the Business Objects to fulfill the request. A use case controller is dedicated to handle all actor requests of a given use case. • Representation: An interface class responsible for interacting with actors of the system. It delegates the actor requests to the Controller and delivers the responses from the Controller to the actor.
Benefits	• It decouples the Representation and the Business Objects. • It reduces the change impact of Representation and Business Objects to one and other. • It supports multiple Representations.
Liabilities	A controller may be assigned too many responsibilities, resulting in a so-called bloated controller. A bloated controller is complex, difficult to understand, implement, test and maintain.
Guidelines	• Adopt use case controllers whenever possible. • Avoid using one controller for more than one use cases.
Related Patterns	Controller is a special case of the Model-View-Controller or MVC pattern.
Uses	In the design of all interactive systems to decouple the representation from business objects.

Applying the Expert Pattern (cont.)



Expert Pattern Trade-off (cont.)

- What are the trade-offs for implementing these checks in each?

Implement in:	Advantages	Disadvantages	Conclusion
Controller	Maintains the idea that all business logic is in the controller	The controller is now addressing logic that is in the business layer	Not a good idea - don't do this
DBMgr	Most efficient since the DB has all the information about the book being checked out	1) We have a lot of un-related logic in the DBMgr about various objects in the system. 2) DBMgr is prone to change everytime an object usage rule changes	For very simple verifications it might be acceptable to do this in the DBMgr (e.g. Login) if rules get more complicated move this into an object
Loan	1) All of the processing related to the object is contained within that class. It insulates the rest of the system from change. 2) DBMgr simply becomes a series of gets/sets of data in the DB.	Not as efficient as capturing this in the DB Mgr	For non-trivial processing its best to keep the logic internal to that object and use the Expert pattern.

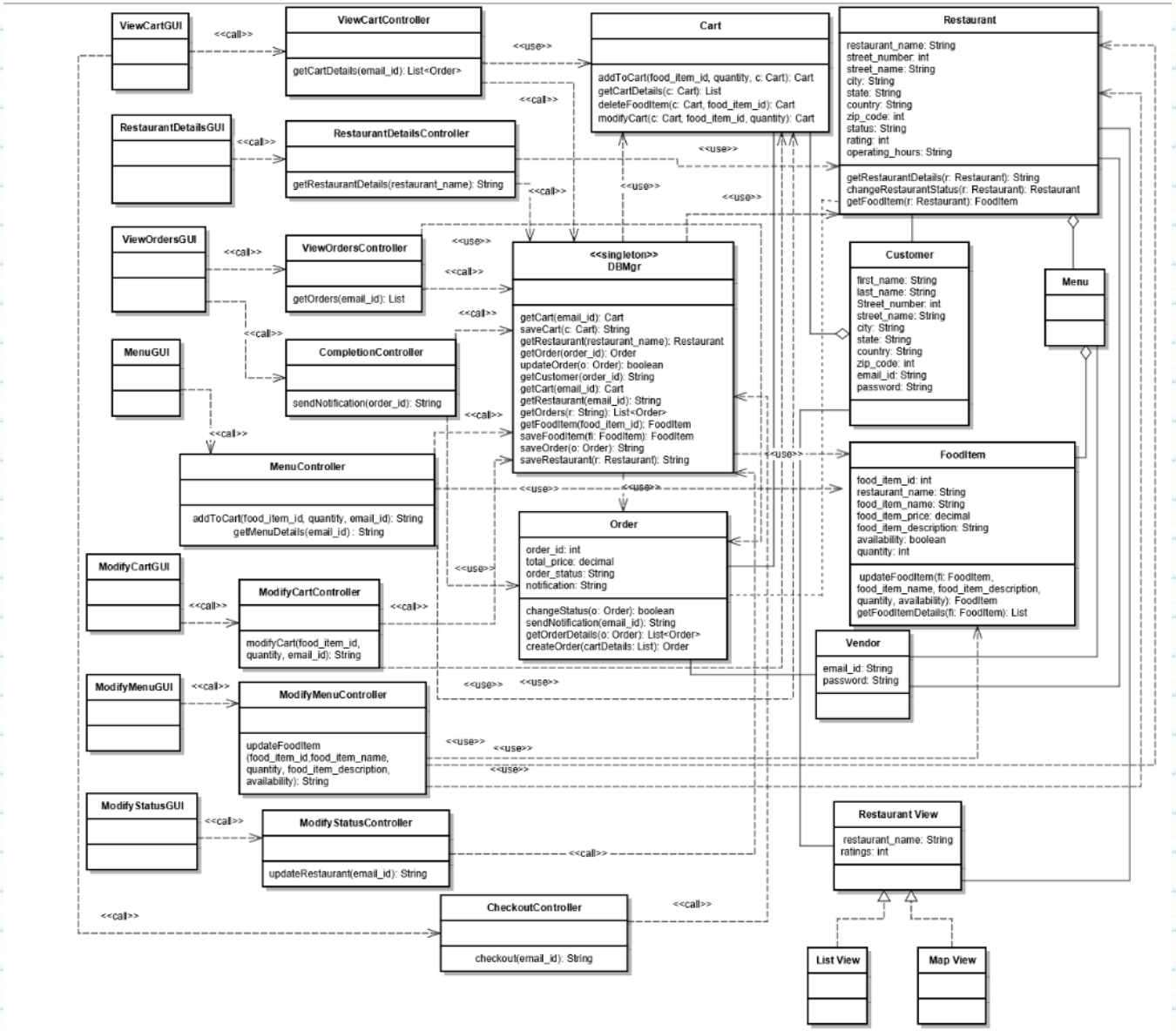
The Creator Pattern

- Object creation is a common activity in OO design – it is useful to have a general principle for assigning the responsibility.
- Assign class B the responsibility to create an object of class A if
  - B is an aggregate of A objects.
  - B contains A objects, for example, the dispenser contains vending items.
  - B records A objects, for example, the dispenser maintains a count for each vending item.
  - B closely uses A objects.
  - B has the information to create an A object.

Steps for Deriving DCD - Step 1

- Identify all classes used in each of the sequence diagrams and put them down in the DCD:
  1. Identify classes of objects that send or receive messages (the next slide shows 5 objects)
  2. Identify classes of objects that are passed as parameters or return types/values. Messages between objects represent function calls from one object to another (the next slide shows d:Document, and l:Loan are parameters, so these are identified)
  3. Identify classes that serve as return types. In the next slide Document is the only class used as a return type.
  4. Identify classes from the domain model. Classes from the domain model that are parent classes may be added to the DCD is it promotes understanding.

Student Project DCD



User Interface Design Process (cont.)

- Explanation of steps
  1. Identify major system displays.

     The major system displays, user input and user actions are identified from the expanded use cases produced in the current iteration. These form the basis for the design of the look and feel in the next two steps.
  2. Produce a draft design of the user interface.

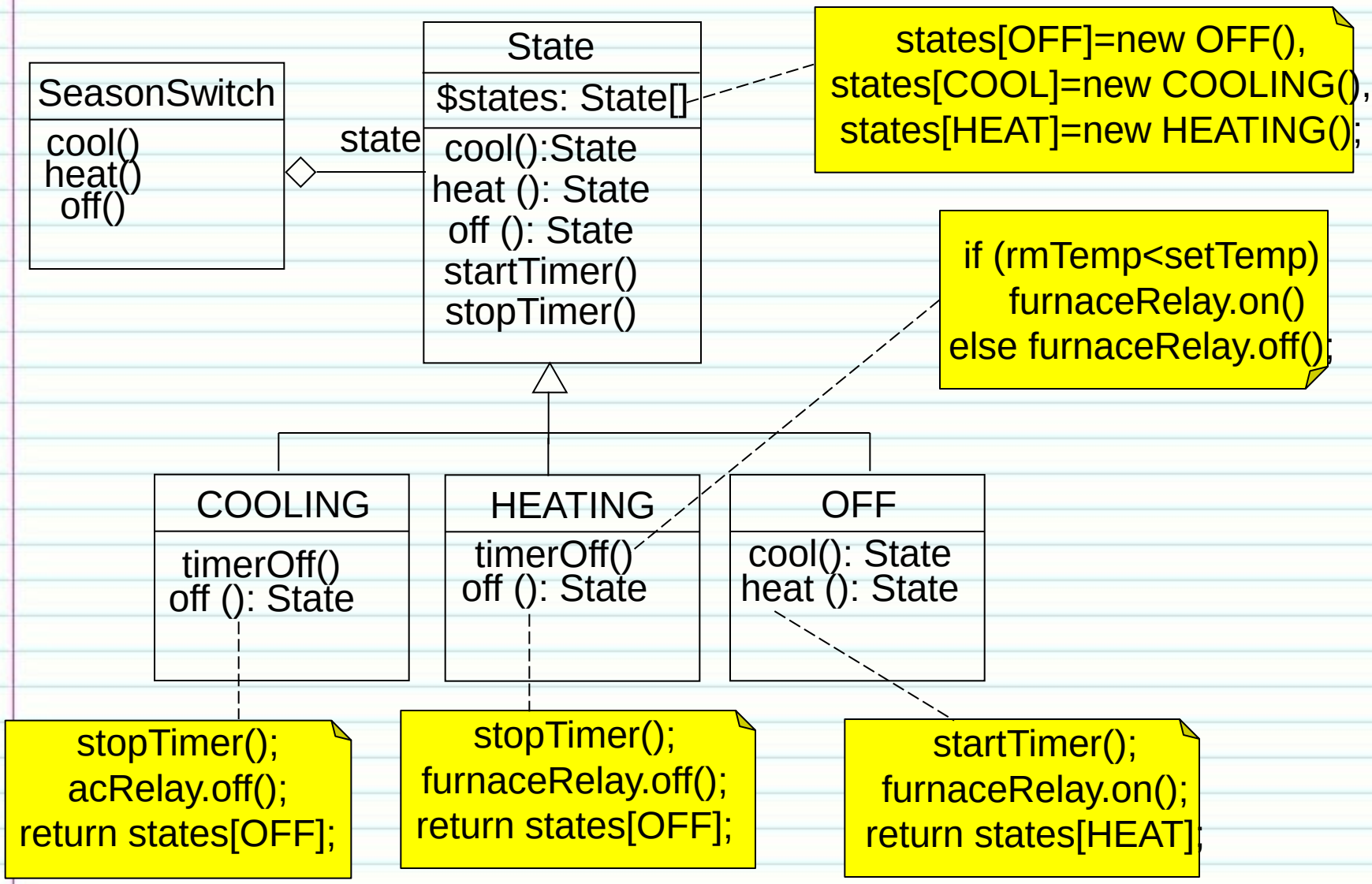
     Develop a draft layout of the user interface (including windows and dialog boxes) corresponding to the major systems displays (previous step) is produced. This step designs the “look” of the user interface.
  3. Specify the interaction behavior

     A state diagram is produced to specify the navigation relationships between the user interface items. This step develops the “feel” of the user interface.
  4. Construct a user interface prototype

     This step produces a user interface prototype to show the look and feel as designed in the previous two steps.
  5. Evaluate the design with users

     In this step, the user interface design, and possibly the prototype, is presented to a group of user representatives to solicit their feedback.

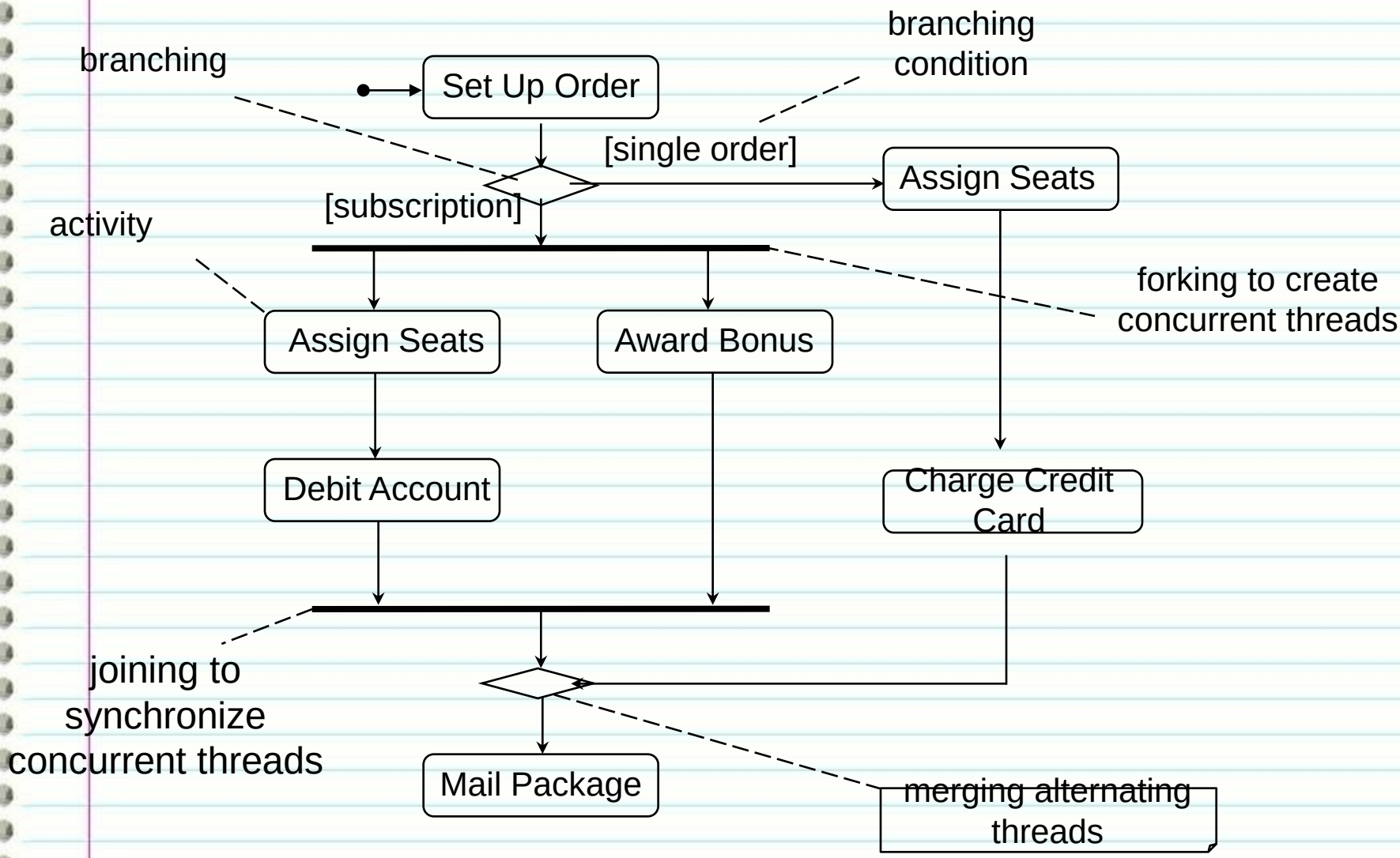
The Season Switch State Pattern



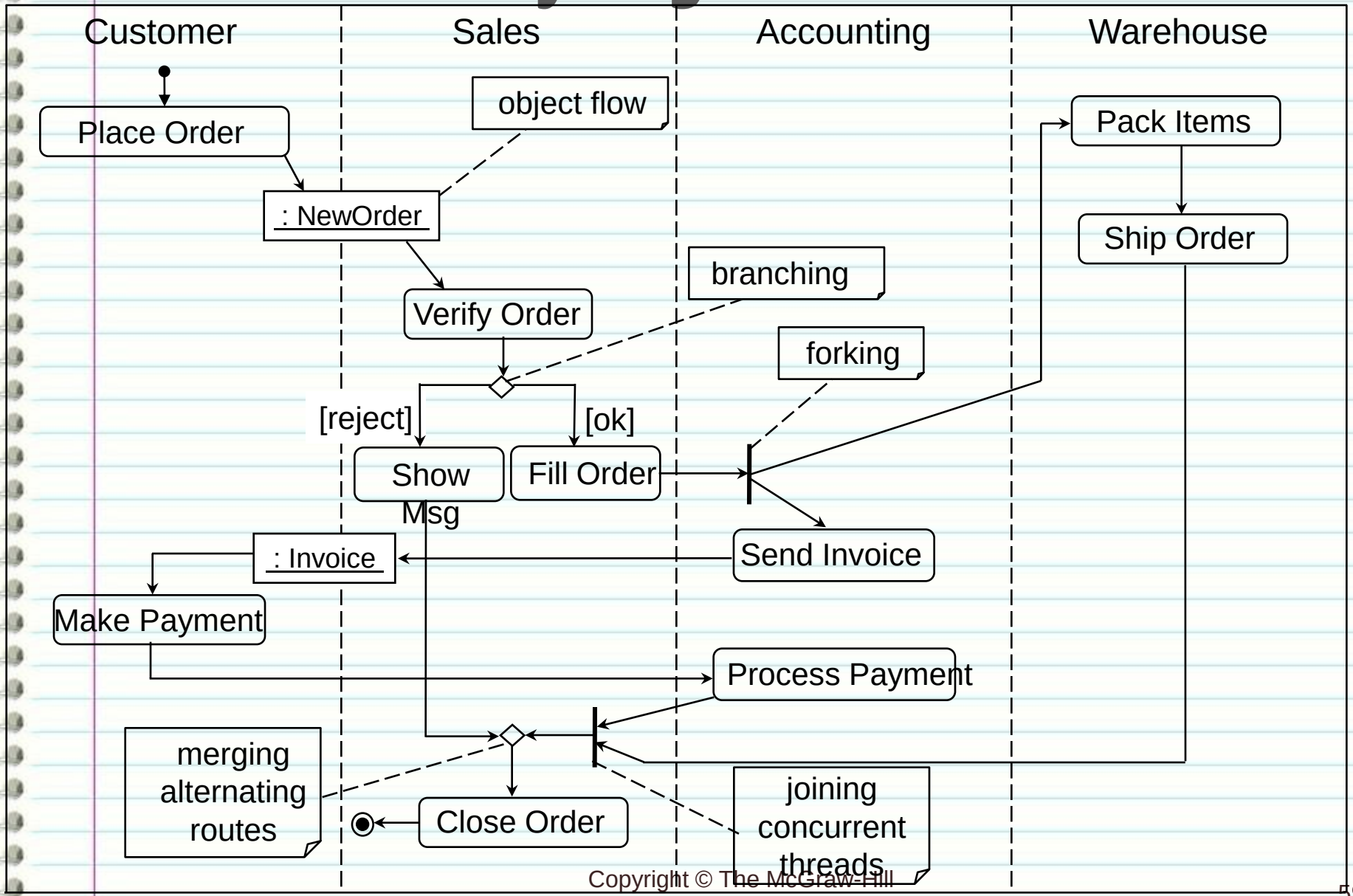
Guidelines for State Design Pattern

- Know the pattern - others use it so you need to be able to recognize it and converse about it
- Software Engineering is about making trade-offs
- Possible choices
  1. Switch,
  2. nested if,
  3. table,
  4. State pattern, or
  5. Controller approach
- Which is easier to change? Which is easier to understand? Which provides the best isolation?

Activity Diagrams



Activity Diagram: Swim Lane



The Agile Project

- The members in an Agile project communicate with each other early and frequently.

- The Agile Manifesto contains four statements of values:
  - Individuals and interactions *over* processes and tools
  - Working software *over* comprehensive documentation
  - Customer collaboration *over* contract negotiation
  - Responding to change *over* following a plan

- Individuals and Interactions
  - Agile development is very people-centered.
  - It is through continuous communication and interaction that teams work most effectively.

Agile Is A Whole Team Approach

- Whole-Team Approach
  - The whole-team approach means involving everyone with the knowledge and skills necessary to ensure project success.
  - The team includes representatives from the customer and other business stakeholders who determine product features.
  - The team should be relatively small; successful teams have been observed with as few as three people and as many as nine.
  - The whole- team approach is supported through the daily stand-up meetings involving all members of the team, where work progress is communicated and any impediments to progress are highlighted.
  - The whole-team approach promotes more effective and efficient team dynamics.

Scrum

- Scrum is an Agile management framework which contains the following constituent instruments and practices:
  - Sprint: Scrum divides a project into iterations (called sprints) of fixed length (usually two to four weeks).
  - Product Increment: Each sprint results in a potentially releasable/shippable product (called an increment).
  - Product Backlog: The product owner manages a prioritized list of planned product items. The product backlog evolves from sprint to sprint (called backlog refinement).
  - Sprint Backlog: At the start of each sprint, the Scrum team selects a set of highest priority items (called the sprint backlog) from the product backlog.
  - Definition of Done: To make sure that there is a potentially releasable product at each sprint's end, the Scrum team discusses and defines appropriate criteria for sprint completion.

Coding Standards

- Define the required and optional items.
- Define the format and language.
- Define the coding requirements and conventions.
- Define the rules and responsibilities to create and implement the coding standards, as well as review and improve the practice.

Key Takeaway Points

- Software quality assurance encompasses a set of activities to ensure that the software under development or modification will meet functional and quality requirements.
- Software quality assurance activities are life-cycle activities.
- The software engineer is the essential ingredient in a quality product - it is not another department's job to build in the quality you didn't do the first time around

Definition of Terms

- Definitions are from the IEEE 24765 and the ISTQB definitions when they agree no color is used - These terms are used throughout the software literature, green are my terms we will use in class
- Bug: see Defect
- Defect: see Fault see Fault
- Error: "A human action that produces an incorrect result." This a mistake that a human makes
- Fault: "An incorrect step, process, or data definition in a computer program" "A flaw in a component or system that can cause the component or system to fail to perform its required function"
- Failure: "Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits." "Deviation of a component or system from its expected delivery, service, or result"
- Mistake: see Error
- Deficiency: A discovered defect in delivered software
- Latent Defect: An undiscovered defect in delivered software
- Test Oracle: "A source to determine expected results to compare with the actual result of the software under test."

Verification/Validation and the Software Life Cycle

- The following are definitions from the IEEE SWEBOK V3 section 10.2.2
- Verification : "… ensure[s] that the product is built correctly." Is the product built to its requirements?
- Validation : "… ensure[s] that the right product is built—that is, the product fulfills its specific intended purpose." Are the requirements correct?
- Which of these activities detect defects?
- Common Software Development Life Cycles (SDLCs) include the v-model, prototyping, iterative and incremental development, spiral development, rapid application development, extreme programming and agile.
- In terms of V&V(which is a defect detection activity) most of what we care about from these SDLCs are what activities detect defects and when they occur

Verification/Validation Activities and the SDLC

Note: this does not imply a specific SDLC

Activity	Verification	Validation
Software Requirements	•Requirements Technical Reviews •Requirements Based Testing	Customer Review, Expert Review, Modeling, Prototyping
High Level Design	•High Level Design Technical Reviews •Integration Level Testing	Modeling, Prototyping
Detailed Design	•Detailed Design Technical Reviews •Integration Level Testing	Modeling, Prototyping
Coding	•Code Technical Reviews •Unit Level Testing	Modeling, Prototyping

•Technical Reviews (SWEBOK) can be Formal Inspections, Walkthroughs, or Peer Reviews

•For high maturity teams, the Technical Review activity can detect up to 90 percent of the software defects!

A Few More Introductory Terms

- Quality: “The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.”
- Defect prevention: “A structured problem-solving methodology to identify, analyze and prevent the occurrence of defects.” Defect prevention changes the process to prevent occurrence.
- How does this compare with defect detection and removal?
- What is quality control vs. quality assurance (defect prevention)?
- Debugging: “The process of finding, analyzing and removing the causes of failures in software.”
- How does debugging differ from defect detection? From troubleshooting?

Seven Testing Principles

1) Testing shows presence of defects: Testing can show the defects are present, but cannot prove that there are no defects.
2) Exhaustive testing is impossible.
3) Early testing: In the software development life cycle testing activities should start as early as possible to reduce cost.
4) Defect clustering: A small number of modules contains most of the defects discovered. Sometimes referred to as the 80/20 rule.
5) Pesticide paradox: If the same kinds of tests are repeated again and again, eventually the same set of test cases will no longer be able to find any new bugs. Does not apply to regression testing.
6) Testing is context depending: Different software products have varying requirements, functions and purposes. For example, safety – critical software is tested differently from an e-commerce site.
7) Absence-of-errors fallacy: Declaring that a test has unearthed no errors is not the same as declaring the software “error-free”.

Black Box Testing Techniques

- Typical “black-box” test analysis and design techniques include:
  - Equivalence partitioning
  - Boundary value analysis
  - Decision table testing
  - State transition analysis
  - Use Case testing
  - Decision logic and Karnaugh maps

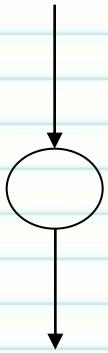
- Black box is a bit of a misnomer – we don’t and can’t test strictly black box

- So these are correctly called Specification based test analysis and design techniques, but I use the term “black box” because it is so commonly used

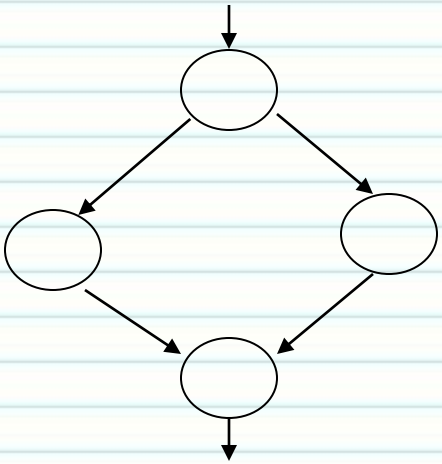
White-Box Testing Techniques

- Basis Path Testing
  - Flow Graph Notation
  - Cyclomatic Complexity
  - Deriving Test Cases
- Condition Testing
- Data Flow Testing
- Loop Testing
- Symbolic Execution

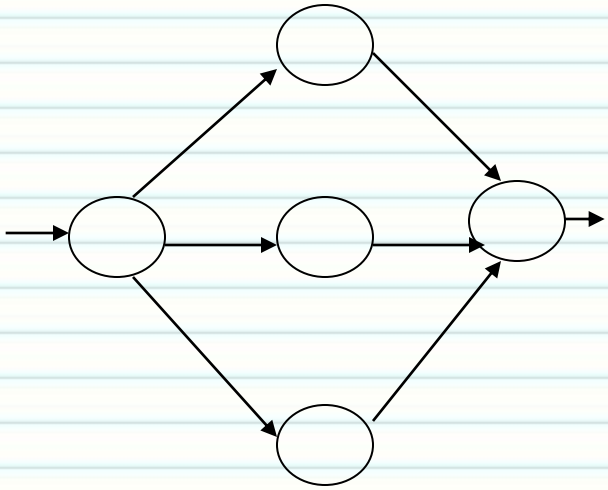
Flow Graph Notations



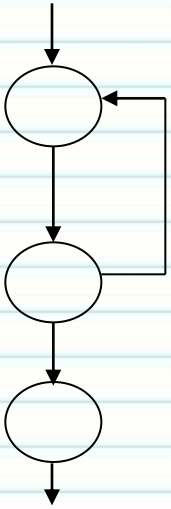
sequential



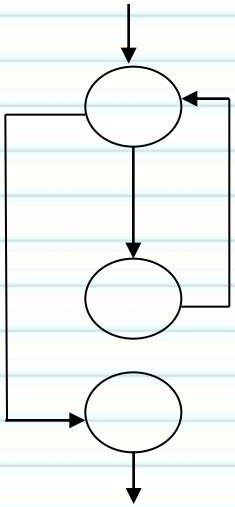
if-then-else



case

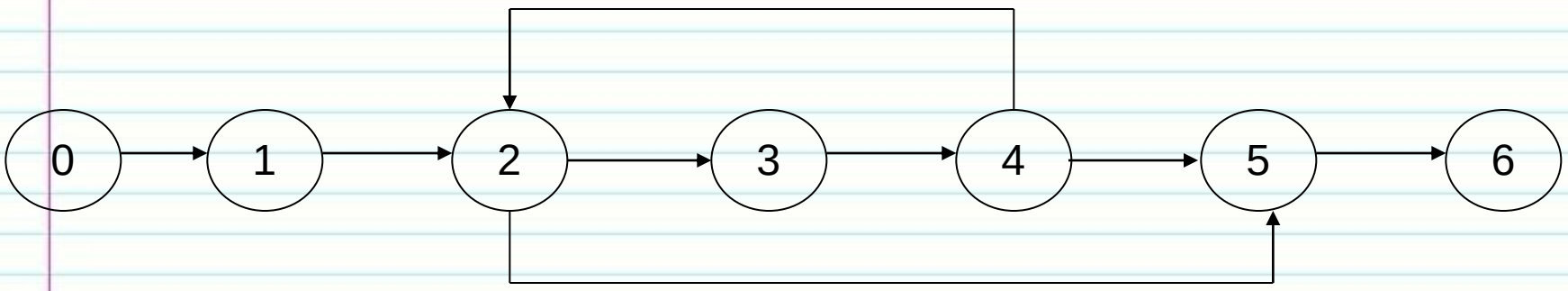


until

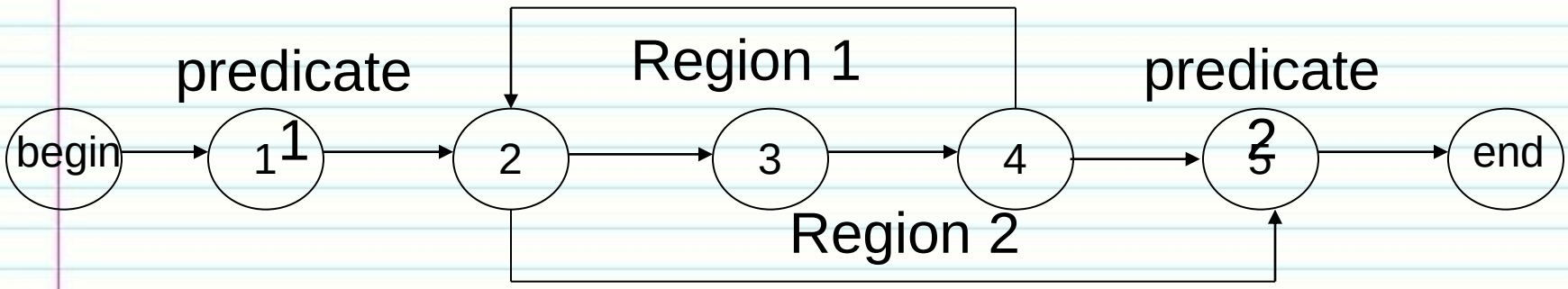


while

An Example Flow Graph



Cyclomatic Complexity



Three ways to compute cyclomatic complexity:

- Number of closed regions plus 1
- Number of atomic binary predicate + 1
- Number of edges - Number of Nodes + 2

The cyclomatic complexity is 2+1=3.