



Copyright Notice

All course materials (including and not limited to lecture slides, handouts, recordings, assessments and assignments), are solely for your own educational purposes at NTU only. All course materials are protected by copyright, trademarks or other rights.

All rights, title and interest in the course materials are owned by, licensed to or controlled by the University, unless otherwise expressly stated. The course materials shall not be uploaded, reproduced, distributed, republished or transmitted in any form or by any means, in whole or in part, without written approval from the University.

You are also not allowed to take any photograph, video recording, audio recording or other means of capturing images and/or voice of any of the course materials (including and not limited to lectures, tutorials, seminars and workshops) and reproduce, distribute and/or transmit in any form or by any means, in whole or in part, without written permission from the University.

Appropriate action(s) will be taken against you (including and not limited to disciplinary proceedings and/or legal action) if you are found to have committed any of the above or infringed copyright.

1 August 2022

Software Engineering

sc2006/ce2006/cz2006

Introduction

Outline of Lecture #1

- 1. What is Software Engineering?**
- 2. Software Engineering Activities**
- 3. Course Logistics**

1. What is Software Engineering?

Software Engineering is:

The production of maintainable, fault-free software that meets the user's requirements and is delivered on time and within budget.

2. Software Engineering Activities

Is Software Engineering **just** Coding?

What are the possible activities in engineering a software product?

What were the engineering activities needed to achieve this?



Analogy with Construction Engineering



What were the engineering activities needed to achieve this?

Deciding the purpose of the building.

Deciding how many room, how big are the rooms, what the building will look like.....

Designing the building.
Deciding on the materials for construction, the power, plumbing and other services....

Project planning. Scheduling.
Teamwork.

Simulations and testing.

Construction.

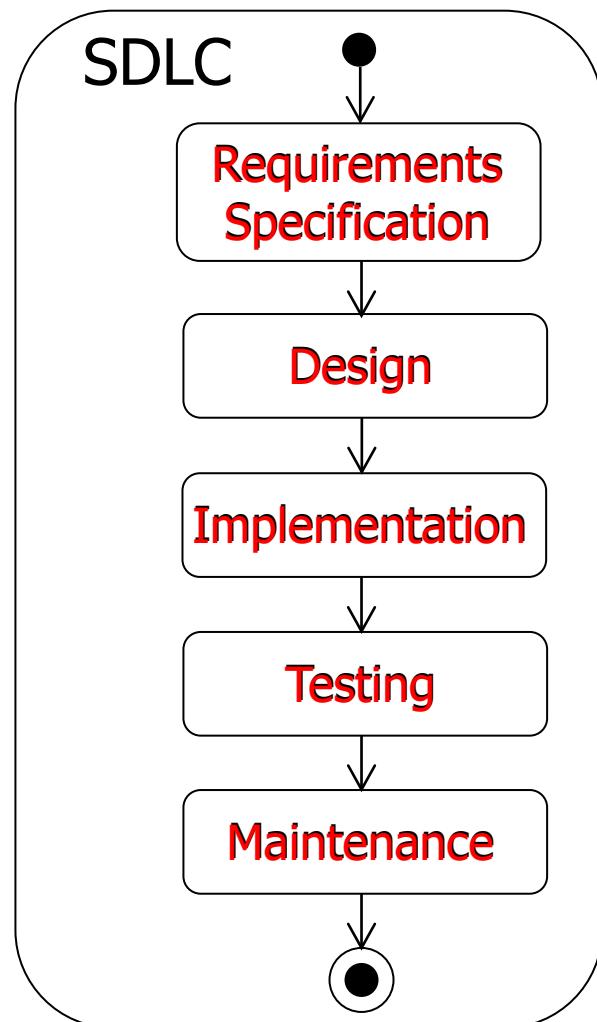
Operation and Maintenance.

All engineering requires the same activities

- 1. **Requirements elicitation** – specify what are we going to make, what it will do and how it will do it.
- 2. **Design it** – What will it look like and how will we make it?
- 3. **Build it** – to the design and requirements specification.
- 4. **Test it** – to ensure it meets the requirements and is error free.
- 5. **Release it and maintain it.**

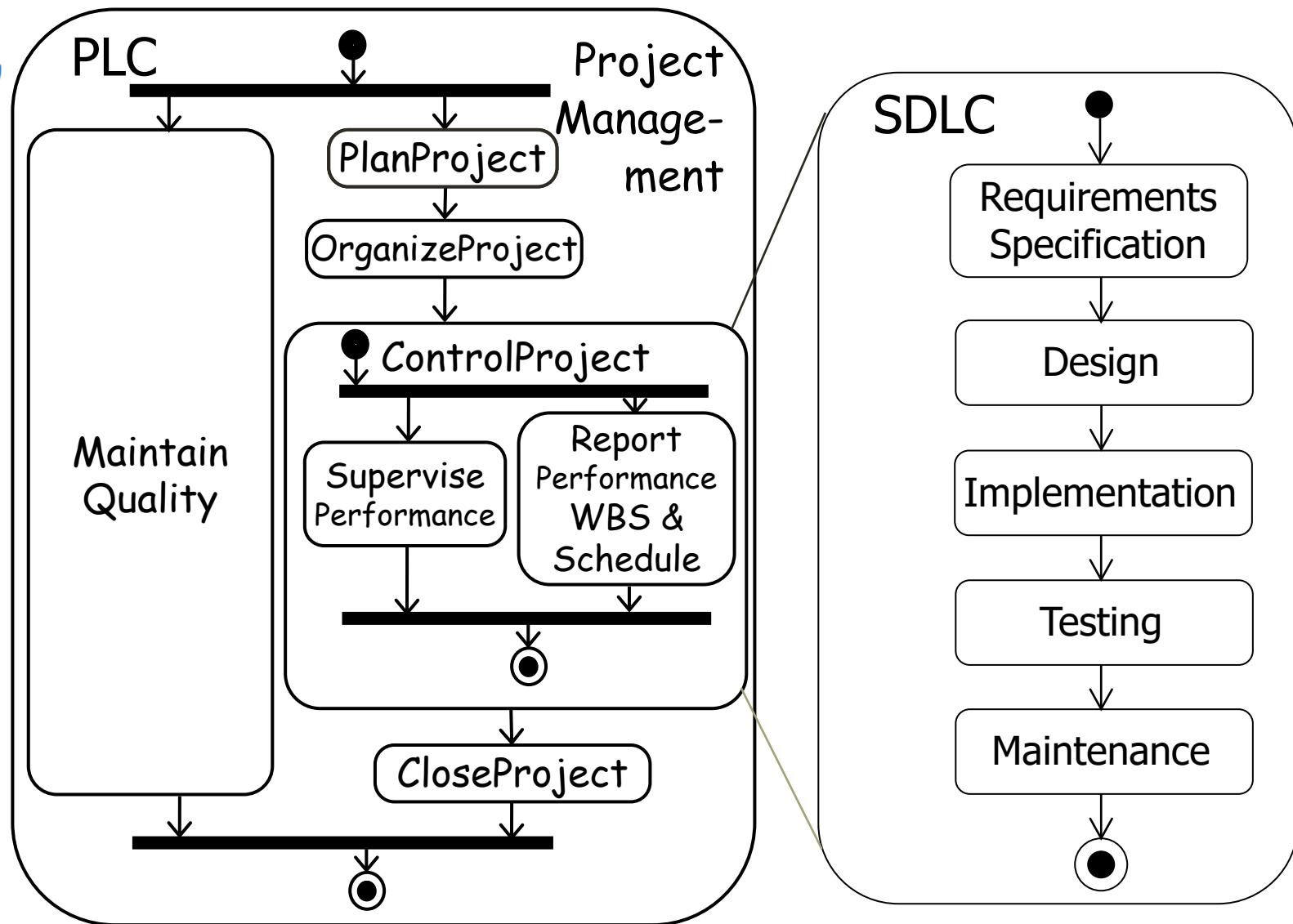


Software Engineering Activities



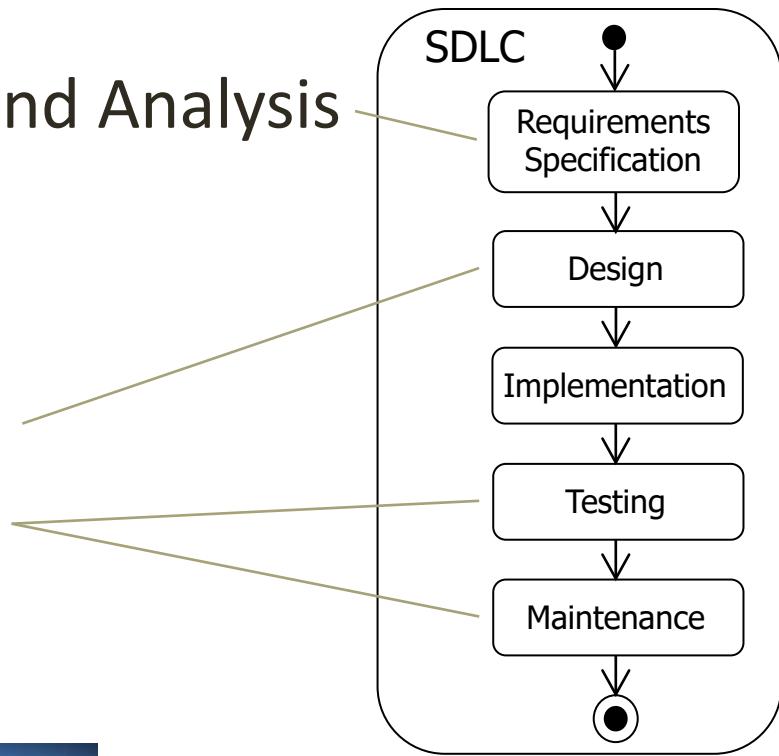
Work Products	Milestones
<ul style="list-style-type: none">• Software Requirement Specification• Prototype	<ul style="list-style-type: none">• Requirements Review
<ul style="list-style-type: none">• Software Design Document• Interface Design Document• Test Cases• Models	<ul style="list-style-type: none">• Preliminary & Final Design Reviews
<ul style="list-style-type: none">• Source Code• System Build• Technical Documentation• User Manual• Test Report	<ul style="list-style-type: none">• User Acceptance Test• General Availability Release
<ul style="list-style-type: none">• Change Request• Bug Report	<ul style="list-style-type: none">• Patch Release• Upgrade

Software Engineering Activities



Main Topics to be covered

- Requirements Elicitation and Analysis
- Software Process Models
- Agile Software Process
- System and Object Design
- Testing and Maintenance



Lectures for large and medium projects

Practices in lab project

3. Course Logistics and Mechanics

- 20+ lectures
 - 9 tutorials
 - 1 group project (50%)
- With 5 x 2 hours lab sessions
- Exam (50%)



Where learning happens!

We want **Active Learning**: Do more than just listening!

Tutorials

start in Week3 !

- Problem-based learning through Grain Elevator System case study and other questions/problems
 - All tutorials are released on NTULearn during week 1.
- You Should
 - Come to each tutorial prepared to present and discuss
 - Do NOT wait for the tutors answers (most questions do not have model answers as there is often not a single correct answer!)

Group Project (50%)

start in Weeks
2 and 3 !

- The project problem and five lab manuals is available on NTULearn during week 1
- You Must
 - Work in a team (4-6 members)
 - Plan your work, start early, proceed iteratively
 - Contribute to all key Software Engineering activities
 - Meet deliverables for labs #1, #2, #3 (**5% for each lab**)
 - Demonstrate your working product in Lab#5 (**20%**)
 - Deliver documentation at end of the semester (**15%**)

Course Reference Books

- Ian Sommerville, **Software Engineering, 10th Edition, ISBN 10: 1-292-09613-6.**
- Bernd Breugge, Allen H Dutoit, *Object-Oriented Software Engineering: using UML, patterns and Java, 3rd Edition, ISBN 10: 0-13-815221-7*, Pearson, 2010.
- Martin Fowler, **UML Distilled: a brief guide to the standard object modelling language, Third Edition, ISBN 0-321-19368-7, Pearson, 2004.**
- Christopher Fox, *Introduction to Software Engineering Design: Process, Principles, and Patterns with UML2*, 1st edition, Pearson Education/Addison-Wesley, 2006.
- Lee Copeland, *A Practitioner's Guide to Software Test Design*, Artech House, 2004.



Lazada

10% OFF

+

FREE SHIPPING

on PRINT BOOK

LIMITED
UNIT

**15% OFF
EBOOKS**

for 12 month
subscription access



<https://bit.ly/NTUPRIN>

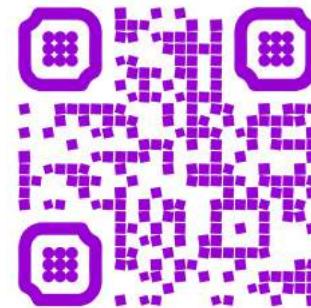


<https://bit.ly/NTUEBOOK>

SCAN to collect coupons!

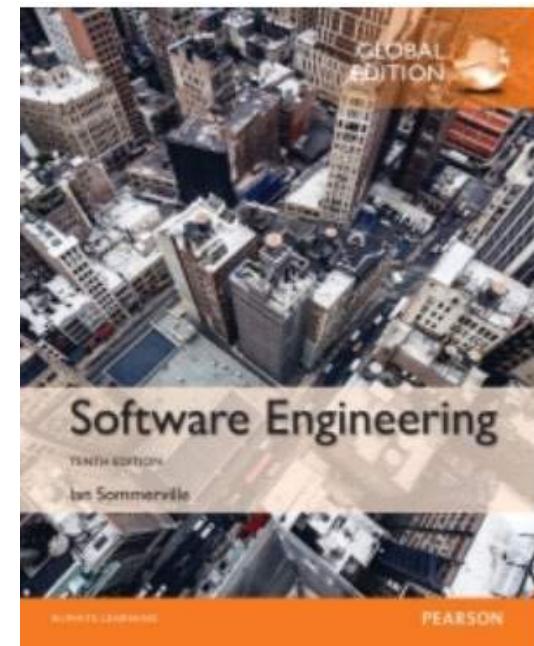


Also available on



ARE YOU A STUDENT?

Log In With Your Student Email
And Enjoy Student Discounts!



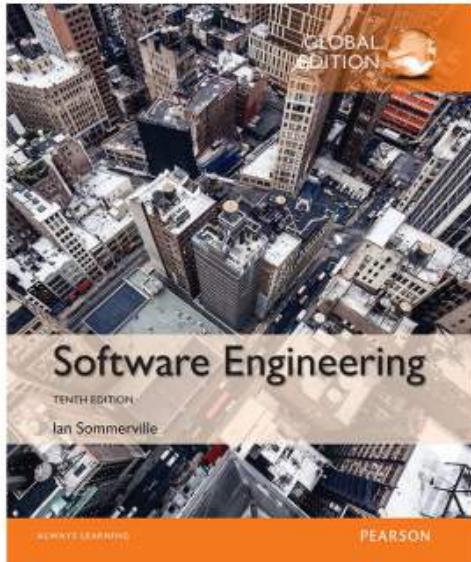
**Software
Engineering 10e by
Sommerville**
ISBN: 9781292096131
eBook ISBN: 9781292096148



CZ2006 Software Engineering



Software Engineering, Global Edition, 10e



FREE
shipping
throughout
August

Get 7% OFF
on your Pearson textbooks on Lazada!

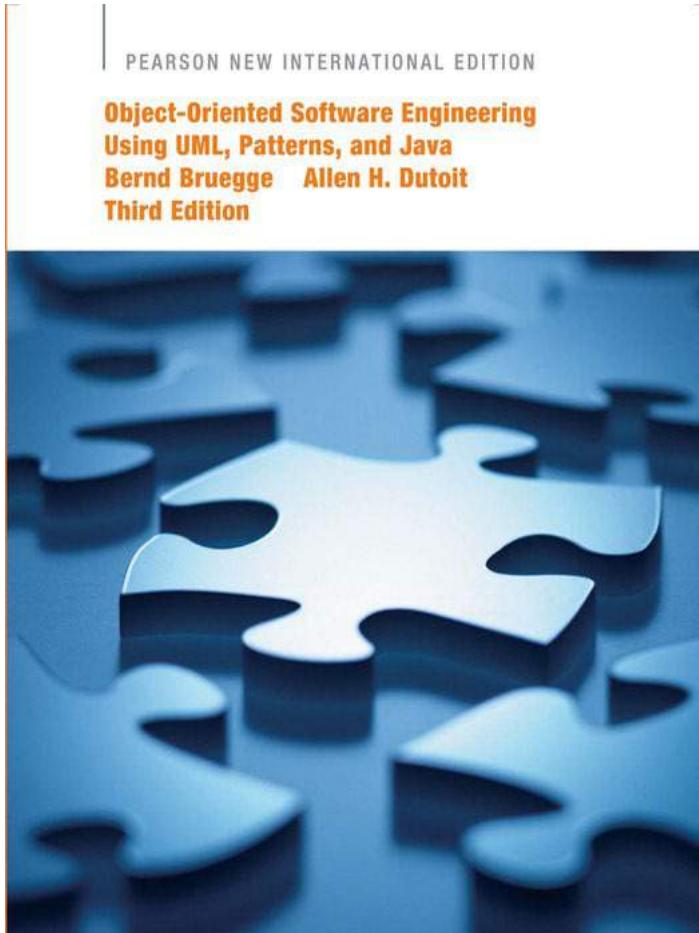


<https://bit.ly/3wVkBd>

Author	: Sommerville
Print ISBN	: 9781292096131
eBook ISBN	: 9781292096148

Also available at **Booklink NTU @ North Spine** <https://www.blinks.com.sg/>

CZ2006 Software Engineering



Object-Oriented Software Engineering
Using UML, Patterns, and Java
Bernd Bruegge Allen H. Dutoit
Third Edition

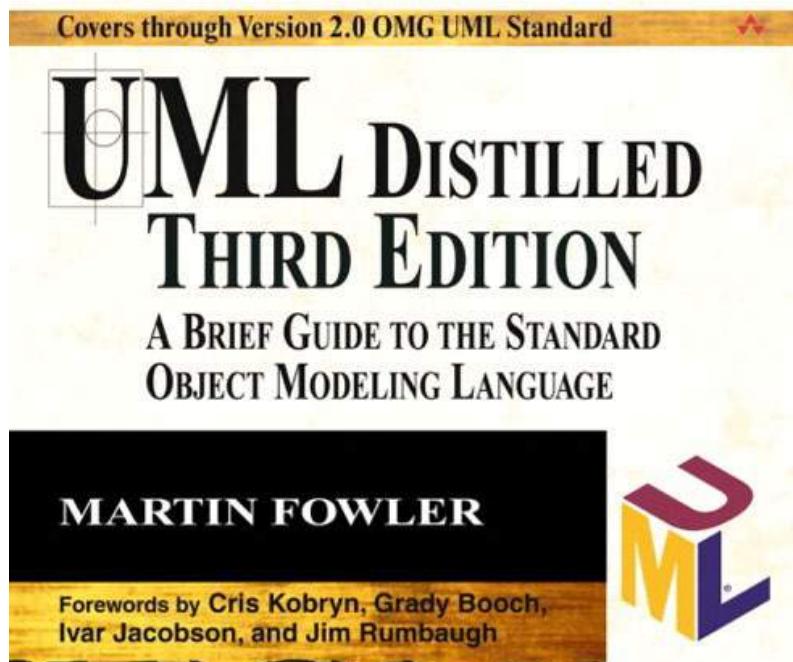
PEARSON NEW INTERNATIONAL EDITION

Author : Bruegge
Publisher : Pearson
ISBN : 9781292024011

Get it now at Book link NTU



CZ2006 Software Engineering



UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3/E

Author : Fowler

Publisher : Pearson

ISBN : 9780321193681

Get it now at Book link NTU



Software Engineering

CE2006/CZ2006

Requirements Elicitation 1 of 3

all engineering requires some activities :

- Requirements elicitation
- Design it
- Built it
- Test it
- Release it n maintain it

{ specify what we going to make
what it will do
how it will do it }

→ Steps :

- 1 Identifying Stakeholder
- 2 Requirement gathering
- 3 Listing out Requirement
- 4 Organizing workshop

Discussion Topics

- Requirements Elicitation
 - Functional & Non-Functional Requirements
 - Data Dictionary
- Requirements Validation
 - Prototype

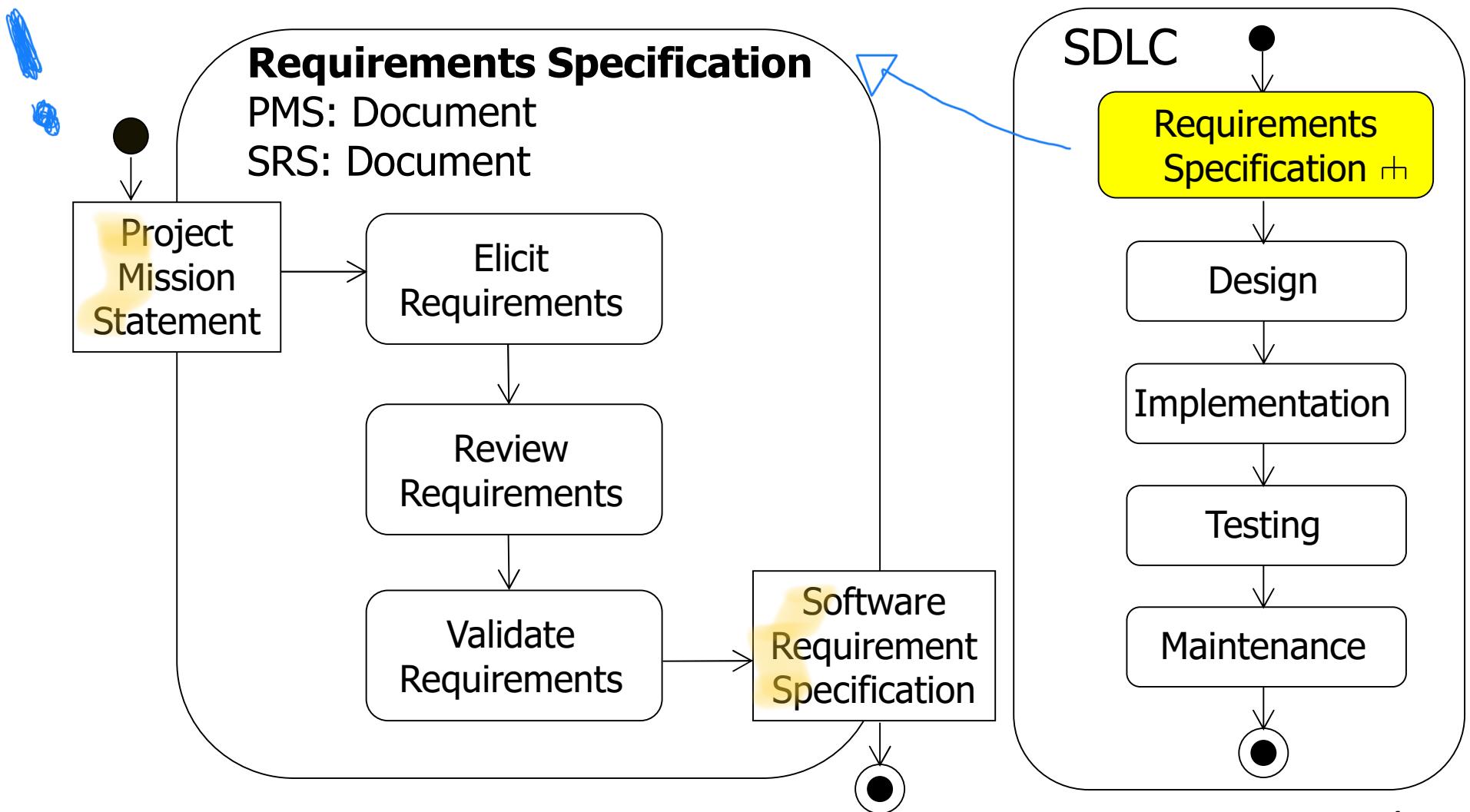
Reading

- Bruegge Chapter 4 sections 4.1-4.3

Correct Requirements Elicitation and Specification are Necessary for a Successful Project



Requirements Specification Activities



Project Mission Statement

A simple short statement of what you intend to accomplish in your project.

Companies and organisations have mission statements:

Google – to organize the world's information and make it universally accessible and useful.

BMW - the BMW Group is the world's leading provider of premium products and premium services for individual mobility.

Apple – Apple designs Macs, the best personal computers in the world, along with OS X, iLife, iWork and professional software. Apple leads the digital music revolution with its iPods and iTunes online store. Apple has reinvented the mobile phone with its revolutionary iPhone and App store, and is defining the future of mobile media and computing devices with iPad.

Project Mission Statement

A project should have a project mission statement that describes the project in **two or three** sentences.



Typically these sentences define

1. The **problem**: what will be done (scope and limits)
2. The **stakeholders, developers and users**
3. The **outcomes and benefits** of the project

Example:

The GoFast team will develop a website that enables airline travellers to rate their travel experiences. This project will be considered complete when the website has been tested and approved for release by the FactFinding Organisation. This project supports the International Travel Watchdogs objective to ensure air passengers can openly compare airlines.

Requirements Elicitation



Eliciting stakeholder needs and desires through:

- Interview
- Observation
- Workshop
- Legacy Product Study
- Competitive Product Study
- Prototype

Learn problem domain
Study user tasks



Types of Requirements

- **Functional requirements** describe **interactions** between the system and the environment, to **map program inputs to program outputs**. Basically the things that the system must do.
- **Non-Functional requirements** describe the **properties** the system must have, that is not directly related to the functional behaviour of the system.

Functional requirement

- help understand function of the system
- explain characteristic that system expected to have
- Identify what system must/must not do
- allow system to perform even if non functional requirements are not met
- System meet user requirement
- essential to system operation
- Straightforward to define n agree on
- define by user
- can be documented n understood through use case

Non functional requirement

- help understand performance of system
- Explain the way product should behave
- Identify how system should do it
- will not work with only non-functional requirement
- Product meets user expectation
- desirable but not always essential
- harder to define n agree on
- define by software engineers, developer, etc.
- can be documented n understood as a quality attribute

Examples of Functional Requirements

System functionality to be performed

e.g., The library member must be able to search the library catalog.

e.g., The bank customer must be able to withdraw cash from the ATM.

Information to be processed

e.g. The system must display the current time in 24 hour format.

e.g. The system must display the temperature in degrees centigrade in the range -10C to +130C to one decimal place of accuracy.

Interface with other systems

e.g. The system must be able to use wifi to communicate all transactions with a clients secure database.

e,g. The system must be able to control up to six robot arms simultaneously.

Examples of Non-Functional Requirements

Usability	Help messages must be displayed in the local language according to the user's locale.
Reliability	After a system reboot, the full system functionality must be restored within 5 minutes.
Performance	When a book is placed in the checkout pad, the system must detect it within 2 seconds.
Supportability	The database must be replaceable with any commercial product supporting standard SQL queries.

! Documenting the Requirements

- Use good technical writing style
 - Write complete, simple, precise, unambiguous sentences using active voice
 - Define terms clearly and use them consistently
 - Use clear layout and formatting (e.g., organizing the requirements in a hierarchy)
- State requirements in an *atomic* manner, such that the specification is *verifiable* and *traceable* and *unambiguous*.

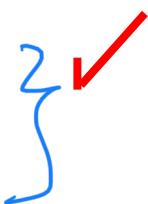
Atomic Requirements

When a computer is added, the tracking system requires the user to specify its type and allow the user to provide a description. Both these fields must be text of length >0 and <512 characters.



1.1 When a computer is added to the tracking system, the user must enter the computer type.

1.1.1 The computer type data must be text of at least one character and less than 512 characters.



1.2 When a computer is added to the tracking system, the user must enter a description of the computer.



1.2.1 The description of the computer must be text of at least one character and less than 512 characters.

Requirements Verifiability

Verifiable - sufficiently specific to be testable

The user interface must be user-friendly. 

80% of first-time users must be able to enter a simple search query within 2 minutes of starting to use the system. 

The system must control more than one drill press. 

The system must control up to seven drill presses concurrently. 

Requirements Traceability

- The ability to track requirements from their expression in an SRS (software requirements specification) to their realization in design, documentation, source code and their verification in reviews and tests
- The ability to track dependencies among requirements, system functions and system components

Requirements Validation

Stakeholders – the system specified meets their needs and desires (i.e. correct) and requirements are prioritized

Development Team – requirements (and underlying assumptions) are properly understood

- Review
 - Walkthrough, inspection, critical review
 - Checklist for completeness, consistency, unambiguity, correctness
- Prototype

User Interface Prototype

A picture is worth a thousand words.

Sketches or a model of what a system will look like brings the requirements to life for all stakeholders.

Can be implemented using

- Story Boarding – sequence of graphics showing different views of the interface in a specific interaction.



- Still images created on a computer.



- Interactive prototype that illustrates some simulated dialogue



The Data Dictionary

The problem domain glossary

Collection of names, definitions, and attributes about data elements that are being used Captured in a database

- Ensures consistent unambiguous terminology that all stakeholders can agree on

E.g. within a university/college define the terms

- Programme of Study
 - Course
 - Degree Programme
-
- Ensures specialised terms are defined E.g.
 - Lecture
 - Tutorial
 - Class
 - Laboratory
 - Seminar

Example of Data Dictionary

Term	Definition
Program of Study	A university program that a student enrolls into. There are three levels of programs: undergraduate, master, and PhD.
Semester	There are four semesters in an academic year. Semester 1 and Semester 4 have 13 teaching weeks. Semester 2 and Semester 3 have 5 teaching weeks.,
Course	A course is a basic unit of teaching. A course must be either compulsory or elective. A course must include lectures and tutorials . Some courses may have labs.
Lecture	The traditional form of class that are delivered in lecture theaters. Each lecture will be video recorded and published in the learning management system
Tutorial	The interactive, small-size classes that are conducted in technology-enhanced tutorial rooms. Tutorials will not be video recorded.
....	

Focus on **problem domain** terms,
not implementation terms.

woodClap ques

↳ What is the 1st step of requirement elicitation?
Identifying Stakeholder ✓

Software Engineering

CE2006/cz2006

Requirements Elicitation 2 of 3

Discussion Topics

- Use Case modeling
- Use Case diagram
- How to develop Use Cases from functional requirements
- Reading
 - Bruegge Chap 4.4
 - Fowler Chap 9

UML: Unified Modeling Language

- Created by Booch, Jacobson & Rumbaugh in 1996
- Version 1.1 was adopted by the Object Management Group (OMG) in 1997. The latest version is 2.5 (August 2015).



- What is it?

It is a **graphical notation with textual annotation** for specifying, documenting and communicating various aspects of the **structure, functionality and dynamic behaviour** of complex software systems.

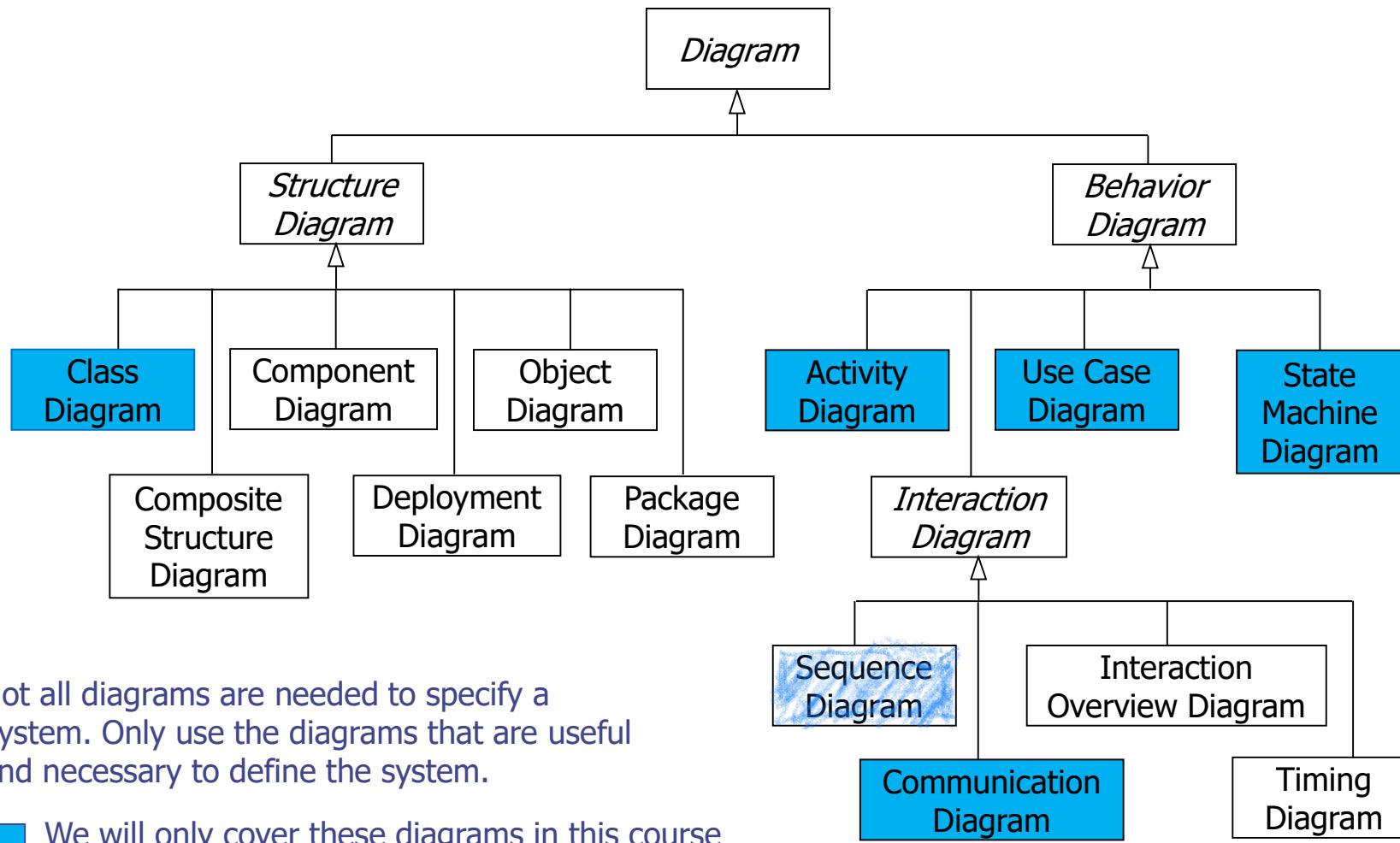
It is **NOT** a programming language (although its supporters are working towards that goal).

It is **NOT** the answer to all your modelling and design problems.

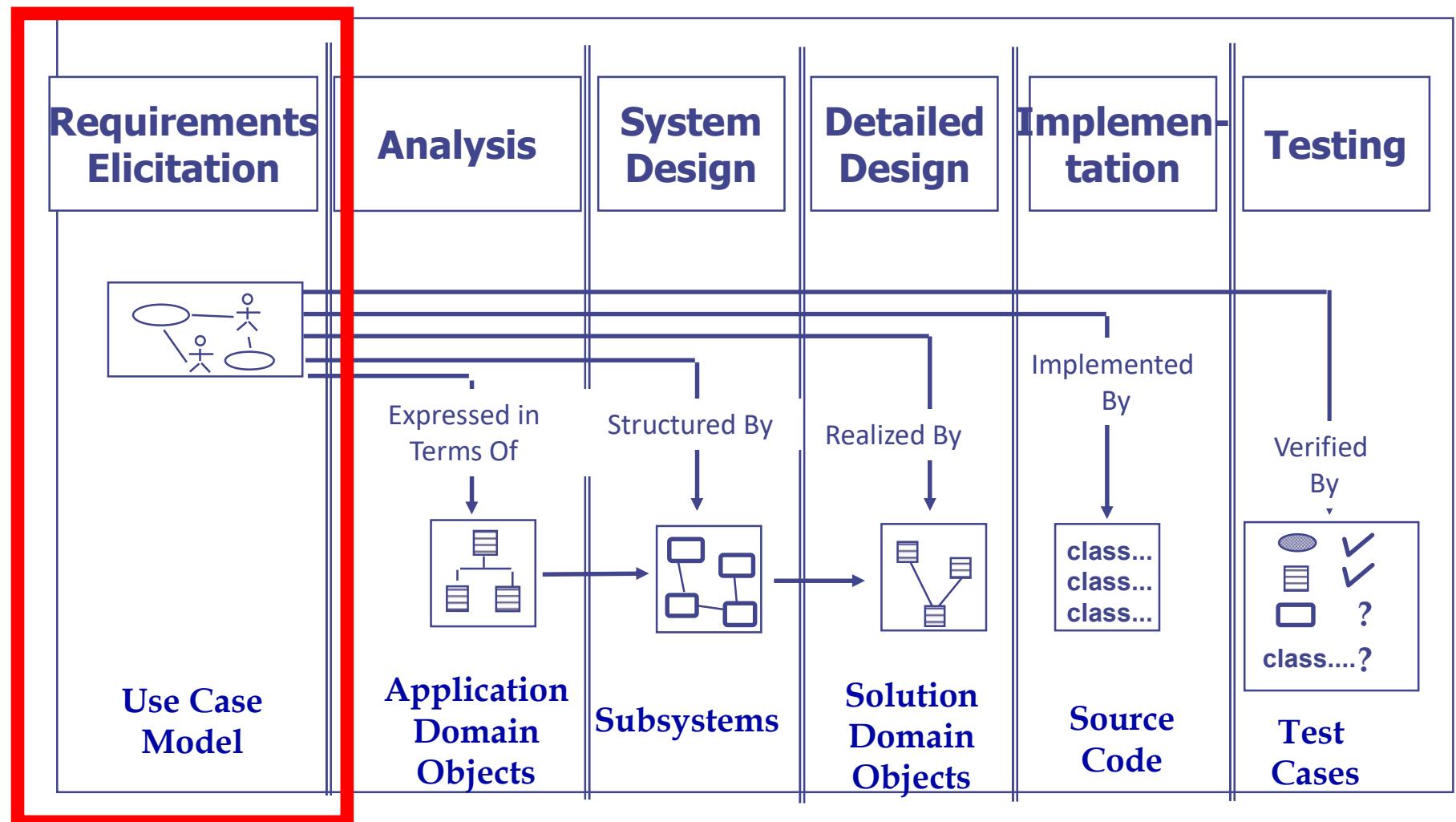
UML is an Industry Standard tool and therefore Provides Standardization!

In the laboratory you can use Visual Paradigm to draw UML diagrams for your project work.

UML Diagram Types



Software Development Lifecycle Activities





Use Cases



- A use case is a software and system engineering term that describes how a user uses a system to accomplish a particular goal. – Google
- A use case is a list of actions or event steps typically defining the interactions between a role (known as an actor) and a system to achieve a goal. – Wikipedia

Functional Requirements versus Use Cases

- Both are about system functionalities
 - Functional requirements: **what** (user, external system, functionality, and information)
 - Use cases: **how** (actors interact with system functionality)

1

Functional requirements are the starting points for use case modeling.

Use Case Model

Use Case Diagram (static)

+

Use Case Description (dynamic)

- Provides coherent visual and textual description of system functionalities
- In use-case-driven iterative development, prioritise use cases for implementation

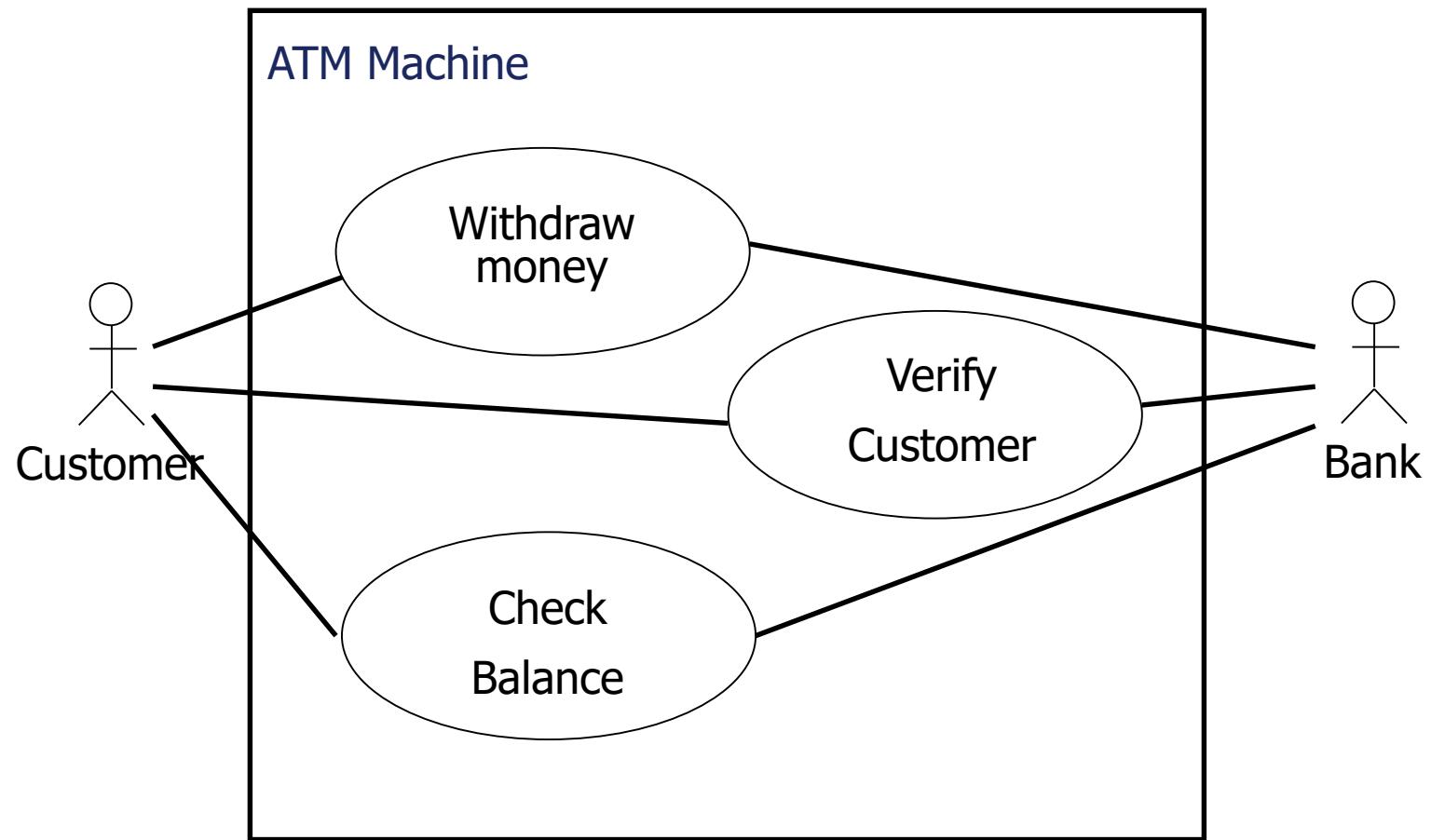
drive from
functional
Requirement



Use Case Diagram Elements

Name	Notation	Description
Actor		An external entity that interacts with system
Use Case		Unit of functionality performed by system, which yields result / value for Actor
Association		Connects Actor to Use Cases(s) in which they participate

Example of Use Case Diagram

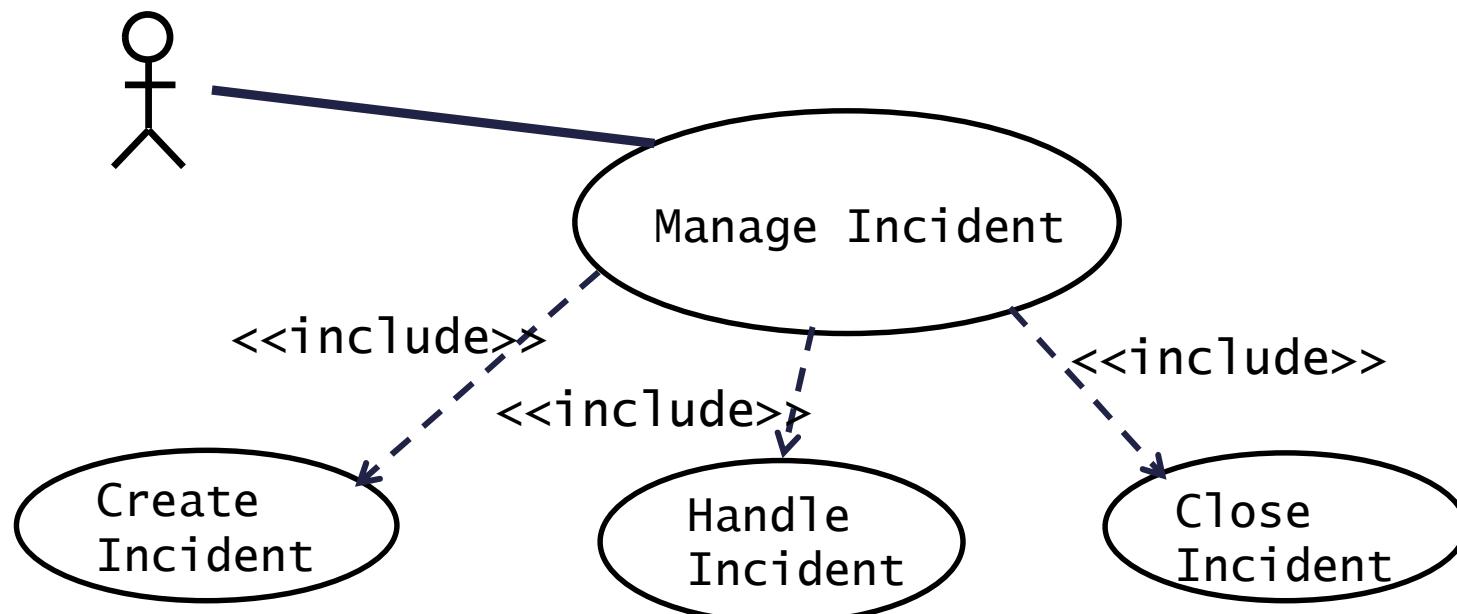


Use Case Associations

- Dependencies between use cases are represented by **use case associations**
- Associations are used to reduce complexity
- Two types of use case associations
 - **Includes**
 - **Extends**

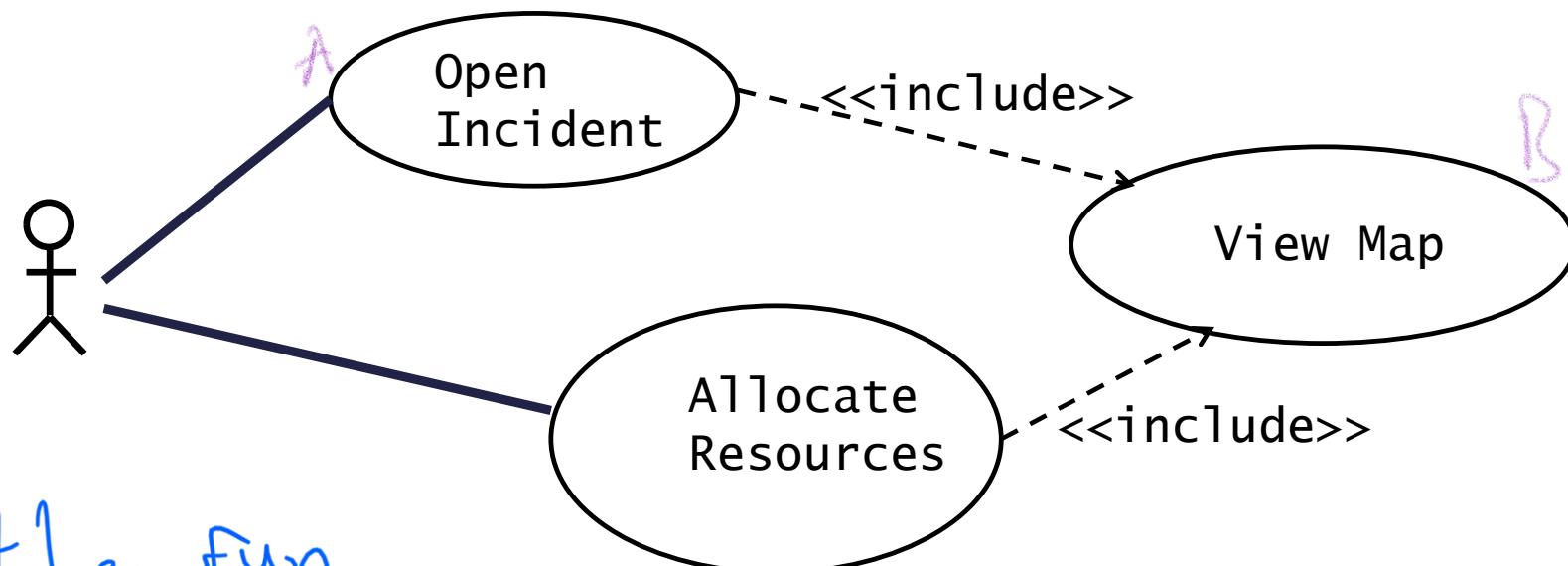
<<include>>:Functional Decomposition

- **Problem:** A function in the original problem statement is too complex
- **Solution:** Describe the function as the aggregation of a set of simpler functions. The associated use case is decomposed into shorter use cases



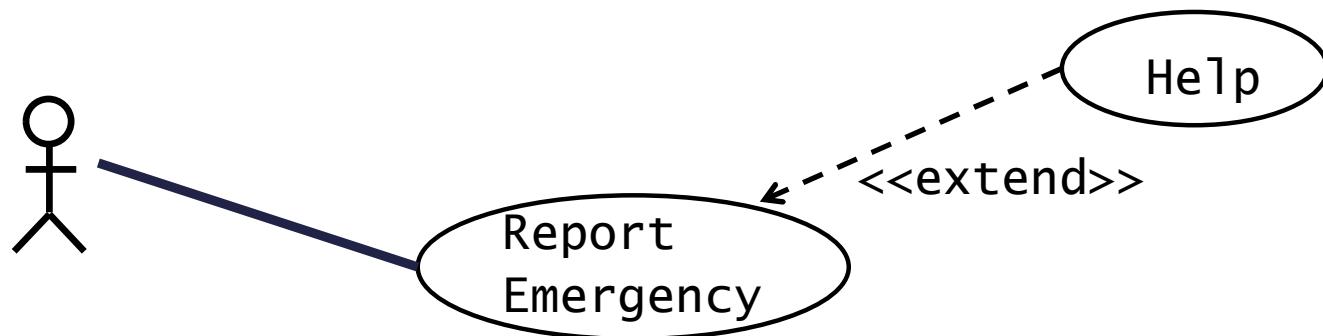
<<include>>: Reuse Existing Functionality

- Problem: There are overlaps among use cases.
- Solution: The ***include association*** from use case A to use case B indicates that an instance of use case A performs all the behaviour described in use case B ("A delegates to B")
- Example: Use case "View Map" describes behaviour that can be used by use cases "Open Incident" and "Allocate Resources"



<<extend>>: Adding Optional Functionality

- Problem: The functionality in the original problem statement needs to be extended.
- Solution: An *extend association* from use case A to use case B
- 1. • Example: “ReportEmergency” is complete by itself, but **can be extended** by use case “Help” for a scenario in which the user requires help



Include, extend associations

→ Original is not complete without include

- Include = **reuse** of functionality
 - denotes a dependency on another use case. Included Use Cases are always used in a parent Use Case.

- Extends = **adding** (optional) functionality. Extends are only used in the parent Use Case in exceptional or unusual circumstances.

→ Original stand on its own without extends

Example Functional Requirements of a Library Management System

1. The library member must be able to search the system for library materials
2. The library member must be able to loan library materials
3. The library management system must verify the library membership with the University Account System before the library member can loan library materials
4. The library member must return library materials via library dropoff kiosk
5. The library management system must send overdue notice using the CITS Email Sever to the library member
6. When there are network issues to connect to CITS Email Server, the library management system must log the event

Identify Potential Actors

1. The **library member** must be able to search the system for library materials
 2. The library member must be able to loan library materials
 3. The library management system must verify the library membership with the **University Account System** before the library member can loan library materials
 4. The library member must return library materials via **library dropoff kiosk** → *Separated device*
 5. The library management system must send overdue notice using the **CITS Email Server** to the library member
 6. When there are network issues to connect to CITS Email Server, the library management system must log the event
- because
not part
of
System
unma
built

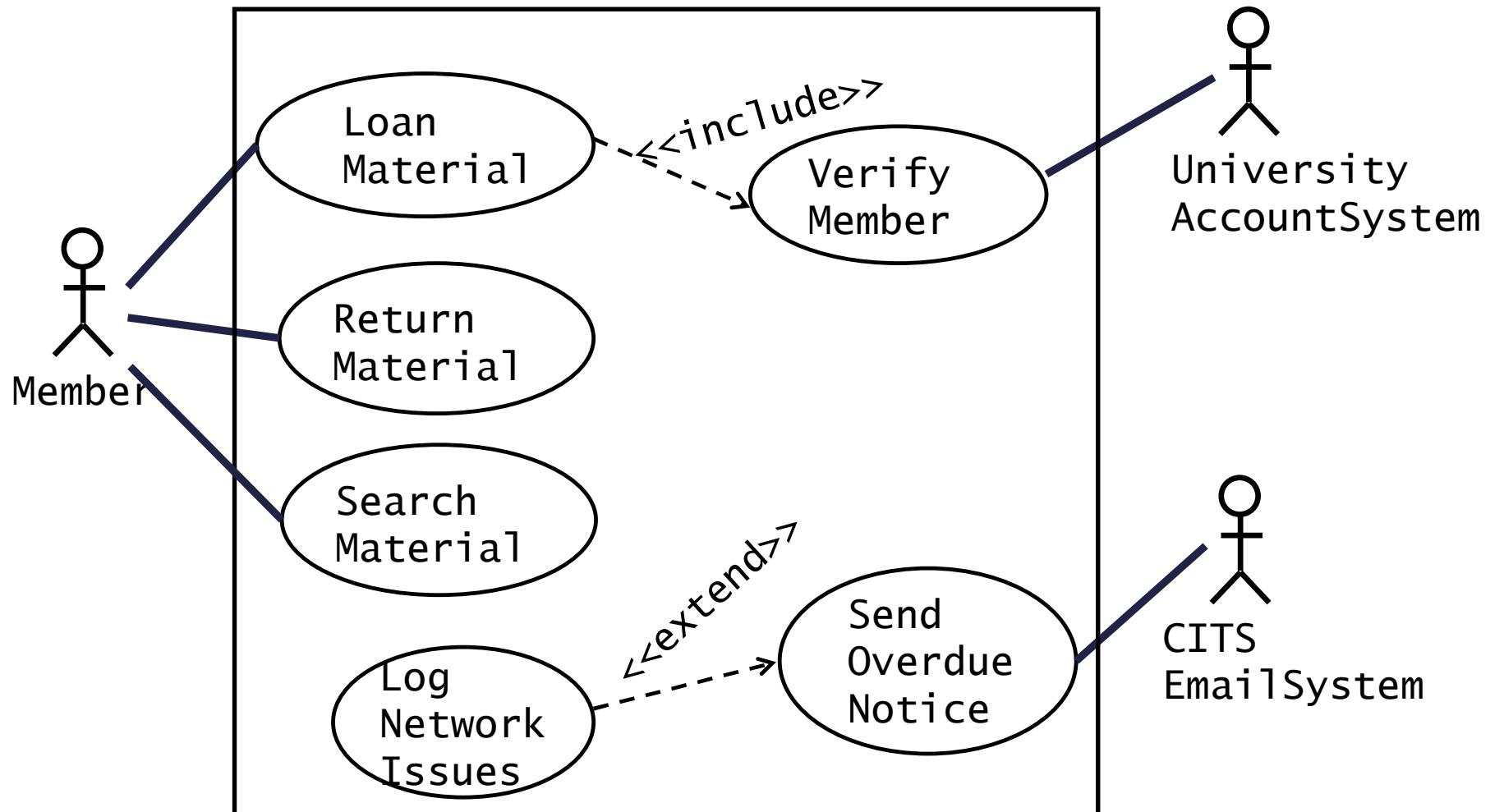
Identify Potential Use Cases

1. The library member must be able to search the system for library materials
2. The library member must be able to loan library materials
3. The library management system must verify the library membership with the University Account System before the library member can loan library materials
4. The library member must return library materials via library dropoff kiosk
5. The library management system must send overdue notice using the CITS Email Sever to the library member
6. When there are network issues to connect to CITS Email Server, the library management system must log the event

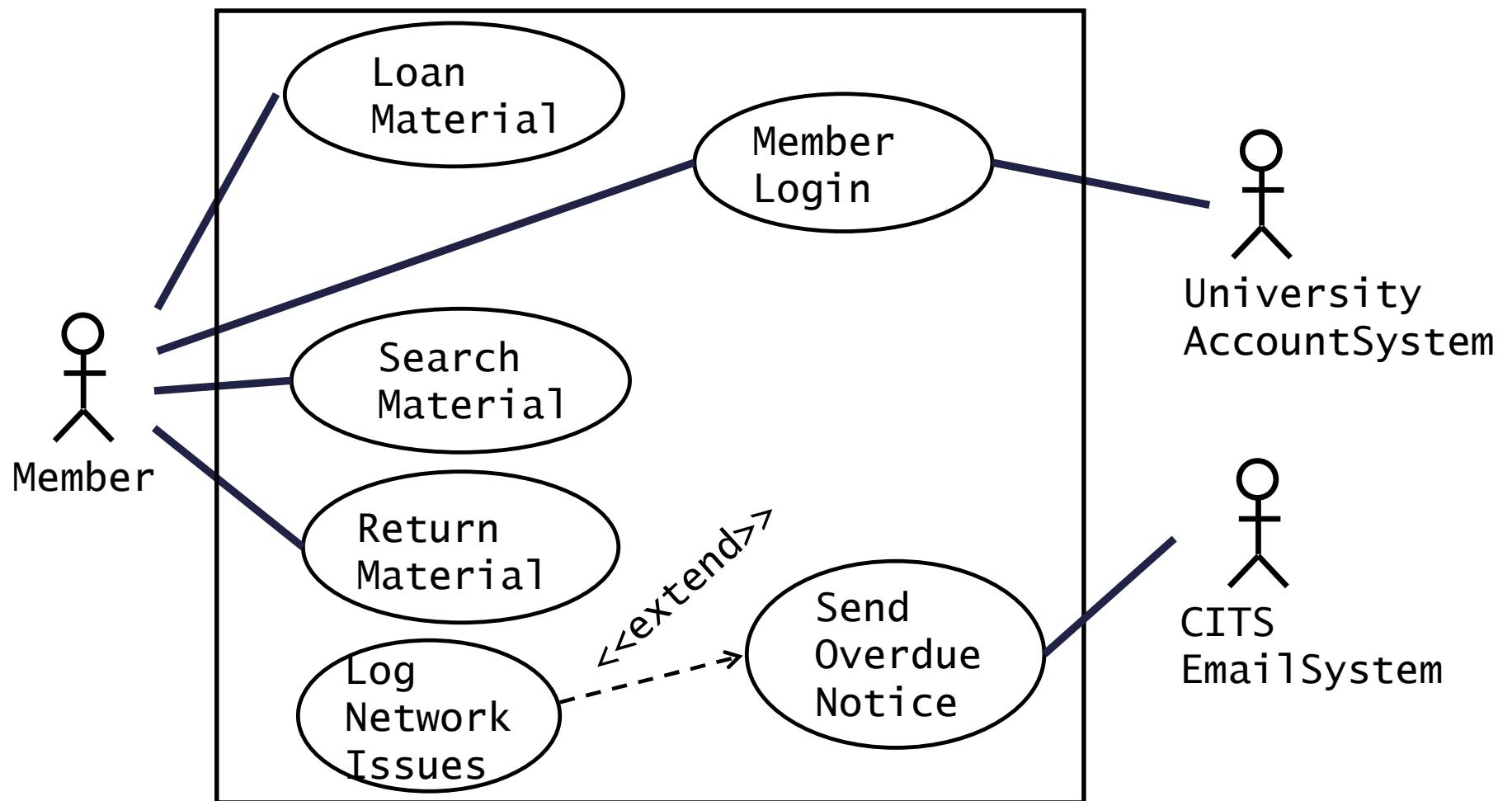
Identify Potential Associations Between Use Cases

1. The **library member** must be able to search the system for library materials
2. The library member must be able to loan library materials
3. The library management system must verify the library membership with the **University Account System** **before** the library member can loan library materials
4. The library member must return library materials via **library dropoff kiosk**
5. The library management system must send overdue notice using the **CITS Email Sever** to the library member
6. **When** there are **network issues** to connect to CITS Email Server, the library management system must log the event

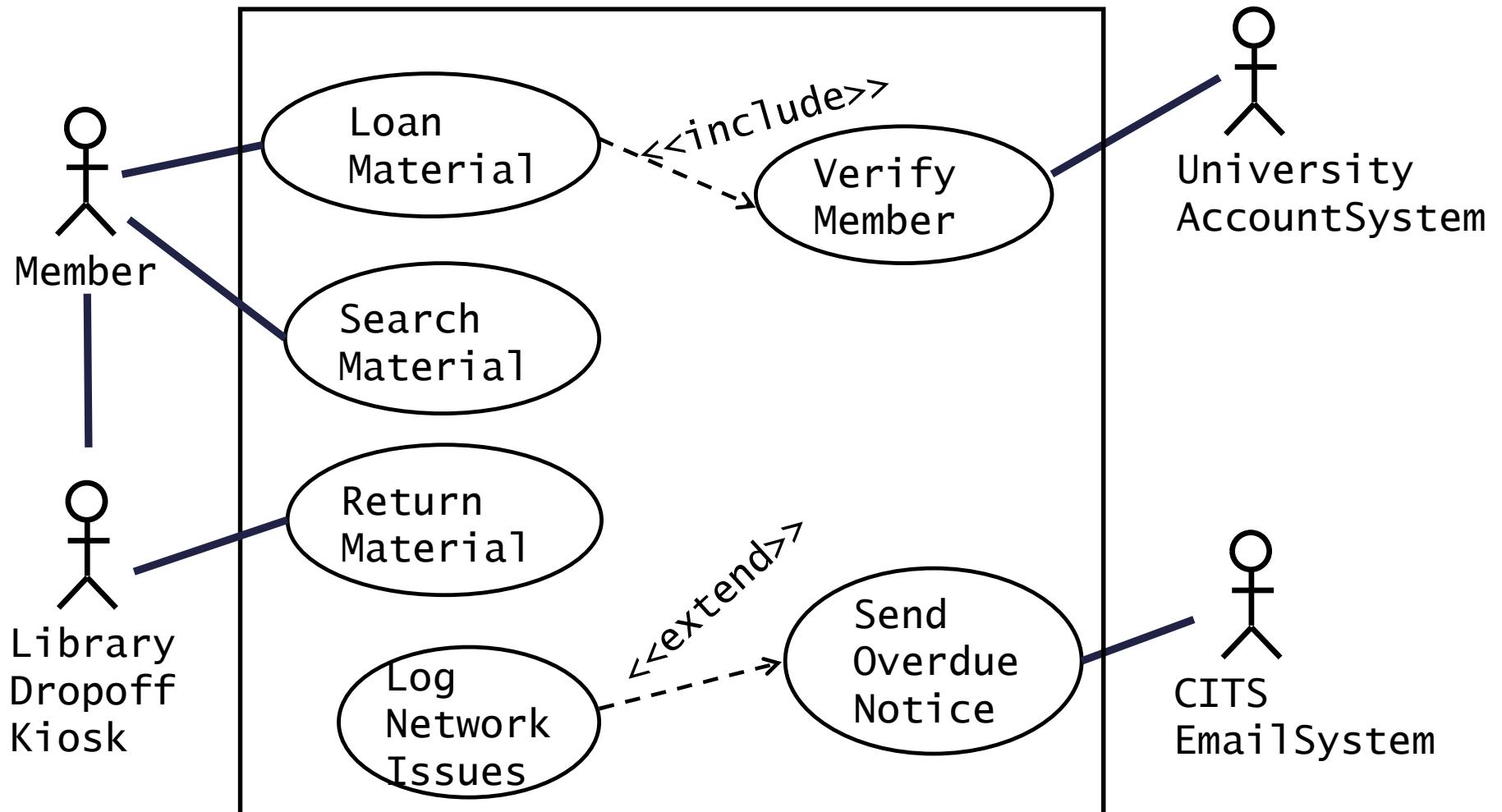
Use Case Diagram 1



Use Case Diagram 2



Use Case Diagram 3



Software Engineering

CE2006/CZ2006

Requirements Elicitation 3 of 3

Discussion Topics

- Use Case description
- Review of requirement elicitation process
- Software Requirements Specification (SRS)
- Reading
 - Bruegge Chap 4.4
 - Fowler Chap 9

Use Case Model

Use Case Diagram (static)

+

Use Case Description (dynamic)

- Provides coherent visual and textual description of system functionalities
- In use-case-driven iterative development, prioritise use cases for implementation



Each Use Case Must have a Use Case Description that contains

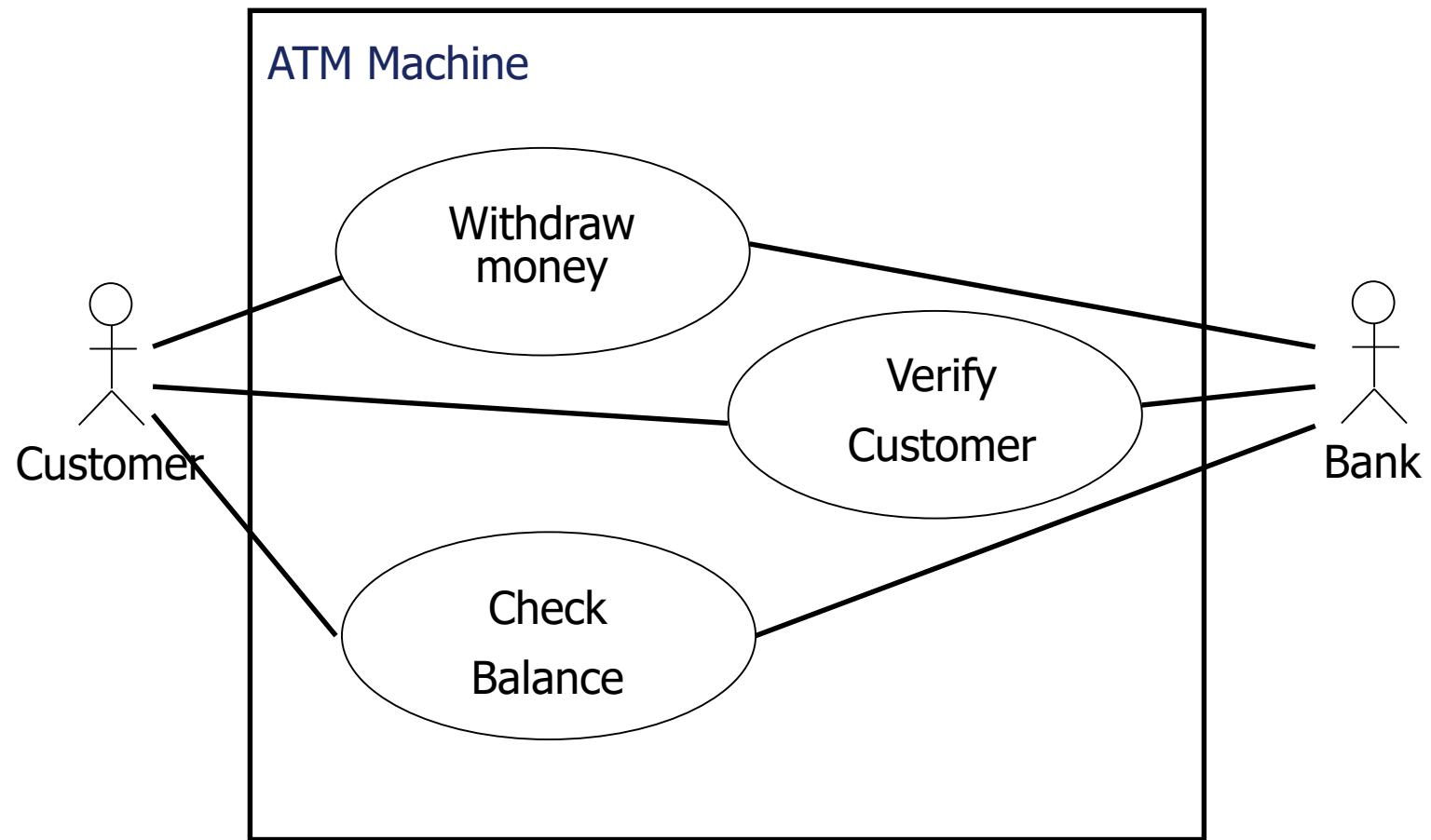
- Participating Actors (Initiating Actor)
 - A use case can have more than one participating actors
 - Initiating actor triggers the use case
 - If the use case is triggered by a system event or another use case, the initiating actor is NULL
- Entry conditions (Pre-condition)
 - The system state must be satisfied to execute a use case
- Exit Conditions (Post-condition)
 - The system state must be reached after executing a use case



Each Use Case Must have a Use Case Description that contains

- Flow of Events (a.k.a Normal or Main Successful Flow)
 - Step #1 – How Actor triggers the use case
 - Step #2 – How System responds
 - Step #3 – ...
- Alternative Flows
 - Variations or errors in the interaction (e.g., wrong username/password)
 - Return back to the normal flow of events
- Exceptions
 - Exceptional situations that cause the failure of the use case (e.g., network not available)

Example of Use Case Diagram



Example #1 Actor, Entry/Exist Conditions

The Bank Customer **Withdraw Money** Using ATM

Participating actor:

Bank Customer (Initiating Actor), Bank

Entry Conditions:

1. Bank Customer has a Bank Account with the Bank
and
2. Bank Customer has an ATM Card and PIN

Exit Conditions:

1. Bank Customer receives the requested cash
or
2. Bank Customer receives an explanation from the ATM why the cash could not be dispensed

Example #1 Flow of Events (Normal Flows)

Actor steps

1. The Bank Customer inputs the card into the ATM
3. The Bank Customer enters a PIN
6. The Bank Customer enters an amount
9. The Bank Customer removes card, receipt and money

System steps

2. The ATM requests the input of a six-digit PIN.
4. The ATM verifies the card and PIN with the account information in the Bank
5. If the card and PIN is verified the ATM requests the amount to withdraw
7. The ATM verifies the amount is available in the customer account
8. If the customer account has sufficient funds, the ATM outputs the money and card and a receipt

Example #1 Flow of Events (Normal Flows)

1. The Bank Customer inputs the card into the ATM
2. The ATM requests the input of a six-digit PIN.
3. The Bank Customer enters a PIN
4. The ATM verifies the card and PIN with the account information in the Bank
5. If the card and PIN is verified the ATM requests the amount to withdraw
6. The Bank Customer enters an amount
7. The ATM verifies the amount is available in the customer account
8. If the customer account has sufficient funds, the ATM outputs the money and card and a receipt
9. The Bank Customer removes card, receipt and money

(Variation that doesn't lead to abandonment)

Example #1 Alternative Flows

AF-S5: If the card and PIN are invalid

1. The ATM displays the message "Invalid card and PIN. Please try again!" for 2 seconds
2. The ATM returns to the step 2

AF-S8: If the customer account has insufficient funds

1. The ATM displays the message "Insufficient funds in your account! Please enter a smaller amount!" for 2 seconds
2. The ATM returns to the step 6

(non-recoverable error) (lead to abandon
of system)

Example #1 Exceptions

EX1: If the Bank Customer enters invalid PIN for three times

1. The ATM displays the message "Card is suspended! Please call the customer service (1633) to reactive the card" for 2 seconds
2. The ATM outputs the card

EX2: If the ATM has insufficient funds

1. The ATM displays the message "This ATM is off-service due to insufficient funds!" at the start screen
2. The ATM does not accept the card

Example #2 Course Registration System

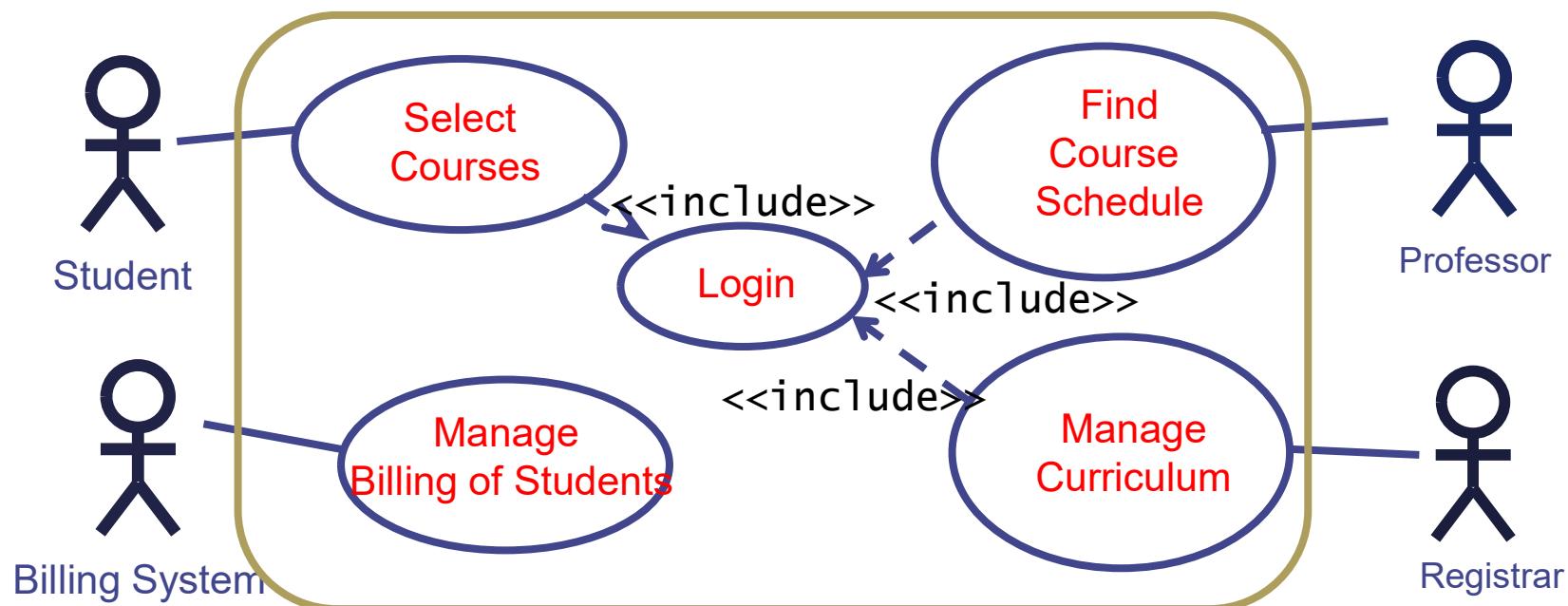
- Registrar manages the course curriculum for a semester
- Students select 4 primary courses and 2 alternate courses
- Students use the system to add and drop courses till the end of the second week of the semester
- Once a Student registers courses for a semester, the registration system notifies the Billing System so the student may be billed for the semester
- Professors use the system to find their course schedule
- All users of the registration system must login the course registration system with the assigned password before performing the tasks

Example #2 Course Registration System

- Registrar manages the course curriculum for a semester
- Students select 4 primary courses and 2 alternate courses
- Students use the system to add and drop courses till the end of the second week of the semester
- Once a Student registers courses for a semester, the registration system notifies the Billing System so the student may be billed for the semester
- Professors use the system to find their course schedule
- All users of the registration system must login the course registration system with the assigned password before performing the tasks

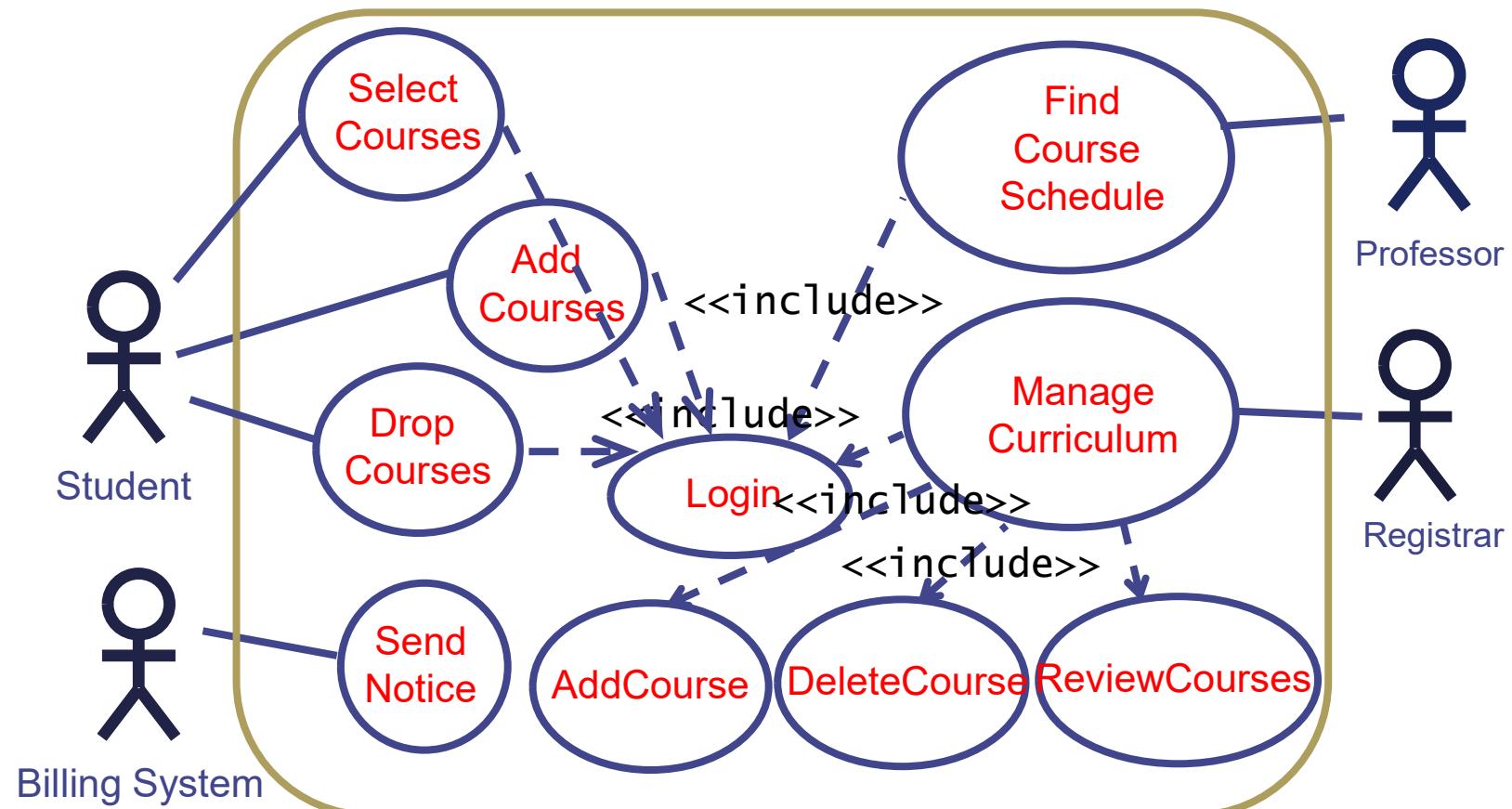
Example#2 Use Case Diagram

- Actors are identified and Use Case diagram are created to visualize the relationships between actors and Use Cases



Example#2 Use Case Diagram

- More use cases



Example#2 Use Case Description: Manage Curriculum - Flow of Events

1. The system uses the included use case Login to verify the Registrar.
2. On successful login, the system prompts the Registrar to select the current semester or a future semester.
3. The Registrar selects the desired semester.
4. The system prompts the Registrar to select the desired activity: ADD, DELETE, REVIEW, or QUIT.
5. If the Registrar selects the activity ADD, then he uses the included use case AddCourse to add a course.
6. If the Registrar selects the activity DELETE, then he uses the included use case DeleteCourse to delete a course.
7. If the Registrar selects the activity REVIEW, then he uses the included use case ReviewCourses to review existing courses.
8. If the Registrar selects the activity QUIT, the system returns to the semester selection screen.

A Use Case Template should be used to describe each Use Case

See the Use Case Description Template available on NTULearn course website.

Use Case ID:			
Use Case Name:			
Created By:		Last Updated By:	
Date Created:		Date Last Updated:	

Actor:	
Description:	
Preconditions:	
Postconditions:	
Priority:	
Frequency of Use:	
Flow of Events:	
Alternative Flows:	
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	



Review of Requirement Elicitation Process

Key Concepts and Deliverables

- Functional requirements
- Use case model
 - Use case diagram
 - Use case description
- Non-functional requirements
- Data dictionary (problem domain glossary)
- UI prototype

Steps 1. Project Team

Review clients specification and review any legacy or competitive products (if available) to:

1. Atomise requirements.
2. Identify Actors
3. Identify main Use Cases (Functional Requirements)
4. Identify Non-Functional requirements
5. Initiate generation of Data Dictionary
6. Identify uncertainties to clarify with client (TBD – To Be Determined). Do not guess the user requirements.

Step 2. Project Team meet Client

(Choose appropriate meeting - Interview, Observation, Workshop)

The agenda and necessary outcomes of the meeting must be set and agreed before the meeting.

Do not waste the Project Teams or Clients time in unstructured unproductive meetings.

Typical Agenda

1. Present and walk through the initial requirements and Data Dictionary. Get agreement.
2. Clarify uncertainties (the TBD issues).
3. Discuss any new requirements the client identifies.

Repeat Steps 1 & 2. Project Team and Client

The refining process continues iteratively until all uncertainties and ambiguities of the functional and non-functional requirements and the definition of terms in the data dictionary are agreed with the clients.

A prototype demonstrating the appearance or user interface and dialogues of the functionality may be produced to show the intended functionality.

Try to minimize number of meetings by making meetings productive.

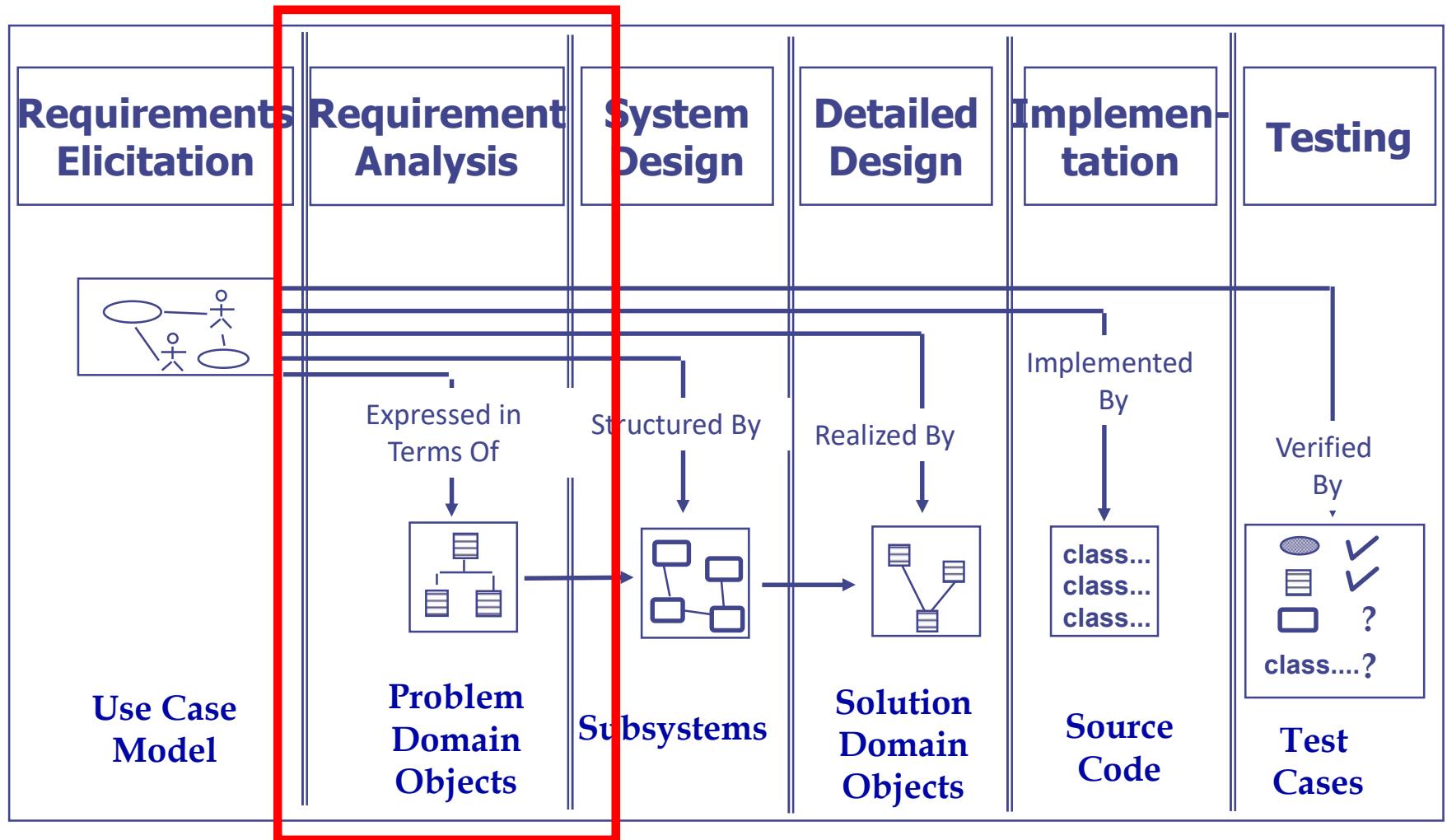
Software Engineering

CE2006/cz2006

Requirements Analysis: Conceptual Models

Discussion Topics

Software Development Lifecycle Activities





Requirements Elicitation versus Requirements Analysis

- Requirements Elicitation:
 - **Purpose:** finding out what **customers** want
 - **Output:** a description of the system in terms of actors and use cases
- Requirements Analysis:
 - **Purpose:** produce a system model that is correct, complete, consistent, unambiguous based on use cases
 - **Output:** conceptual model (system structure)
 - + dynamic model (system behavior)



Formalizing Requirements with Analysis Models

- Clarifies structural and dynamic aspects of system to be build
- Validates requirements
- Underpins solution modelling

Requirement Analysis Goals

- Conceptual model (structural aspect)
 - Analyze use cases to identify the **objects** and **roles** of objects involved in the system
- Dynamic model (dynamic aspect)
 - Determine how to **fulfill the processes** defined in the use cases and which **objects** do these processes

Conceptual Model

(What object to use
for dynamical aspect)

- **Objects:** Anything that has a state and exhibits behavior
 - e.g., Bus, Dog, Company
 - e.g., LoginForm, UserVerifier, UserList
- **Operations:** Procedures through which objects communicate amongst themselves
 - e.g.: Bus: Stop, Start. Dog: Bark, Growl. Company: Sell, Quote.
 - e.g., LoginForm: submit. UserVerifier: authenticate. UserList: SearchUser
- **Attributes:** Variables that hold state information
 - e.g. Bus: Seats, Colour. Dog: Name, Breed. Company: Name, Employees

Conceptual Model in Class Diagram

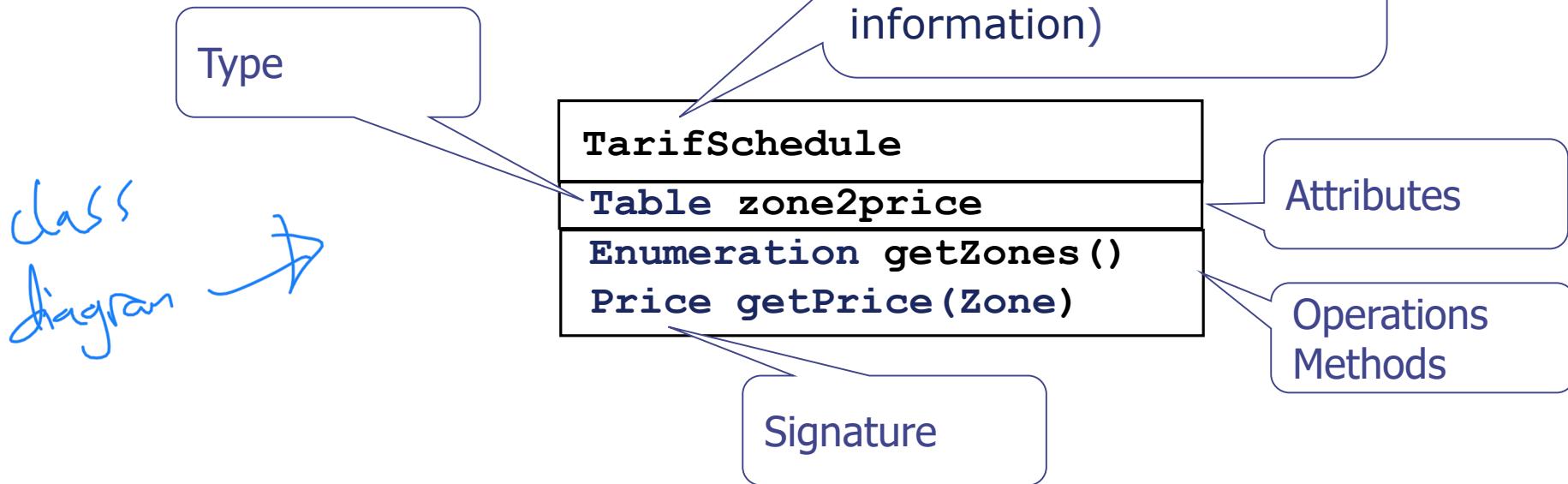
- Document and visualize conceptual model in class diagram
 - Object ⇒ Class
 - Roles (Attributes and operations) of object ⇒ Attributes and methods of class



Class Diagrams

- Class diagrams represent the structure of the system
- The classes define the responsibilities for doing various activities
- Used during
 - requirements analysis: model application domain concepts
 - system design: model subsystems and role of object
 - object design: specify the detailed behavior and attributes of classes

Classes

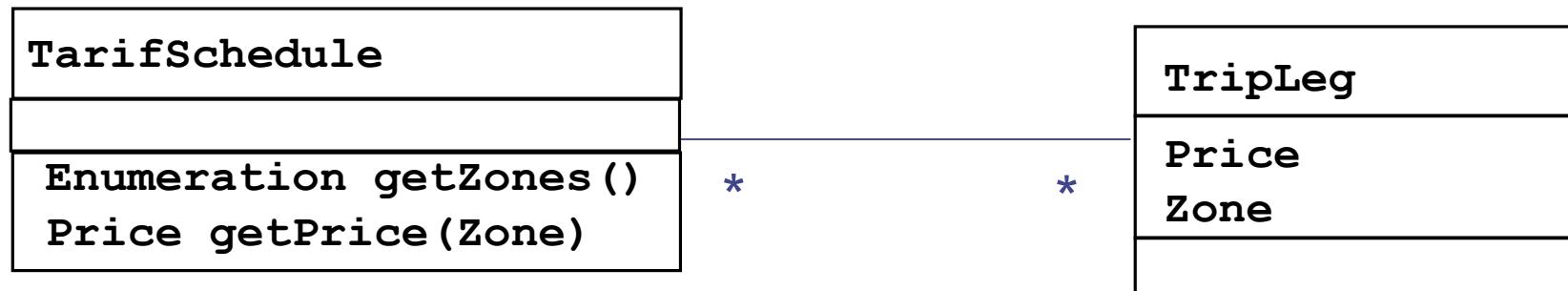


- A **class** represents a concept
- A class encapsulates states (**attributes**) and behaviour (**operations**)

Each attribute has a *type*

Each operation has a *signature*

Associations

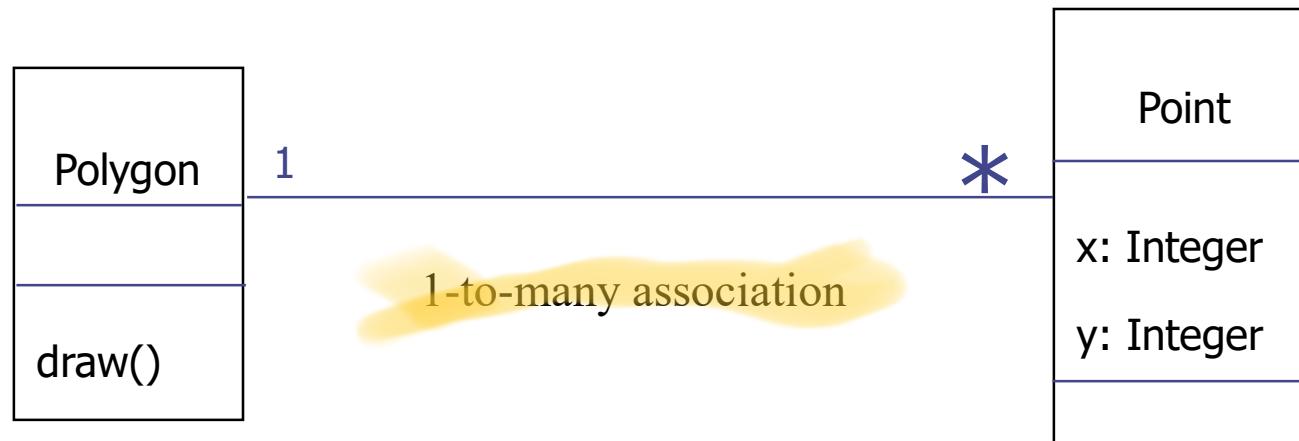
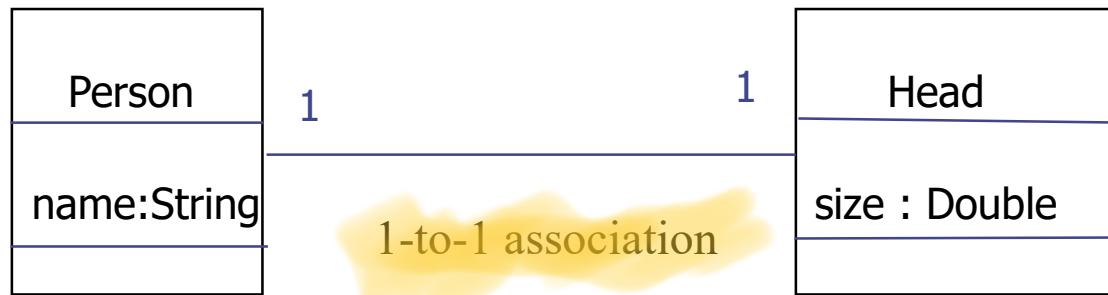


Associations denote relationships between classes

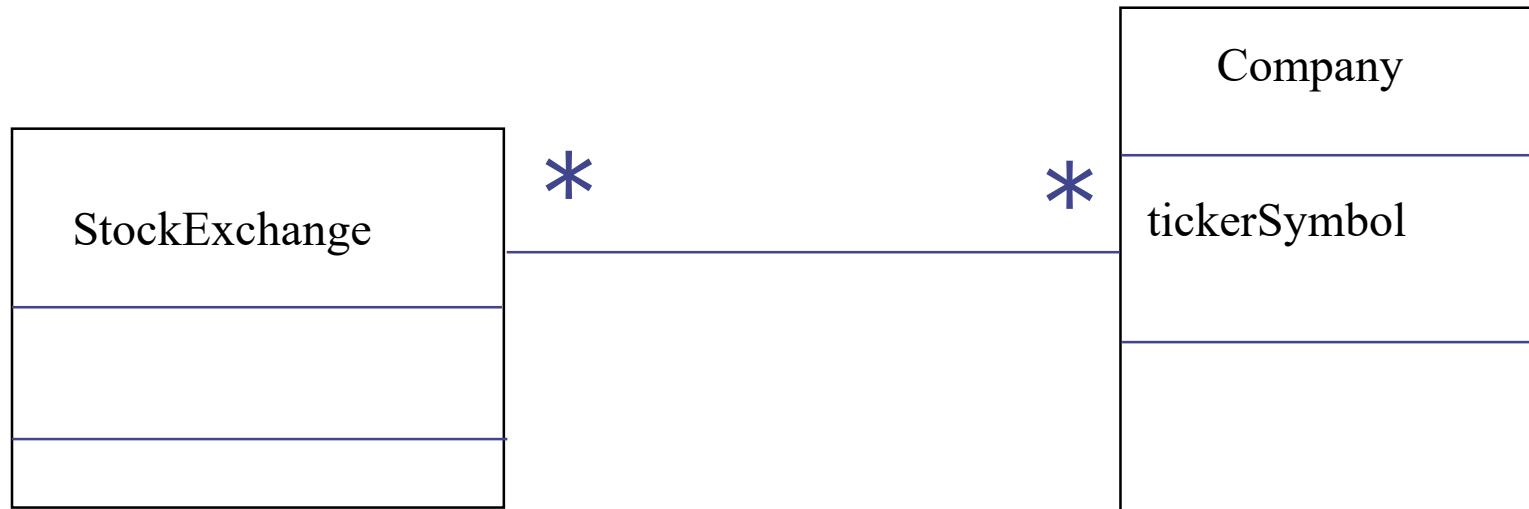
The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

Multiplicity is a number (e.g. 3), a range of numbers (1..6) or a large unspecified number (many) (*)

1-to-1 and 1-to-many Associations



Many-to-Many Associations



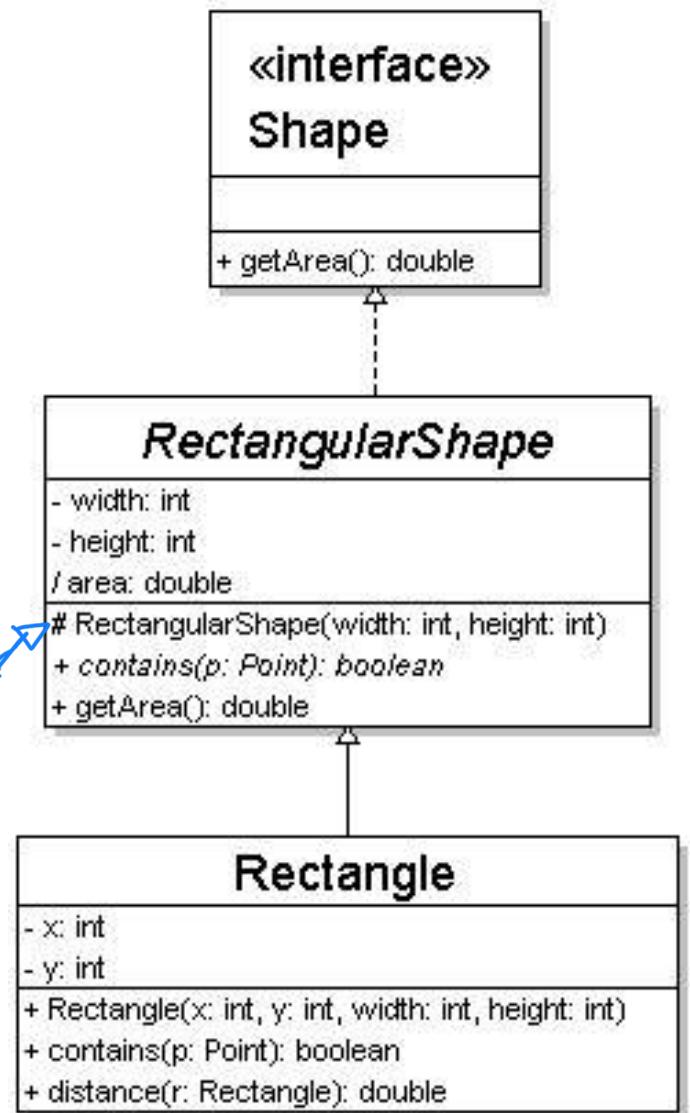
Class Relationships

- **generalization:** an inheritance relationship
 - inheritance between classes
 - interface implementation
- **association:** a usage relationship
 - dependency
 - aggregation
 - composition

Class Relationships

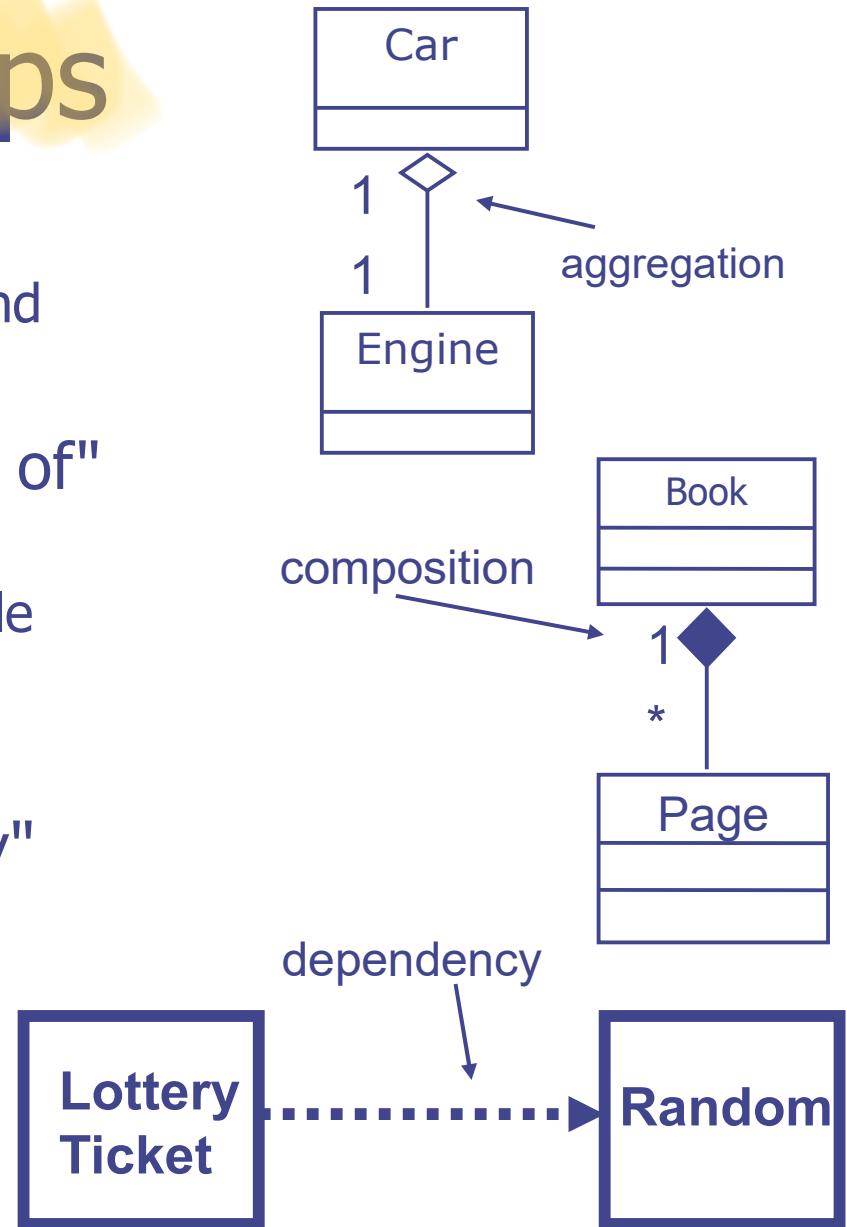
- generalization (inheritance) relationships
 - hierarchies drawn top-down with arrows pointing upward to parent
 - line/arrow styles differ, based on whether parent is a(n):
 - class: → solid line, black arrow
 - abstract class: → solid line, white arrow
 - interface: - - - → dashed line, white arrow
 - we often don't draw trivial / obvious generalization relationships, such as drawing the Object class as a parent

not method



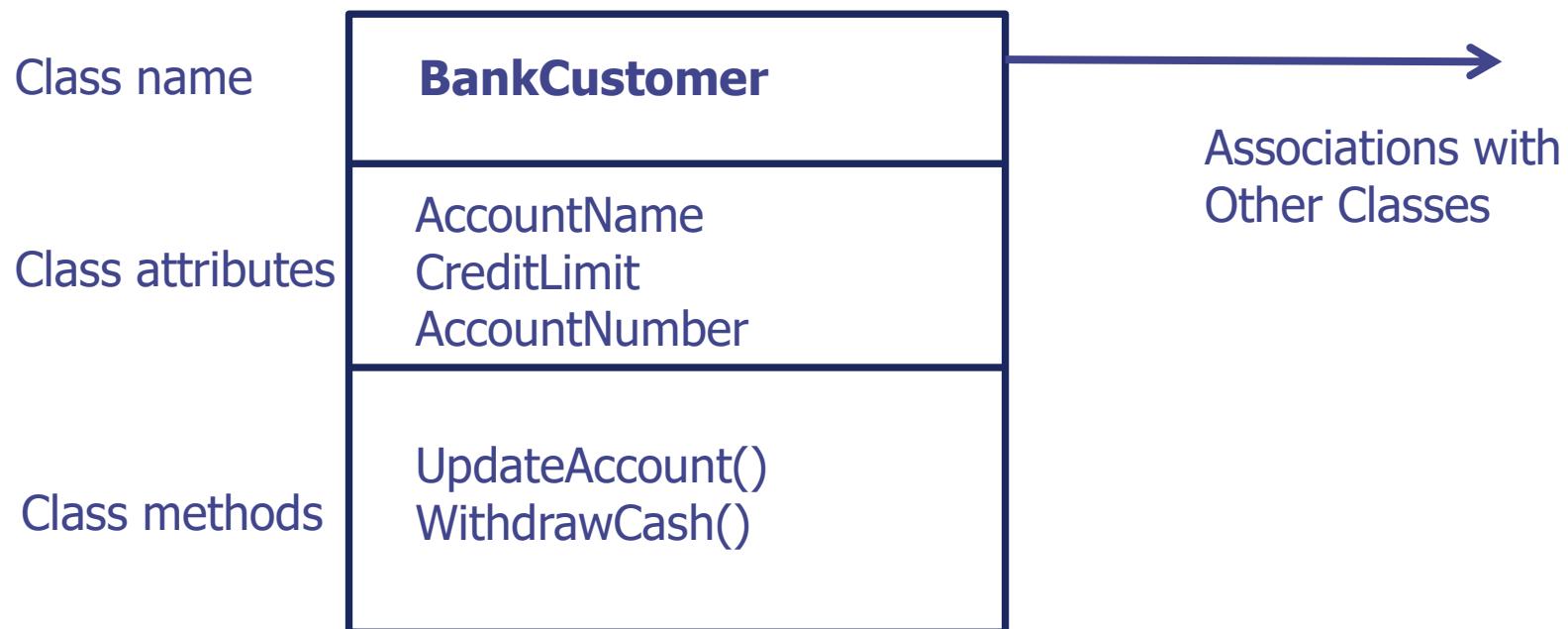
Class Relationships

- **aggregation:** "is part of"
 - symbolized by a clear white diamond
- **composition:** "is entirely made of"
 - stronger version of aggregation
 - the parts live and die with the whole
 - symbolized by a black diamond
- **dependency:** "uses temporarily"
 - symbolized by dotted line
 - often is an implementation detail, not an intrinsic part of that object's state



Requirement Analysis and Conceptual Modeling Activities

- ! 1. Identify the Classes Use case model → Class diagram
- 2. Find the attributes for each Class
- 3. Find the methods/operations for each Class
- ! 4. Find the associations between Classes



Identification of Initial Objects or Classes

Look for

1. Recurring nouns / concepts in the use cases
2. Real-world entities the system must track
3. Application (problem) domain terms in data dictionary

we focus ↗

Review definitions and attributes with stakeholders

Stereotypes of Classes



Boundary Class <<boundary>>

– interaction between Actor & System



Control Class <<control>>

– logic to realize use case

data
base



Entity Class <<entity>>

– information tracked by System



Types of Entity Classes

- Things remembered or data that persists
 - UserInfo, Event, Course, Book
- Organisational units
 - Company, TutorialGroup
- Structures
 - OrderList

Start with data dictionary

Example: EnterSecureFacility Use Case Description

Identify recurring nouns, real-world entities,
domain terms – these may be Classes

Flow of Events:

1. User presses finger on FingerprintReader.
2. FingerprintReader scans fingerprint and sends image to the FacilityAccessSystem.
3. The FacilityAccessSystem validates fingerprint and unlocks EntryGate.
4. EntryGate sends a signal to the FacilityAccessSystem that it has opened.
5. The FacilityAccessSystem locks EntryGate, logs entry event and increments occupant count.

Example: EnterSecureFacility Use Case Description

Identify recurring nouns, real-world entities,
domain terms – these may be Classes

Flow of Events:

1. User presses finger on FingerprintReader.
2. FingerprintReader scans fingerprint and sends image to the FacilityAccessSystem.
3. The FacilityAccessSystem validates fingerprint and unlocks EntryGate.
4. EntryGate sends a signal to the FacilityAccessSystem that it has opened.
5. The FacilityAccessSystem locks EntryGate, logs entry event and increments occupant count.



Example: EnterSecureFacility - Possible Classes

FingerprintReader

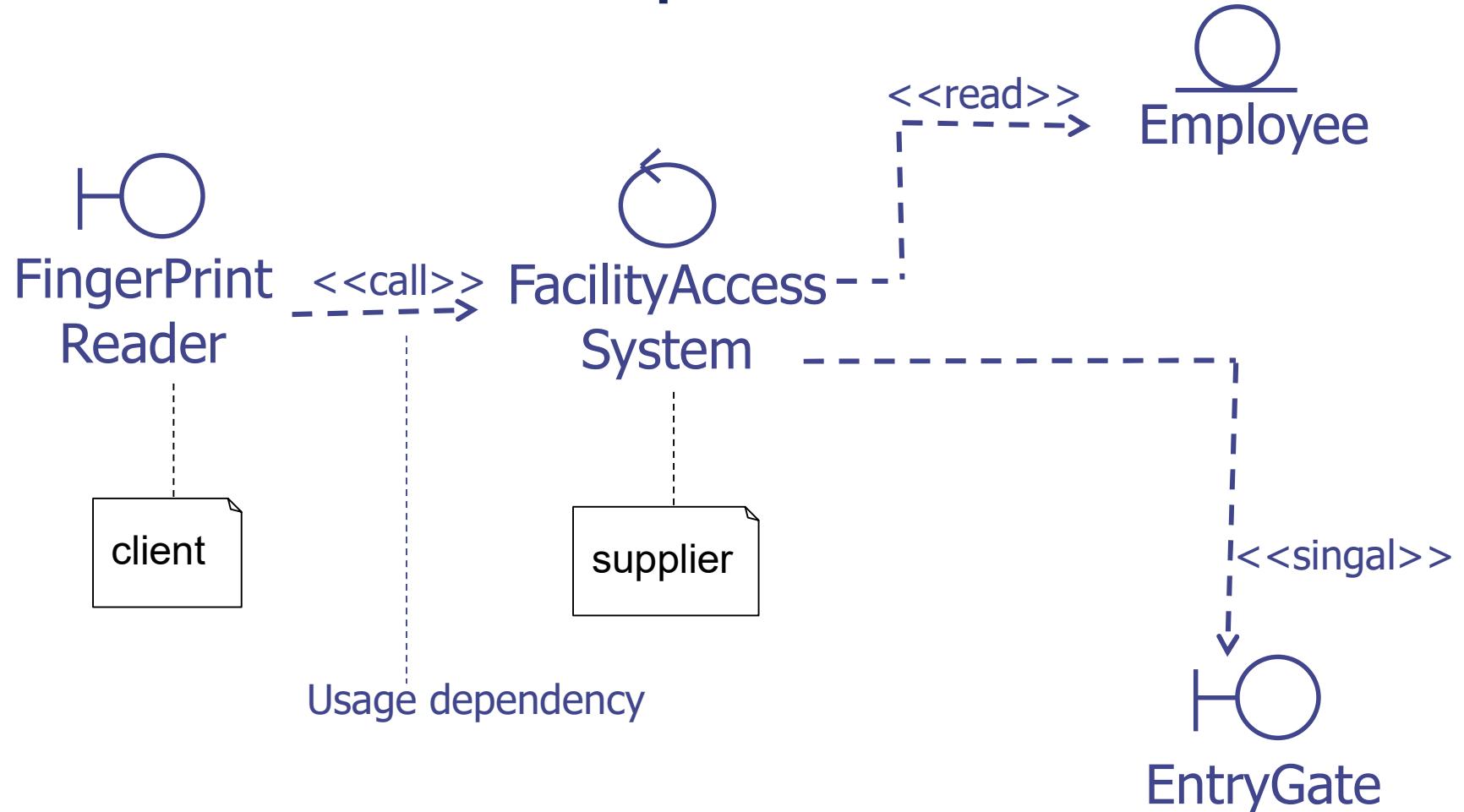
EntryGate

FacilityAccessSystem

Employee
(Fingerprint?)

related
use & first

Stereotyping to Indicate Class Responsibilities



Summary

- Analysis Modelling
- Class Diagram

Software Engineering

CE2006/CZ2006

Requirements Analysis: Dynamic Models 1

Discussion Topics

- Interaction Diagrams
 - Sequence Diagram
 - Communication Diagram
- Reading
 - Bruegge chap. 2.4.3, 5.4.4
 - Fowler chap.4 and chap. 12 Bruegge chap.2.4.4, 5.4.9
 - Fowler chap.10

Requirement Analysis Goals

- Conceptual model (structural aspect)
 - Analyze use cases to identify the objects and roles of objects involved in the system
- Dynamic model (dynamic aspect)
 - Determine how to fulfill the processes defined in the use cases and which objects do these processes

Document and Visualize Dynamic Model

- Using the following types of diagrams
 - Sequence diagram
 - Communication diagram
 - State machine
 - Activity diagram

Sequence Diagram and Communication Diagram

Captures how **a group of objects** (in a Class Diagram) **interact or collaborate** to achieve an activity as described in a Use Case.

Usually captures the dynamics of **a single scenario** and the **sequence of messages** that are passed between the objects and the **sequence of events** that happen during the scenario.

Both diagrams essentially show the same information –

- **Sequence diagrams** emphasise the sequence and timing of calls and messages between the objects
- **Communication diagrams** emphasise the communication links between the objects

UML in Development Process

- Ultimate Goal: effective communication!
 - Requirement Elicitation (users & customers)
 - Design (designers & developers)
 - Documentation (other developers/maintainers)



How to Describe Systems' Interactions Using UML
Sequence Diagram

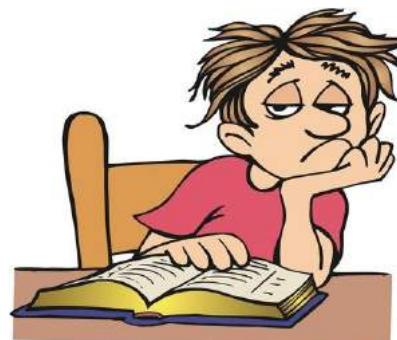
<https://youtu.be/Mqy569thFN8>

Example: EnterSecureFacility Use Case Description

Flow of Events:

1. User presses on FingerprintReader.
2. FingerprintReader scans fingerprint and validates image with the FacilityAccessValidator.
3. The FacilityAccessValidator finds fingerprint in the EmployeeList.
4. The FacilityAccessValidator validates fingerprint and unlocks EntryGate.
5. ...

Too much text!



Class Diagram

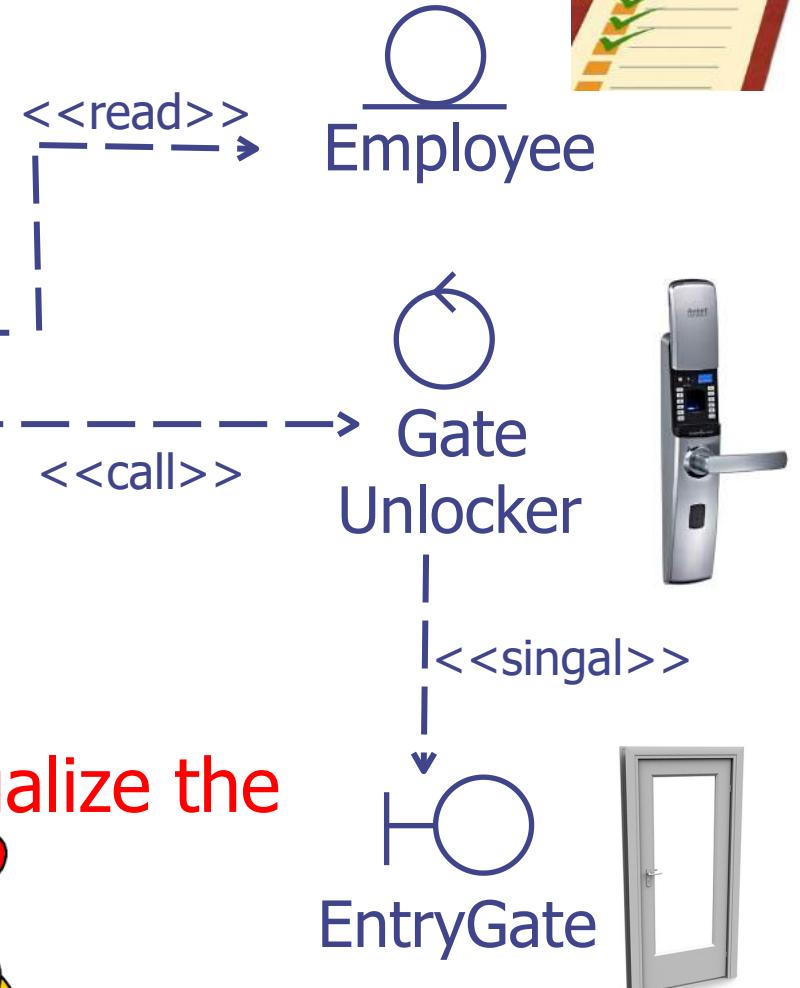
Time ordering info is lost!



FingerPrint Reader



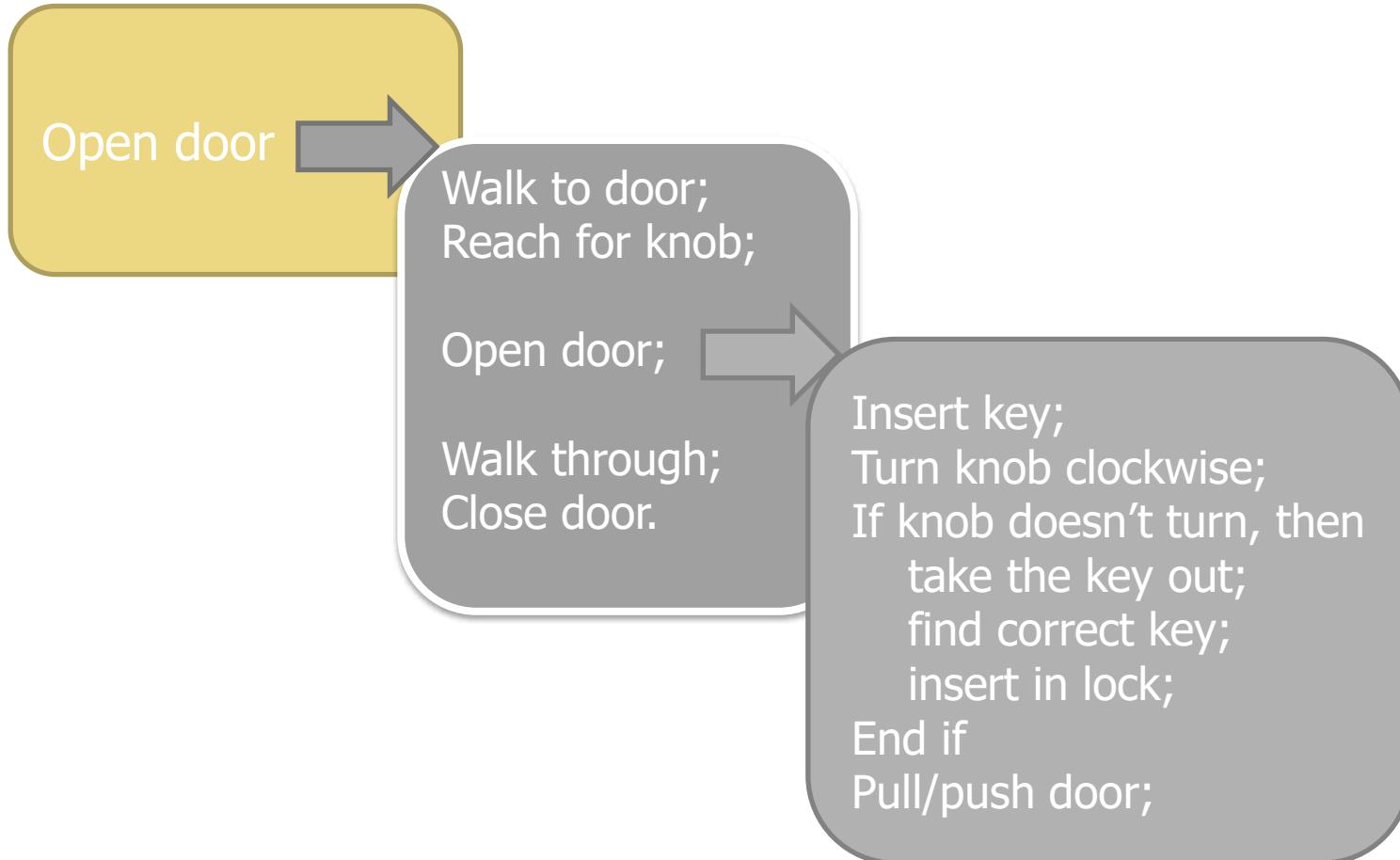
FacilityAccess Validator



How to visualize the workflow?



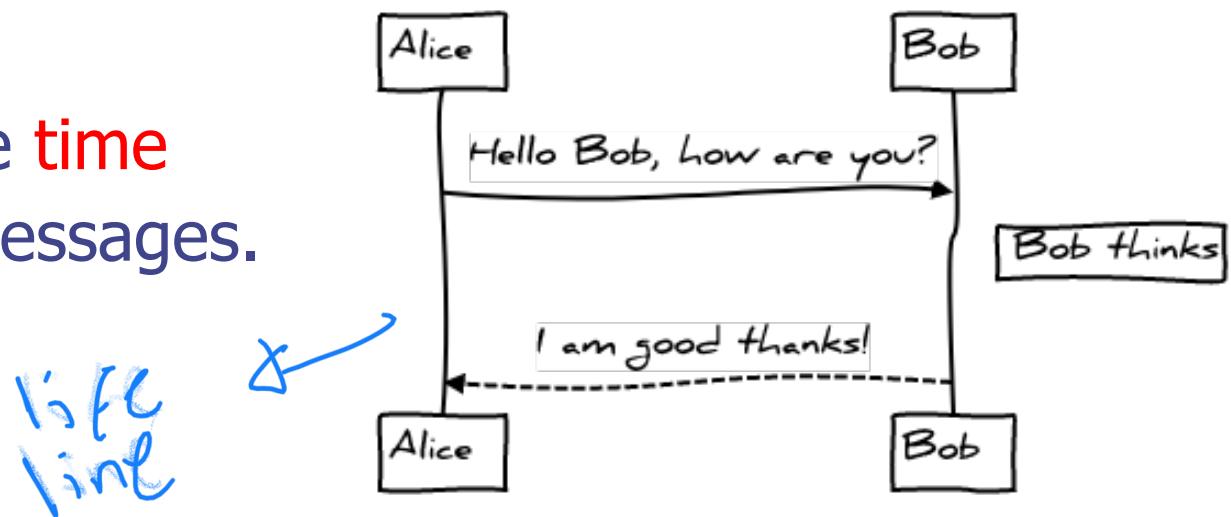
Incremental Design Refinement



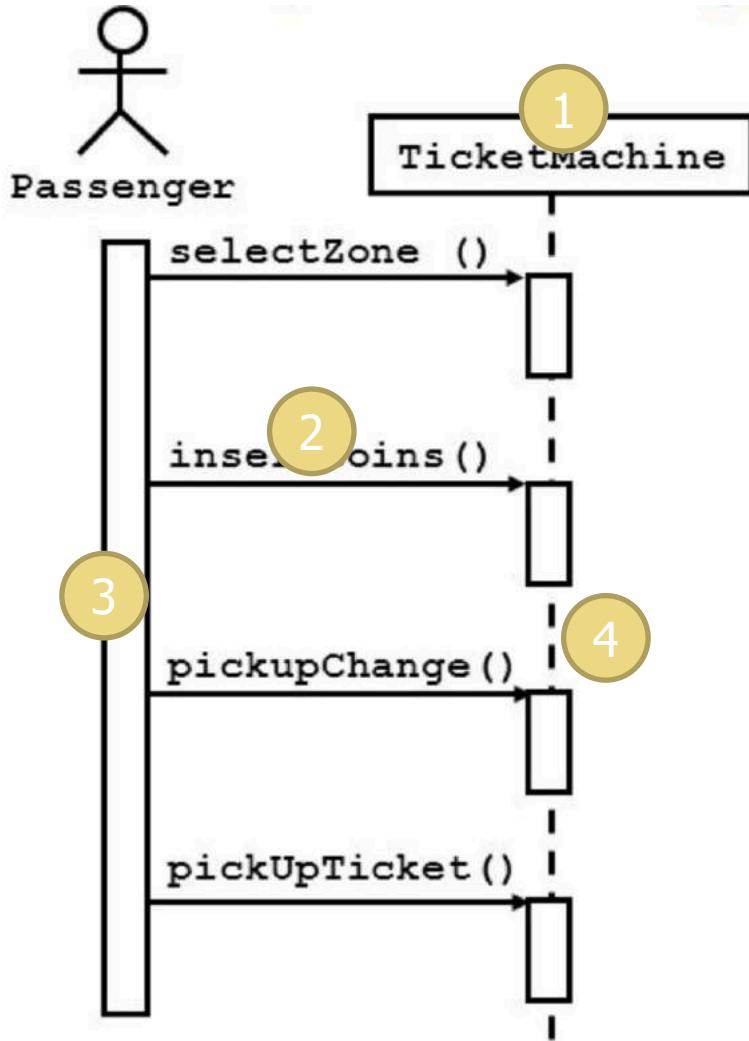
Sequence Diagram

- Captures how a group of objects interact or collaborate to achieve a use case scenario.
- Usually captures the dynamics of **a single scenario** and the **sequence of messages** that are passed between the objects.

- Emphasizes the **time ordering** of messages.



Sequence Diagram: Definitions



- Used during **requirement analysis**
 - To refine use case descriptions
 - Find additional/missing objects
- 1. **Objects** are represented by rectangles
- 2. **Messages** are represented by arrows
- 3. **Activations** are represented by narrow bars
- 4. **Lifelines** are represented by dashed lines

Example: EnterSecureFacility Use Case Description

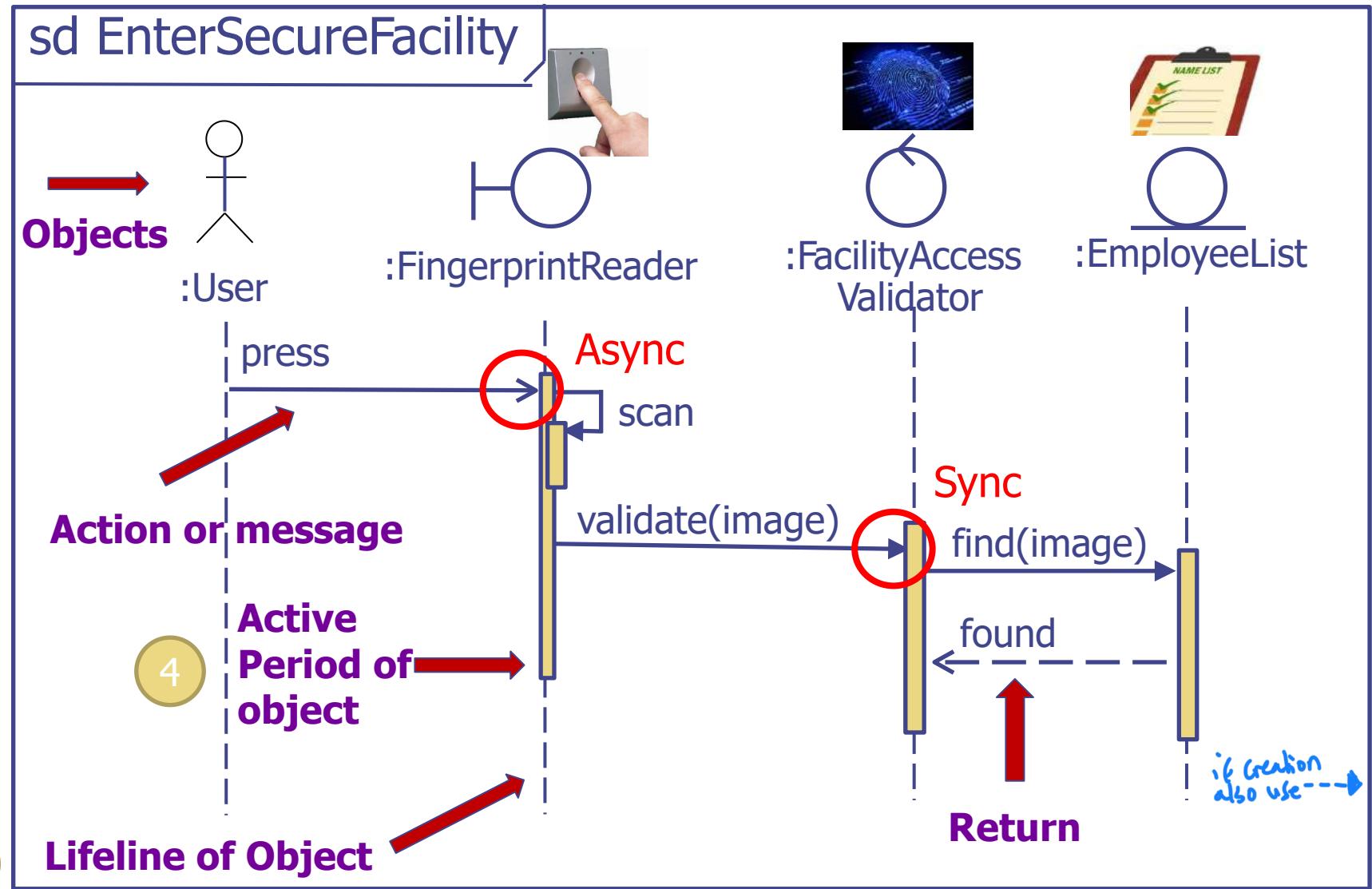
Flow of Events:

1. User presses on FingerprintReader.
2. FingerprintReader scans fingerprint and validates image with the FacilityAccessValidator.
3. The FacilityAccessValidator finds fingerprint in the EmployeeList.
4. The FacilityAccessValidator validates fingerprint and unlocks EntryGate.
5. ...

Drawing sequence diagram:
Identify important verbs



Example: EnterSecureFacility Sequence Diagram



Refine EnterSecureFacility Use Case Description

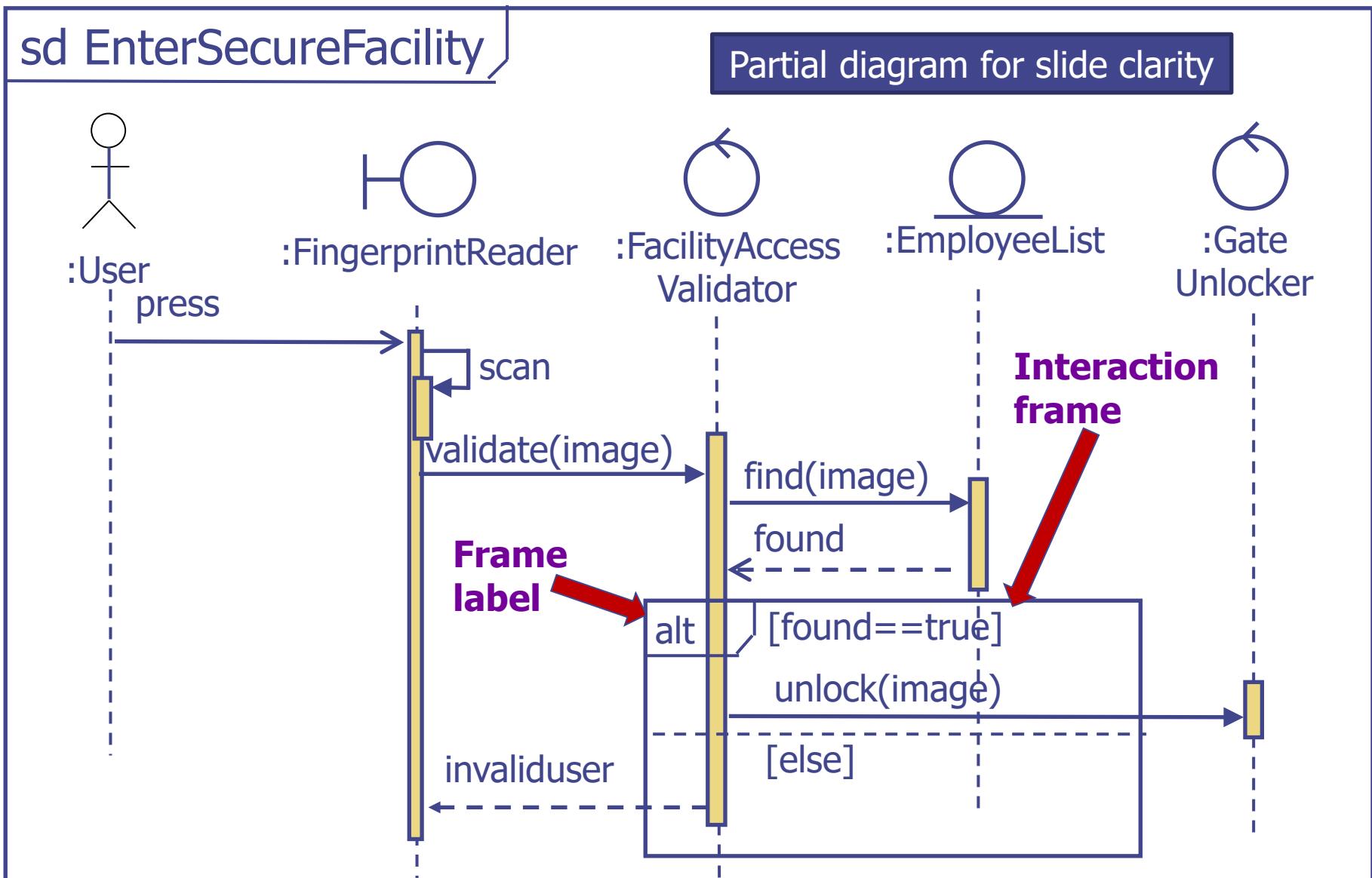
Flow of Events:

1. User presses on FingerprintReader.
2. FingerprintReader scans fingerprint and sends the image to FacilityAccessValidator.
3. The FacilityAccessValidator validates fingerprint with the scanned fingerprint image.
4. If the fingerprint image is found in the employee list, the FacilityAccessValidator unlocks EntryGate.
5. ...

Alternative Flows:

- AF-4. If the fingerprint image is not found in the employee list
1. FingerprintReader displays a "Please try again" message
 2. Go back to Step 1

Example: EnterSecureFacility Sequence Diagram



Types of Interaction Frames

sd sequence diagram

ref reference another sequence diagram

loop the actions loop until guard condition is false
(as in while, for)

opt/alt optional/alternative flow
(as in if-then, if-then-else, switch-case)

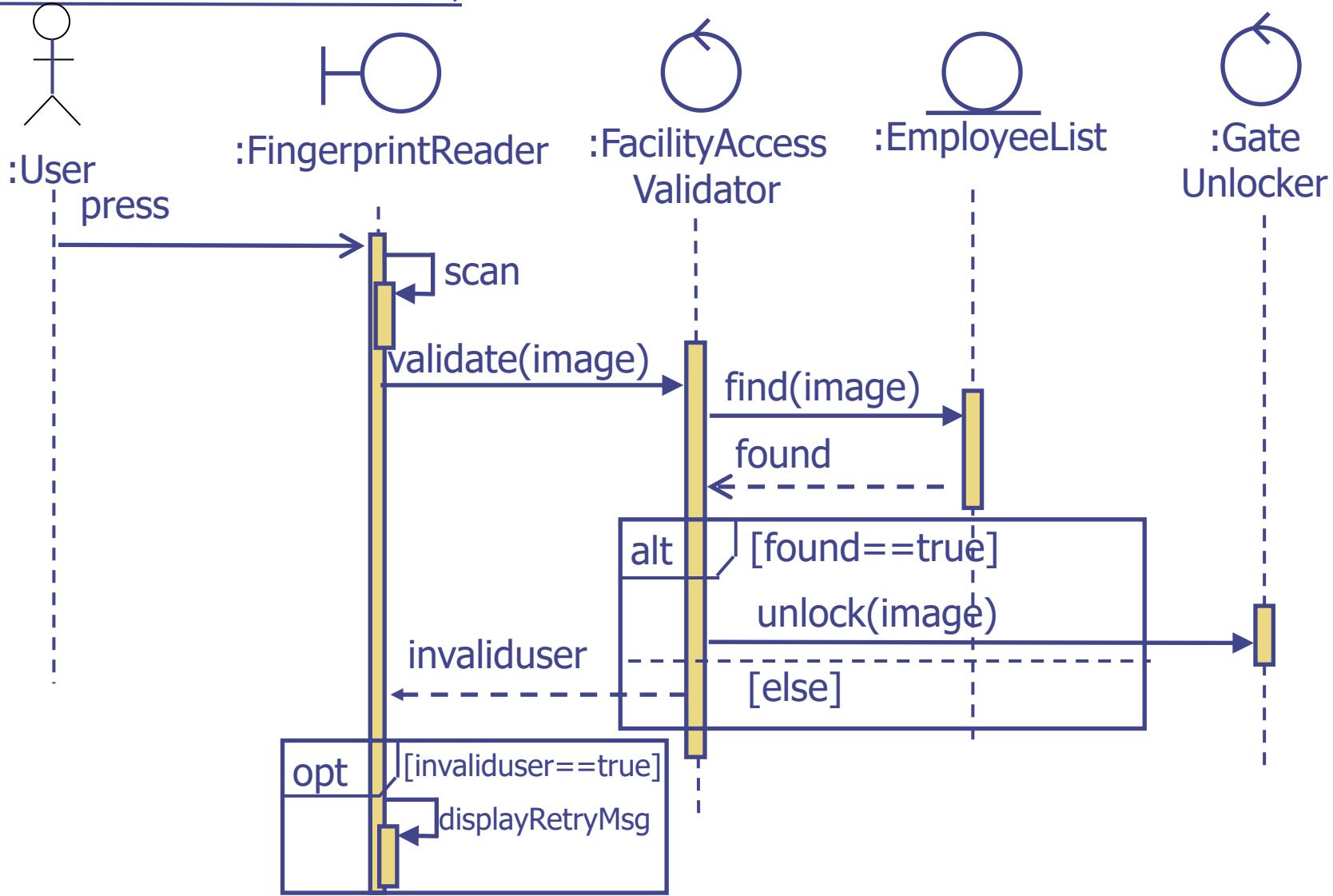
par each action is run in parallel

See UML cheatsheet for examples!

Refine EnterSecureFacility Sequence Diagram

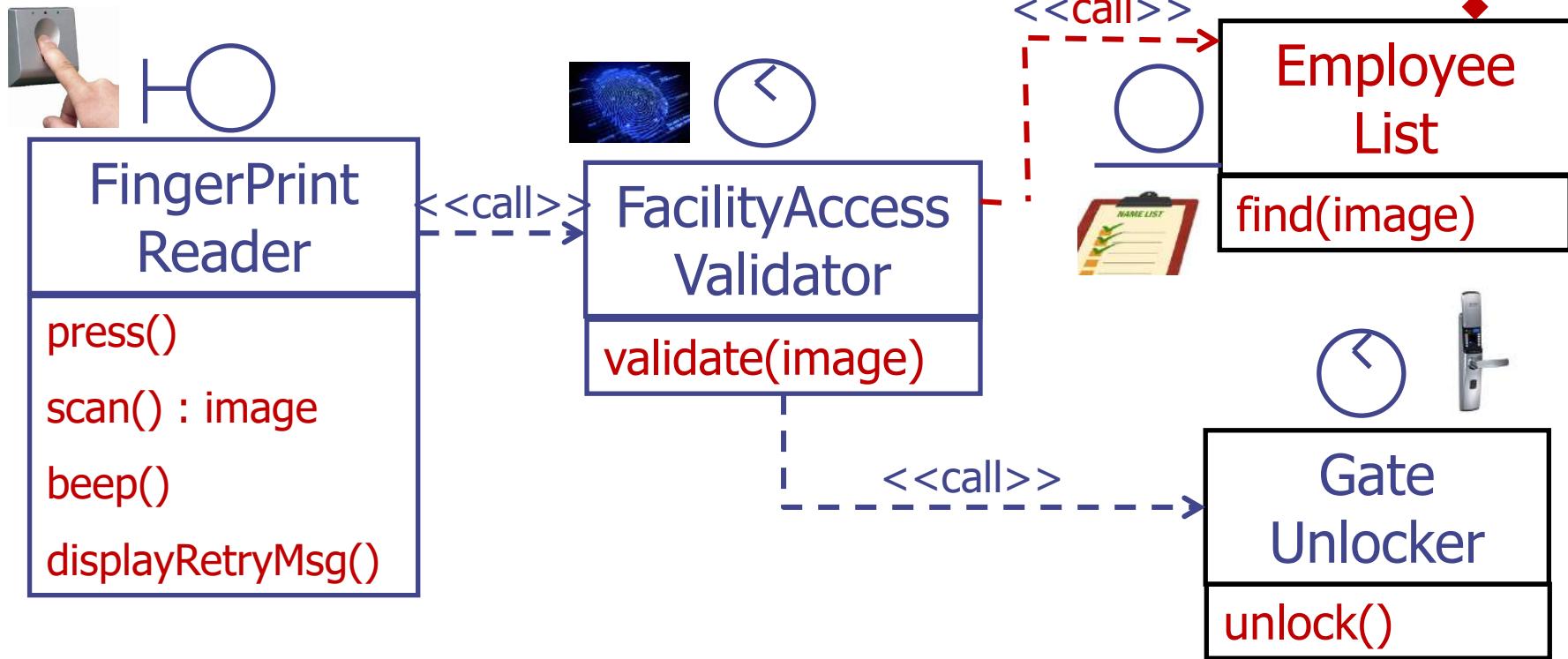
sd EnterSecureFacility

Partial diagram for slide clarity

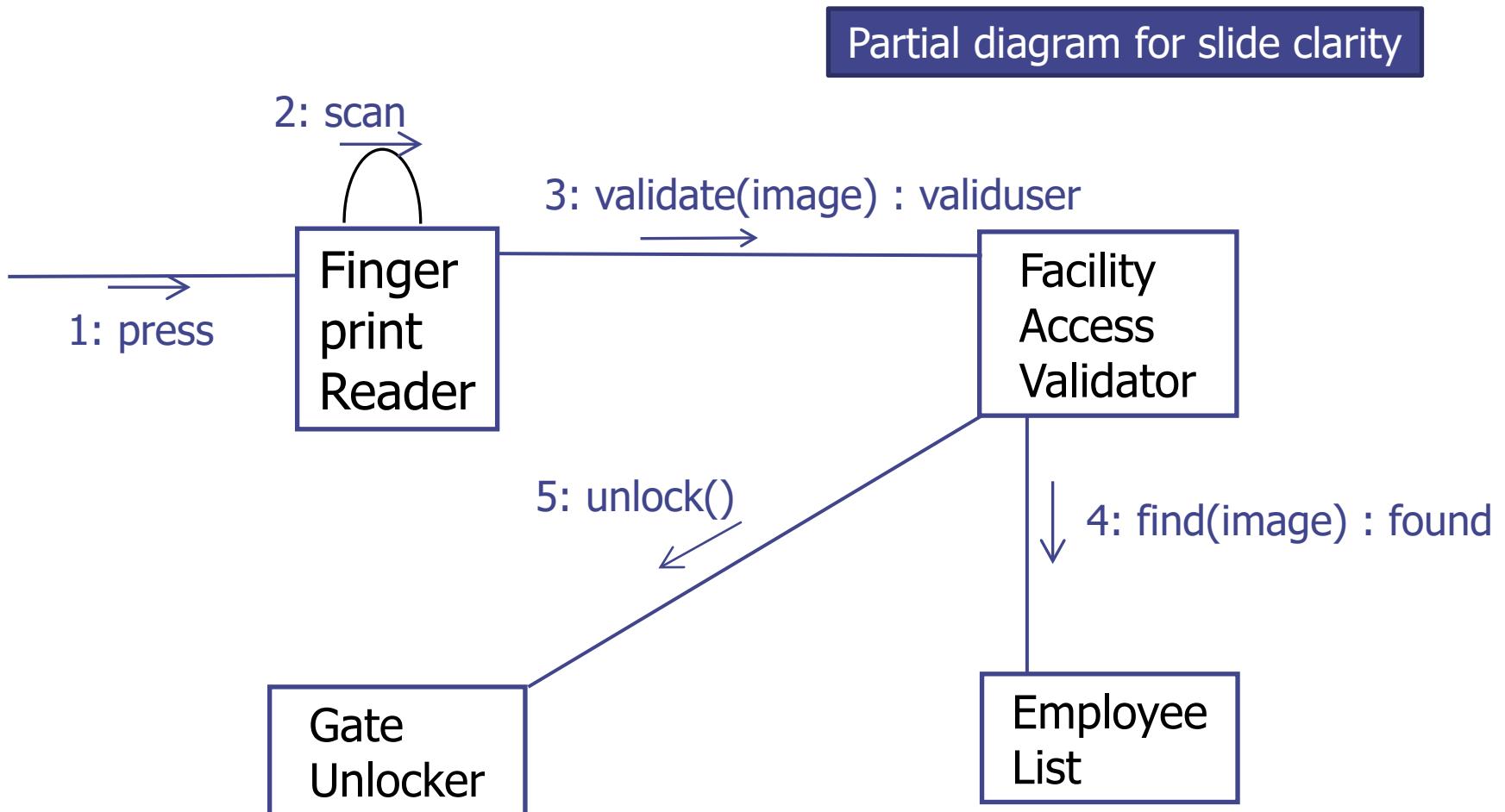


Refine Class Diagram

Partial diagram for slide clarity



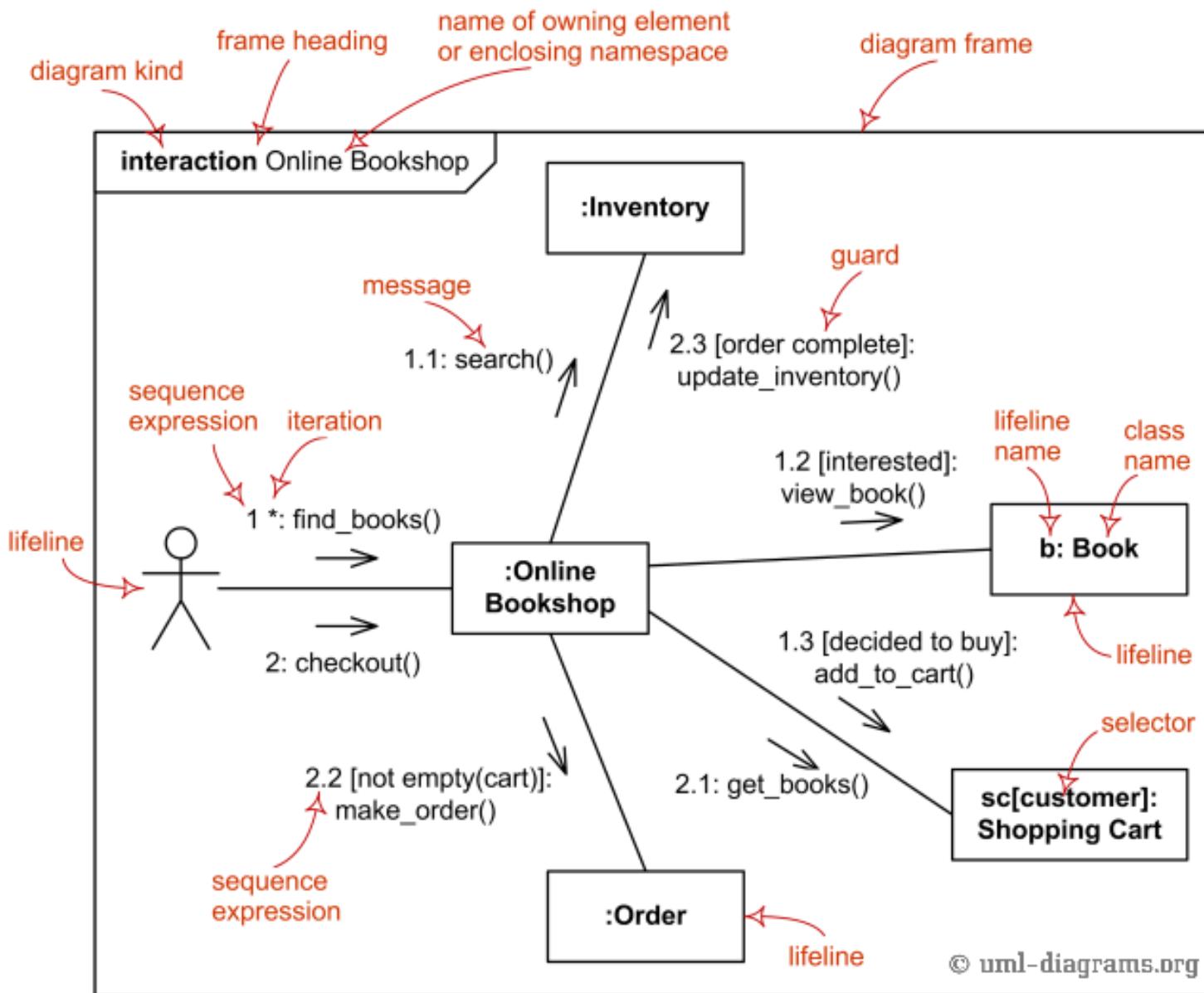
Example: EnterSecureFacility Communication Diagram



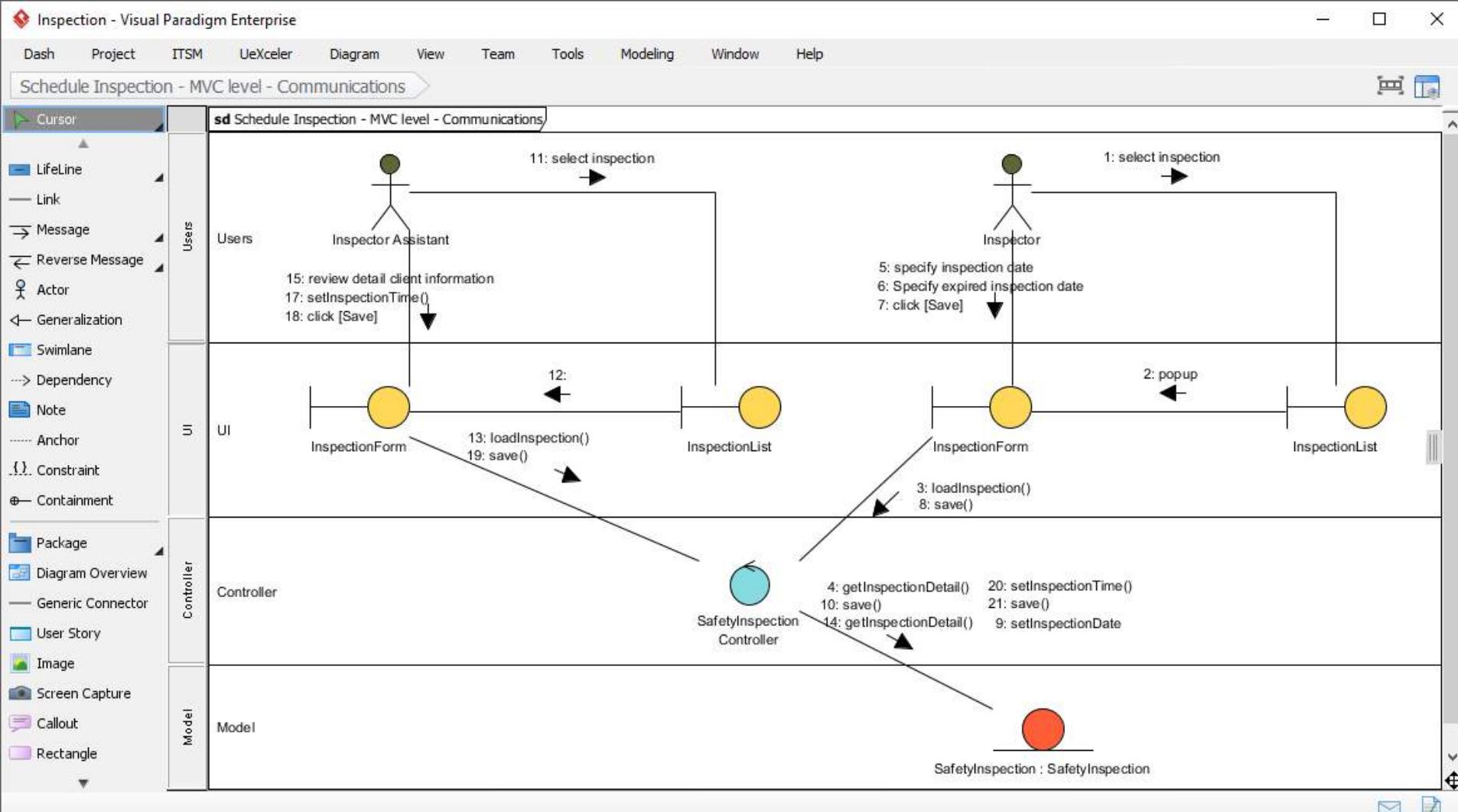
Communication diagram

- Models the interaction between objects or parts in terms of sequenced messages.
- Represents a combination of information taken from Class, Sequence, and Use-Case Diagrams describing both the static structure and dynamic behavior of a system.
- Messages are labeled with a chronological number.
- Compare with sequence diagram:
 - Communication diagrams show which elements each one interacts with better
 - Sequence diagrams show the order in which the interactions take place more clearly.

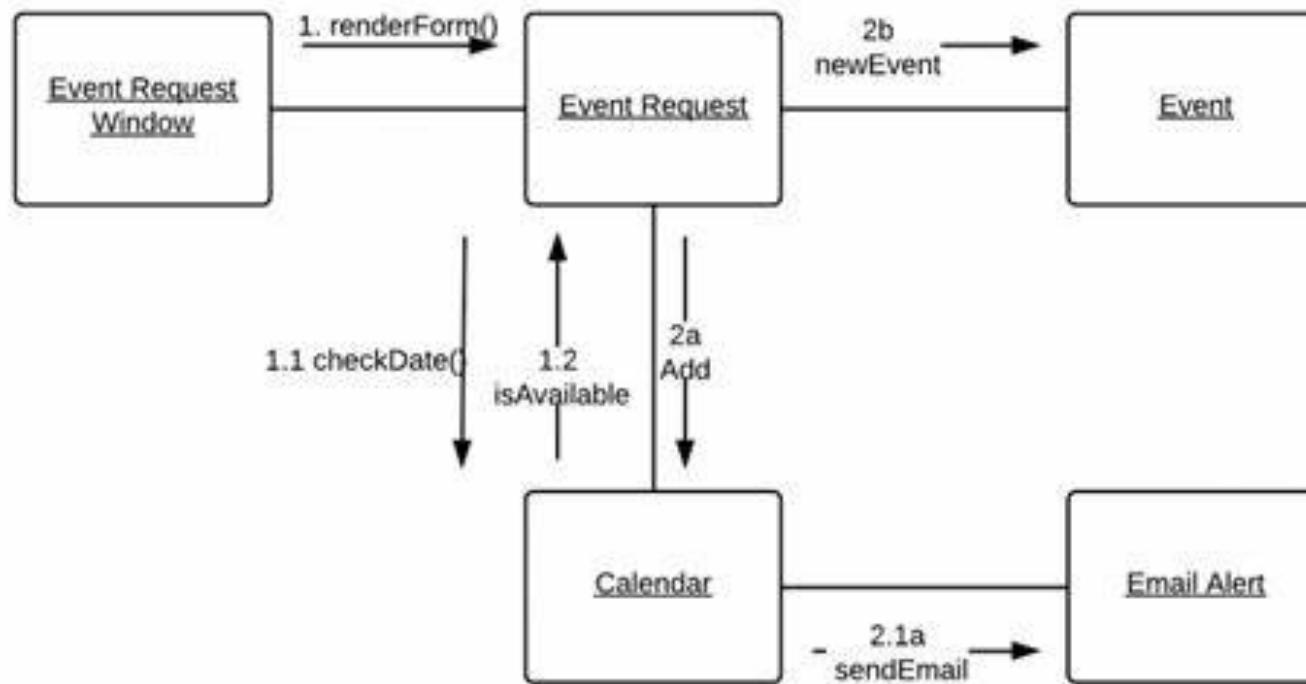
Communication diagram (example)



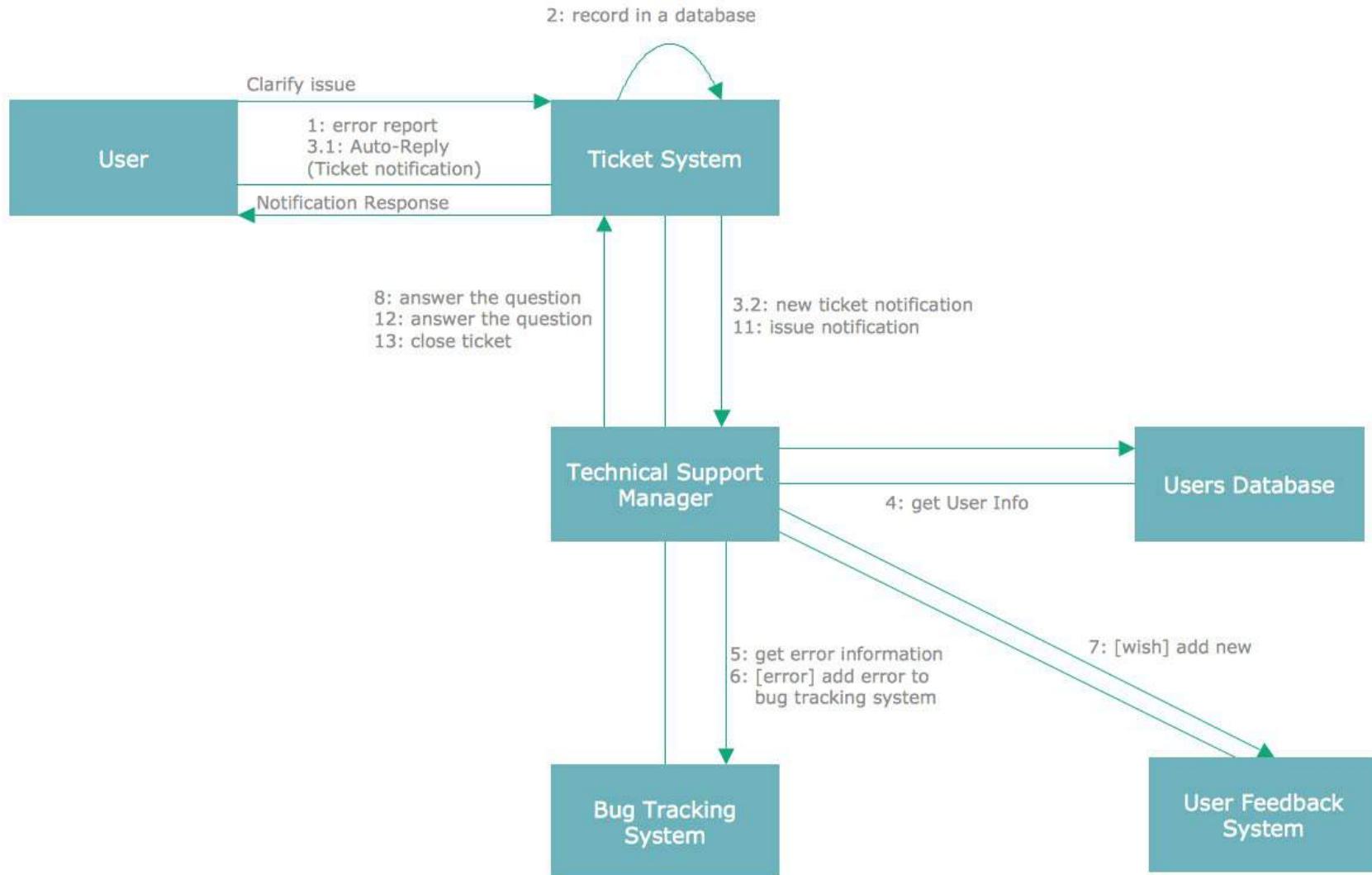
Communication diagram (example)



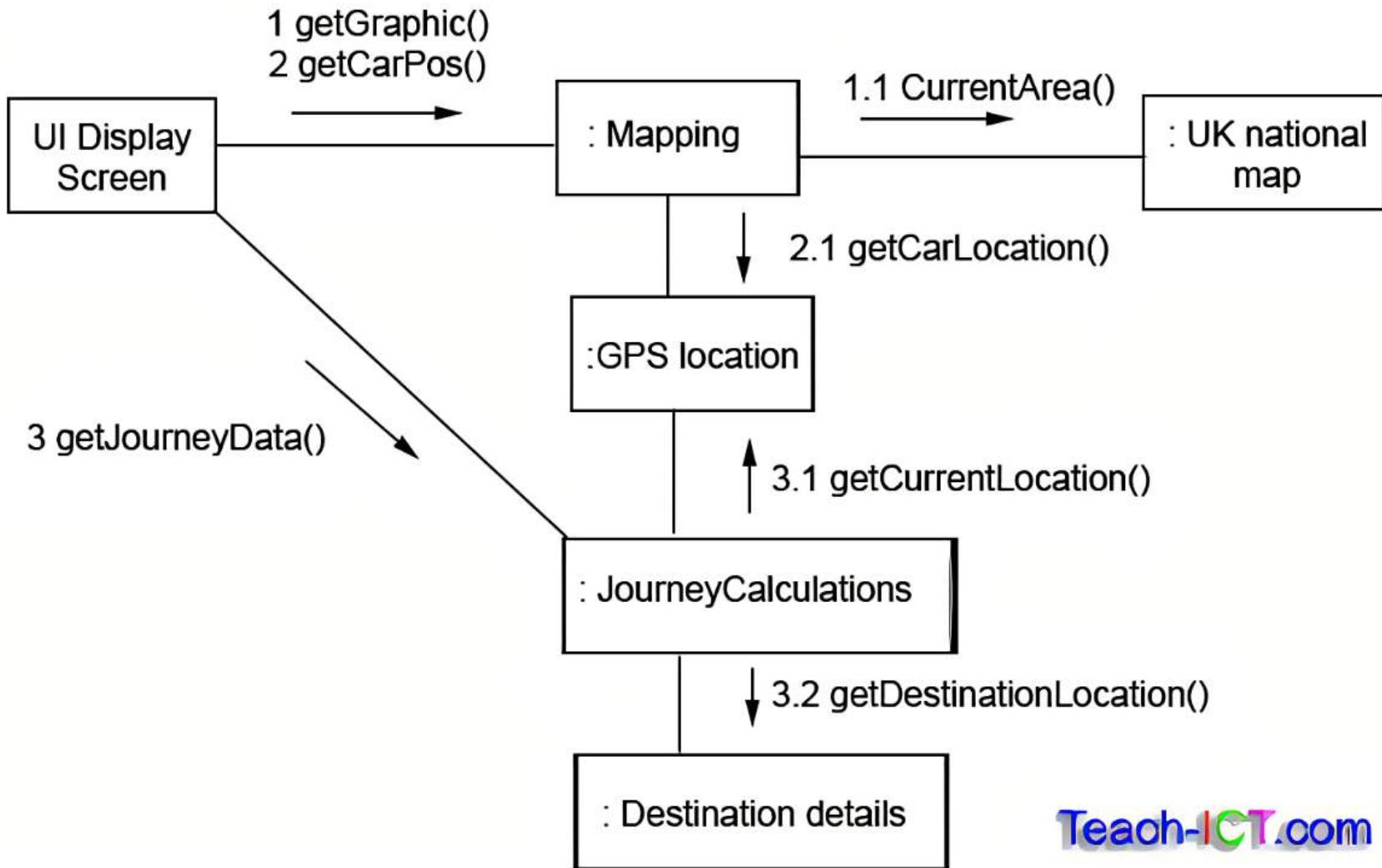
Communication diagram (example)



Communication diagram (example)



Communication diagram (example)



Summary

Sequence Diagrams are useful for:

Refining the Use Cases

- Identifying missing steps and flows
- Identifying unnecessary steps and flows

Refining the Class Diagrams (objects)

- Identifying missing classes
- Identifying unnecessary classes
- Identifying class operations
- Identify class attributes

Identifying all relationships between classes

Visual Paradigm can automatically switch between the two if you create the Sequence Diagram first.

Communication Diagrams serve a similar purpose by visualizing the sequence of messages differently.

In general, you need a sequence or communication diagram for each use case.

Summary

- Why sequence diagram?
 - Use cases are too wordy and also hard to read
 - Class diagram loses information on time ordering
 - Sequence diagram is more detailed than use case, not for non-technical communication in general.
- When to use sequence diagram?
 - Good at capturing interactions between objects
 - Use sequence diagram to look at the behaviour of **several objects within a single use case**.
 - Not so good at precise definition of behaviours
 - State diagrams (single object in many use cases).
 - Activity diagram (many use cases and many threads).
 - We will look at them later in the course.

Software Engineering

CE2006/CZ2006

Requirements Analysis: Dynamic Models 2

Discussion Topics

- Interaction Diagrams
 - State Machine Diagrams
 - Activity Diagrams
- Software Requirement Specifications
- Reading
 - Bruegge chap. 2.4.3, 2.4.4, 2.4.5
 - Fowler chaps. 4, 10 and 12

State Machine Diagram

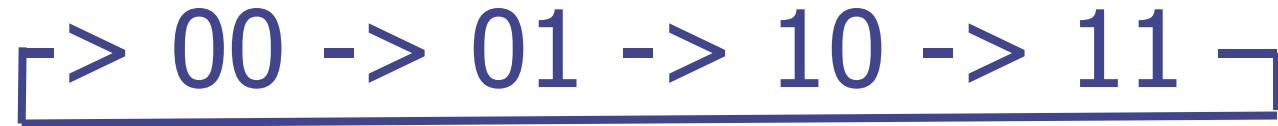
Also known as **Statecharts**, **Finite State Automata**.

Models **a system** or **an object** based on the "**States**" it can be in.

A "**State**" is a stable condition of a system, that can exist for a period of time, or for all time, until some **ACTION** drives the system into another stable state.

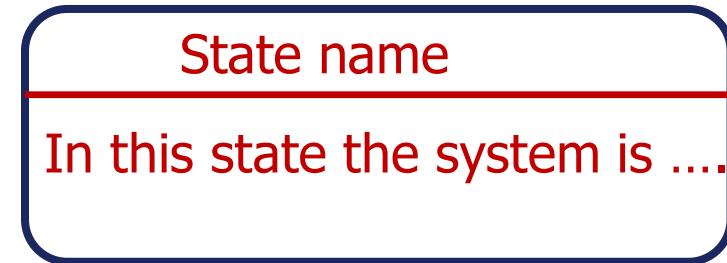
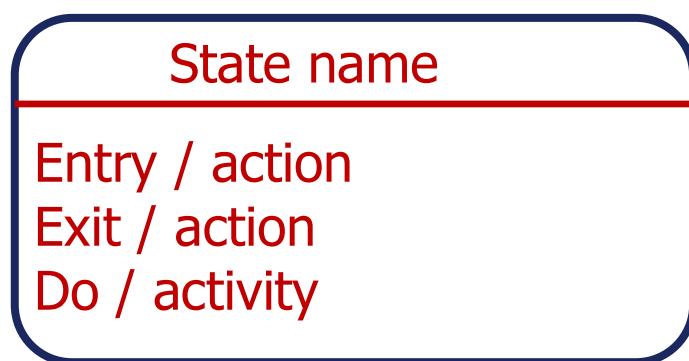
Most systems will have a **finite number of states** in which it can be depending on the previous state and actions which move it to another state.

- e.g. a 2-bit binary counter has 4 states and moves from one state to the next when a clock signal drives it to the next state.



State Machine Diagram

In UML, a State is represented as a **rectangle** with **rounded corners** with a **name** at the top and **optionally** a **description** of the state and/or **actions and activities** performed while in that state.

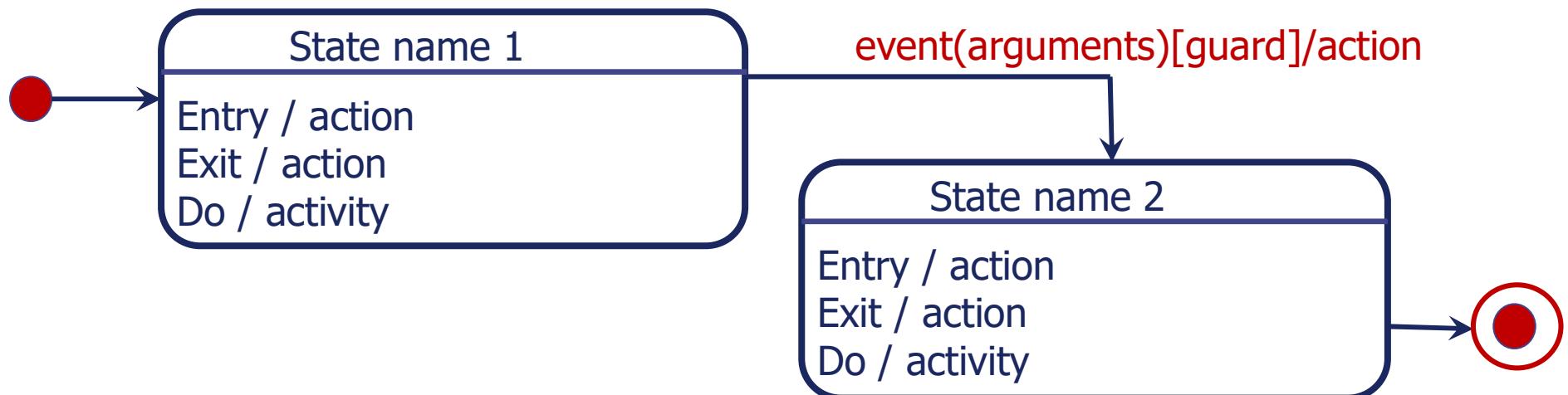


State Machine Diagram

A state machine diagram begins with a **start node** followed by a series of states connected by **transitions** and finally an **end node** (unless the system is in an infinite loop and there is no end node).

A transition from one state to the next is assumed to be instantaneous in response to an event. A transition can have **optionally**:

- event: event triggers the transition
- guard: the condition which must be true to take the transition
- action: action to do during the transition



Dialog Map: State Machine Diagrams in Prototyping

Prototyping is a technique to reduce the risk of building the wrong product, or of building the right product badly.

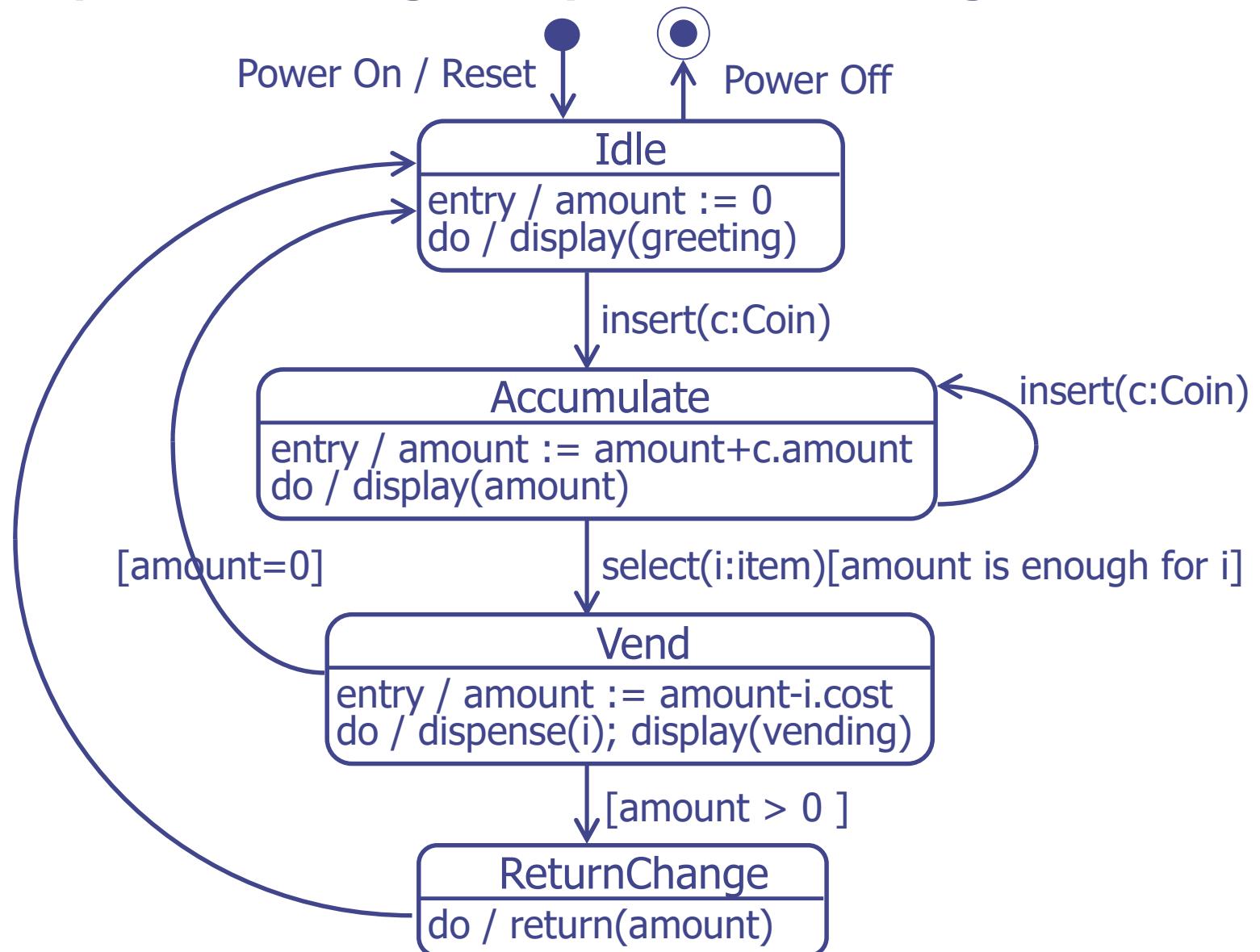
Using a State Machine Diagram and a prototype users can **experiment** with the **navigation options** and **preview** the **functionality** available at each screen to assess the look and feel of the system.

Many **user interfaces** can be thought of as **finite state machines**, with only one display screen or window active at a time (a state), and defined conditions for moving from one state to another (a transition).

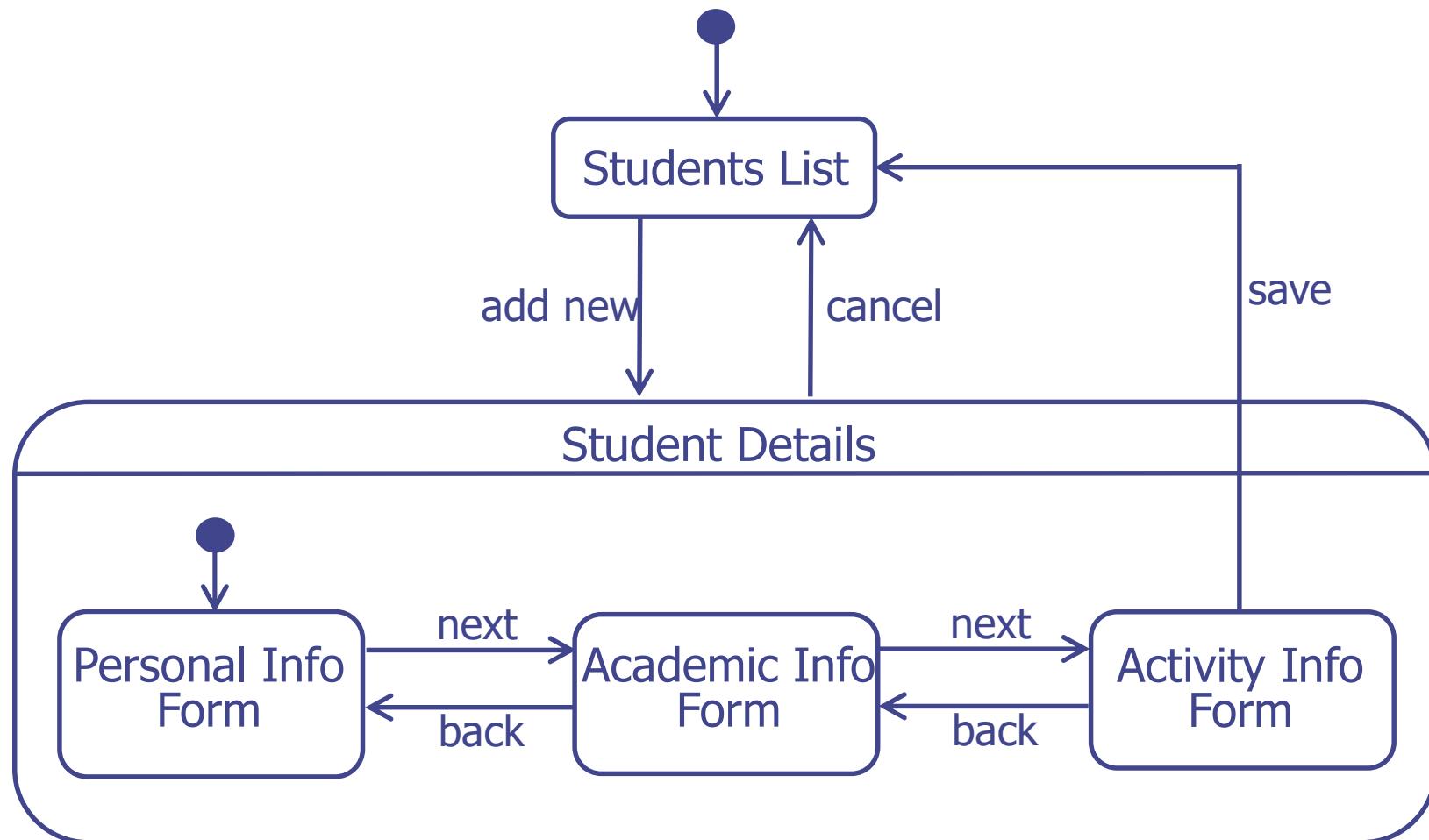
Such diagrams are sometimes called **dialog maps**.

- The dialog map helps us **find missing or incorrect navigation pathways early**, when they are cheap to fix.
- We can also spot opportunities for reuse and redundancies in the user interface, without worrying about the details of screen design prematurely.

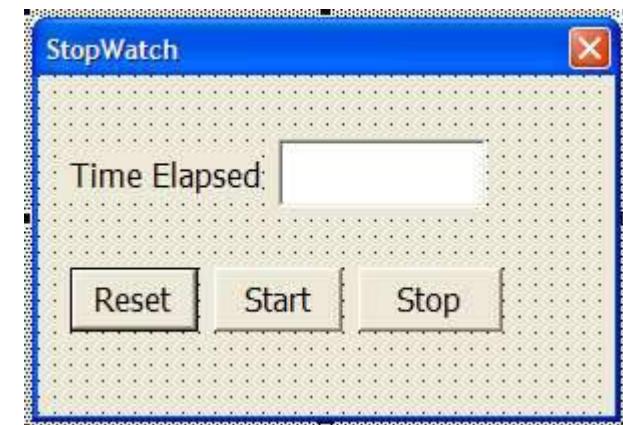
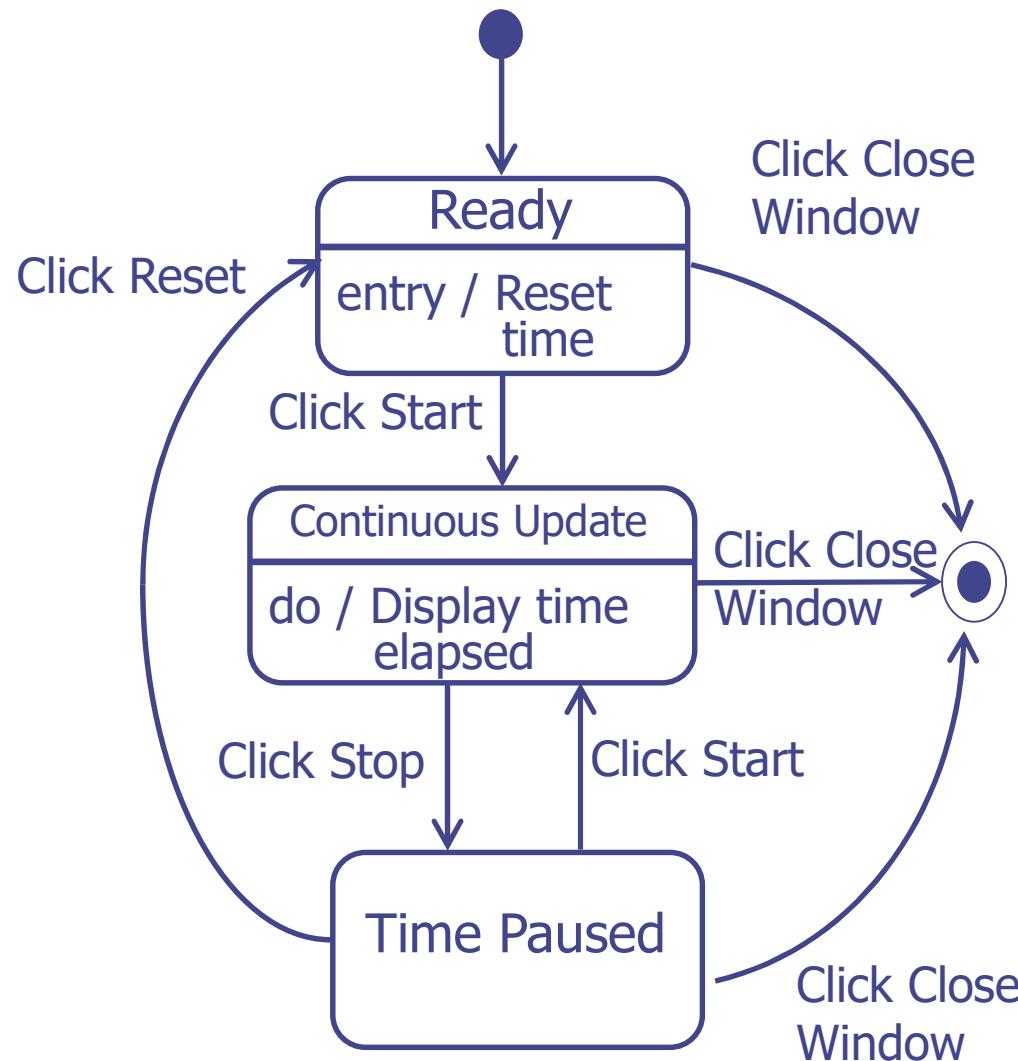
Example: Dialog Map of Vending Machine



Example: Dialog Map of Student Management System



Practice: Stopwatch Functionality



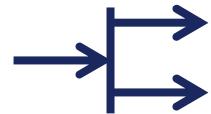
UML Activity Diagram



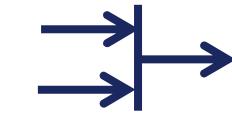
- Activities are the execution of one or more basic operations. Represented by a rounded rectangle.



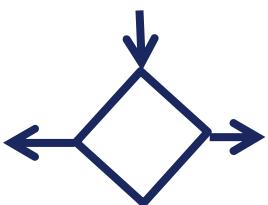
- Control flow between activities represented by arrows.



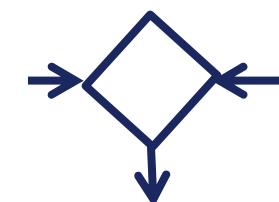
- Forks are when control flow has one input and two or more outputs (go to concurrent activities).



- Joins are when two or more control flows come together into a single control flow.



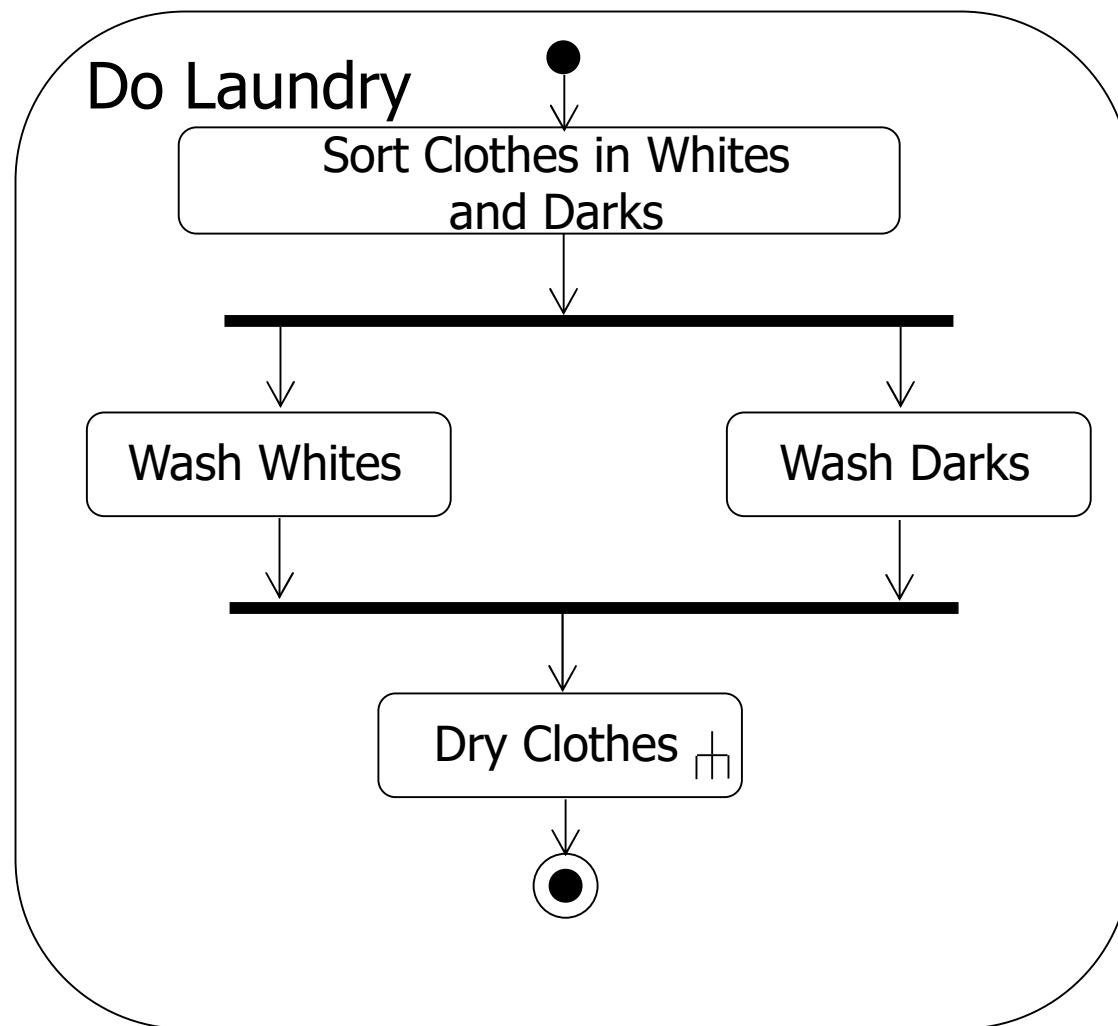
- Decision boxes (branches) are when control flow can go in one of several directions depending on a condition.



- Merge several branches into one branch

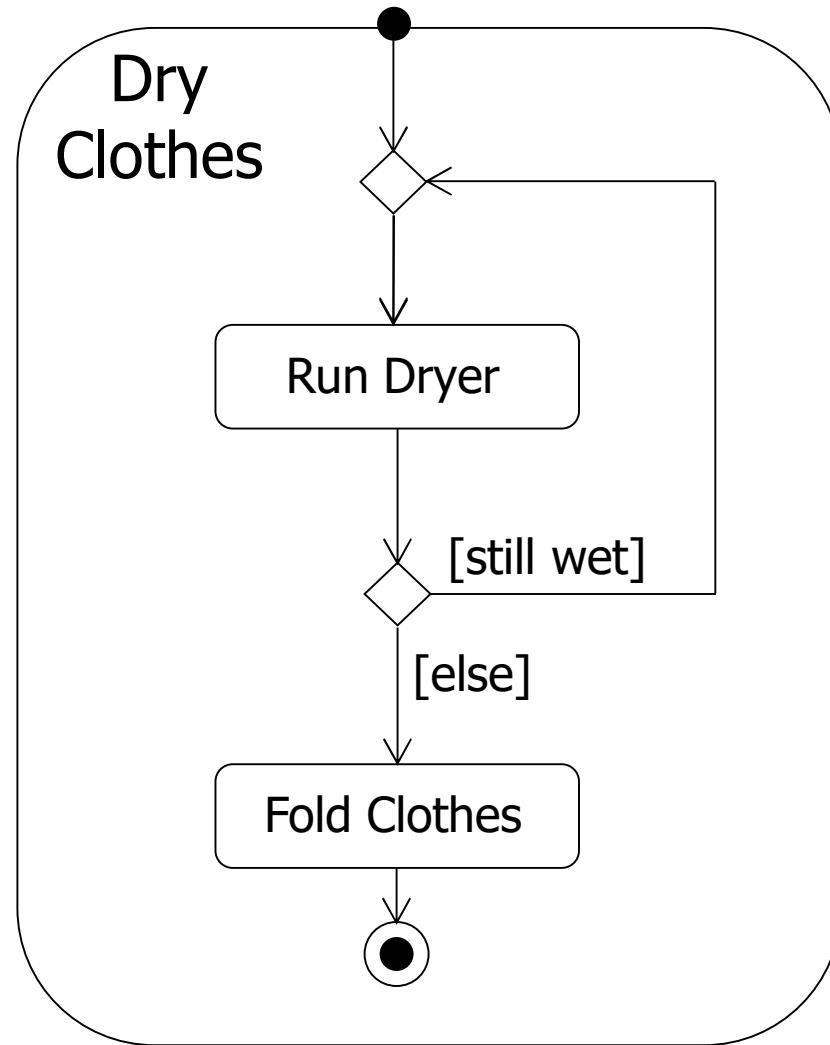
Example:

Activity Diagram of Do Laundry

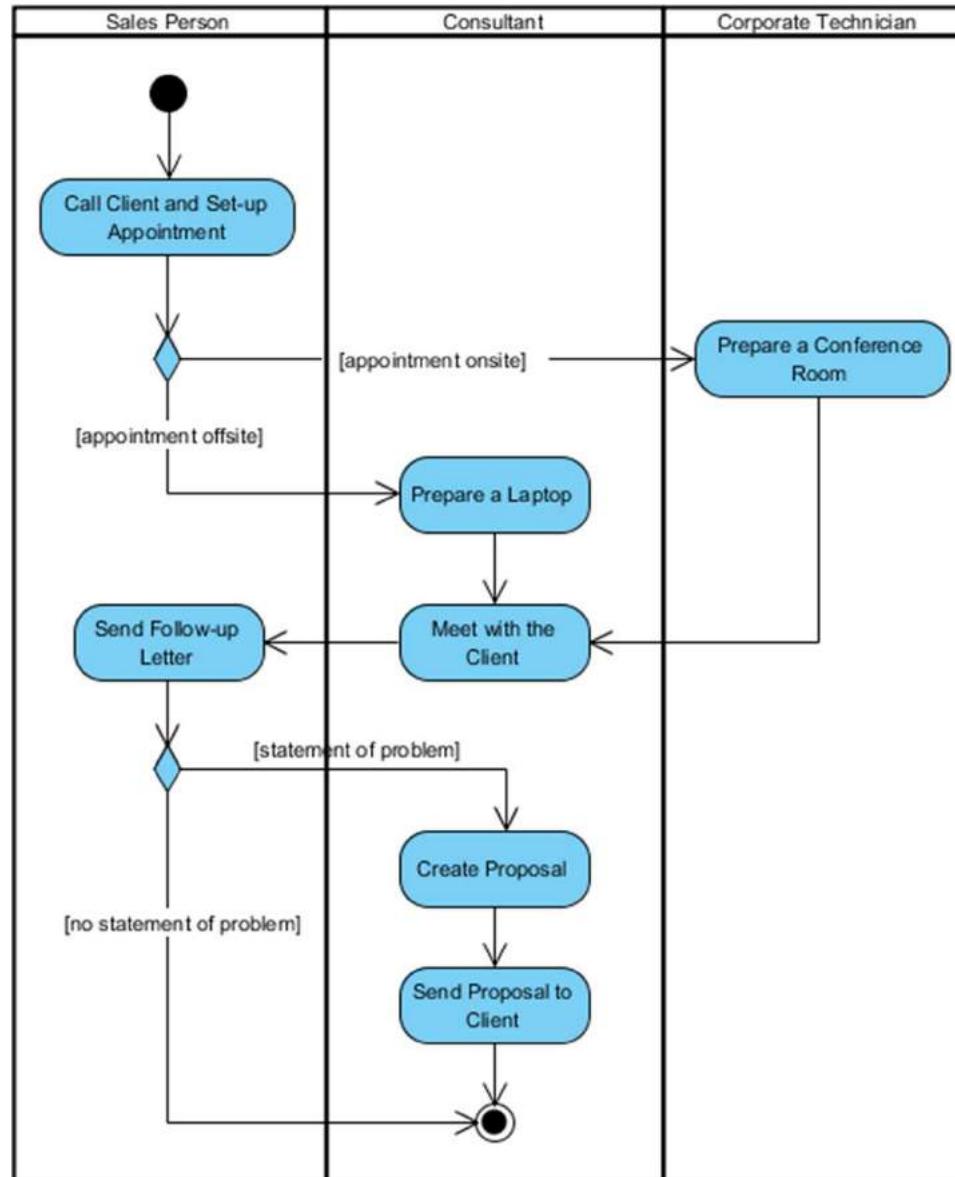


Example:

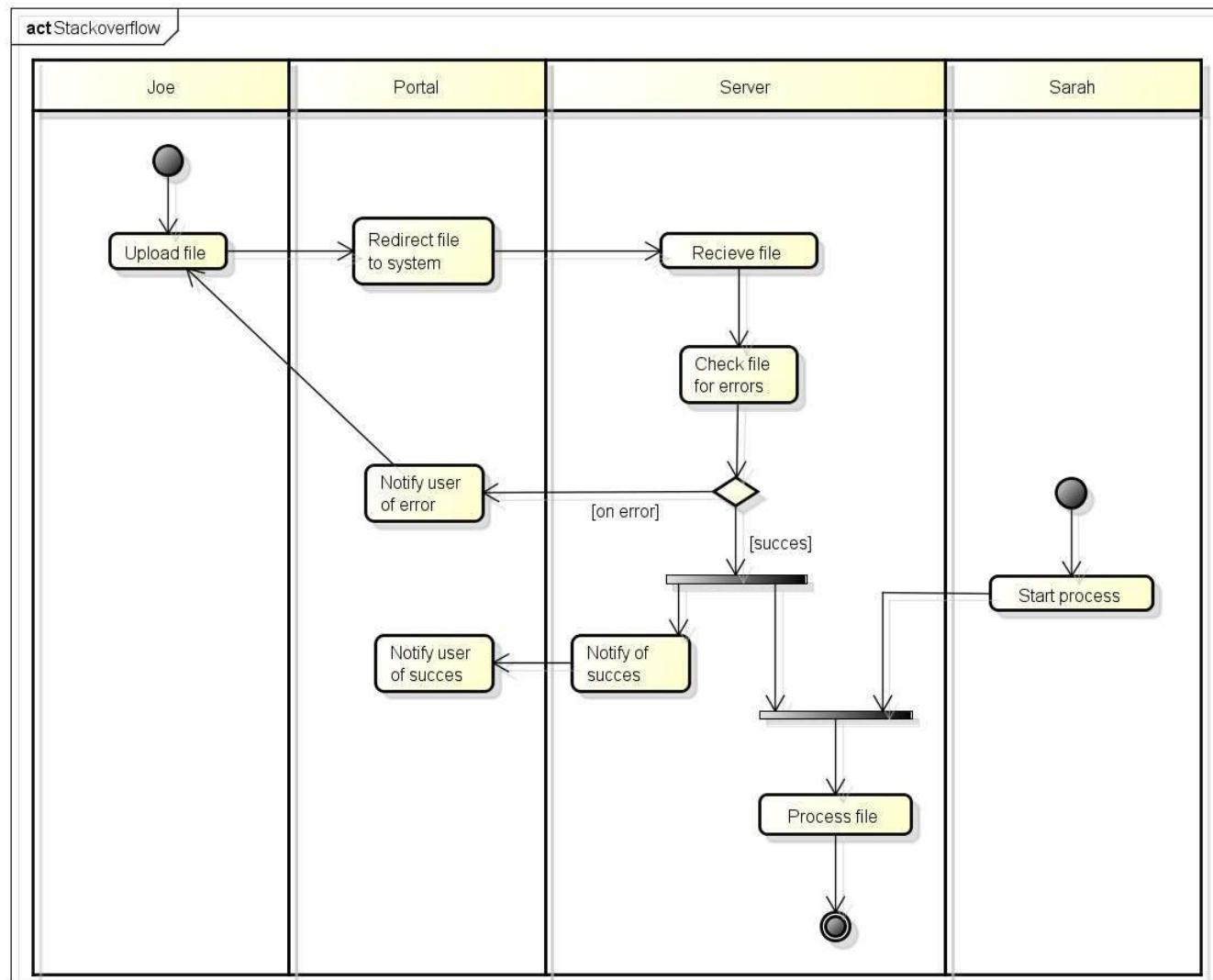
Activity Diagram of Dry Clothes



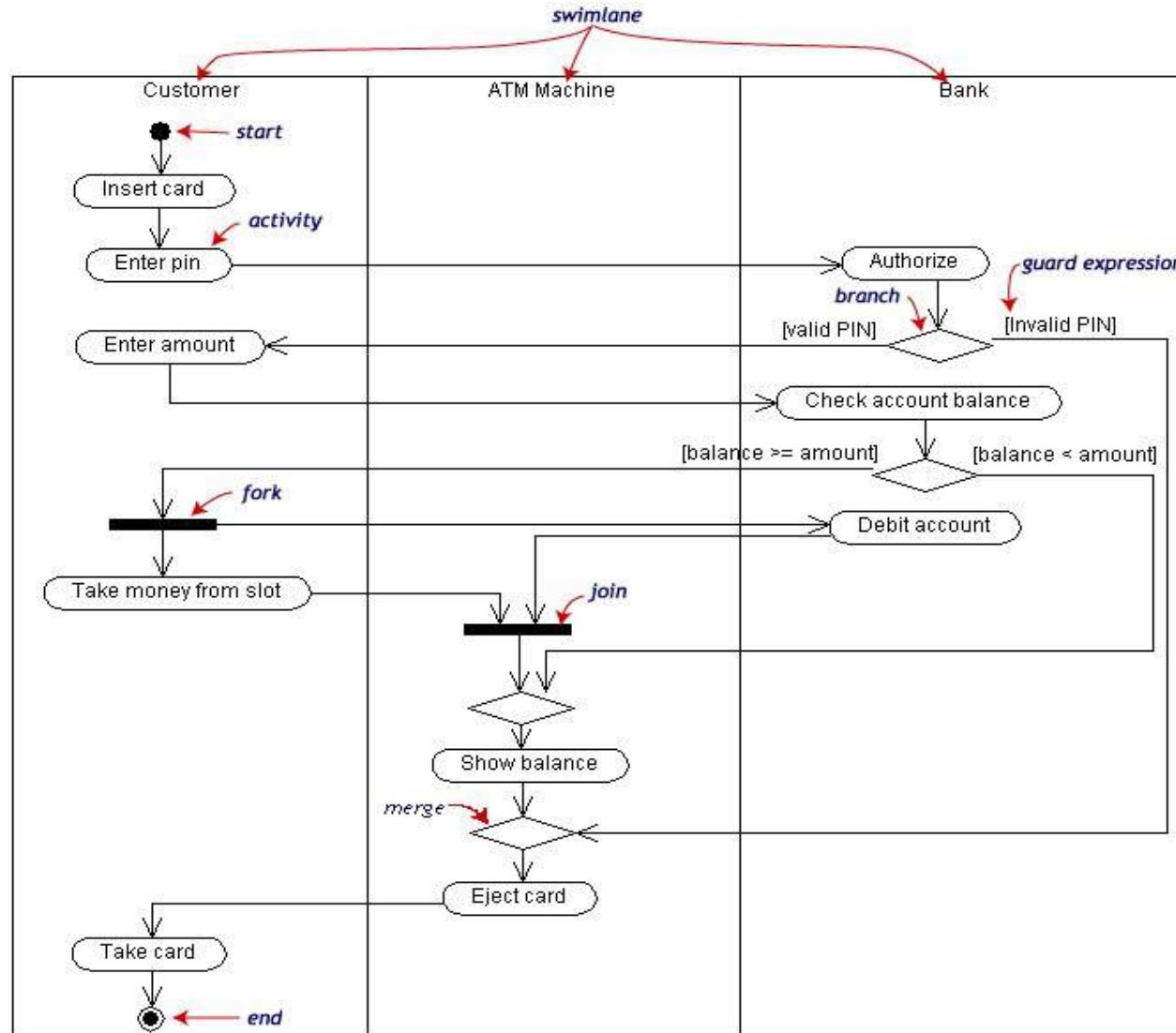
More Examples



More Examples



Example of Activity Diagram

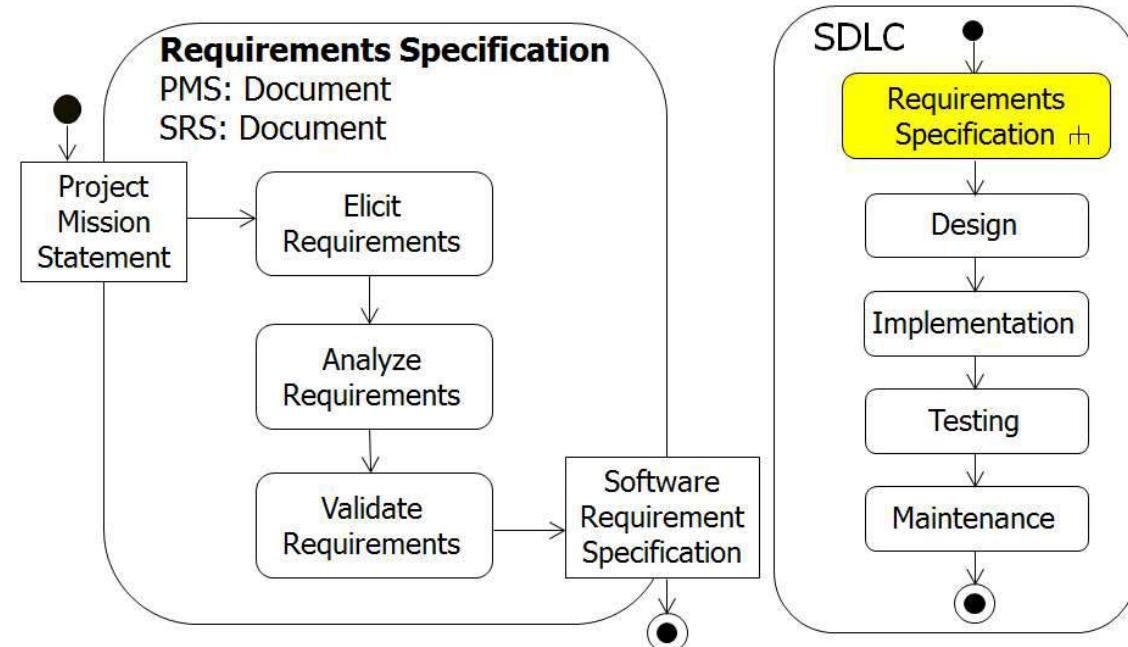


Difference between State Machine Diagram and Activity Diagram

Activity diagrams are used as a flow chart of **activities** performed by the system. They describe the **workflow behavior** of a system in terms of activities.

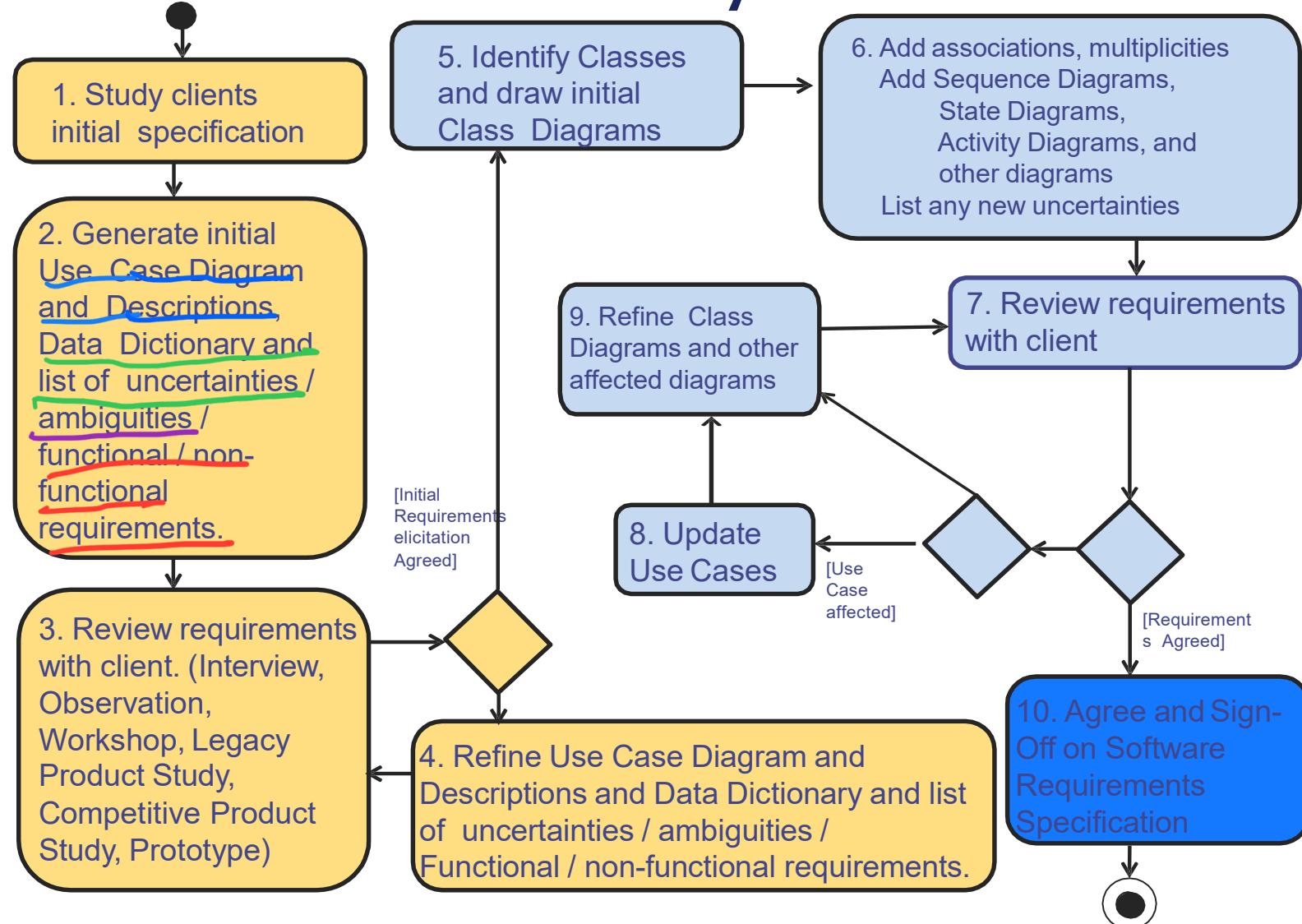
State machine diagram models **a system or an object** based on the **"States"** it can be in.

We have already used
Activity Diagrams

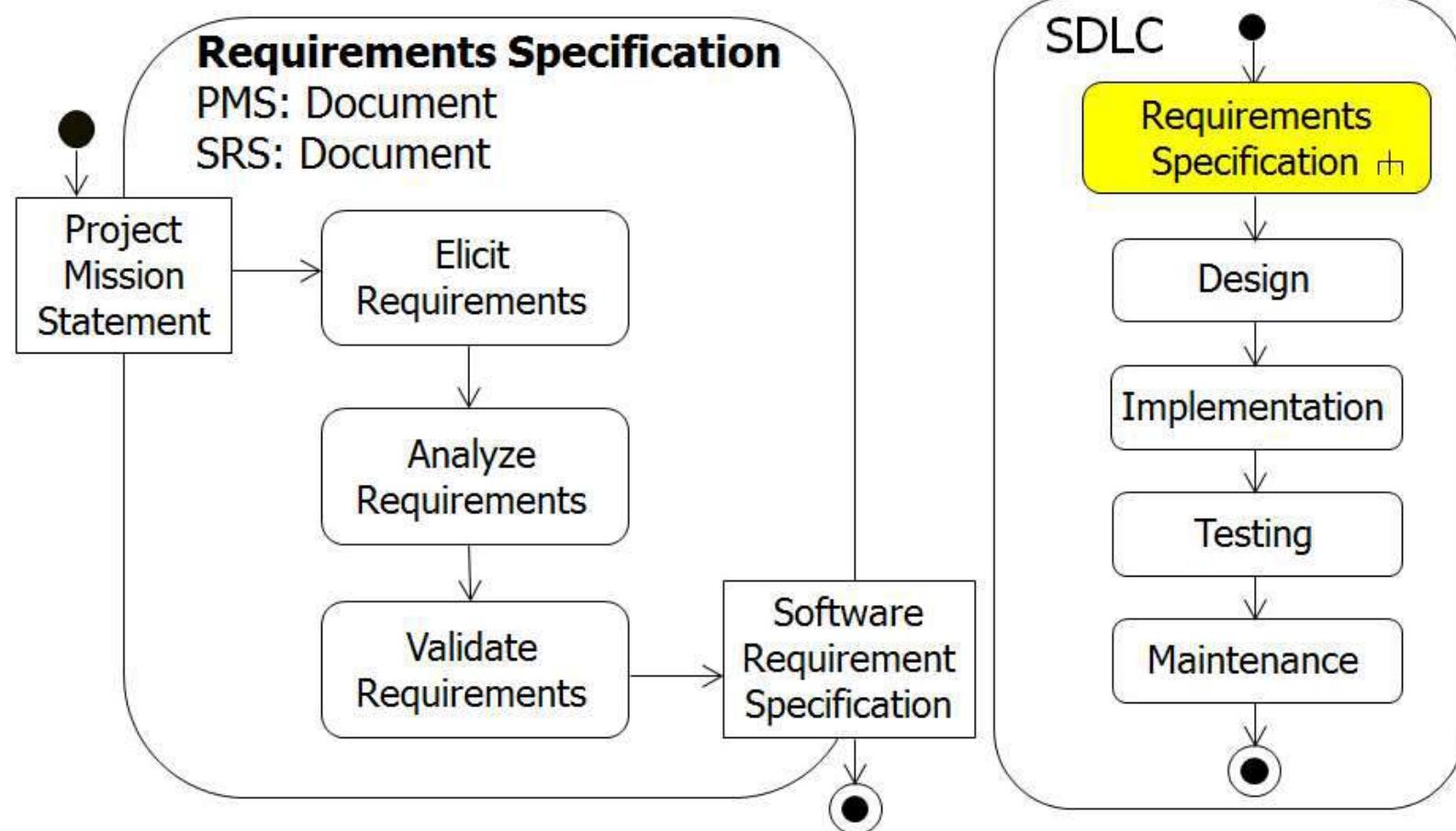


Review of Requirement Elicitation and Analysis Process

Activity Diagram of Requirement Elicitation and Analysis



Software Requirement Specification (SRS)



Contents of the SRS

1. Each organisation will have its own template for an SRS.
2. There are key content that must be addressed in an SRS
3. There is an IEEE standard

830-1998 - IEEE Recommended Practice for Software Requirements Specifications.

This is a document that you can download from IEEEXplore.

How and Who should write the SRS?

1. It should be written in **natural language**, in an **unambiguous** manner that may also include diagrams as necessary.
2. At the same time the detailed functional and non-functional **requirements need to be defined and understood** by the development team.
3. **Development team** members (programmers and managers) are **seldom the best writers of formal documents**. However, they often have to write it!
4. It is good practice to involve a **technical writer** with the development team in the drafting of the SRS because they are usually better at assessing and planning documentation projects and better meet customer document needs.

Contents of the SRS

- 1) Product Description**
 - a) Purpose of the System (mission statement)**
 - b) Scope of the System**
 - c) Users and Stakeholders**
 - d) Assumptions and Constraints**
- 2) Functional Requirements**
- 3) Non-Functional Requirements**
- 4) Interface Requirements**
 - a) User**
 - b) Hardware**
 - c) Software**
- 5) Data Dictionary**

See detailed SRS description and SRS template on NTULearn course website.

Summary

- Interaction Diagrams
 - State Machine Diagrams
 - Activity Diagrams
- Software Requirement Specifications

Software Engineering

CE2006/cz2006

Software Processes

Topics covered

- ✧ Software process models
- ✧ Process activities

The software process

- ✧ A structured set of activities required to develop a software system.
- ✧ Many different software processes but all involve:
 - Specification – defining what the system should do;
 - Design and implementation – defining the organization of the system and implementing the system;
 - Validation – checking that it does what the customer wants;
 - Evolution – changing the system in response to changing customer needs.
- ✧ A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

Software process descriptions

- ✧ When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- ✧ Process descriptions may also include:
 - Products, which are the outcomes of a process activity;
 - Roles, which reflect the responsibilities of the people involved in the process;
 - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.

Plan-driven and agile processes

- ✧ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ✧ In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- ✧ In practice, most practical processes include elements of both plan-driven and agile approaches.
- ✧ There are no right or wrong software processes.

Software process models

Software process models

✧ The waterfall model

- Plan-driven model. Separate and distinct phases of specification and development.

✧ Incremental development

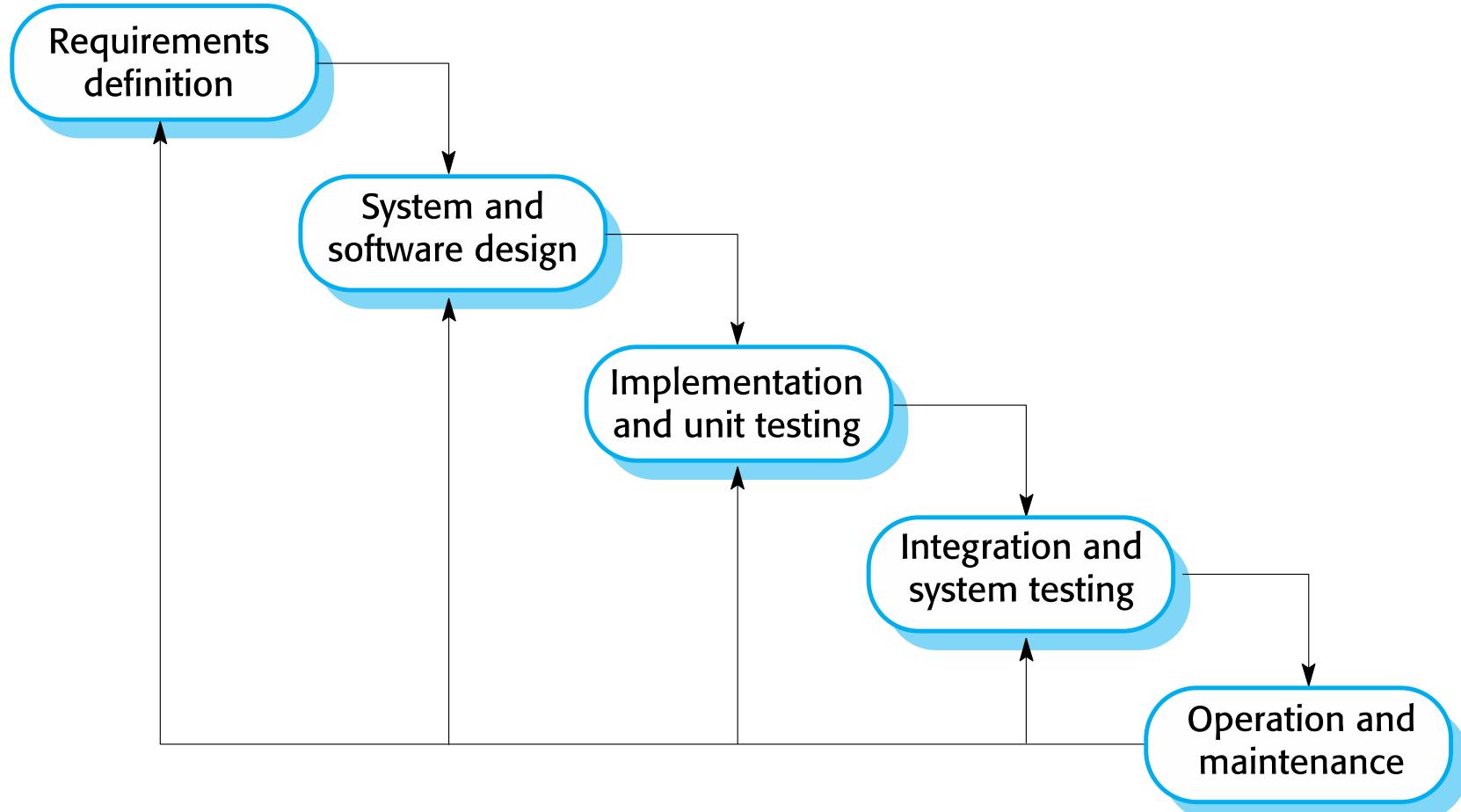
- Specification, development and validation are interleaved. May be plan-driven or agile.

✧ Integration and configuration

- The system is assembled from existing configurable components. May be plan-driven or agile.

✧ In practice, most large systems are developed using a process that incorporates elements from all of these models.

The waterfall model



Software Processes (adapted from:
Software Engineering by Ian Sommerville)

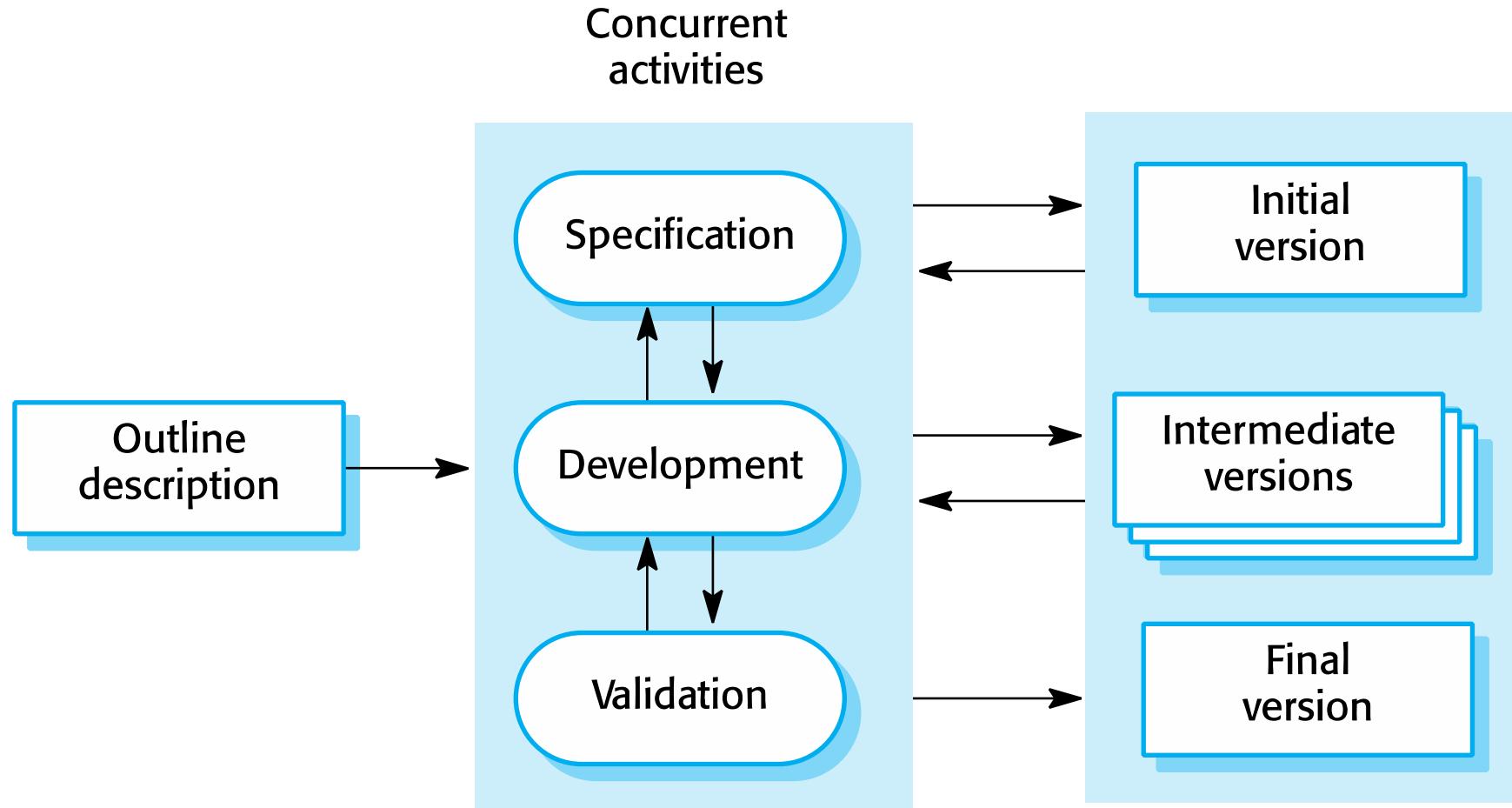
Waterfall model phases

- ✧ There are separate identified phases in the waterfall model:
 - Requirements analysis and definition
 - System and software design
 - Implementation and unit testing
 - Integration and system testing
 - Operation and maintenance
- ✧ The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

Waterfall model problems

- ✧ Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
 - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
 - Few business systems have stable requirements.
- ✧ The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
 - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

Incremental development



Incremental development benefits

- ✧ The cost of accommodating changing customer requirements is reduced.
 - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- ✧ It is easier to get customer feedback on the development work that has been done.
 - Customers can comment on demonstrations of the software and see how much has been implemented.
- ✧ More rapid delivery and deployment of useful software to the customer is possible.
 - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental development problems

✧ The process is not visible.

- Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.

✧ System structure tends to degrade as new increments are added.

- Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

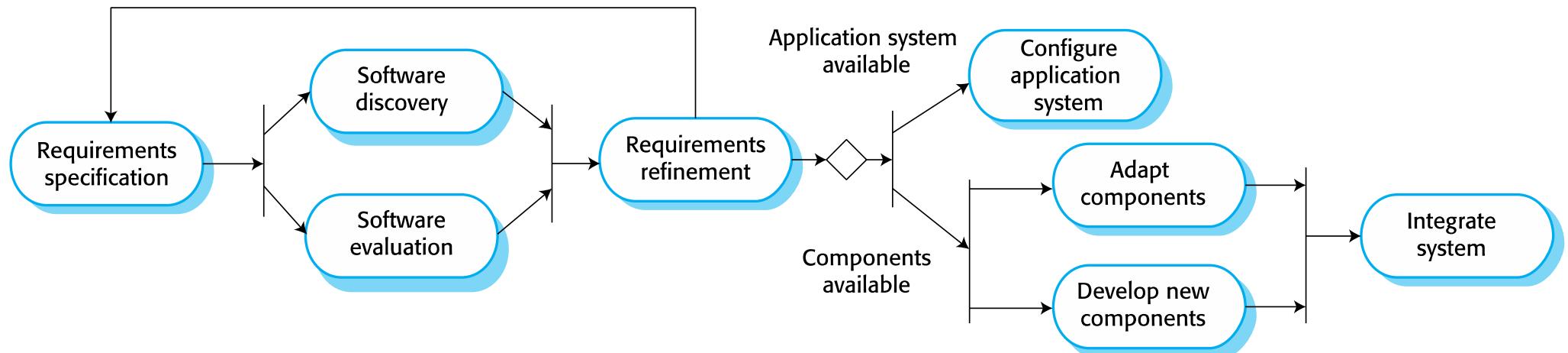
Integration and configuration

- ✧ Based on software reuse where systems are integrated from existing components or application systems (sometimes called COTS -Commercial-off-the-shelf) systems).
- ✧ Reused elements may be configured to adapt their behaviour and functionality to a user's requirements
- ✧ Reuse is now the standard approach for building many types of business system

Types of reusable software

- ✧ Stand-alone application systems (sometimes called COTS) that are configured for use in a particular environment.
- ✧ Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- ✧ Web services that are developed according to service standards and which are available for remote invocation.

Reuse-oriented software engineering



Key process stages

- ✧ Requirements specification
- ✧ Software discovery and evaluation
- ✧ Requirements refinement
- ✧ Application system configuration
- ✧ Component adaptation and integration

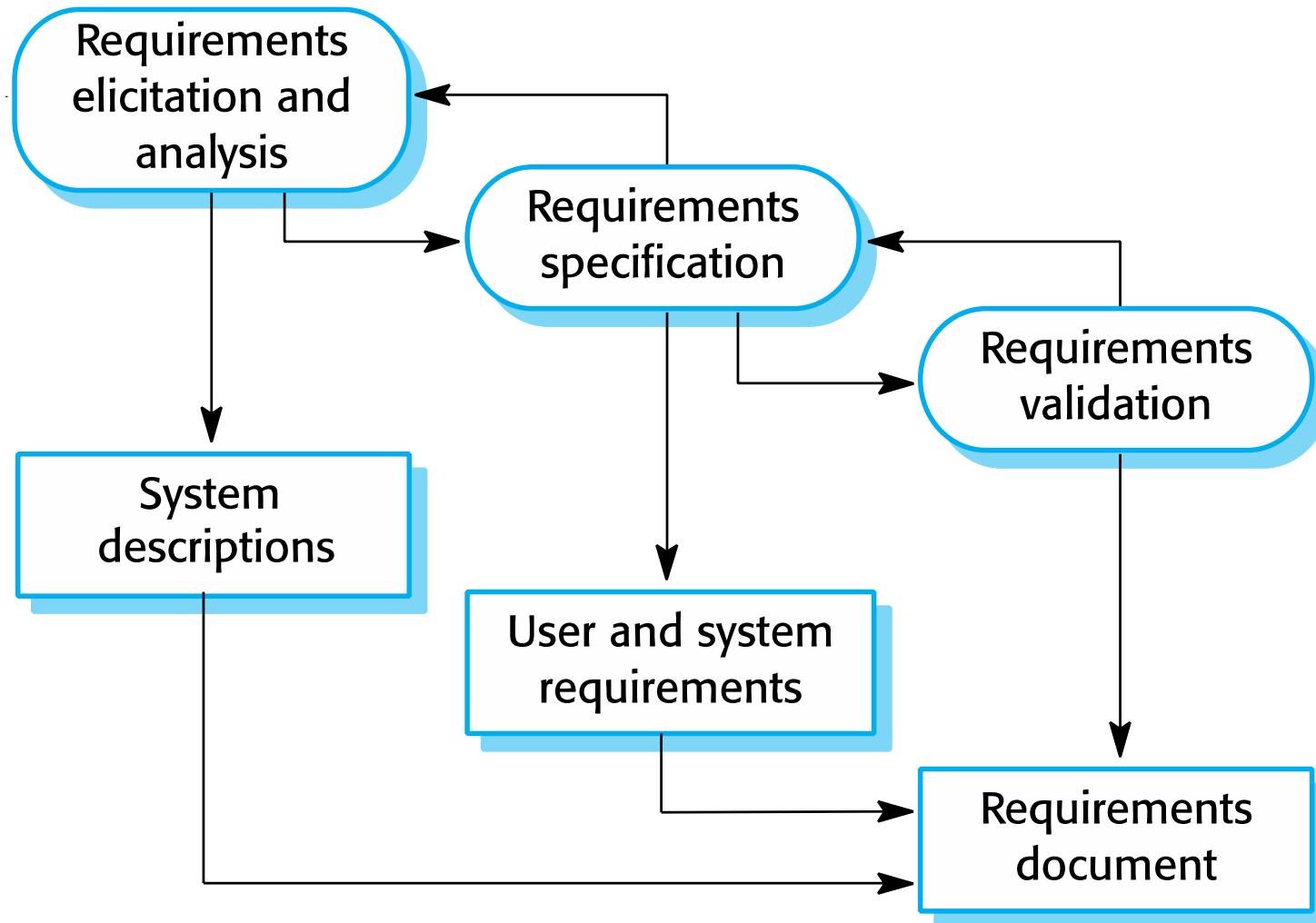
Advantages and disadvantages

- ✧ Reduced costs and risks as less software is developed from scratch
- ✧ Faster delivery and deployment of system
- ✧ But requirements compromises are inevitable so system may not meet real needs of users
- ✧ Loss of control over evolution of reused system elements

Process activities

- ✧ Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- ✧ The four basic process activities of specification, development, validation and evolution are organized differently in different development processes.
- ✧ For example, in the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved.

The requirements engineering process



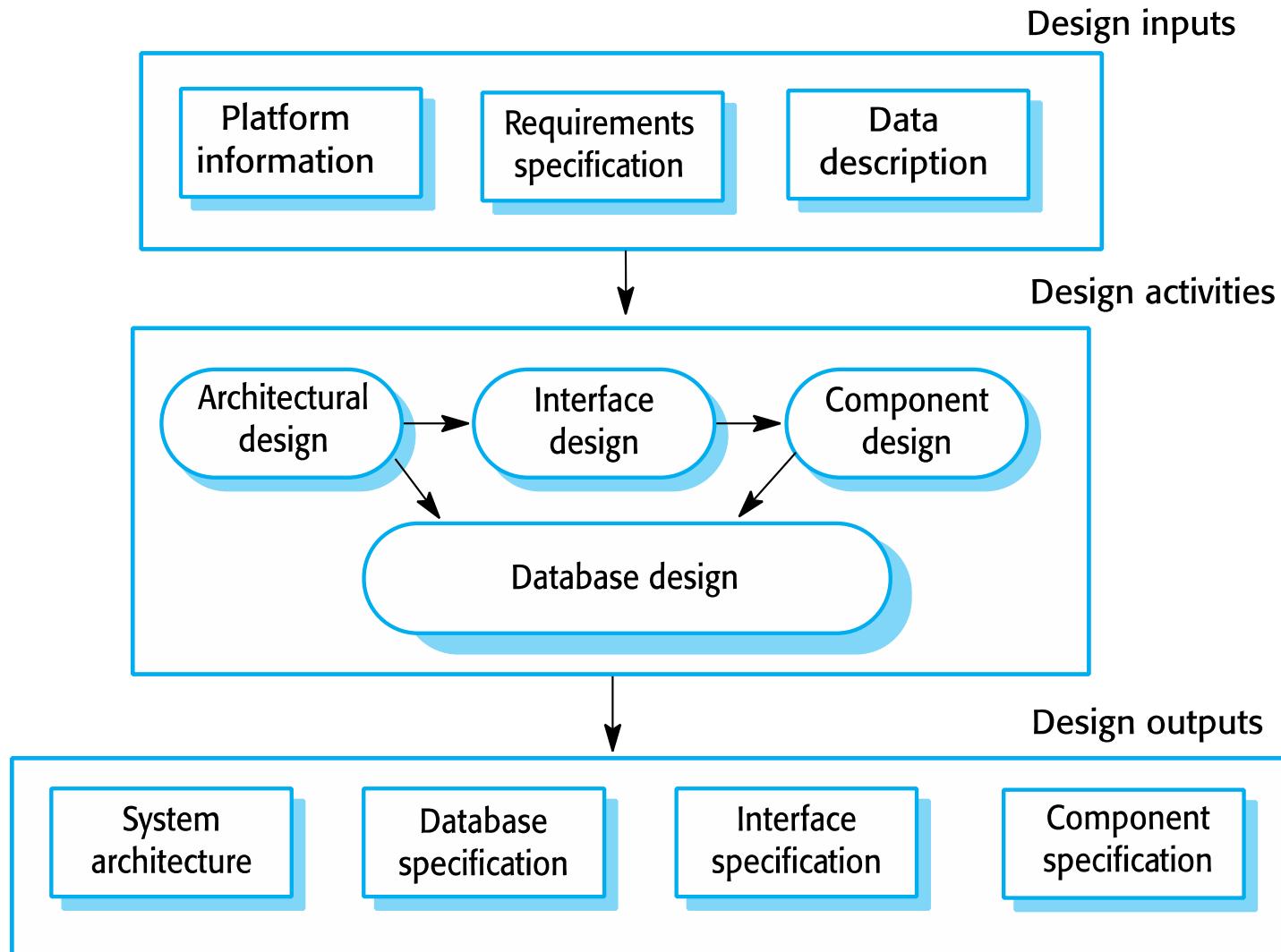
Software specification

- ✧ The process of establishing what services are required and the constraints on the system's operation and development.
- ✧ Requirements engineering process
 - Requirements elicitation and analysis
 - What do the system stakeholders require or expect from the system?
 - Requirements specification
 - Defining the requirements in detail
 - Requirements validation
 - Checking the validity of the requirements

Software design and implementation

- ✧ The process of converting the system specification into an executable system.
- ✧ Software design
 - Design a software structure that realises the specification;
- ✧ Implementation
 - Translate this structure into an executable program;
- ✧ The activities of design and implementation are closely related and may be inter-leaved.

A general model of the design process



Design activities

- ✧ *Architectural design*, where you identify the overall structure of the system, the principal components (subsystems or modules), their relationships and how they are distributed.
- ✧ *Database design*, where you design the system data structures and how these are to be represented in a database.
- ✧ *Interface design*, where you define the interfaces between system components.
- ✧ *Component selection and design*, where you search for reusable components. If unavailable, you design how it will operate.

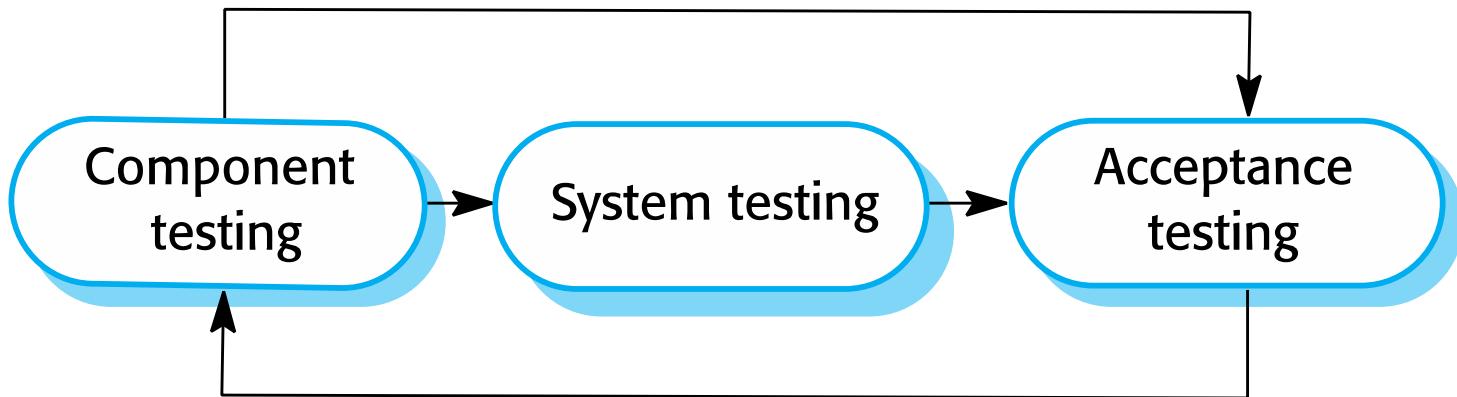
System implementation

- ✧ The software is implemented either by developing a program or programs or by configuring an application system.
- ✧ Design and implementation are interleaved activities for most types of software system.
- ✧ Programming is an individual activity with no standard process.
- ✧ Debugging is the activity of finding program faults and correcting these faults.

Software validation

- ✧ Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- ✧ Testing is the most commonly used V & V activity.

Stages of testing



Testing stages

✧ Component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

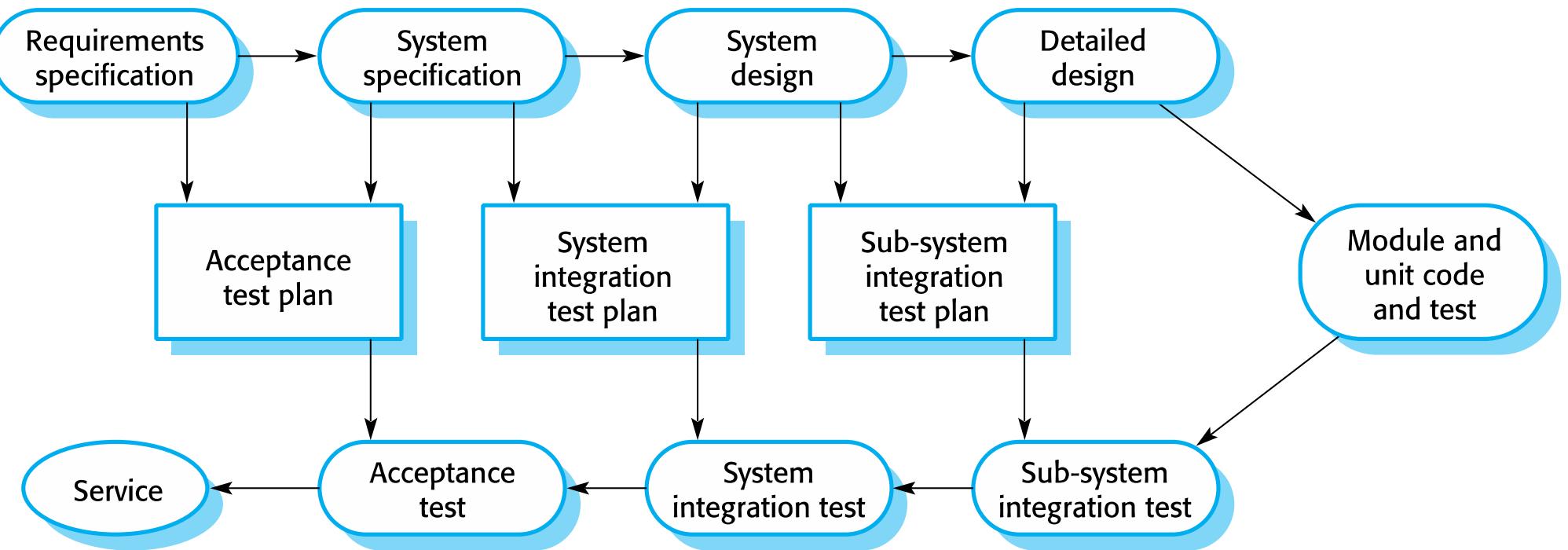
✧ System testing

- Testing of the system as a whole. Testing of emergent properties is particularly important.

✧ Customer testing

- Testing with customer data to check that the system meets the customer's needs.

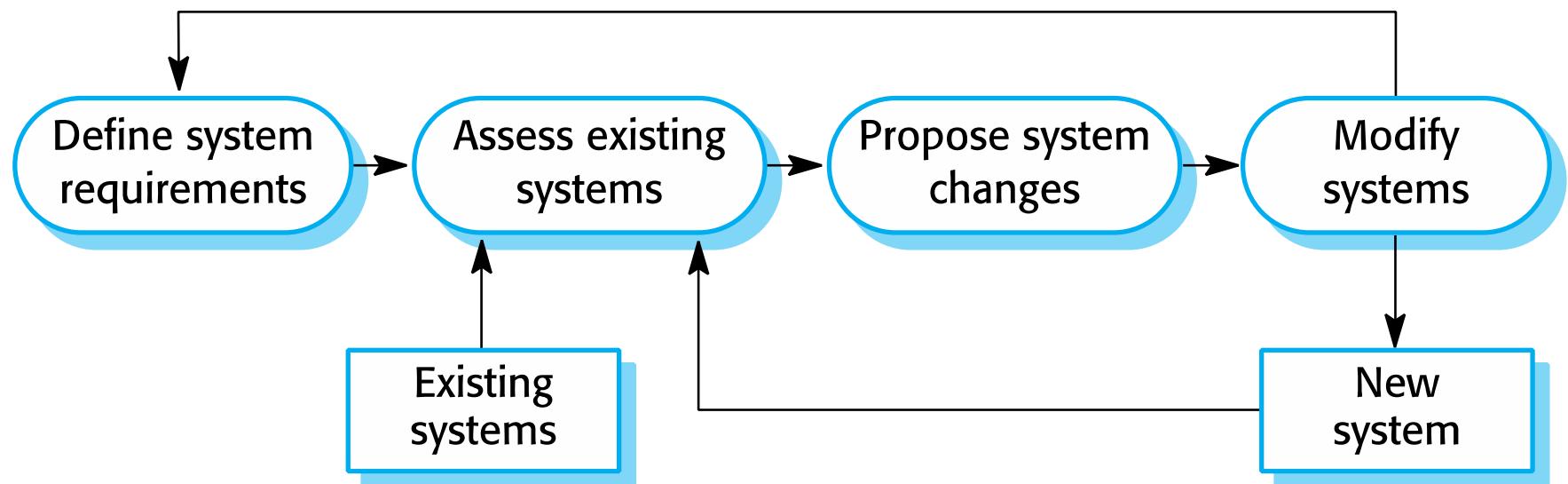
Testing phases in a plan-driven software process (V-model)



Software evolution

- ✧ Software is inherently flexible and can change.
- ✧ As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- ✧ Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

System evolution



Key points

- ✧ Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- ✧ General process models describe the organization of software processes.
 - Examples of these general models include the ‘waterfall’ model, incremental development, and reuse-oriented development.
- ✧ Requirements engineering is the process of developing a software specification.

Key points

- ✧ Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- ✧ Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- ✧ Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.

Software Engineering

CE2006/CZ2006

Agile Software Development

Topics covered

- ✧ Agile methods
- ✧ Agile development techniques
- ✧ Agile project management

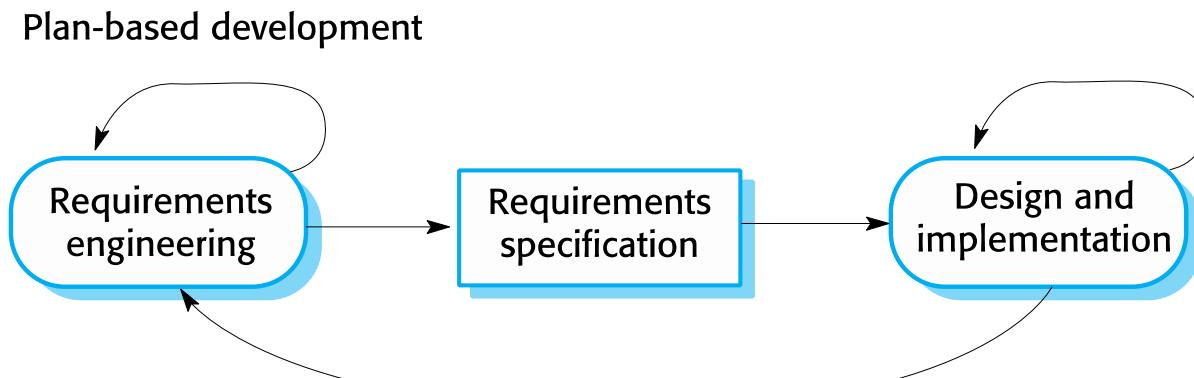
Rapid software development

- ✧ Rapid development and delivery is now often the most important requirement for software systems
 - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
 - Software has to evolve quickly to reflect changing business needs.
- ✧ Plan-driven development is essential for some types of system but does not meet these business needs.
- ✧ Agile development methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems

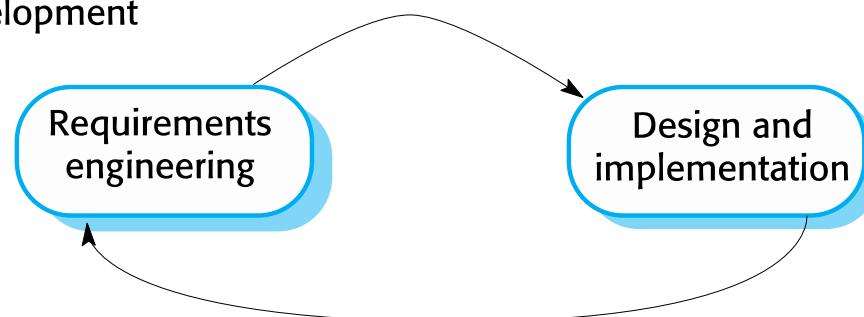
Agile development

- ✧ Program specification, design and implementation are inter-leaved
- ✧ The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- ✧ Frequent delivery of new versions for evaluation
- ✧ Extensive tool support (e.g. automated testing tools) used to support development.
- ✧ Minimal documentation – focus on working code

Plan-driven and agile development



Agile development



Agile Software Development (Adapted
from: Software Engineering by Ian
Sommerville)

Plan-driven and agile development

✧ Plan-driven development

- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.

✧ Agile development

- Specification, design, implementation and testing are interleaved and the outputs from the development process are decided through a process of negotiation during the software development process.

Agile methods

Agile methods

- ✧ Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
 - Focus on the code rather than the design
 - Are based on an iterative approach to software development
 - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- ✧ The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

Agile manifesto

- ✧ We are uncovering better ways of developing ~~processes~~ software by doing it and helping others do it. ~~Through~~ Through this work we have come to value:
 - Individuals and interactions over processes and tools
 - Working software over comprehensive documentation
 - Customer collaboration over contract negotiation
 - Responding to change over following a plan
- ✧ That is, while there is value in the items on ~~the right~~, we value the items on the left more.

The principles of agile methods

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Agile method applicability

- ✧ Product development where a software company is developing a small or medium-sized product for sale.
 - Virtually all software products and apps are now developed using an agile approach
- ✧ Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external rules and regulations that affect the software.

Agile methods across organizations

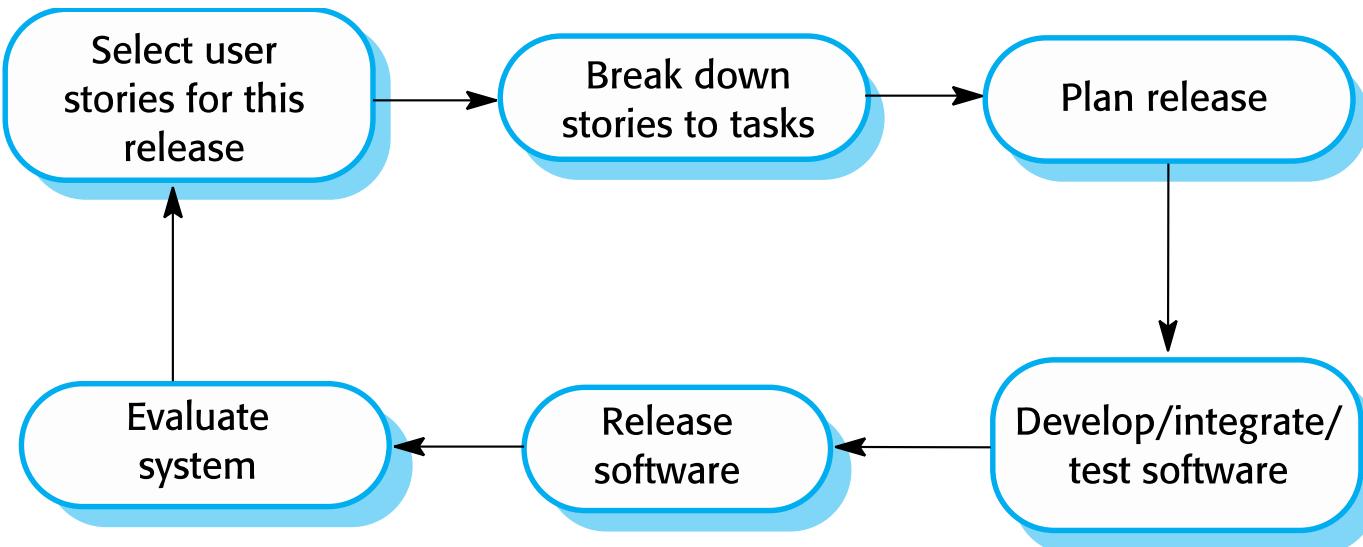
- ✧ Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- ✧ Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- ✧ Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.
- ✧ There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Agile development techniques

Extreme programming

- ✧ A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques.
- ✧ Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.
 - New versions may be built several times per day;
 - Increments are delivered to customers every 2 weeks;
 - All tests must be run for every build and the build is only accepted if tests run successfully.

The extreme programming release cycle



Extreme programming practices (a)

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Extreme programming practices (b)

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Influential XP practices

- ✧ Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.
- ✧ Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.
- ✧ Key practices
 - User stories for specification
 - Refactoring
 - Test-first development
 - Pair programming

User stories for requirements

- ✧ In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- ✧ User requirements are expressed as user stories or scenarios.
- ✧ These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- ✧ The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

A ‘prescribing medication’ story

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Examples of task cards for prescribing medication

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

Refactoring

- ✧ Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- ✧ XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- ✧ Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.

Refactoring

- ✧ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- ✧ This improves the understandability of the software and so reduces the need for documentation.
- ✧ Changes are easier to make because the code is well-structured and clear.
- ✧ However, some changes requires architecture refactoring and this is much more expensive.

Examples of refactoring

- ✧ Re-organization of a class hierarchy to remove duplicate code.
- ✧ Tidying up and renaming attributes and methods to make them easier to understand.
- ✧ The replacement of inline code with calls to methods that have been included in a program library.

Test-first development

- ✧ Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- ✧ XP testing features:
 - Test-first development.
 - Incremental test development from scenarios.
 - User involvement in test development and validation.
 - Automated test harnesses are used to run all component tests each time that a new release is built.

Test-driven development

- ✧ Writing tests before code clarifies the requirements to be implemented.
- ✧ Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
 - Usually relies on a testing framework such as Junit.
- ✧ All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.

Customer involvement

- ✧ The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- ✧ The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- ✧ However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test case description for dose checking

Test 4: Dose checking

Input:

1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:

OK or error message indicating that the dose is outside the safe range.

Test automation

- ✧ Test automation means that tests are written as executable components before the task is implemented
 - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- ✧ As testing is automated, there is always a set of tests that can be quickly and easily executed
 - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Pair programming

- ✧ Pair programming involves programmers working in pairs, developing code together.
- ✧ This helps develop common ownership of code and spreads knowledge across the team.
- ✧ It serves as an informal review process as each line of code is looked at by more than 1 person.
- ✧ It encourages refactoring as the whole team can benefit from improving the system code.

Pair programming

- ✧ In pair programming, programmers sit together at the same computer to develop the software.
- ✧ Pairs are created dynamically so that all team members work with each other during the development process.
- ✧ The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- ✧ Pair programming is not necessarily inefficient and there is some evidence that suggests that a pair working together is more efficient than 2 programmers working separately.

Agile project management

Agile project management

- ✧ The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.
- ✧ The standard approach to project management is plan-driven. Managers draw up a plan for the project showing what should be delivered, when it should be delivered and who will work on the development of the project deliverables.
- ✧ Agile project management requires a different approach, which is adapted to incremental development and the practices used in agile methods.

Scrum

- ✧ Scrum is an agile method that focuses on managing iterative development rather than specific agile practices.
- ✧ There are three phases in Scrum.
 - The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
 - This is followed by a series of sprint cycles, where each cycle develops an increment of the system.
 - The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.



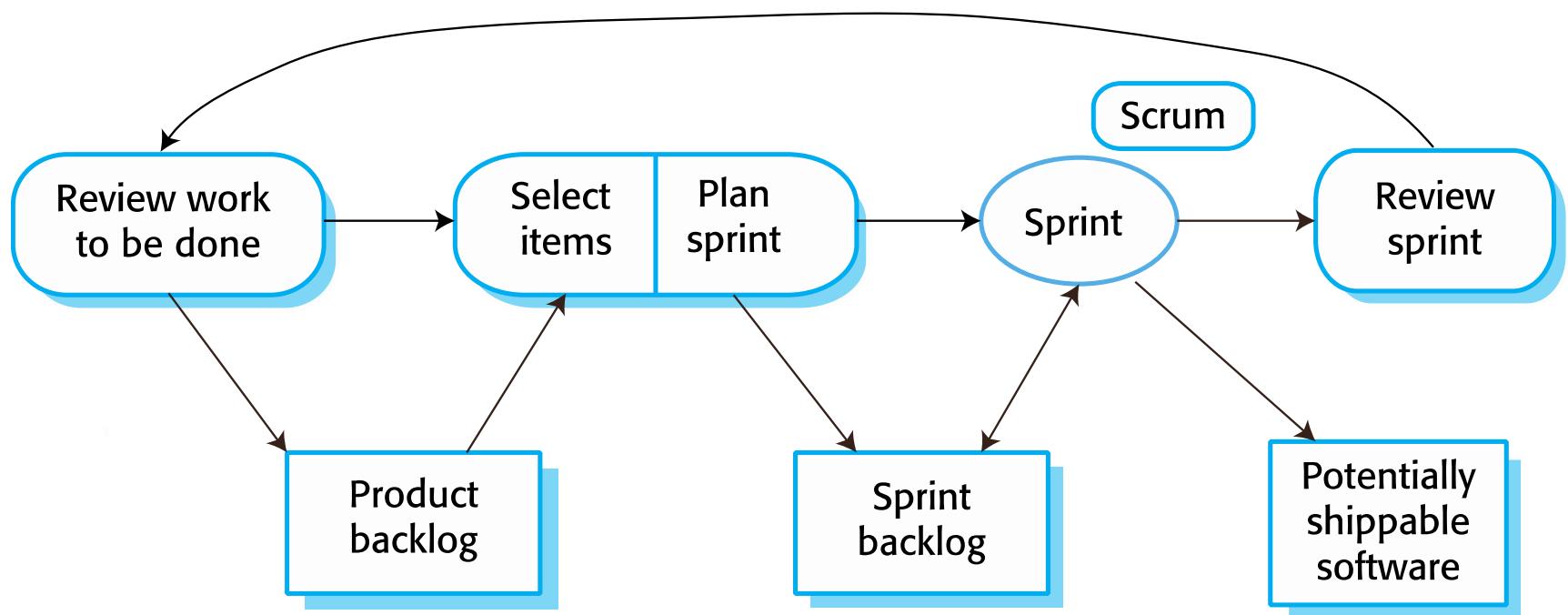
Scrum terminology (a)

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.

Scrum terminology (b)

Scrum term	Definition
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

Scrum sprint cycle



The Scrum sprint cycle

- ✧ Sprints are fixed length, normally 2–4 weeks.
- ✧ The starting point for planning is the product backlog, which is the list of work to be done on the project.
- ✧ The selection phase involves all of the project team who work with the customer to select the features and functionality from the product backlog to be developed during the sprint.

The Sprint cycle

- ✧ Once these are agreed, the team organize themselves to develop the software.
- ✧ During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called ‘Scrum master’.
- ✧ The role of the Scrum master is to protect the development team from external distractions.
- ✧ At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

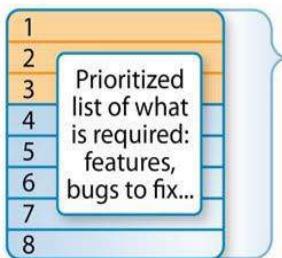
Practical Aspects of Scrum

The Agile Scrum Framework at a glance

Inputs from
Customers, Team,
Managers, Execs



Product Owner



Product Backlog



The Team

Team selects starting at top as much as it can commit to deliver by end of Sprint

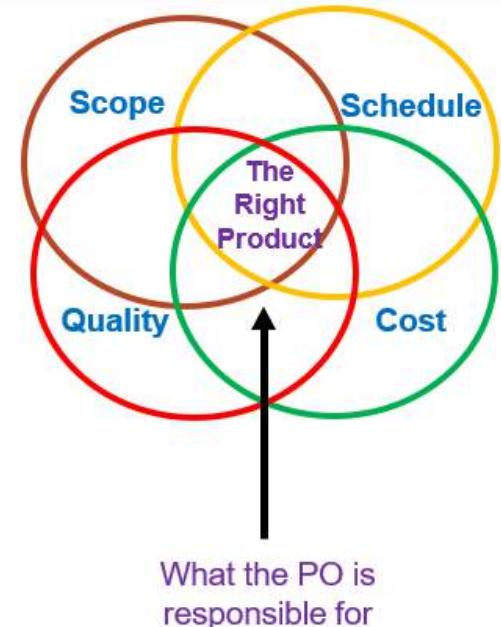
Sprint Planning Meeting

Agile Software Development (Adapted
from: Software Engineering by Ian
Sommerville)



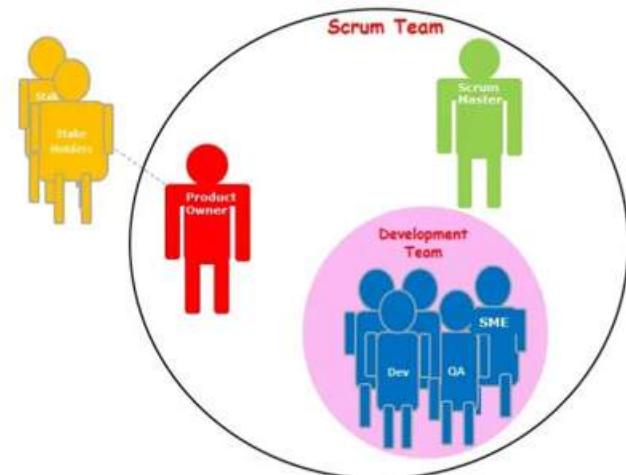
SCRUM ROLE – PRODUCT OWNER

- Has a clear vision and goals for the Product. Represents the interests of the customers/stakeholders.
- Responsible for **maximizing the value of the Product** resulting from the work of the Development Team.
- Owns the Product Backlog.
 - Creation
 - Management
 - Prioritization



SCRUM ROLE – SCRUM MASTER

- A **servant-leader** - focus is on the needs of the team members and those they serve (the customer), with the goal of achieving results in line with the organization's values, principles and business objectives
- Teaches and coaches the organization and team in adopting and using Scrum process to perform at the highest level.
- Facilitates Scrum events.
- Helps team to remove and prevent **impediments**.



SCRUM ROLE – DEVELOPMENT TEAM

- A team of 5 to 9 professionals who **is responsible for the working product** at the end of each Sprint.
- Flat structure: no titles other than developer, no sub-team.
- Owns Sprint backlog and decides how much is realistic for each Sprint
- Produces thoroughly tested, defect-free “Done” Product Increment in each Sprint.

SCRUM ARTIFACT- PRODUCT BACKLOG

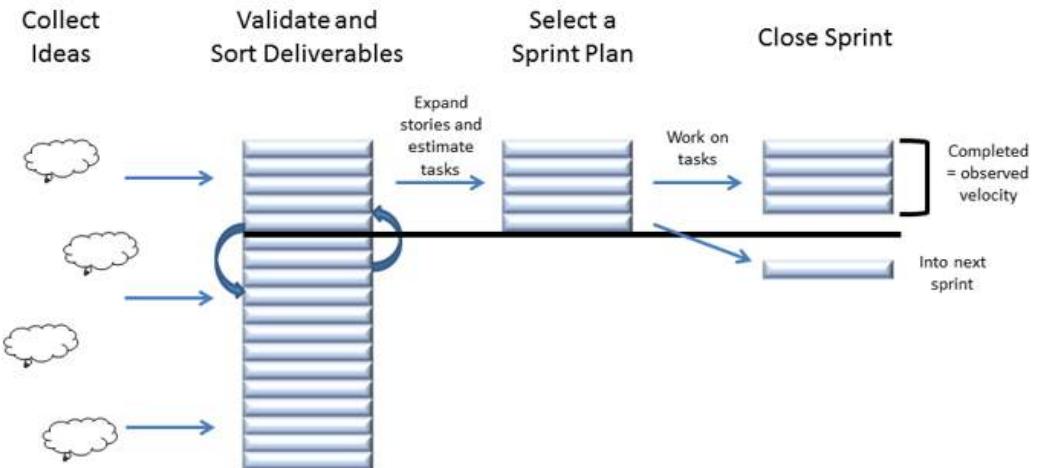
- An **ordered list** of everything that is known to be needed in or for the Product to achieve maximum value.
- Is the **single source of requirements** for any changes to be made to the product.
- Product Owner is responsible for it, including its content, availability and ordering
- **User Stories** are a way of expressing **Features** on the Product Backlog.

SCRUM ARTIFACT- SPRINT BACKLOG

- The set of **Product Backlog items (PBIs)** selected for the Sprint, plus a plan for delivering the product Increment and realizing the Sprint Goal.
- The sprint backlog is **unchanged** during the period of the **Sprint**.
- Items not “Done” by end of the Sprint - they will be added back to the Product Backlog and addressed during the next Sprint.
- Anything needs improvement - can be put back to the Product Backlog. Either add additional Sprints to the schedule or remove lower priority items from the Product Backlog.

SCRUM ARTIFACT- SPRINT BACKLOG

- PBIs are broken down into tasks.
- Tasks are estimated in hours, usually 1 - 16. Task that more than 16 hours are broken down further later on.
- Team members sign up for task. They are not assigned.
- Estimated remaining work is updated daily.



SCRUM ARTIFACT- PRODUCT INCREMENT

- At end of each Sprint, “Done” items will be integrated and tested with the increments of all previous Sprints. -> **Potentially Shippable Product**
- The Product Increment is in a potentially releasable state – Product Owner could release it if they wanted to do so with **the collection of completed features** (ie: the delivery of a Release)

SCRUM EVENT - SPRINT

- A time-box event of **2 to 4 weeks** when a Scrum Team works to complete a set amount of work in which a “Done” product increment is created.
- Each sprint has a goal of what is to be built. The **Sprint Goal** does not change during the sprint.
- The time-box is never extended. Sprint ends when time box expires.
- At every sprint, the Scrum Team will **inspect** and **adapt** the **Product** and the **Process**.
- ***Can the Sprint be cancelled before its duration is over? If so, who can cancel the Sprint?***

WHAT WORK CAN GET DONE IN THIS SPRINT?

- Product Owner presents the **ordered** product backlog items (PBIs) to the Development team.
- Scrum Master **facilitates**, gives advice, asks question and helps identify risks etc.
- The PBIs are broken down to tasks and tasks are estimated jointly in hours.
- Only the Development Team can assess how much can be accomplished in the Sprint.

HOW WILL THE CHOSEN WORK GET DONE?

- The Development Team decides how it will build this functionality into a “Done” Product Increment during the Sprint.
- The Development Team does planning and design to make sure that they will achieve the Sprint Goal. May invite other people to attend to provide technical or domain advice.
- Scrum Master helps the team in case any arrangements need to be done for achieving the Sprint Goal.

SCRUM EVENT - DAILY SCRUM

- A **stand-up meeting** between the Scrum Team. It's a **15-minute time-box** daily event generally held at the same place and the same time.

- **Input ??** 3 questions –

"What did I do yesterday?", "What will I do today?" & "Are there any issues or Impediments?"

- Optimises team collaboration and performance by inspecting the work since last Daily Scrum and adapts its progress towards the Sprint Goal.
- Helps the team in terms of better communication, decision-making and improved level of knowledge.

SCRUM EVENT - SCRUM REVIEW

- A time-box event of **1 hr x num. of weeks of a Sprint** (incl. Product Owner & key stakeholders) held at the end of each Sprint.

- Hands-on with what's been built, and active discussion of :
 - **What** is the current state of the Product?
 - **What** should be added to the Product or improved in future Sprints to deliver maximum value?

- **Output:** ?? Revised Product Backlog by Product Owner

SCRUM EVENT -

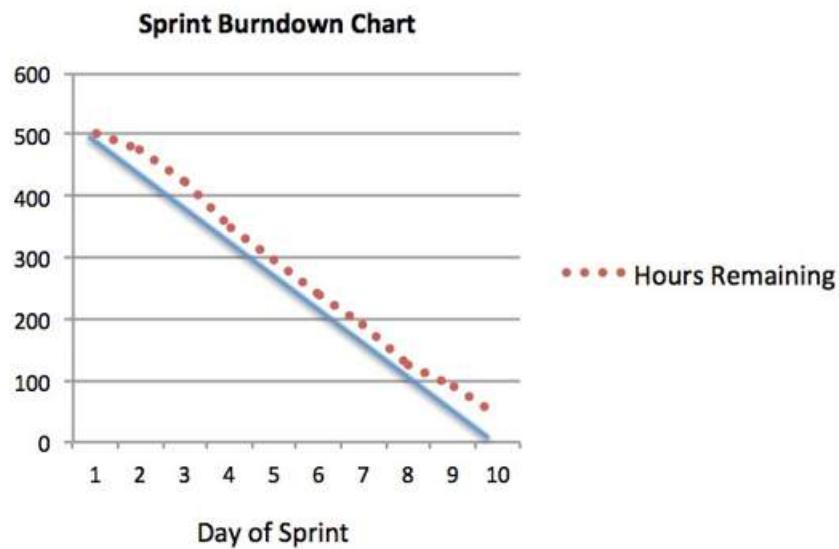
SCRUM RETROSPECTIVE

- A time-box event of **45 mins x num. of weeks of a Sprint** between the Scrum Team at the end of the sprint after Sprint Review.

- Active discussion of:
 - Inspect **how** the sprint went with regard to process, tool, and people.
 - Identify items that went well and potential improvements.
 - Create an action plan to implement improvements in the Scrum Team.

- **Input ??** Results from the Sprint
- **Output ??** Lesson learned, Improvements, and action list for the next Sprint.

SPRINT BURNDOWN CHART

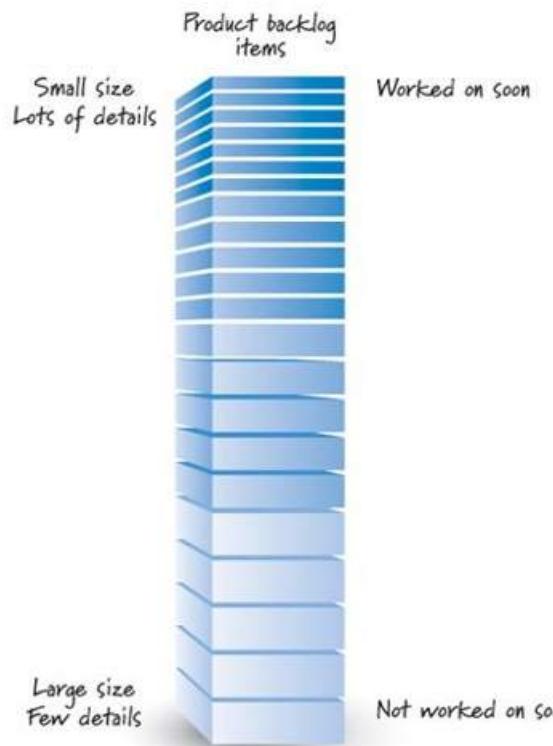


- A graphical representation of work left to do versus time.
- Purpose:
 - Monitoring the project scope creep
 - Keeping the team running on schedule
 - Comparing the planned work against the team progression

VELOCITY

- Velocity is the average size of Product Backlog items, in number of points, that the team can deliver during a Sprint.
- Measures the **amount of work done** during a Sprint. A key metric in Scrum.
- Evaluated only as an average. One Sprint's number is meaningless
- Tasks are estimated in real units using hours or quarter-days.
- Points cannot be converted to hours.

CHARACTERISTICS OF A GOOD PRODUCT BACKLOG



Detailed appropriately

- Not all items in a PB will be at the same level of detail at the same time.
- PBIs that we plan work on soon should be at the top of the backlog, small in size and very detail so that they can be work on in the near sprint. BPIs that will be worked on later can be larger and less detail.

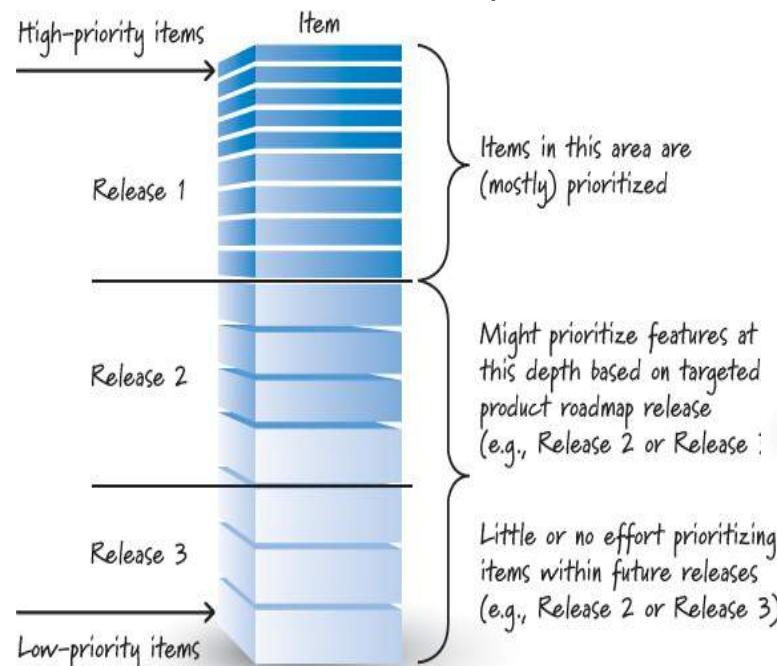
Emergent

- The product backlog is never complete or frozen. Instead, it is continuously updated based on a stream of economically valuable information that is constantly arriving.

CHARACTERISTICS OF A GOOD PRODUCT BACKLOG

▪ Estimated

- Each PBI has a size estimate corresponding to the effort required to develop the item. PO uses these estimates as one of several inputs to help determine a PBI's priority (and therefore position) in the PB.



Item	Size
	2
	3
	2
	5
	13
	13
	20
	20
	40
	L
	XL

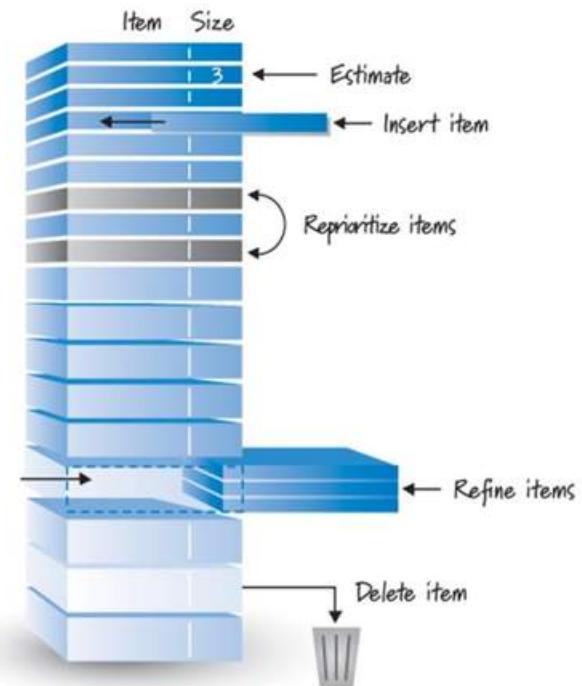
Each item has a size estimate
Most estimates are story point or ideal day estimates
Very large items near the bottom may not have an estimate or may be estimated in T-shirt sizes.

▪ Prioritized

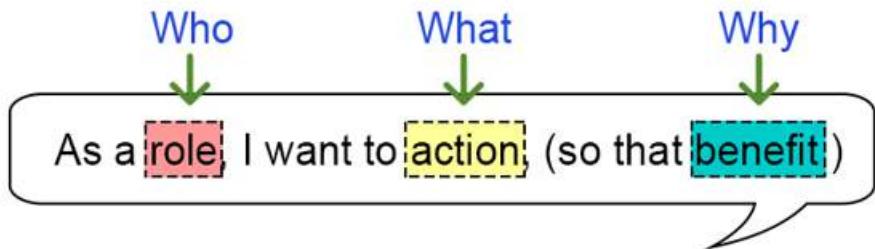
- Useful to prioritize the near-term items that are destined for the next few Sprints

PRODUCT BACKLOG GROOMING (REFINEMENT)

- Refers to a set of three principal activities:
 - Creating and refining (adding details to) PBIs
 - Estimating PBIs
 - Prioritizing PBIs
- An **ongoing** process - Items are reviewed and revised.
- As a general rule, the Development Team spend up to 10% of its time each sprint assisting with refinement activities.



USER STORIES



- User Stories are expressed in everyday language and describe a specific **goal** (what) from the **perspective of a user** (who) along with the reason (why) he/she wants it.
- **Epics**- a **large** body of work that can be broken down into multiple features. It may start out very big with limited detail.
- The Scrum Team splits these Epics into a smaller, more detailed User Stories in Product Backlog grooming/refinement.

3CS APPROACH

Card

- represents the capture of the statement of intent of the user story.
- traditionally written on index cards or post-it notes.



Cards can be re-organized when agile requirements change.

3CS APPROACH

Conversation

- details behind the User Stories ,where the clearer understanding comes from.
- provides shared context that cannot be achieved via formal documentation.

Confirmation

- represents acceptance criteria of the user story which confirms the intended value is delivered

WHAT IS ACCEPTANCE CRITERIA

- A set of predefined scope and requirements that must be executed by developers to consider the user story finished.
- May include:
 - **Functional Criteria:** Identify specific user functions or business processes that must be in place. A functional criterion might be “A user is able to access a list of available reports.”
 - **Non-functional Criteria:** Identify specific non-functional conditions the implementation must meet, such as design elements. A non-functional criterion might be “Edit buttons and Workflow buttons comply with the Site Button Design.”

CHARACTERISTICS OF A GOOD ACCEPTANCE CRITERIA

- **State an intent not a solution** (e.g. “The user can choose an account” rather than “The user can select the account from a drop-down”)
- **Are independent of implementation** (ideally the phrasing would be the same regardless is implemented on e.g. web, mobile or a voice activated system)
- **Are relatively high level** (not every detail needs to be in writing)

Teamwork in Scrum

- ✧ The ‘Scrum master’ is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- ✧ The whole team attends short daily meetings (Scrums) where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
 - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

Scrum benefits

- ✧ The product is broken down into a set of manageable and understandable chunks.
- ✧ Unstable requirements do not hold up progress.
- ✧ The whole team have visibility of everything and consequently team communication is improved.
- ✧ Customers see on-time delivery of increments and gain feedback on how the product works.
- ✧ Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Key points

- ✧ Agile methods are incremental development methods that focus on rapid software development, frequent releases of the software, reducing process overheads by minimizing documentation and producing high-quality code.
- ✧ Agile development practices include
 - User stories for system specification
 - Frequent releases of the software,
 - Continuous software improvement
 - Test-first development
 - Customer participation in the development team.

Key points

- ✧ Scrum is an agile method that provides a project management framework.
 - It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- ✧ Many practical development methods are a mixture of plan-based and agile development.
- ✧ Scaling agile methods for large systems is difficult.
 - Large systems need up-front design and some documentation and organizational practice may conflict with the informality of agile approaches.

Software Engineering

CE2006/CZ2006

UML Review 1

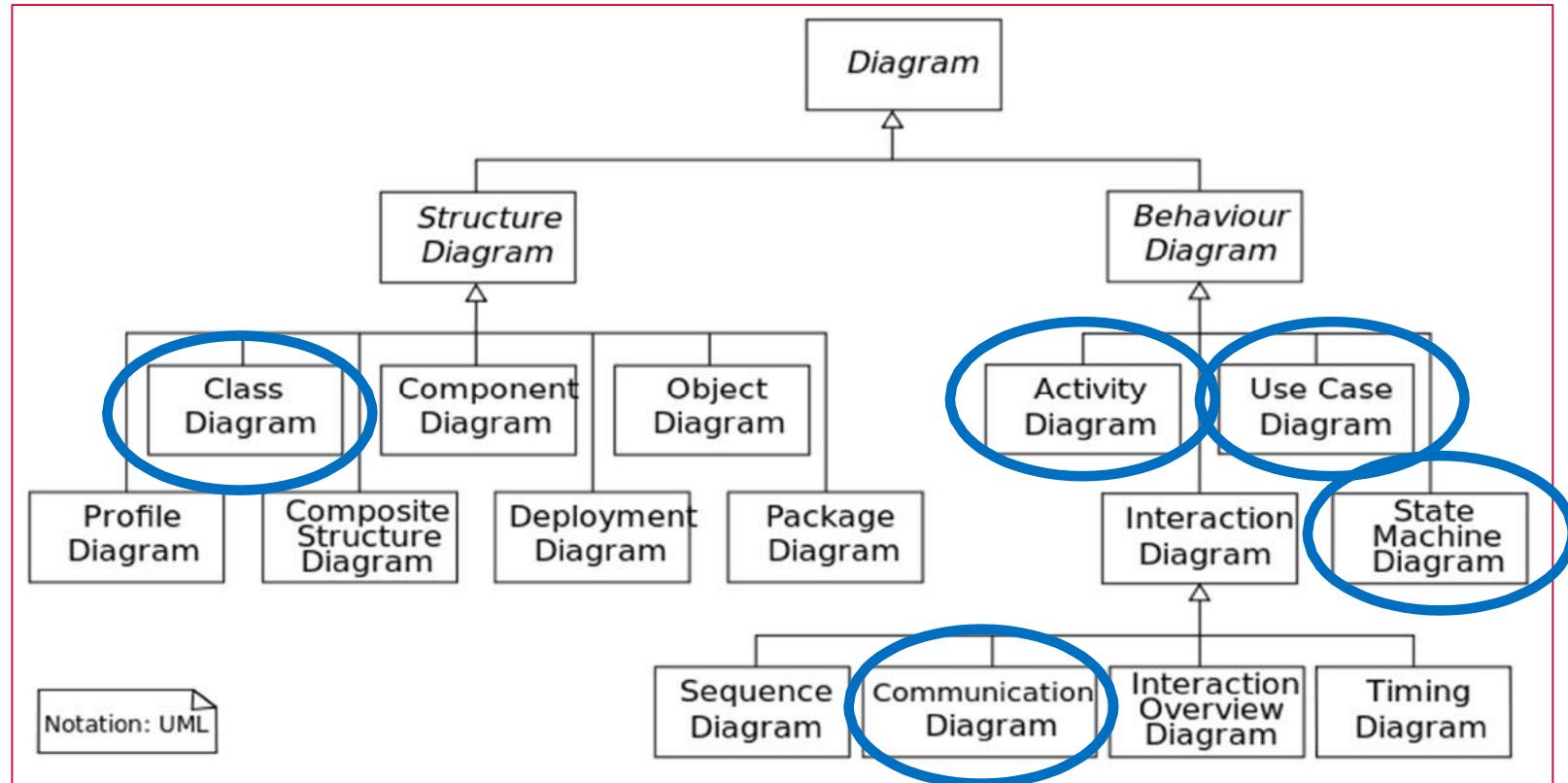
Lesson Objectives

Review UML with an example

At the end of the lesson, you should be able to:

- Describe UML diagrams learned so far
- Explain the purposes of different kinds of UML diagrams
- Use different diagrams at different steps in software requirement elicitation and analysis process

UML diagrams learned



Activity diagram: Flow from one **activity** to another describing the operation of a system.

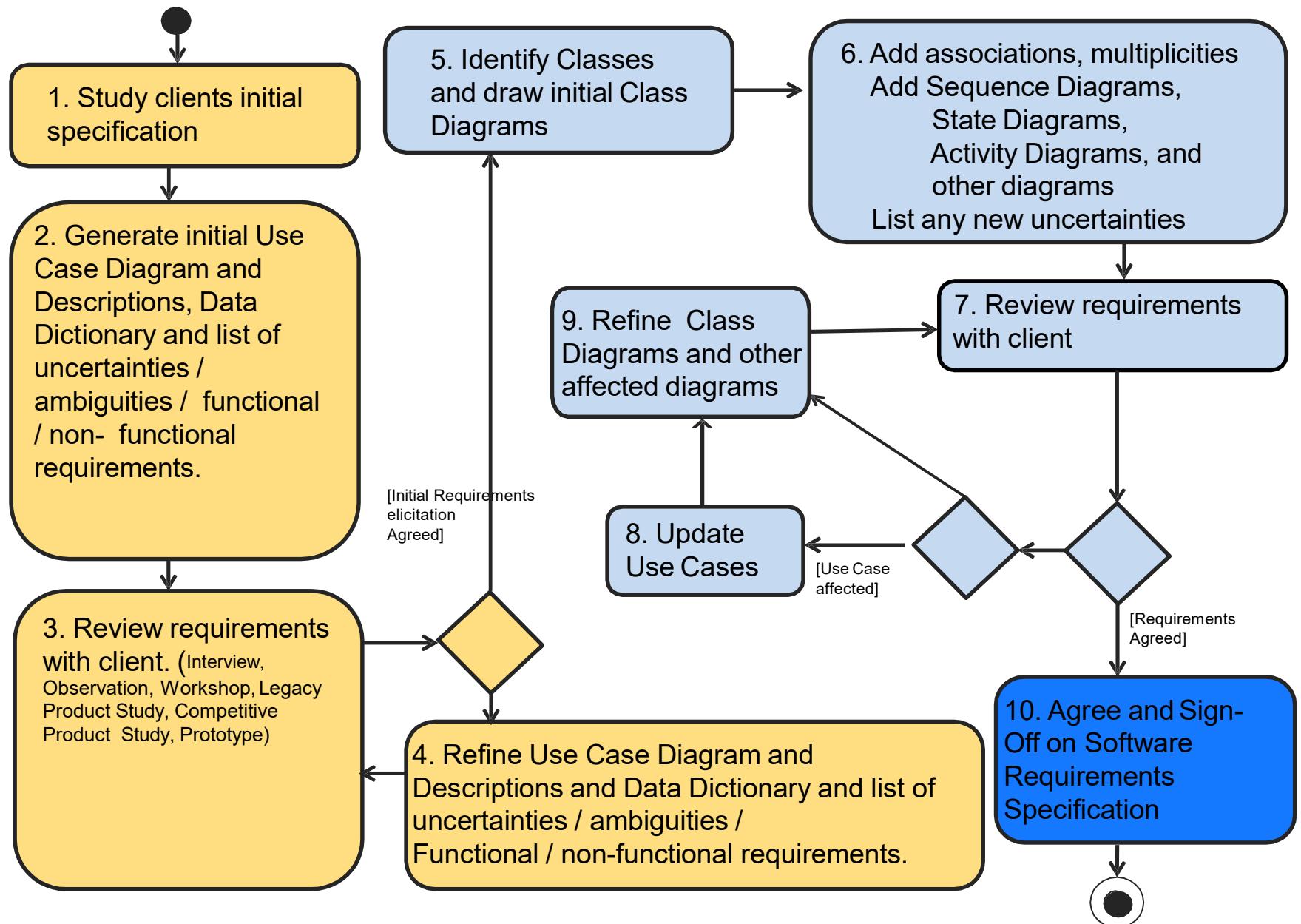
Use Case diagram: describes the **functionality** provided by a system in terms of **actors**, their goals represented as **use cases**, and any dependencies among those use case descriptions.

Class diagram (Static structure diagram): describes the **structure of a system** by showing the system's **classes**, their attributes, and the relationships among the classes.

Sequence diagram and Communication diagram: shows how objects **communicate** with each other in terms of a **sequence of messages**.

State machine diagram: describes the **states** and state **transitions** of the system.

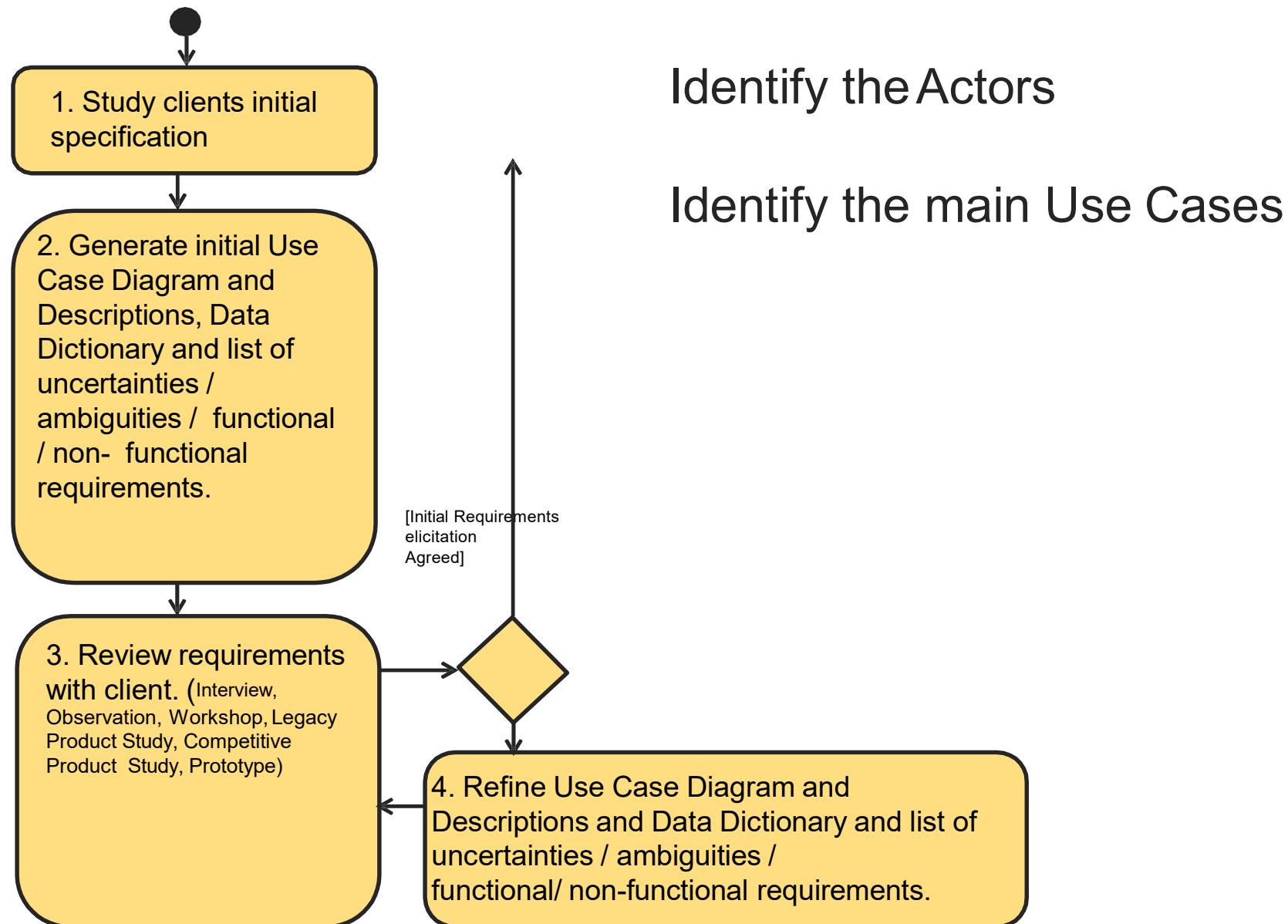
Requirement Elicitation and Analysis Process



Example: A security door access system

1. [A door access system](#) is needed to allow employees to enter a secure area using their fingerprint.
2. When an employee needs to open the door he/she must place their index finger on a fingerprint reader located beside the door.
3. The scanned fingerprint is compared and validated against a company database of employee fingerprints.
4. If a fingerprint match is found the message “Please enter” is displayed and the door is opened to allow the employee to enter.
5. If the fingerprint match fails the message “Access Denied” is displayed and the access system reverts to waiting for a new fingerprint to scan.
6. Every time a door access is attempted a record log is maintained. The information recorded for a successful access is the Time of Access and the name of employee allowed access. The information recorded for a failed access is the Time of Unsuccessful Attempted Access.
7. The company’s Human Resources department maintains a database of employee fingerprints. These are collected once during the new employee enrolment process and are removed when an employee leaves the company.
8. The Human Resources department can view the record log of accesses and attempted accesses and update the employees fingerprint data if necessary.

First Step – Generate Use Case Diagram



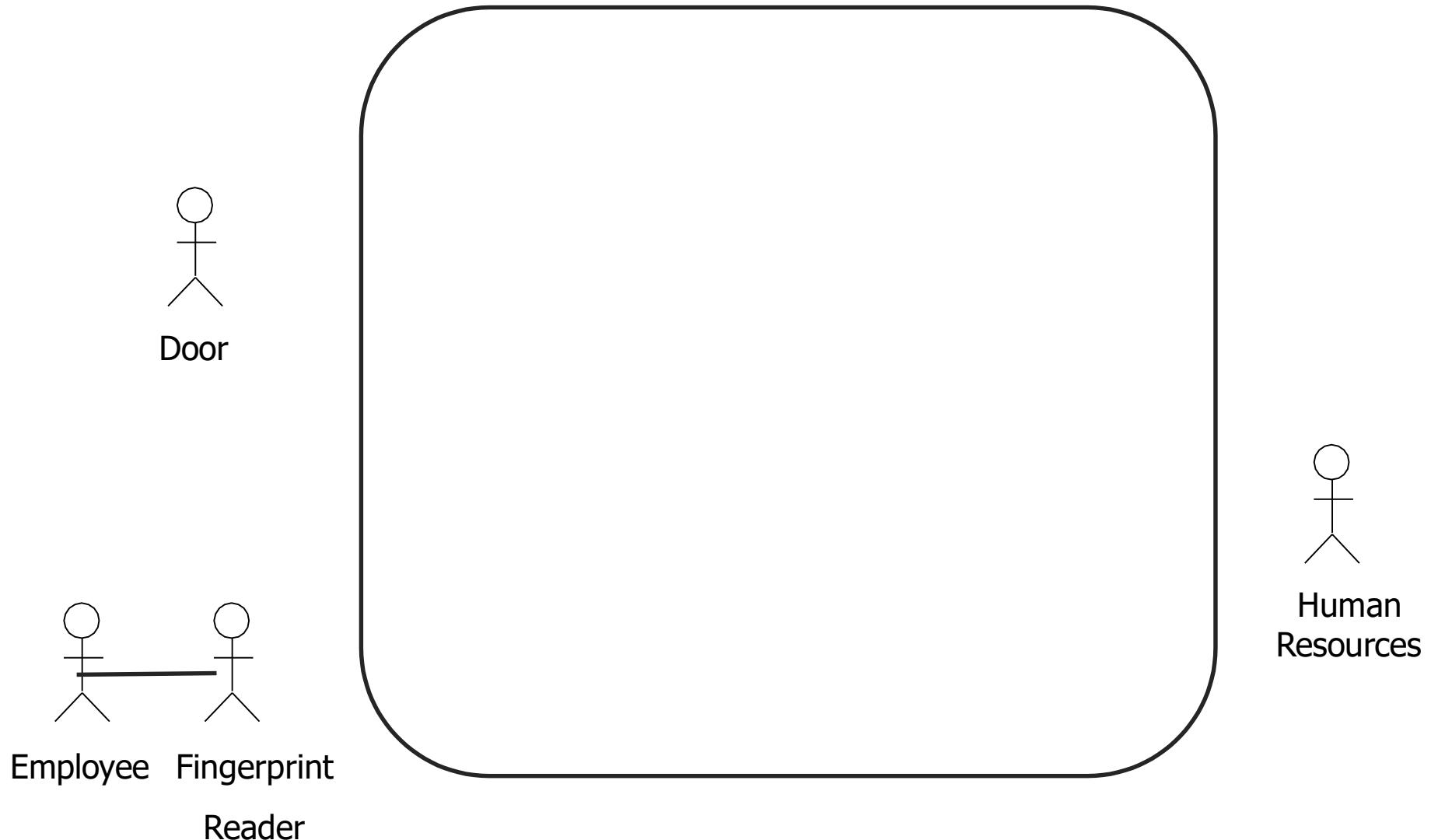
1. Study clients initial specification

IDENTIFY THE ACTORS

1. A door access system is needed to allow employees to enter a secure area using their fingerprint.
2. When an employee needs to open the door he/she must place their index finger on a fingerprint reader located beside the door.
3. The scanned fingerprint is compared and validated against a company database of employee fingerprints.
4. If a fingerprint match is found the message “Please enter” is displayed and the door is opened to allow the employee to enter.
5. If the fingerprint match fails the message “Access Denied” is displayed and the access system reverts to waiting for a new fingerprint to scan.
6. Every time a door access is attempted a record log is maintained. The information recorded for a successful access is the Time of Access and the name of employee allowed access. The information recorded for a failed access is the Time of Unsuccessful Attempted Access.
7. The company’s Human Resources department maintains a database of employee fingerprints. These are collected once during the new employee enrolment process and are removed when an employee leaves the company.
8. The Human Resources department can view the record log of accesses and attempted accesses and update the employees fingerprint data if necessary.

2. Generate initial Use Case diagram

IDENTIFY THE ACTORS



1. Study clients initial specification

IDENTIFY THE MAIN USE CASES (action verb)

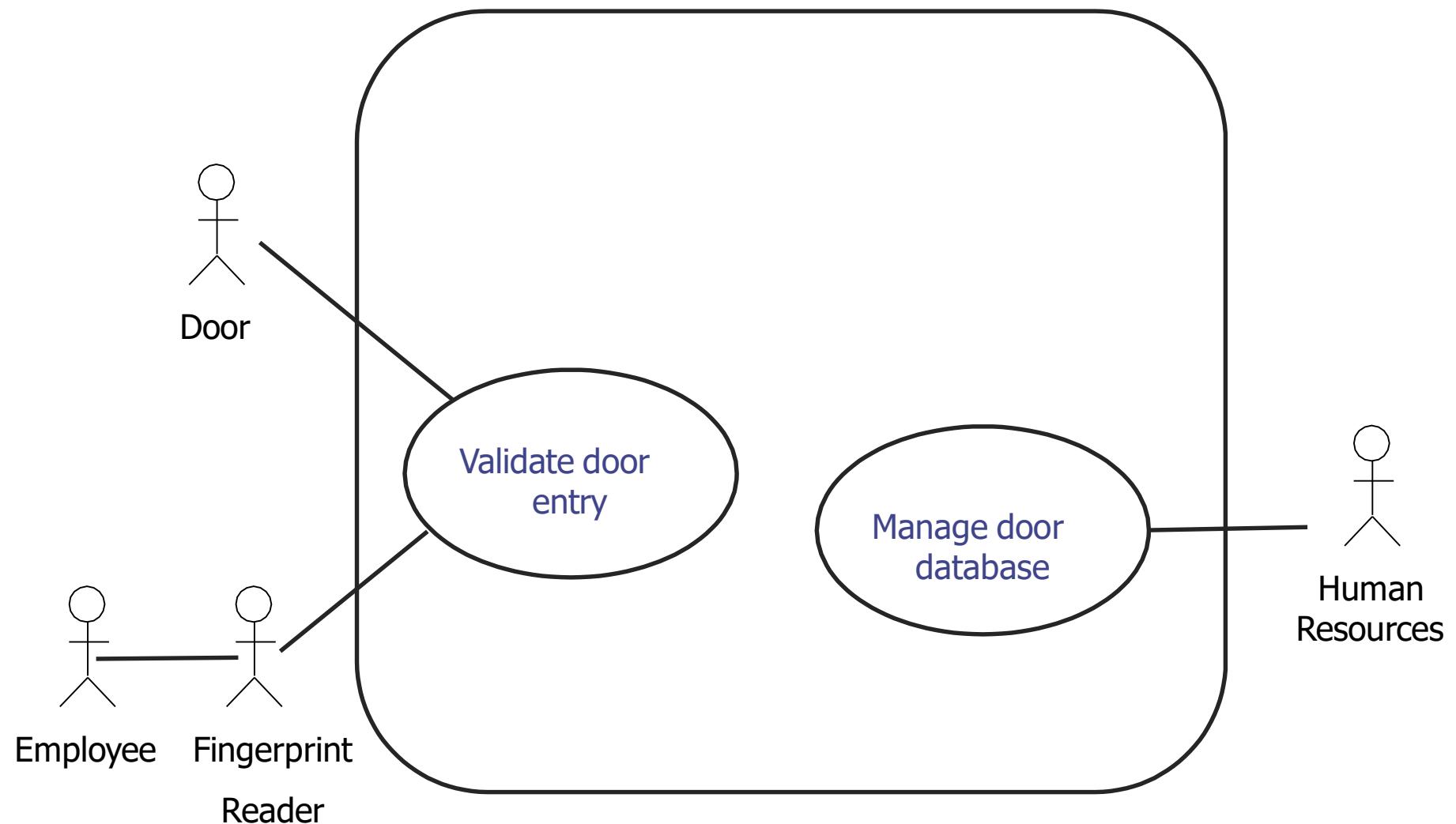
1. A door access system is needed to allow employees to enter a secure area using their fingerprint.
2. When an employee needs to open the door he/she must place their index finger on a fingerprint reader located beside the door.
3. The scanned fingerprint is compared and validated against a company database of employee fingerprints.


4. If a fingerprint match is found the message “Please enter” is displayed and the door is opened to allow the employee to enter.
5. If the fingerprint match fails the message “Access Denied” is displayed and the access system reverts to waiting for a new fingerprint to scan.
6. Every time a door access is attempted a record log is maintained. The information recorded for a successful access is the Time of Access and the name of employee allowed access. The information recorded for a failed access is the Time of Unsuccessful Attempted Access.
7. The company’s Human Resources department maintains a database of employee fingerprints. These are collected once during the new employee enrolment process and are removed when an employee leaves the company.

8. The Human Resources department can view the record log of accesses and attempted accesses and update the employees fingerprint data if necessary.

2. Generate initial Use Case diagram

IDENTIFY THE MAIN USE CASES



2. Generate initial Use Case descriptions

See the Use Case Description Template available on NTULearn course website.

Use Case ID:			
Use Case Name:			
Created By:		Last Updated By:	
Date Created:		Date Last Updated:	
Actor:			
Description:			
Preconditions:			
Postconditions:			
Priority:			
Frequency of Use:			
Flow of Events:			
Alternative Flows:			
Exceptions:			
Includes:			
Special Requirements:			
Assumptions:			
Notes and Issues:			

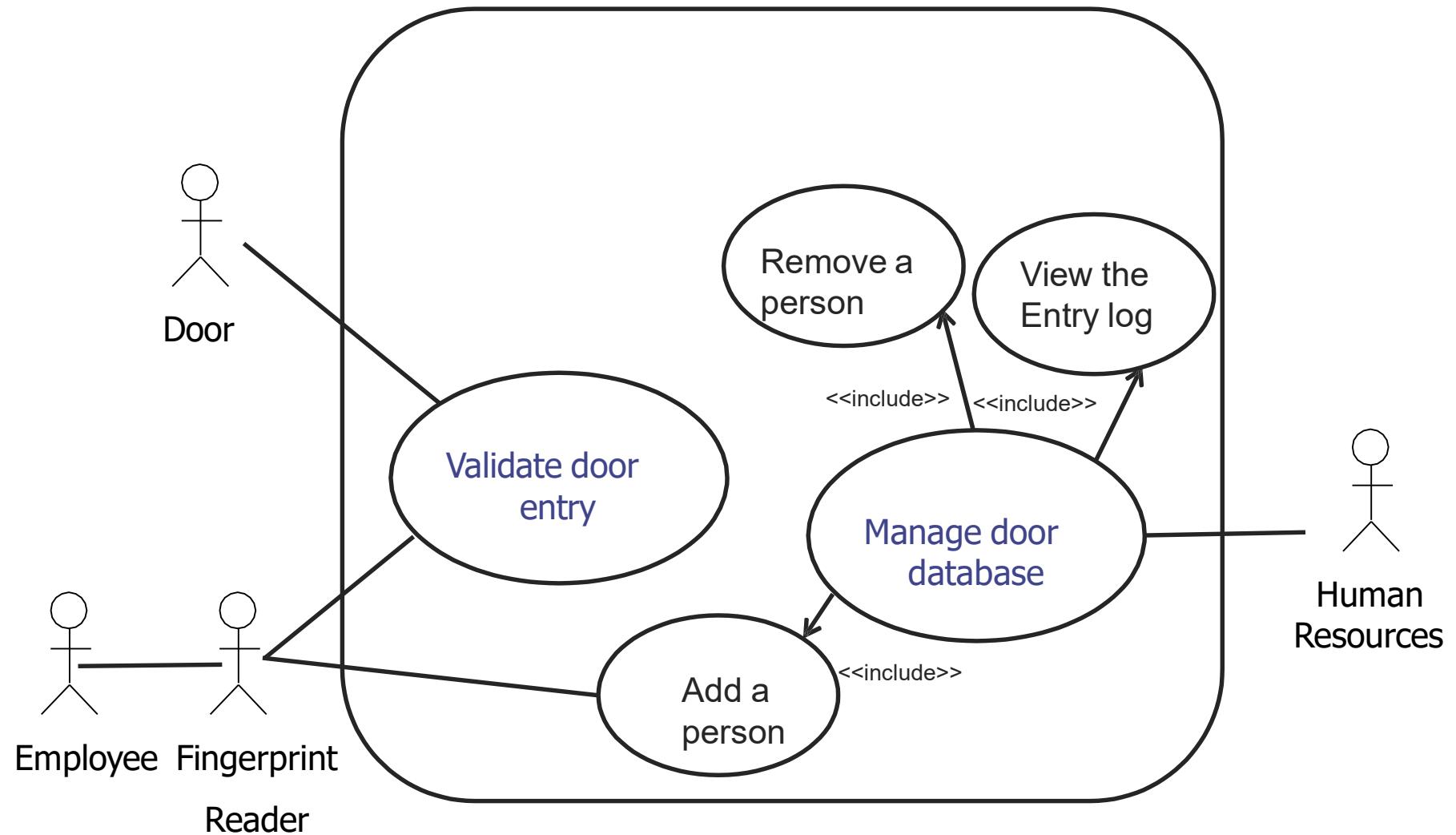
3 & 4. Refine Use Case diagram

REFINE THE MAIN USE CASES

1. A door access system is needed to allow employees to enter a secure area using their fingerprint.
2. When an employee needs to open the door he/she must place their index finger on a fingerprint reader located beside the door.
3. The scanned fingerprint is compared and validated against a company database of employee fingerprints.
4. If a fingerprint match is found the message “Please enter” is displayed and the door is opened to allow the employee to enter.
5. If the fingerprint match fails the message “Access Denied” is displayed and the access system reverts to waiting for a new fingerprint to scan.
6. Every time a door access is attempted a record log is maintained. The information recorded for a successful access is the Time of Access and the name of employee allowed access. The information recorded for a failed access is the Time of Unsuccessful Attempted Access.
7. The company’s Human Resources department maintains a database of employee fingerprints. These are collected once during the new employee enrolment process and are removed when an employee leaves the company.
8. The Human Resources department can view the record log of accesses and attempted accesses and update the employees fingerprint data if necessary.

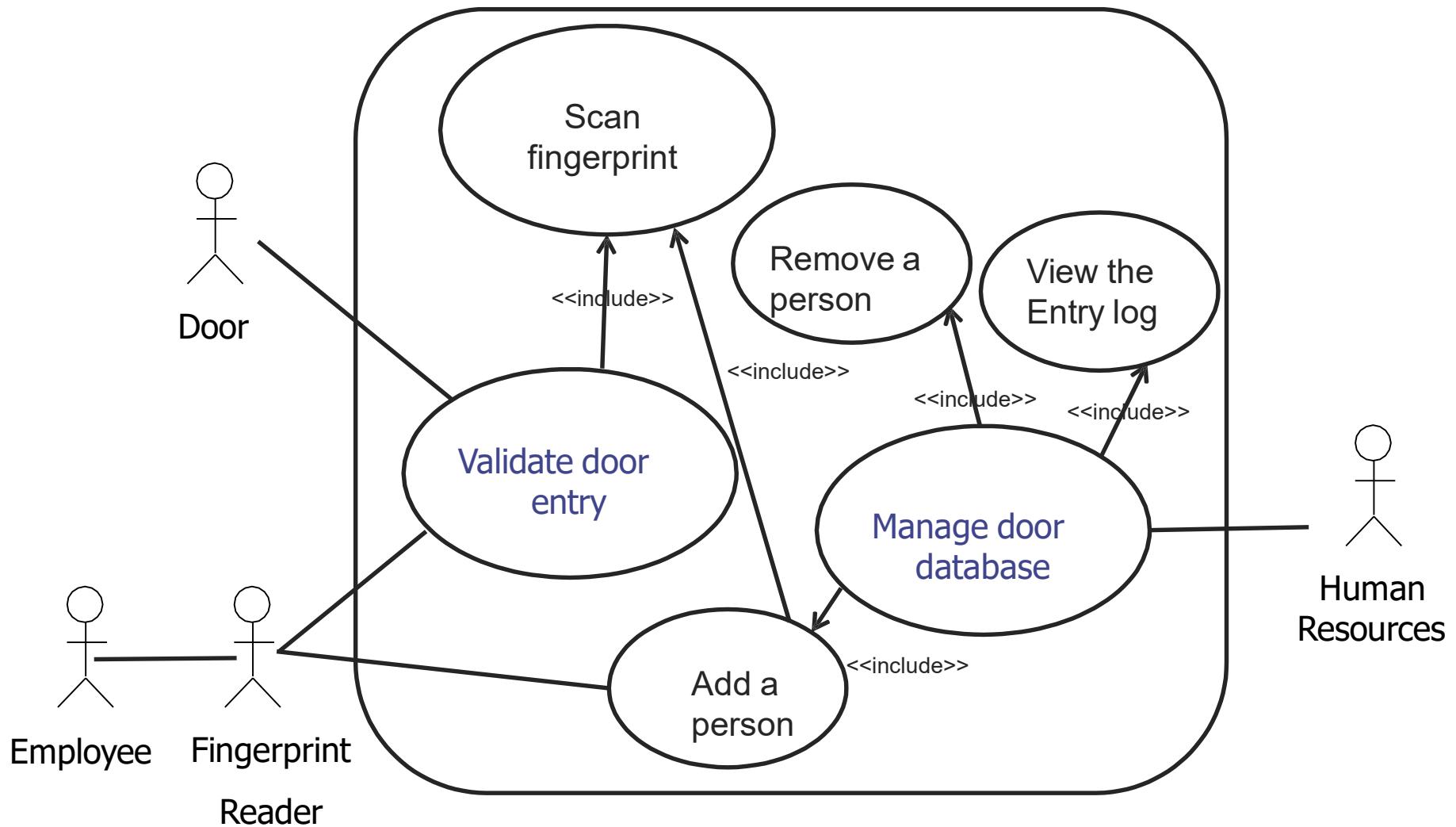
3 & 4. Refine Use Case diagram

REFINING THE MAIN USE CASES



3 & 4. Refine Use Case diagram

REFINING THE MAIN USE CASES



3 & 4. Refine Use Case descriptions

See the Use Case Description Template available on NTULearn course website.

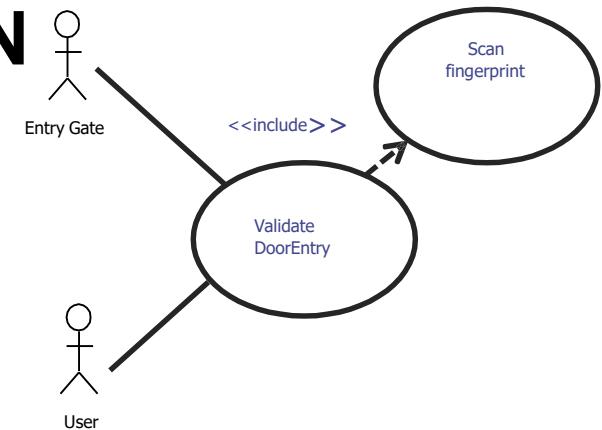
Use Case ID:			
Use Case Name:			
Created By:		Last Updated By:	
Date Created:		Date Last Updated:	
Actor:			
Description:			
Preconditions:			
Postconditions:			
Priority:			
Frequency of Use:			
Flow of Events:			
Alternative Flows:			
Exceptions:			
Includes:			
Special Requirements:			
Assumptions:			
Notes and Issues:			

EXAMPLE USE CASE DESCRIPTION

ValidateDoorEntry Use Case Description

Flow of Events:

1. Employee presses finger on FingerprintReader.
2. FingerprintReader scans fingerprint (using use case Scan fingerprint) and sends image to the FacilityAccessManager.
3. The FacilityAccessManager validates the fingerprint
4. If fingerprint is valid display the message "Please enter" and unlock the EntryGate.
 - 4.1 The EntryGate sends a signal to the FacilityAccessManager that it has opened and closed.
 - 4.2 The FacilityAccessManager locks the EntryGate, and logs the entry time and name of the employee who entered in the entry log.



Alternative Flow:

- AF-S4. If fingerprint is not valid display the message "Access denied" and log the attempted entry time in the entry log.

Example: A security door access system

QUESTIONS THAT ARISE AND NEED CLARIFICATION WITH THE CLIENT

1.What is the detailed process when a fingerprint is validated?

What if the gate is opened and no one enters?

What if the gate is opened and does not close?

Is there a timeout?

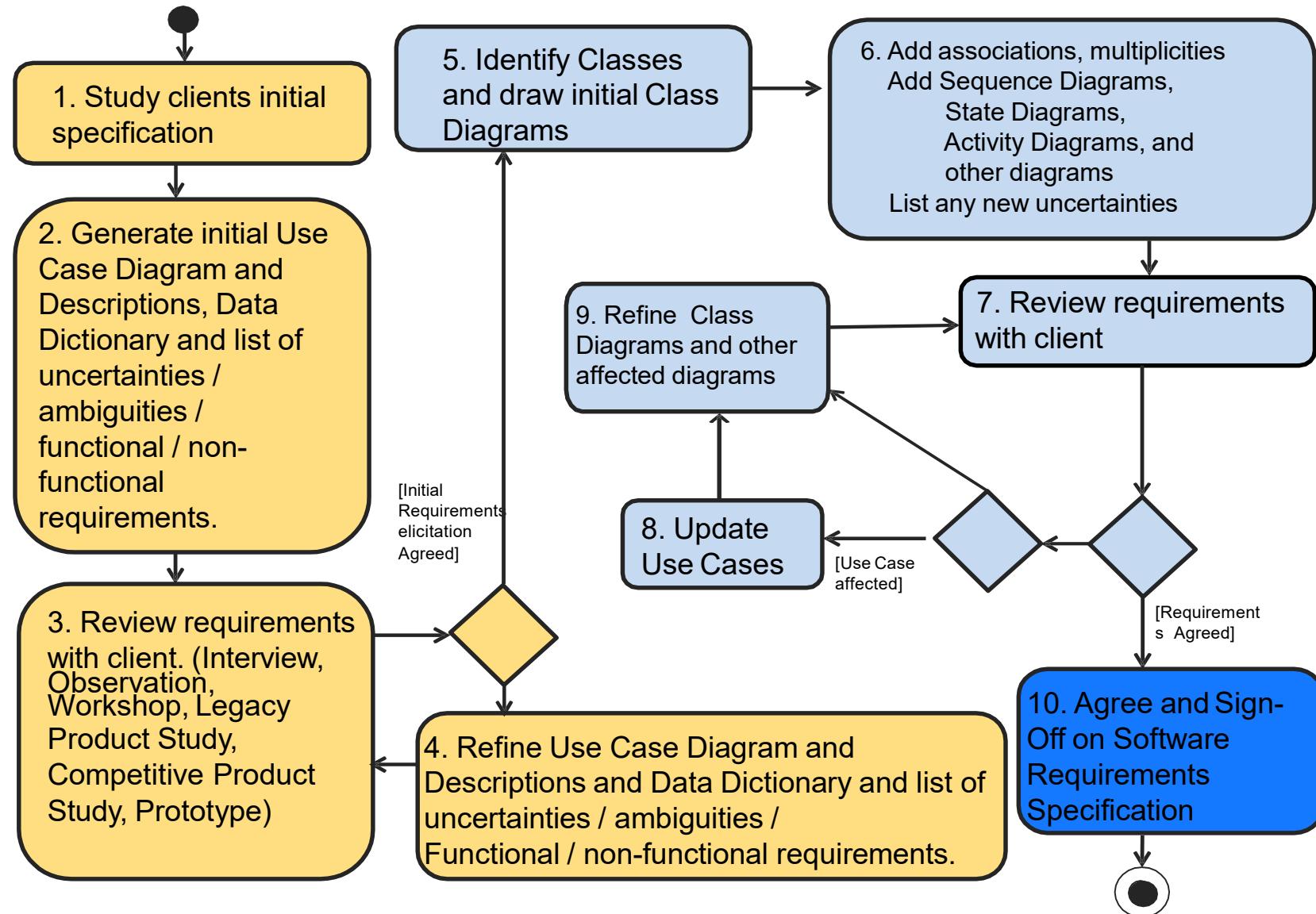
What if more than one person enters?

2.Is there a particular format the Employee data is to be recorded in the log?

Does it need to be exported to another system or printed?

Other questions will arise as detailed Use Cases are written. These need to be clarified with the customer reflected in changes in the Use Case diagrams and descriptions

Summary: Requirement Elicitation and Analysis Process



Software Engineering

CE2006/CZ2006

UML Review 2

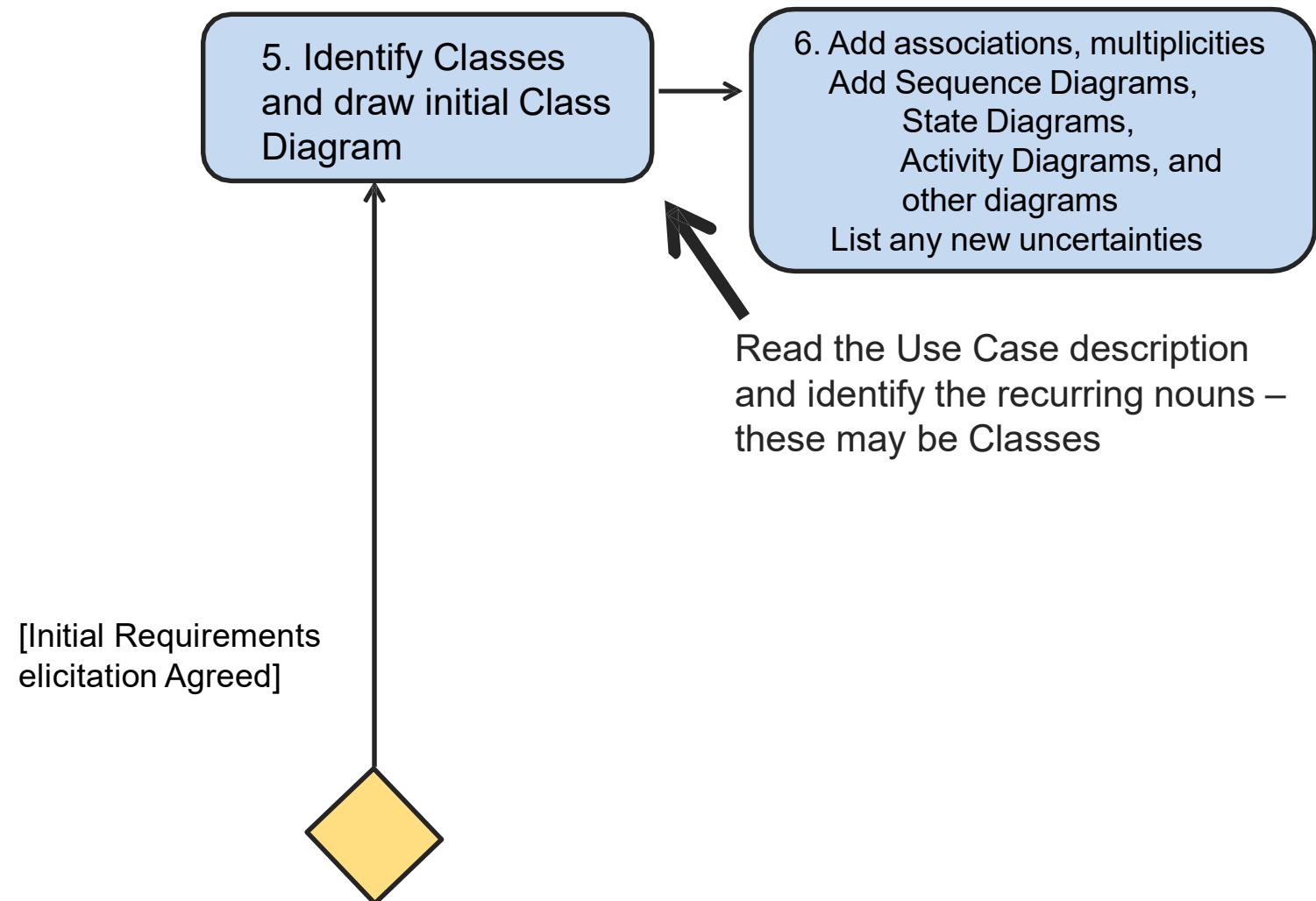
Lesson Objectives

Review UML with an example

At the end of the lesson, you should be able to:

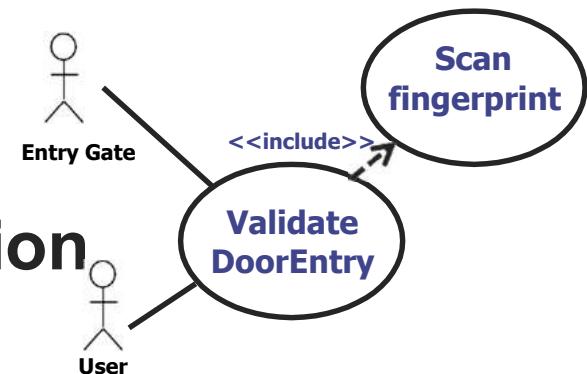
- Describe UML diagrams learned so far
- Explain the purposes of different kinds of UML diagrams
- Use different diagrams at different steps in software requirement elicitation and analysis process

Second Step – Generate Class Diagram



5. Identify Classes

ValidateDoorEntry Use Case Description



1. Employee presses finger on FingerprintReader.
2. FingerprintReader scans fingerprint (using use case Scan fingerprint) and sends image to the FacilityAccessManager.
3. The FacilityAccessManager validates the fingerprint.
4. If fingerprint is valid display the message "Please enter" and unlock the EntryGate.
 - 4.1 The EntryGate sends a signal to the FacilityAccessManager that it has opened and closed.
 - 4.2 The FacilityAccessManager locks the EntryGate, and logs the entry time and name of the employee who entered in the entry log.

AF-S4:

If fingerprint is not valid display the message "Access denied" and log the attempted entry time in the entry log.

5. Draw initial Class Diagram

INITIAL CLASSES AND TYPES



6. Add properties and operations

ADDINIG INITIAL OPERATIONS TO CLASSES



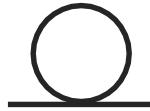
Fingerprint
Reader Interface



EntryGate Interface
Unlock()
Lock()



FASMgr
Validate(Img)
GateOpened()

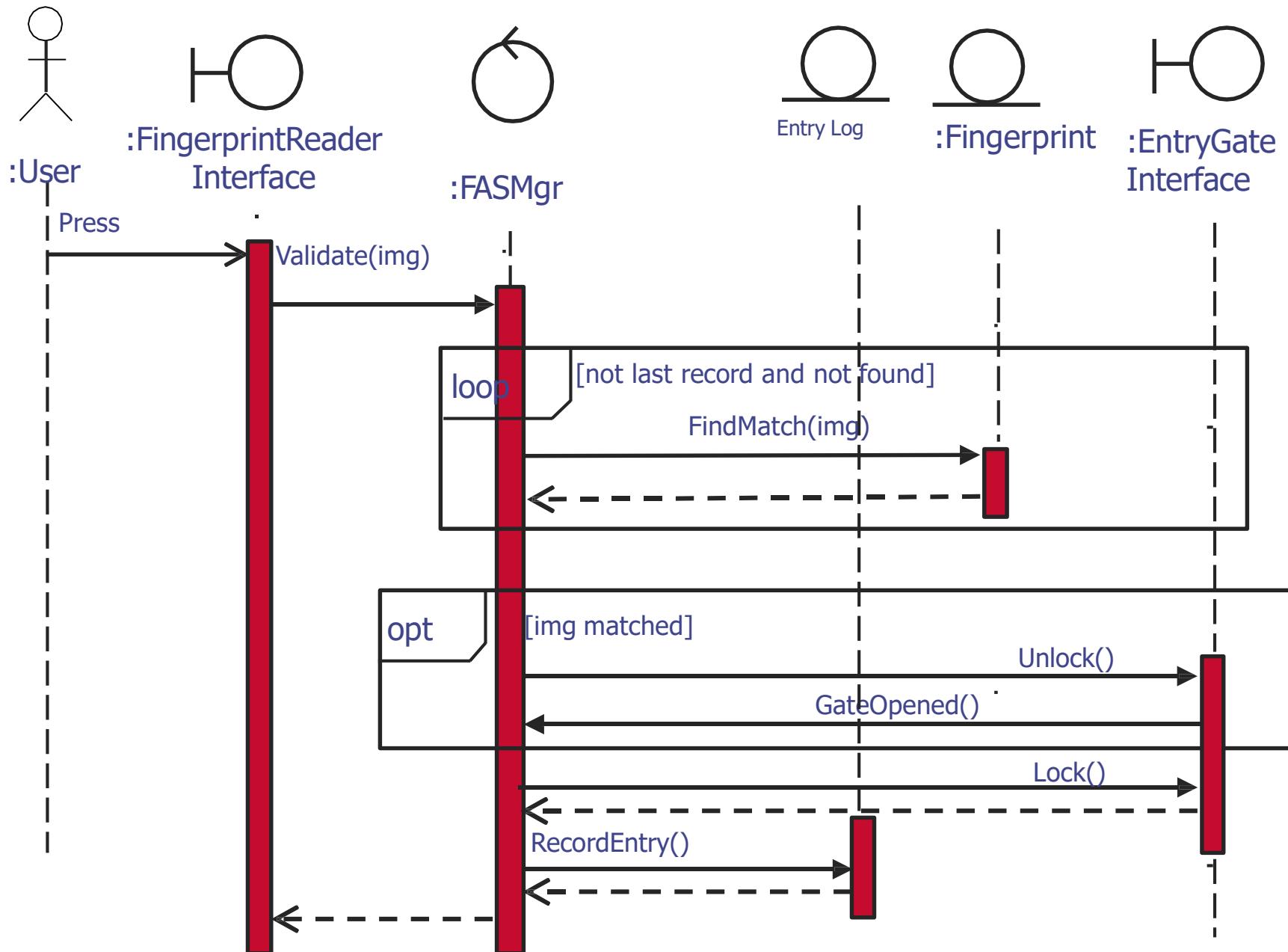


Fingerprint
FindMatch(Img)

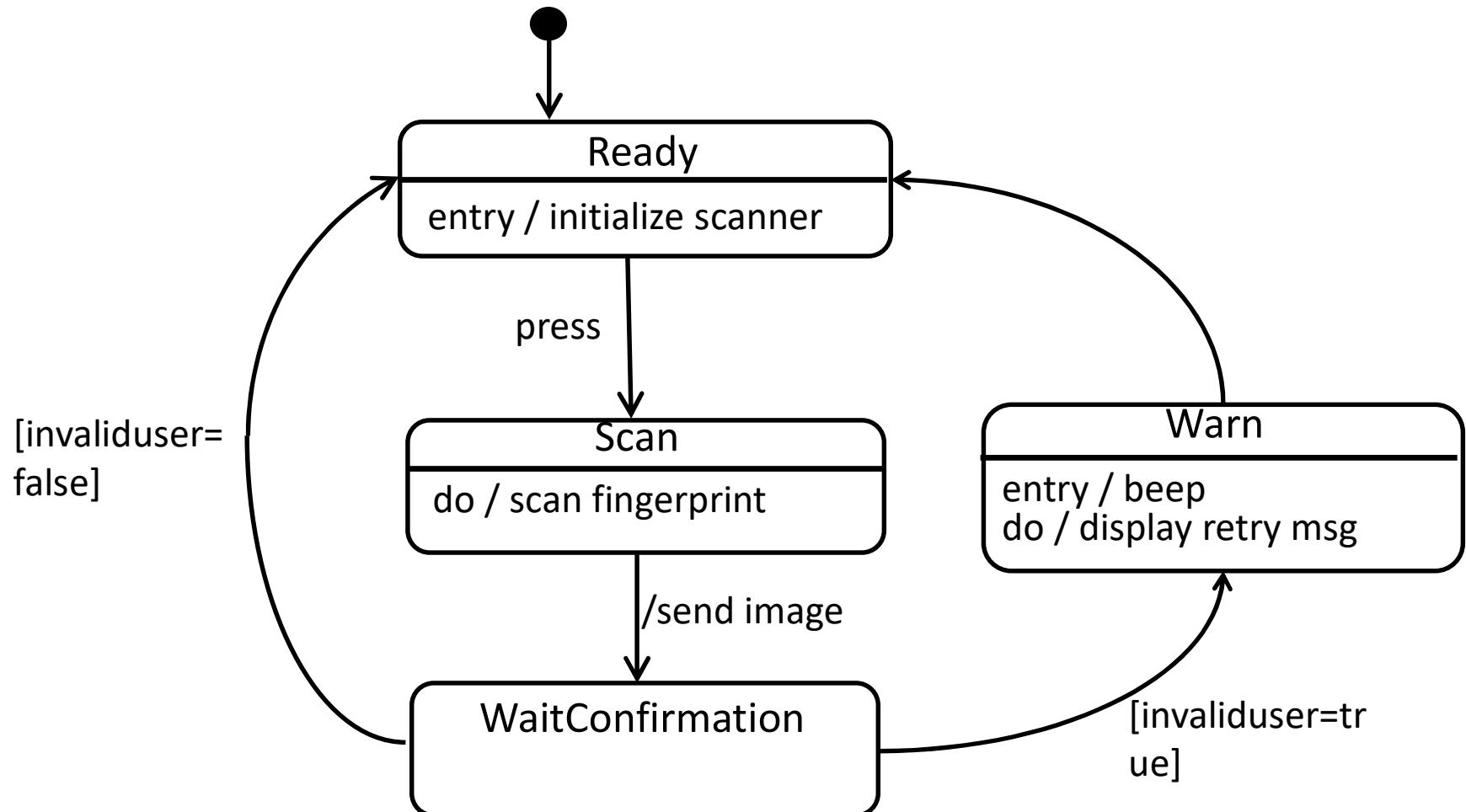


Entry Log
RecordEntry(time: Date)

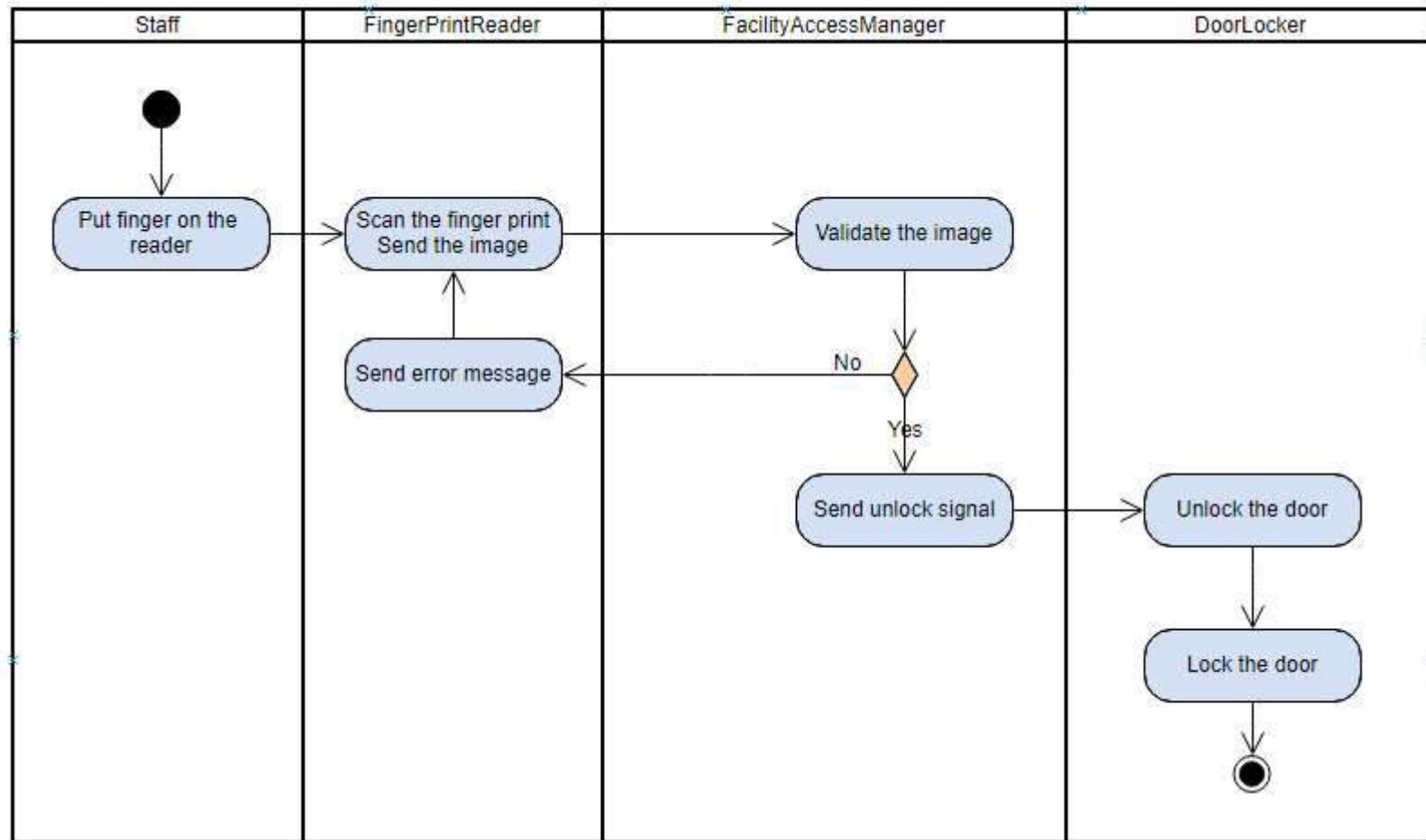
6. Add Sequence Diagrams



Example: FingerprintReader Functionality



Example: Validate Door Entry System

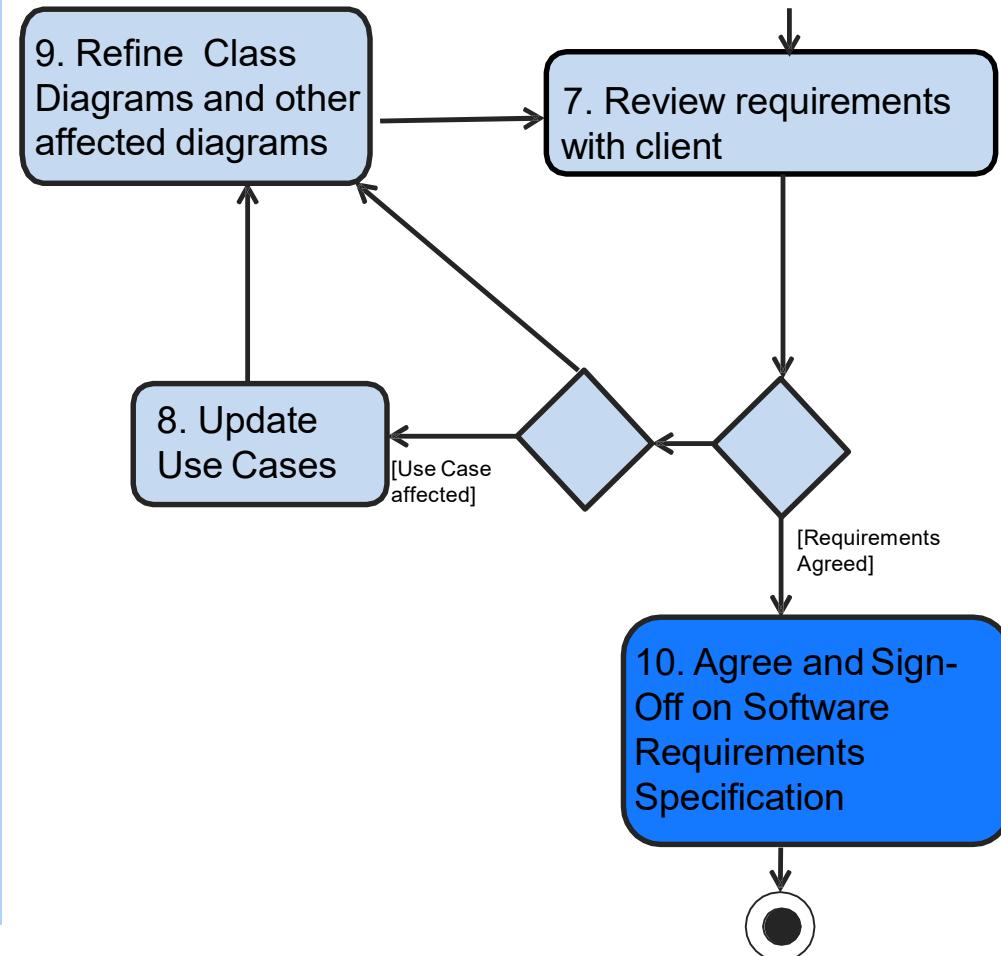


Third Step – Refine Class (& Use Case) Diagrams

Iteratively refine the Class Diagram and Sequence Diagram.

This may involve some refinement of the Use Cases.

Repeat until the Functional Requirements of the System are fully specified.



Summary

UML Review with an Example

First Half of the Course

Introduction to Software Engineering
Requirement Elicitation

 Use Case Diagram

Requirement Analysis

 Class Diagram

 Sequence/Communication Diagram

 State Machine Diagram

 Activity Diagram

Second Half of the course

Software Design

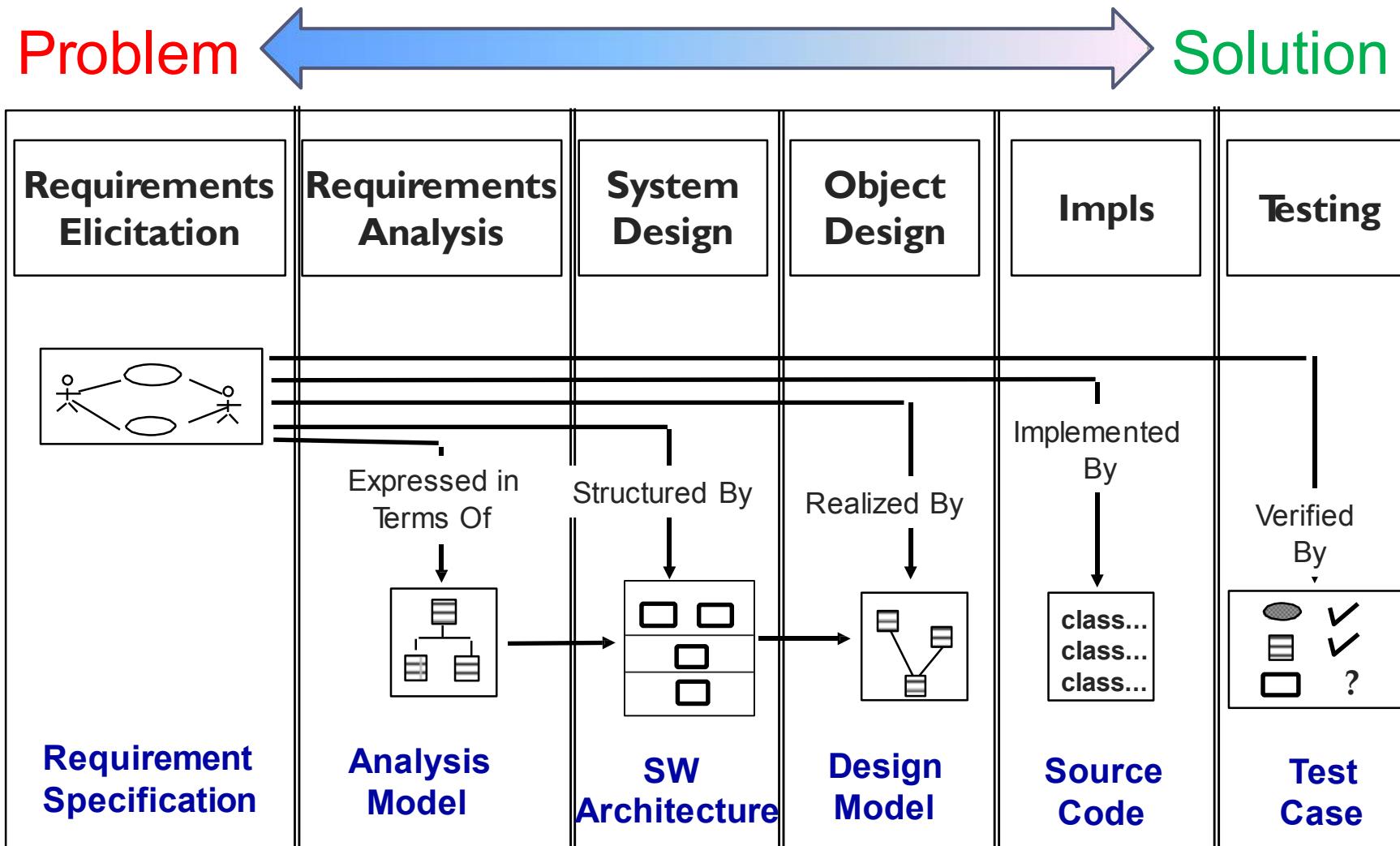
Software Testing

CZ2006/CE2006 Software Engineering

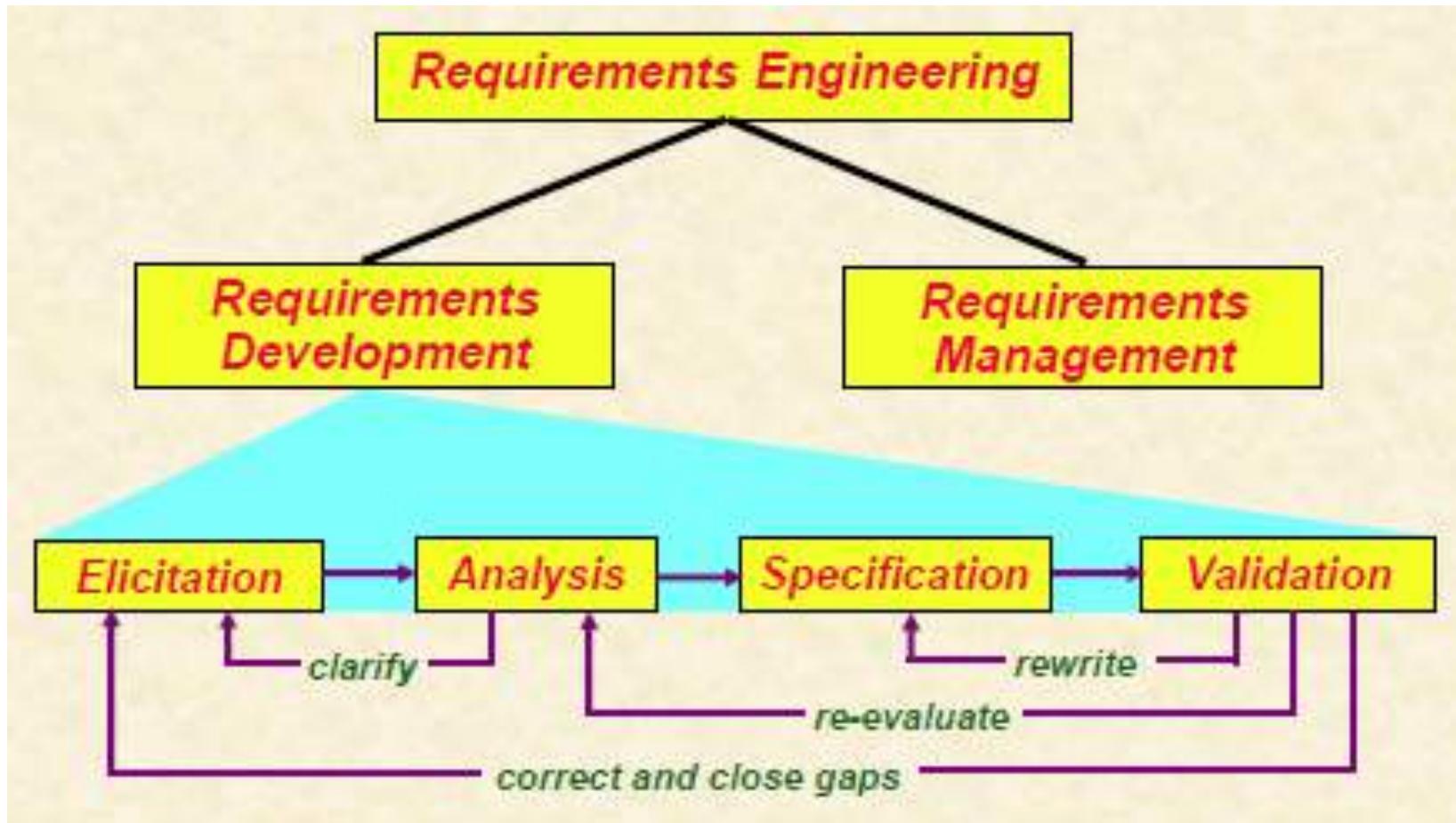
Lecture 12: System Design

Liu Yang

SDLC Activities

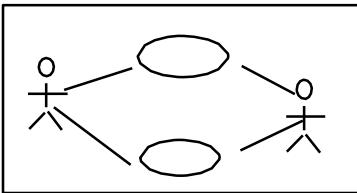


Components of Requirements Engineering



Understand What Customers Want

Requirements Elicitation



Requirement Specification (SRS)

- ▶ From project mission statement
- ▶ To Requirement specification (SRS)
 - ▶ Func & non-func requirements
 - ▶ Data dictionary
 - ▶ Use case model

Informal
Vague
Incomplete

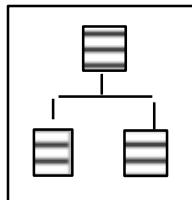
Formal
Technical
Consistent

A vertical double-headed arrow pointing upwards, spanning the height of the two columns of text. The top half of the arrow is blue, transitioning to purple at the bottom, symbolizing a spectrum or progression between the two states described.

Bridge Customer Perspective to Software Engineer Perspective

Requirements Analysis

Expressed in
Terms Of



Analysis Model

From Requirement Specification (SRS)

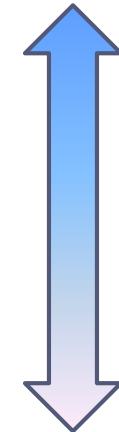
- ▶ Func & non-func requirements
- ▶ Data dictionary
- ▶ Use case model



To analysis model

- ▶ Class diagram (boundary, control, entity)
- ▶ Sequence diagram
- ▶ State machine

More
about
Customer

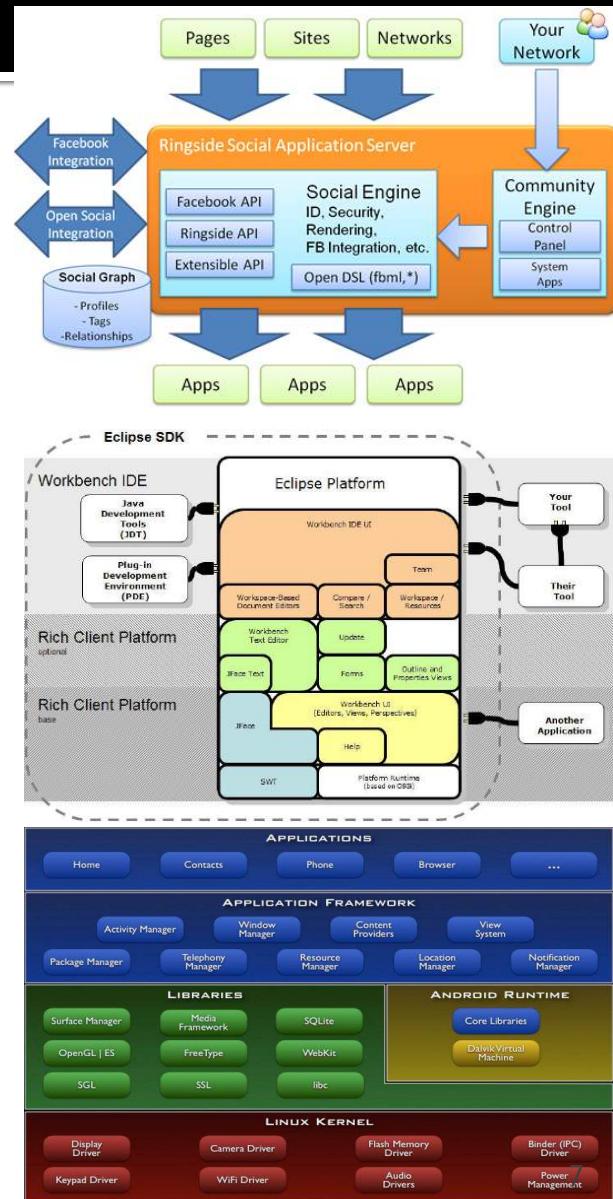


More
about
**Software
Engineer**

System Architecture Design, Why?

The Reasons 1

- Larger
 - Source size, Binary size
 - Number of users, user input
 - Number of developers
 - Life of the project (decades...)
 - Number of changes, versions
- Complexity is increasing
 - Environment
 - Technologies
 - Hardware
 - ...



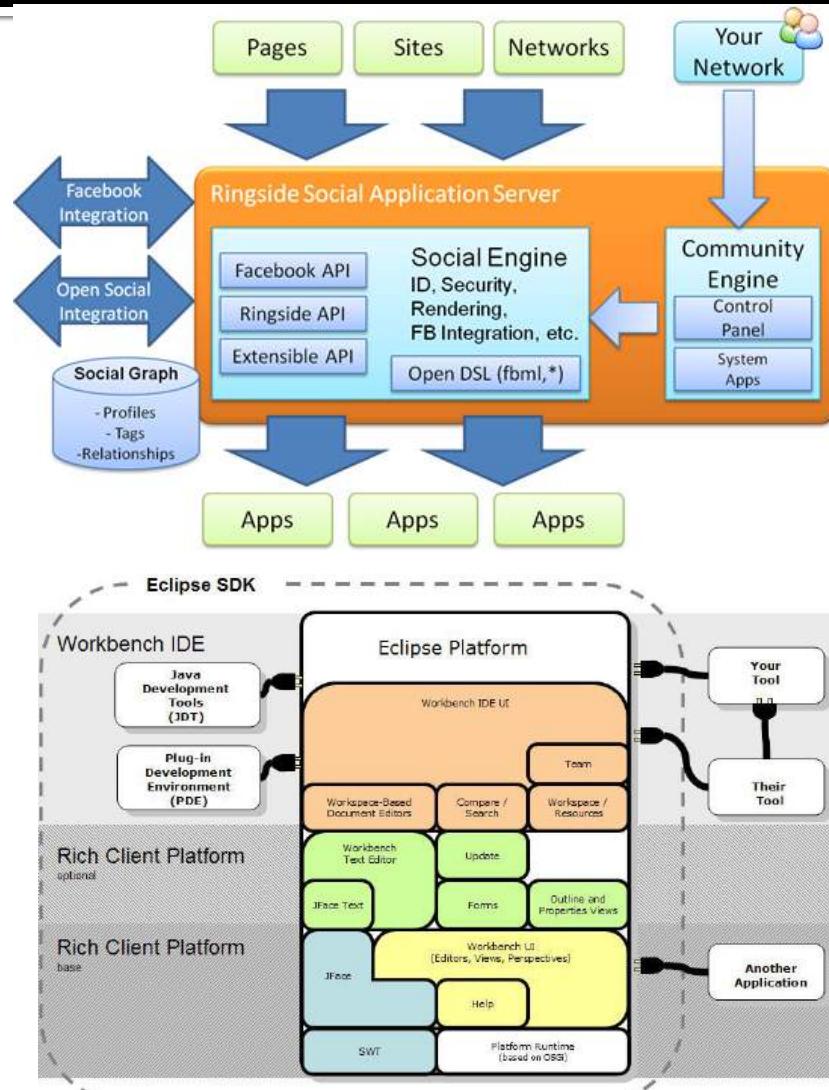
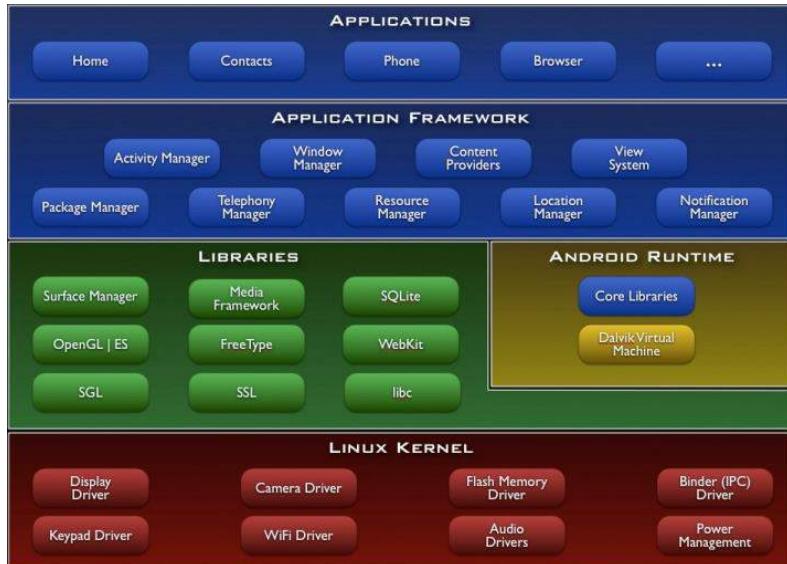
The Reasons 2

- Cost is critical (limited budget/resource)
- Schedule constraints (tight schedule)
- Maintenance

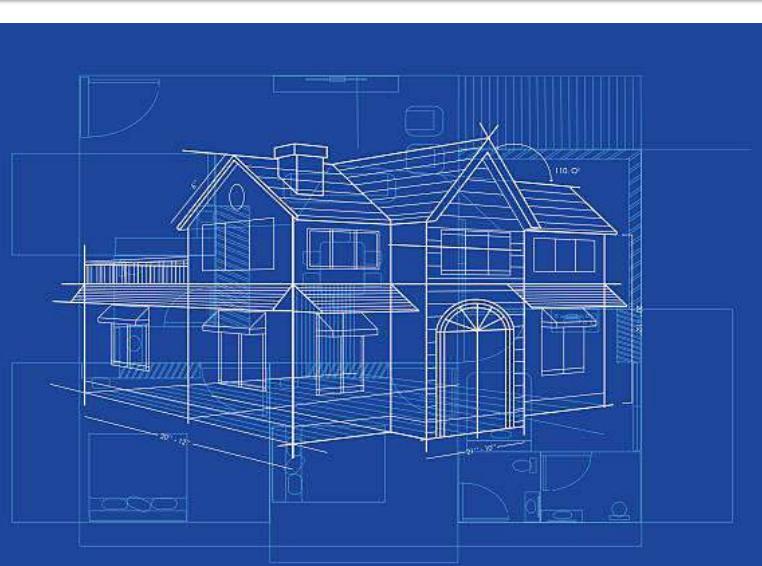


The Reasons 3

- Non-functional Requirements
 - Performance
 - Availability
 - Security
 - Scalability
 - Extensibility



Software System Architecture Design



Software Architecture Definition

Software architecture is considered as a description of the high level structure of a software system in terms of architectural elements and the interactions between them.

*'Abstractly, software architecture involves the description of **elements** from which systems are built, **interactions** among those elements, **patterns** that guide their **composition**, and **constraints** on these patterns.'*

(Shaw and Garlan, 1996)

Shaw and Garlan's Definition

Software Architecture = {Components, Connectors}

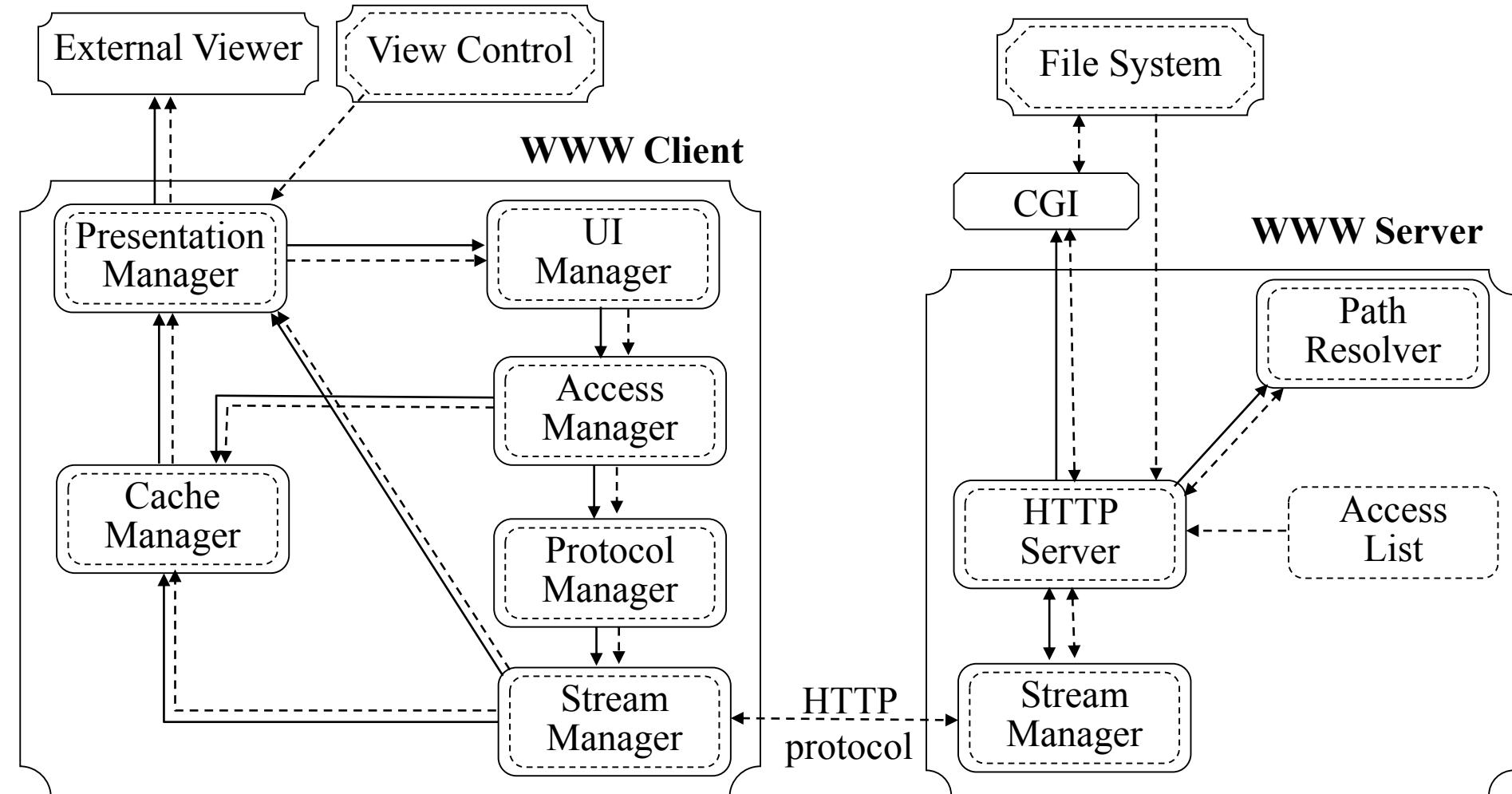
■ *Components*

- A component is a unit of software that performs some function at run-time.
- Examples: programs, objects, processes, clients, servers, databases

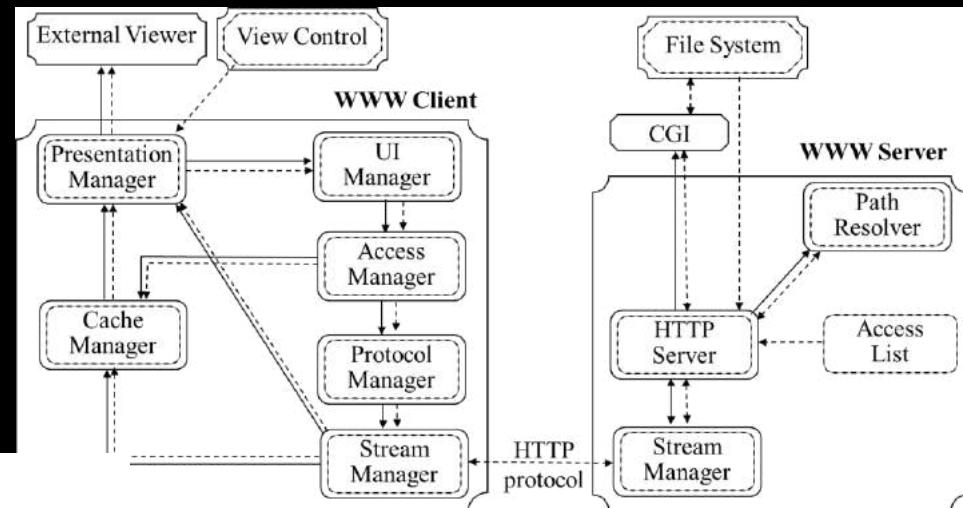
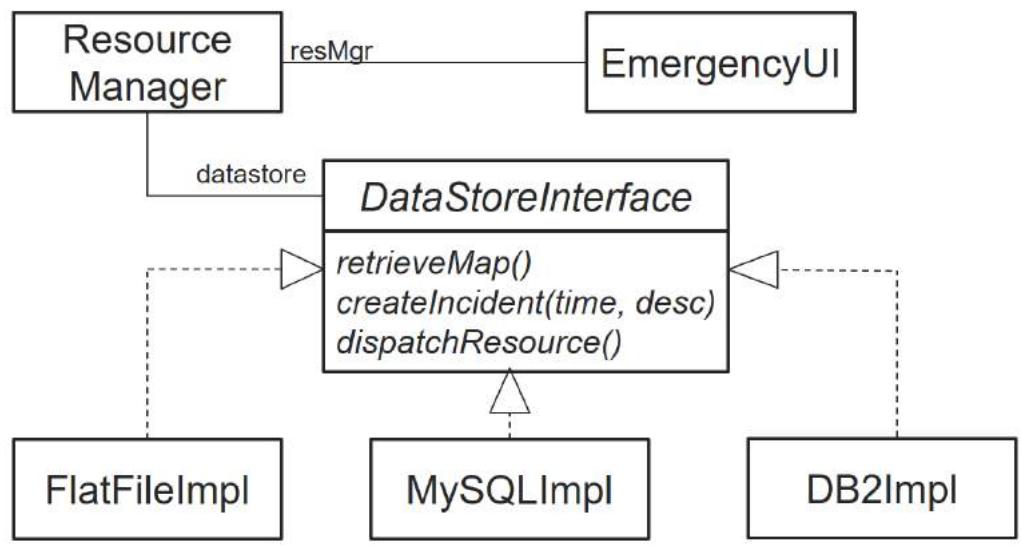
■ *Connectors*

- Interactions among components:
 - communication, coordination, or cooperation among components.
- Examples: shared variable access, procedure calls, remote procedure calls, communication protocols, data streams, transaction streams.
- Implementations of connectors are usually distributed over many system components; often do not correspond to discrete elements of the running systems.

WWW Client-Server Architecture



Differences between class diagram and architecture diagram?



Notes about SA

- Architecture is abstract
 - The details of the components and connectors are hidden
- Architecture is about structure and interactions
 - Focus on the topology
 - Components
 - Data and control communication
- Architecture is purposeful
 - demonstrate or analyse the properties of interest
 - design documentation, as transferable knowledge about software design, evaluation, and so on

How to develop a system architecture?

Functional Requirement

- Dynamic Models
- Analysis Object Models

Non-functional Requirement

- Efficiency
- Reliability
- Robustness
- Security
- Maintainability

Design Principles

- **Modularity**
- **Abstraction**
- **Open-Closed**
- **Reusability**

Design Patterns

- Layered Architecture
- Client and Server
- Data Centered

Constraints

- Languages
- Libraries
- Communications

Don't repeat yourself

Modularity

Separation of Concerns
Low Coupling and High Cohesion

Open Close Principle

Principle of Least Knowledge

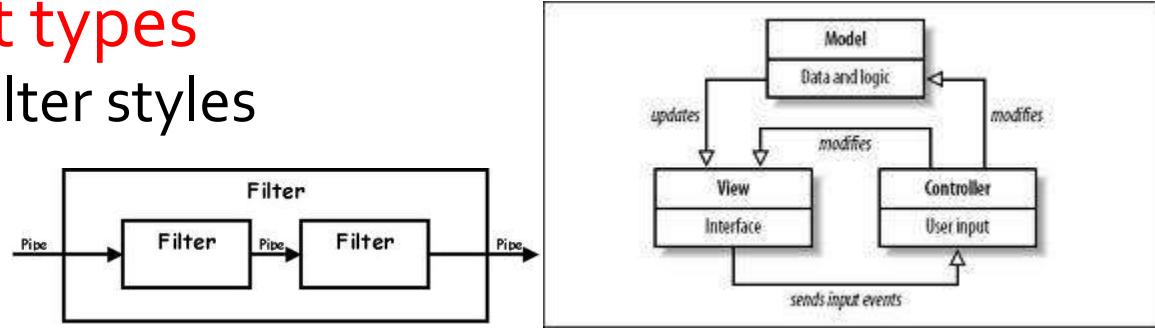
*Single Responsibility
Principle*

Software Architectural Styles

- One of the hallmarks of software architectural design is the use of idiomatic patterns of system organization.
- An architectural style defines a family of systems in terms of a **pattern** of **structural organization**.
 - A vocabulary of component and connector types
 - A set of constraints on how they can be combined
 - One or more semantic models that specify how to determine a system's overall properties from the properties of its parts

What is in a Style

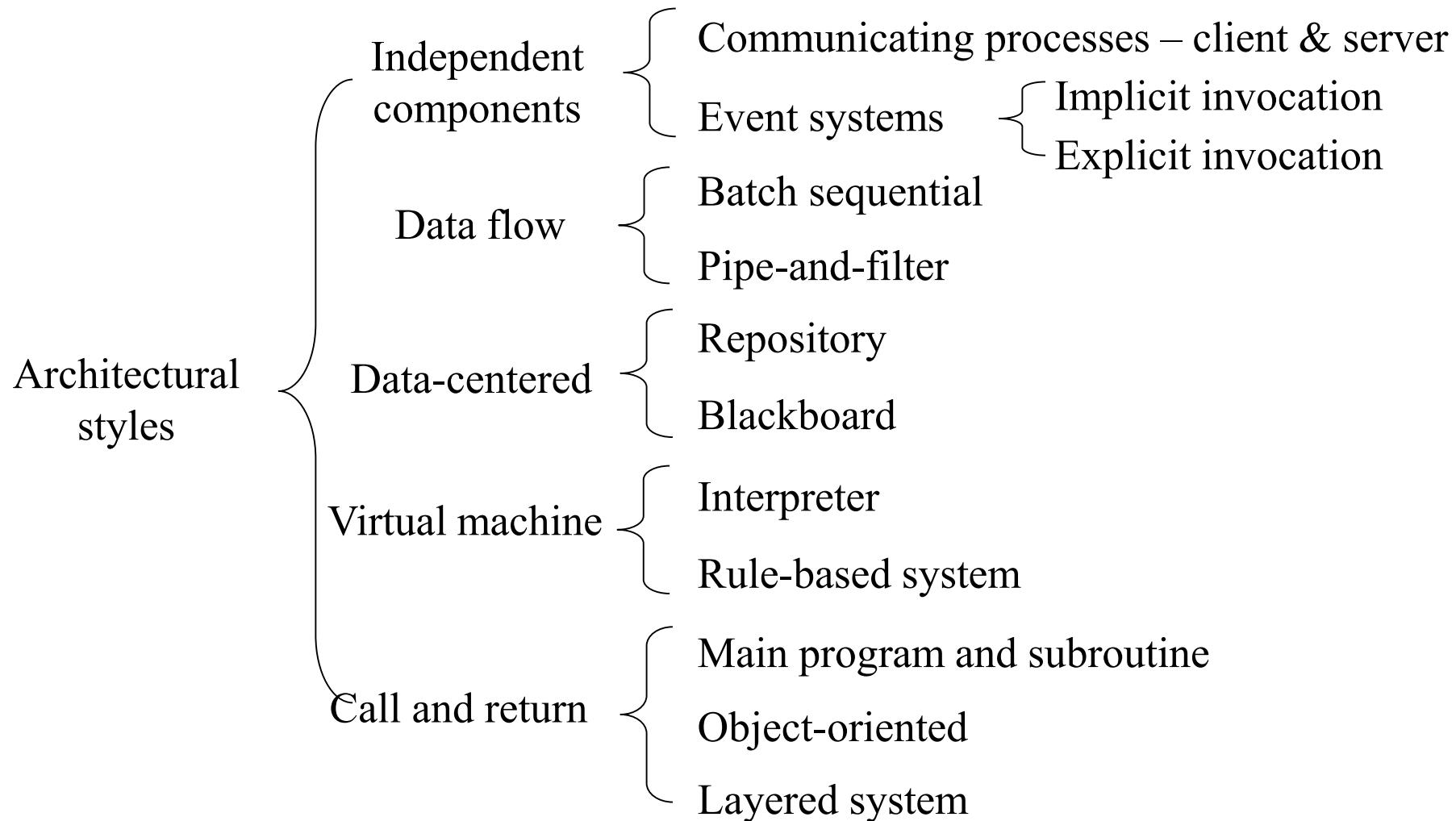
- A set of **component types**
 - Filters in pipe-and-filter styles
 - MVC
- A set of connectors
 - subroutine calls, remote procedure calls, data streams, and sockets
- A topological structure
- A set of semantic constraints
 - Filters in a pipe-and-filter architecture do not share states with each other
 - One layer can only call the service provided by the lower layer in a layered system



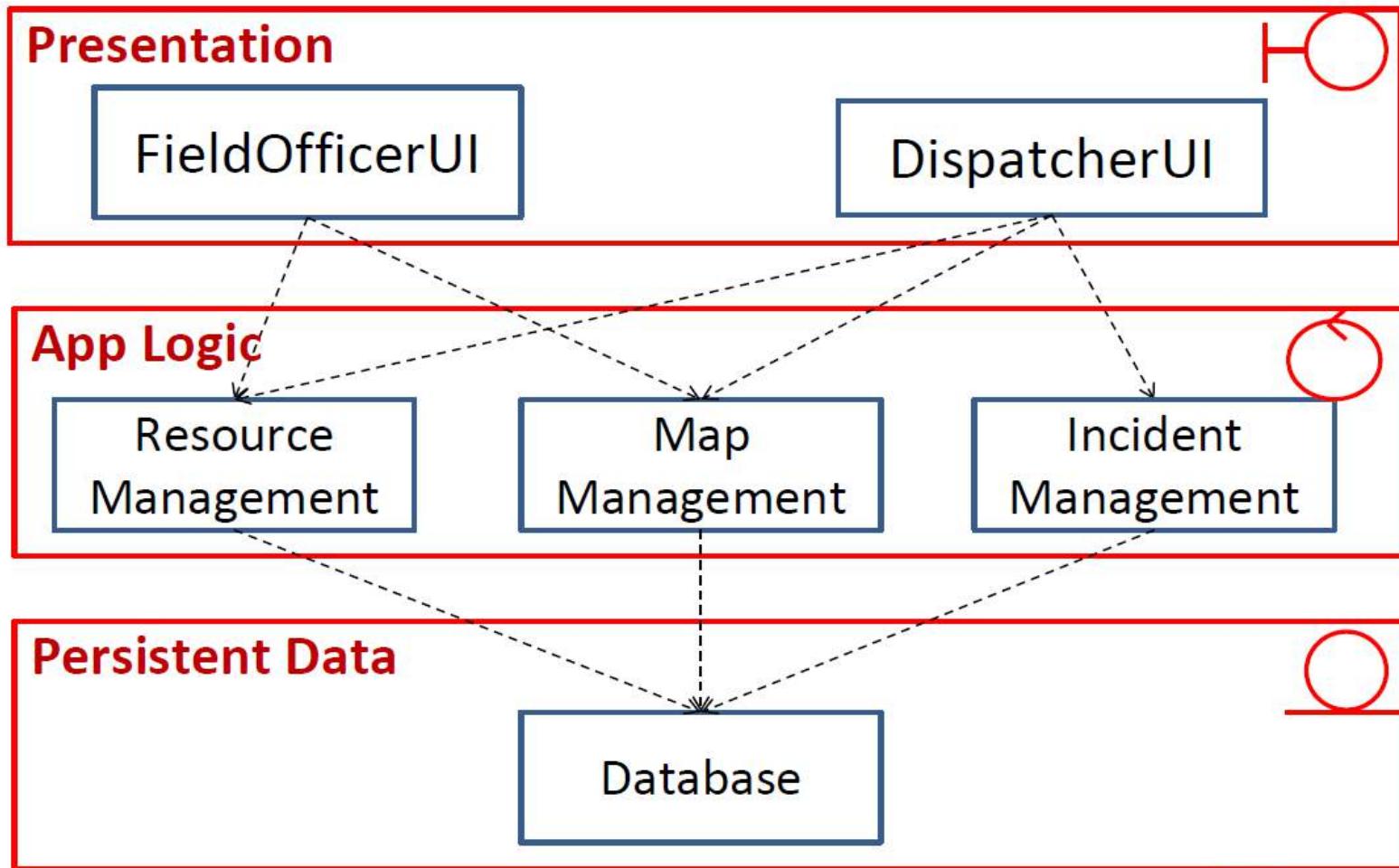
What Is Ambiguous in A Style

- The number of components involved
 - a pipe-and-filter architecture
 - 2 filters connected by 1 pipe or
 - 20 filters connected by 19 pipes
- The mechanism of interaction
 - In a layered system, the call to the lower layer can be
 - local procedure calls, or remote procedure calls, or other process communication mechanisms.
 -
- The function of the system and components
 - one of the components in an architectural style may be a database, but the kind of data may vary.

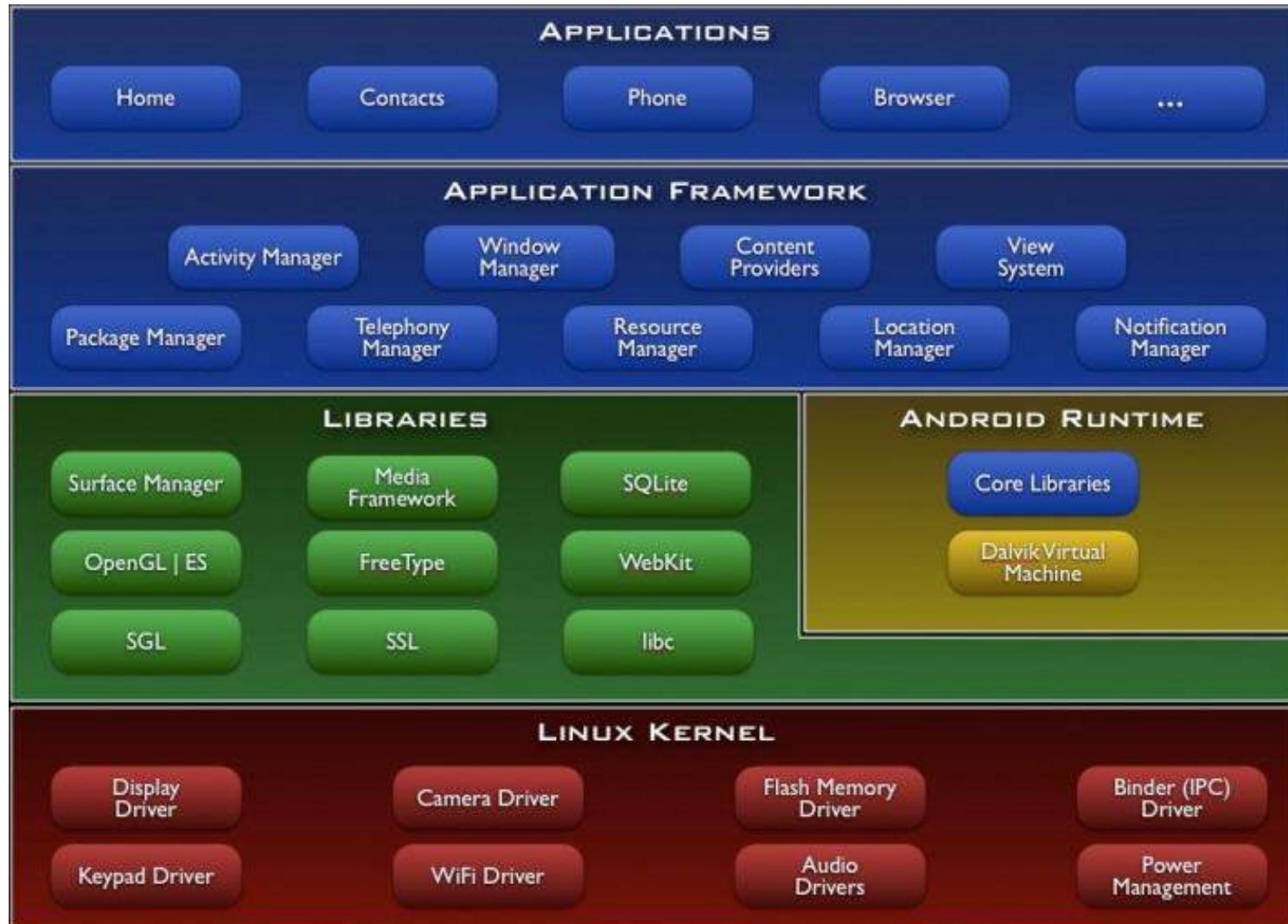
A Catalogue of Software Architectural Styles



Layered Architecture – 3-Layered Architecture



Examples: Android



Analysis of Independent Component

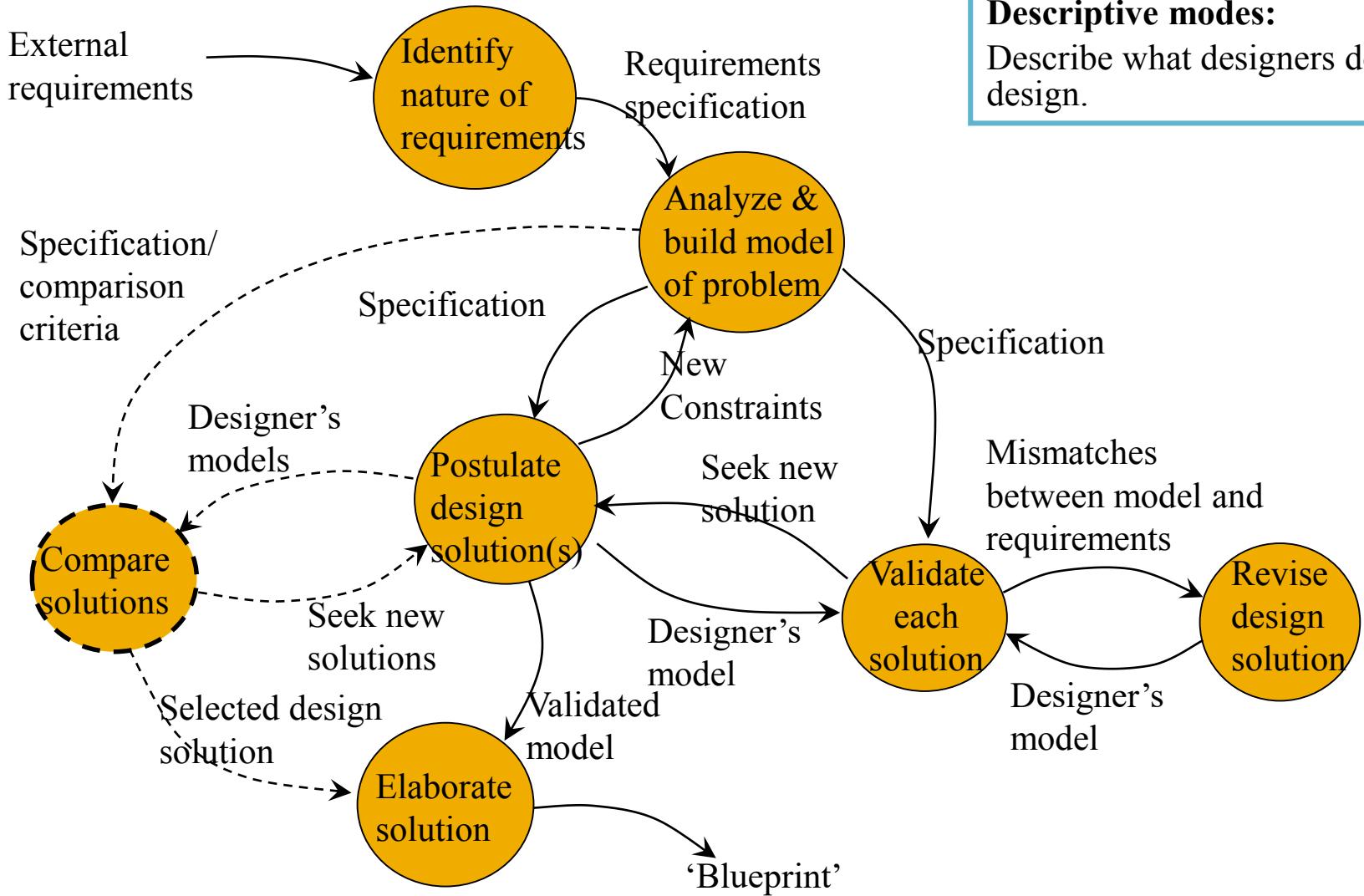
Advantages

- Modifiability by decoupling the computation
- Concurrent execution
- Scalability
- Easy integration

Disadvantages

- Components are independent
 - Message received?
 - Solution: complicated protocols
- Correctness is hard to achieve

A Generic Process Model of Designs



Descriptive modes:

Describe what designers do in design.

Example of Evolution



Maintainability



New Features



Extensibility



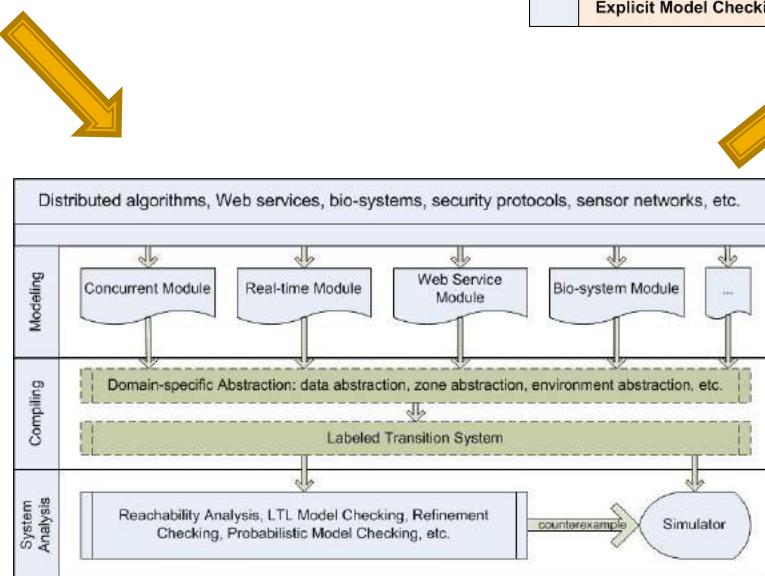
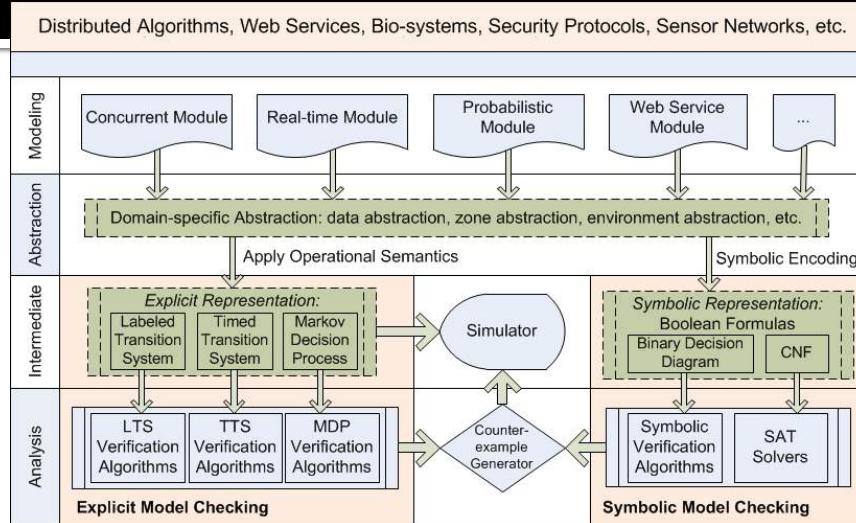
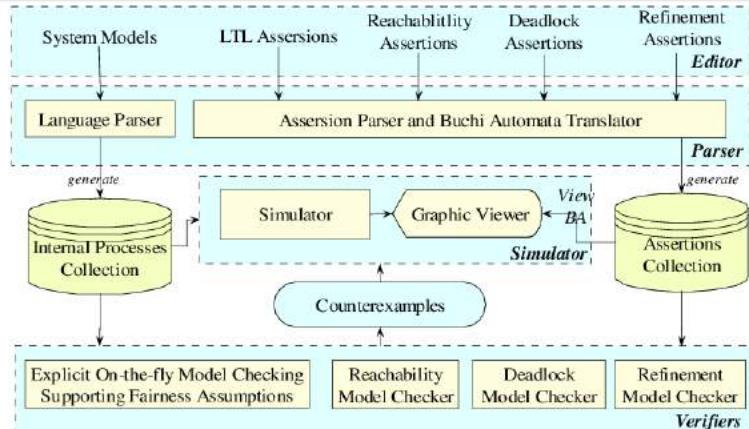
New Features



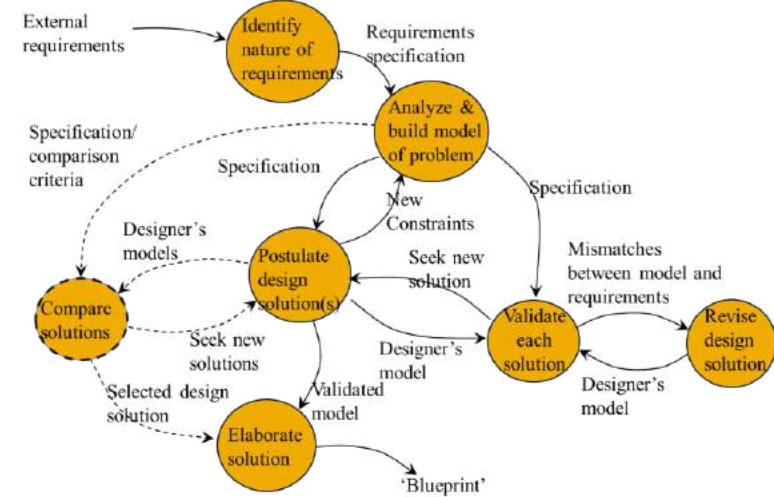
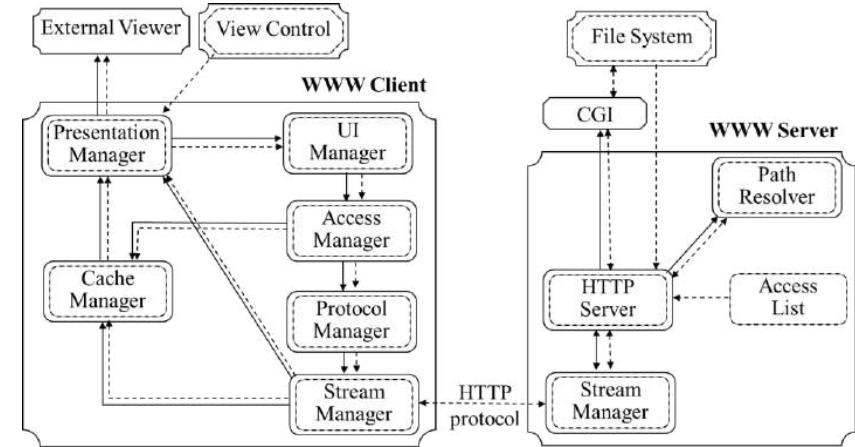
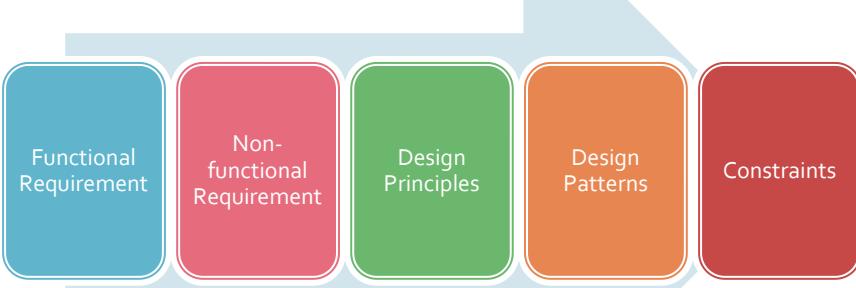
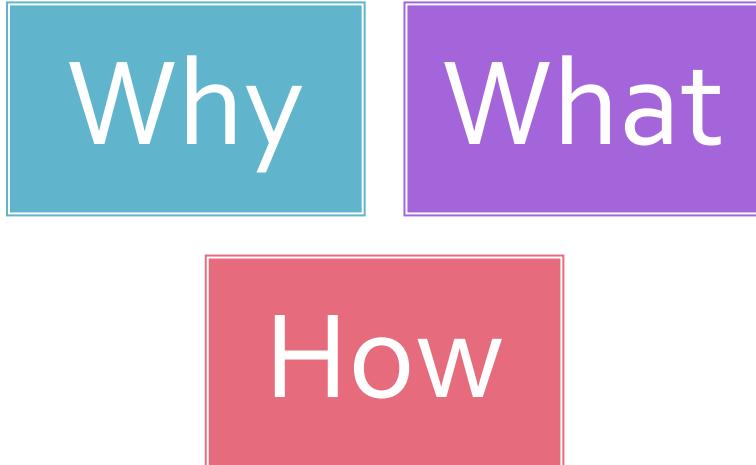
Performance
Scalability



Real Architecture Diagrams



Summary of System Design



CZ2006/CE2006 Software Engineering

Lecture 13: Object Design

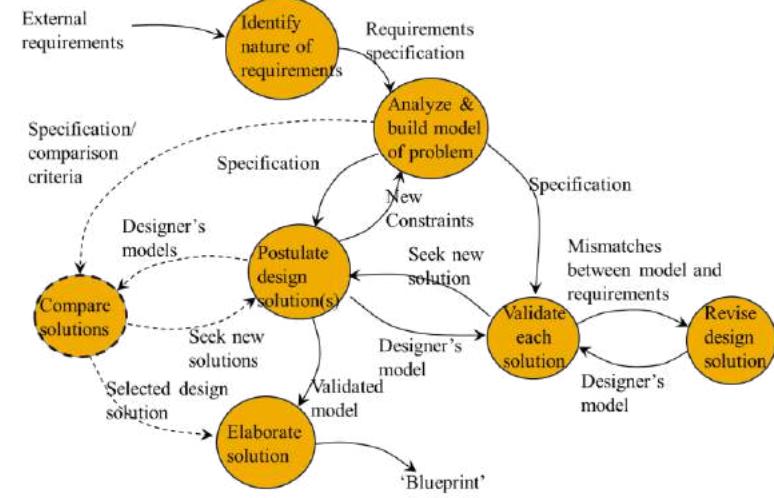
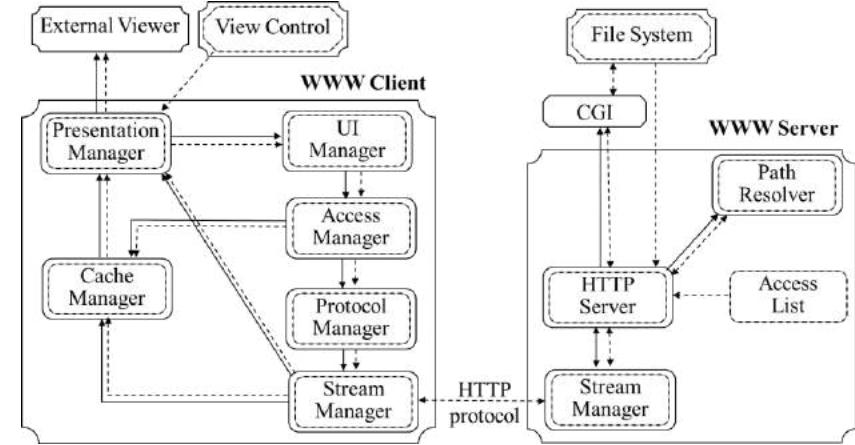
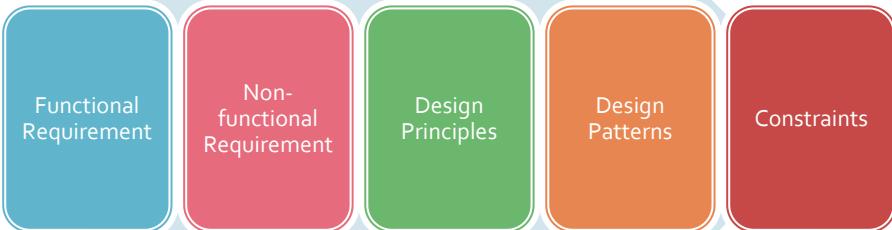
Liu Yang

Recap of the System Design

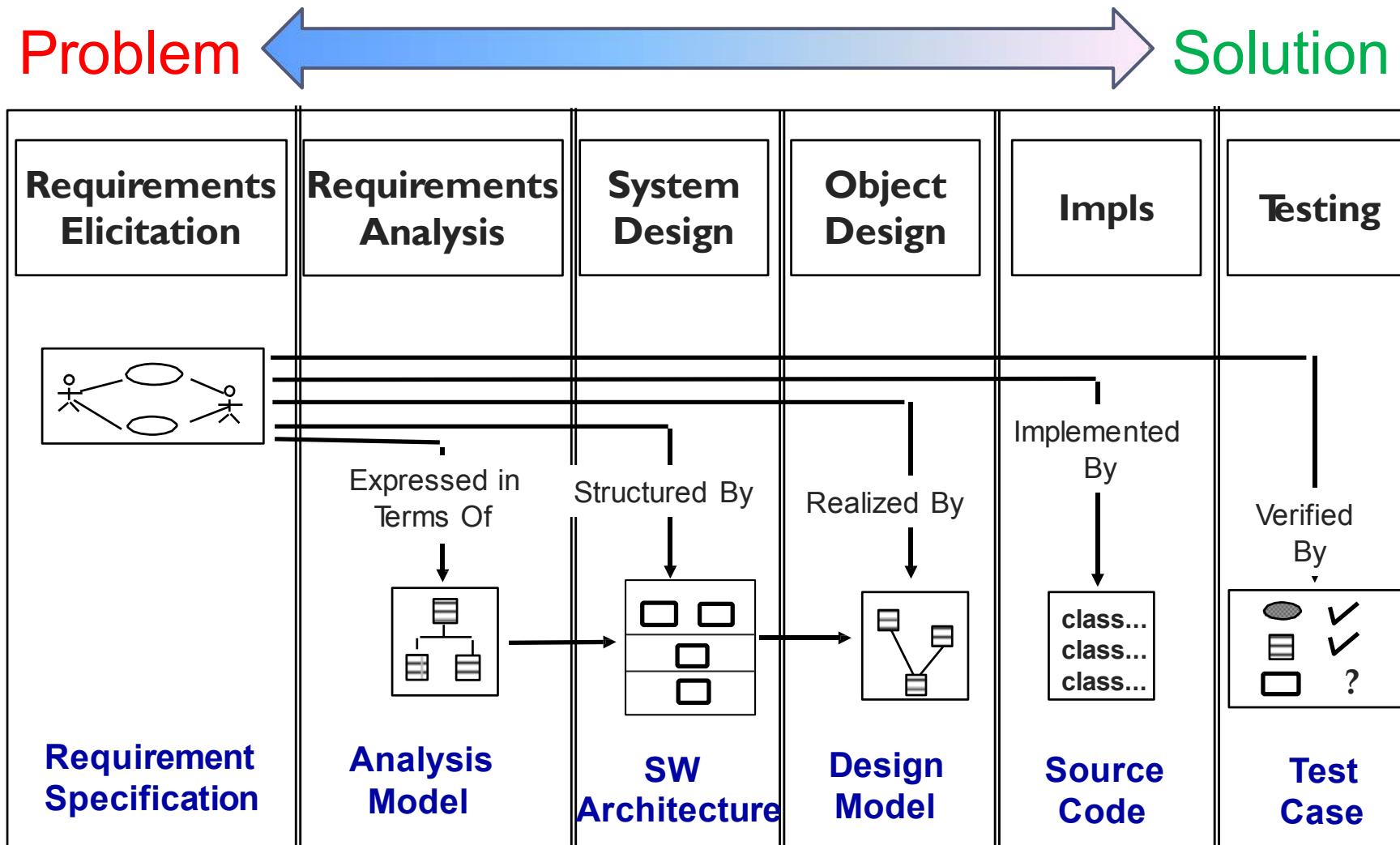
Why

What

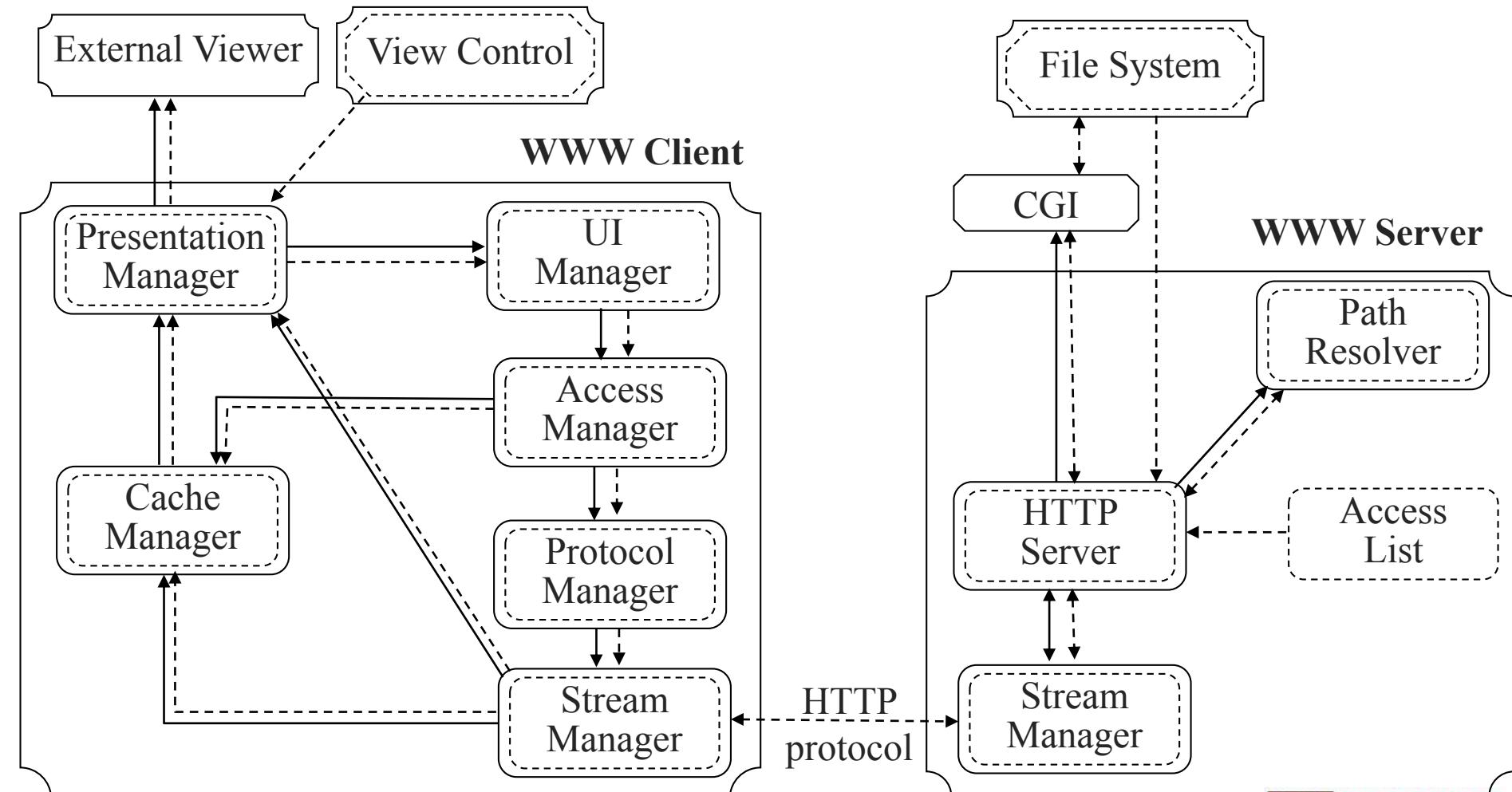
How



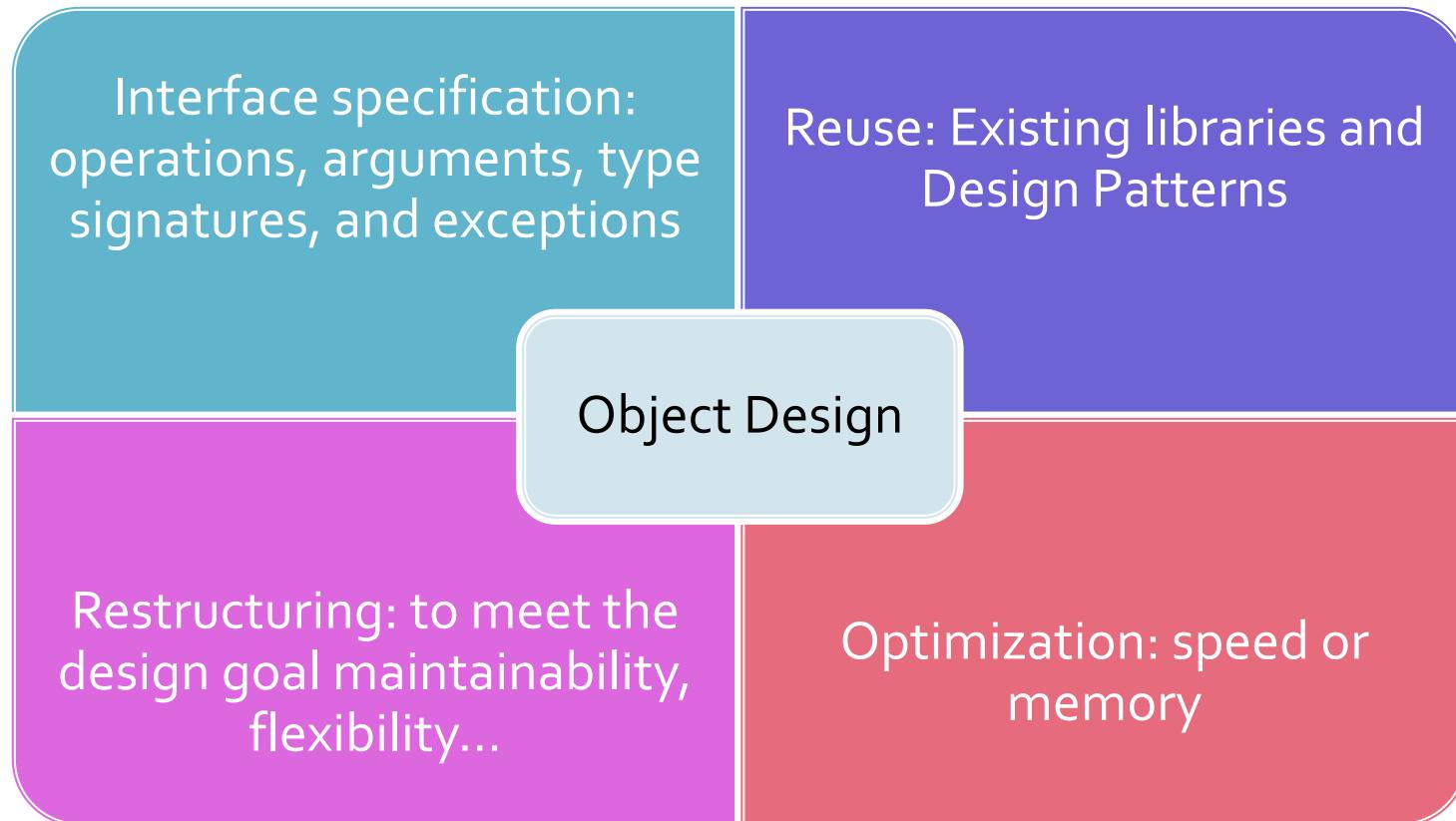
SDLC Activities

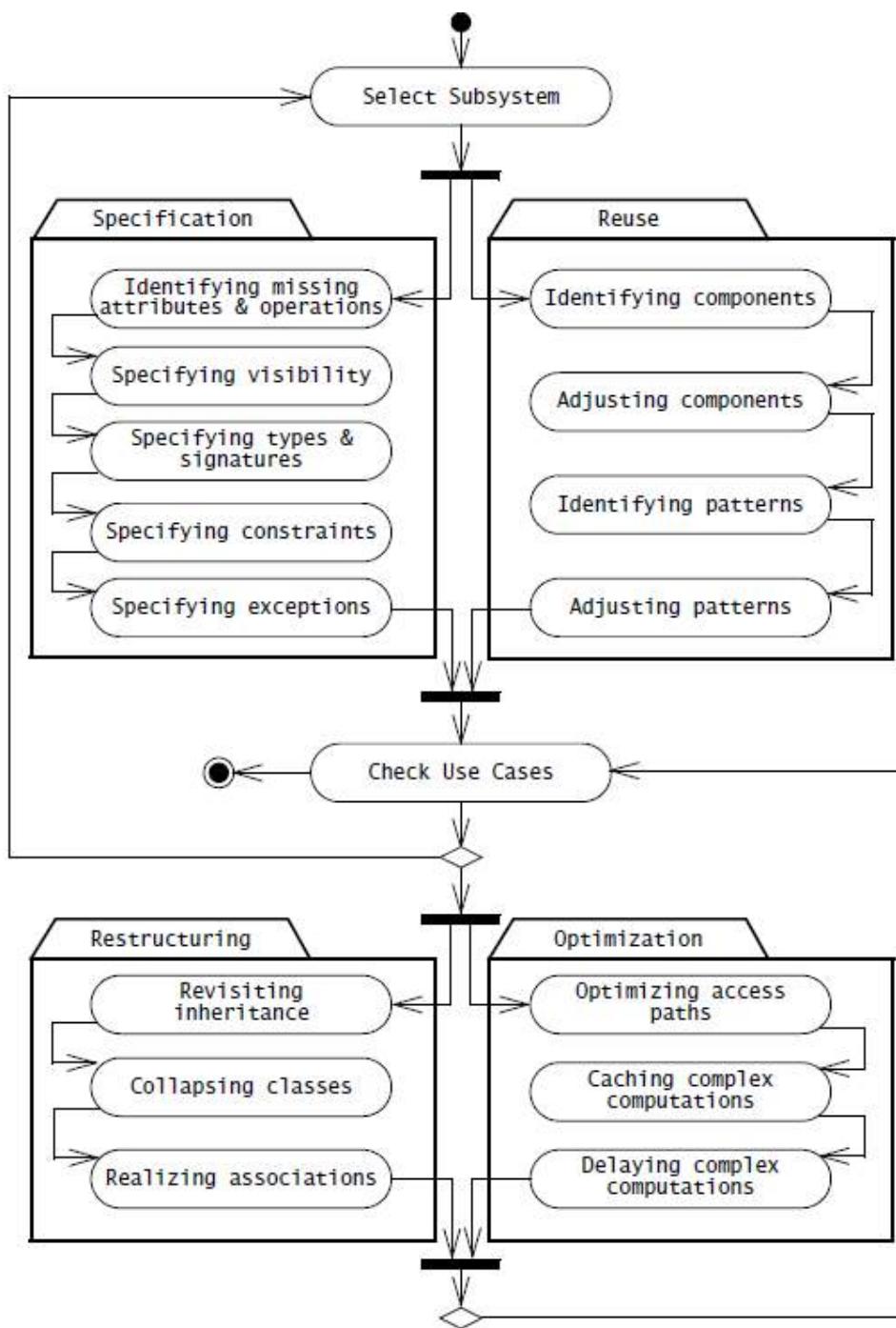


WWW Client-Server Architecture

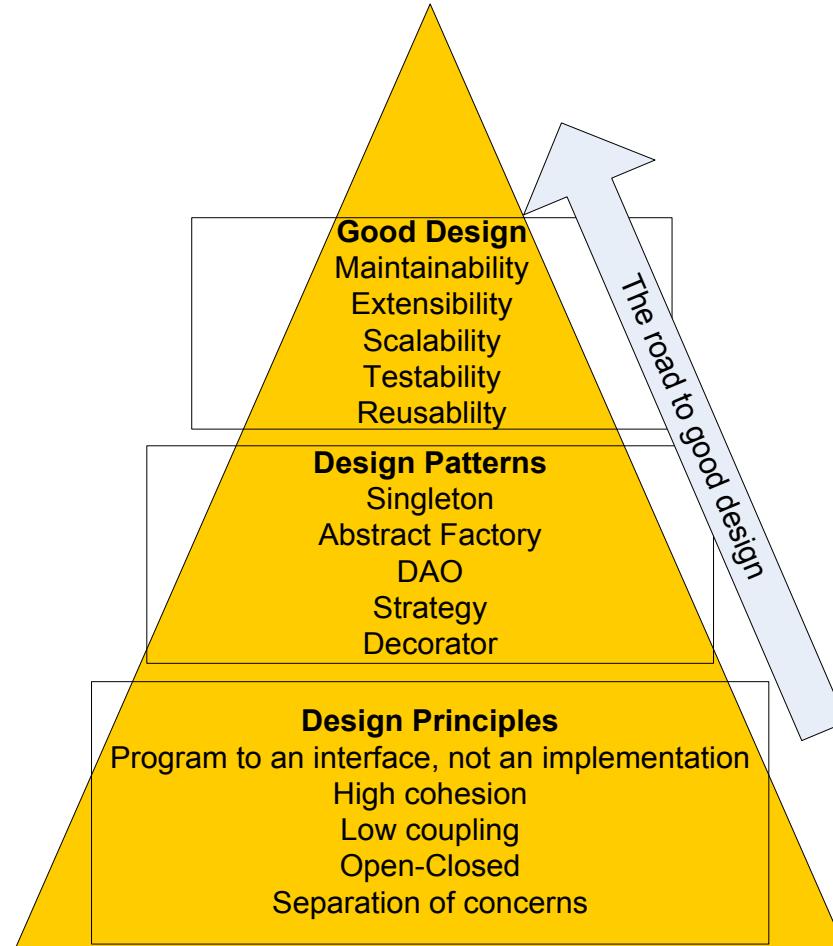


Object Design



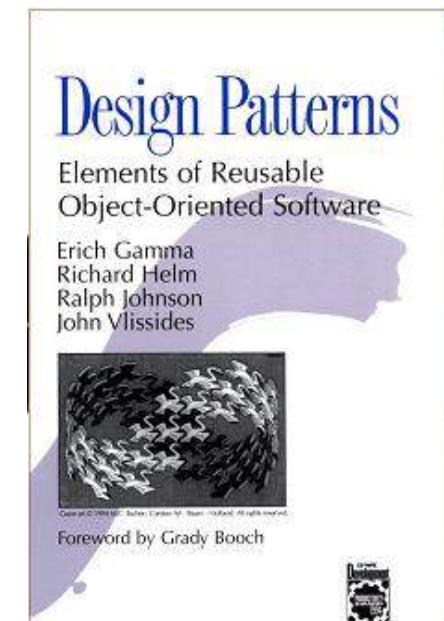


Why use Design Patterns?



What is a Design Pattern?

- A design pattern
 - a proven solution to a problem in a context.
 - abstracts a recurring design structure
 - A template with class and/or object
 - dependencies,
 - structures,
 - interactions, or
 - conventions



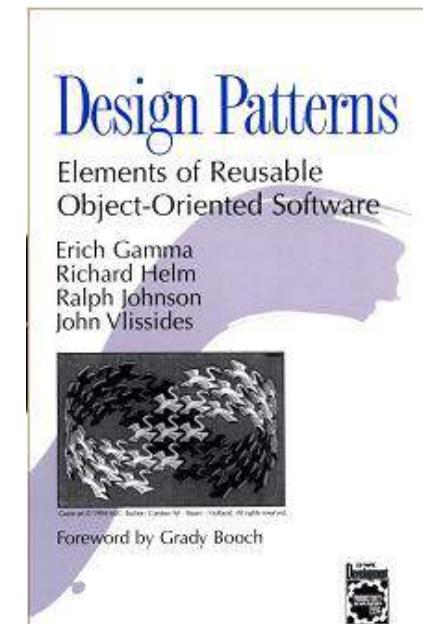
“Gang of Four” (GoF) Book, 1996

4 Elements of a Design Pattern

- **Name**
 - Describes the pattern
 - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
- **Problem**
 - Describes when to apply the pattern
 - Answers - What is the pattern trying to solve?
- **Solution**
 - Describes elements, relationships, responsibilities, and collaborations which make up the design
- **Consequences**
 - Results of applying the pattern
 - Benefits and Costs
 - Subjective depending on concrete scenarios

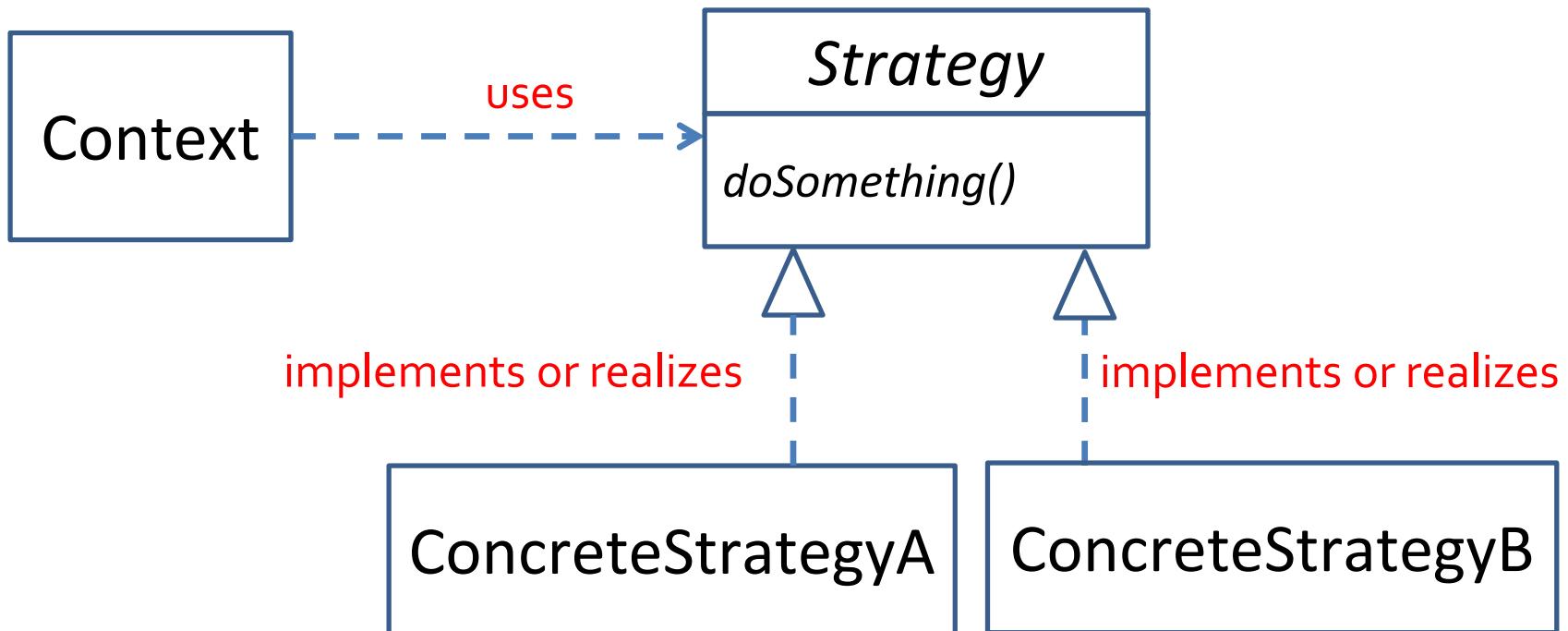
GoF Patterns (23)

- *Creational Patterns*
 - **Abstract Factory**
 - Builder
 - Factory Method
 - Prototype
 - Singleton
- *Structural Patterns*
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - **Façade**
 - Flyweight
 - Proxy
- *Behavioral Patterns*
 - Chain of Responsibility
 - Command
 - Interpreter
 - Iterator
 - Mediator
 - Memento
 - **Observer**
 - State
 - **Strategy**
 - Template Method
 - Visitor



Strategy Pattern

- ▶ Design problem: A set of algorithms or objects should be **interchangeable**.
- ▶ Solution: **Strategy Pattern**

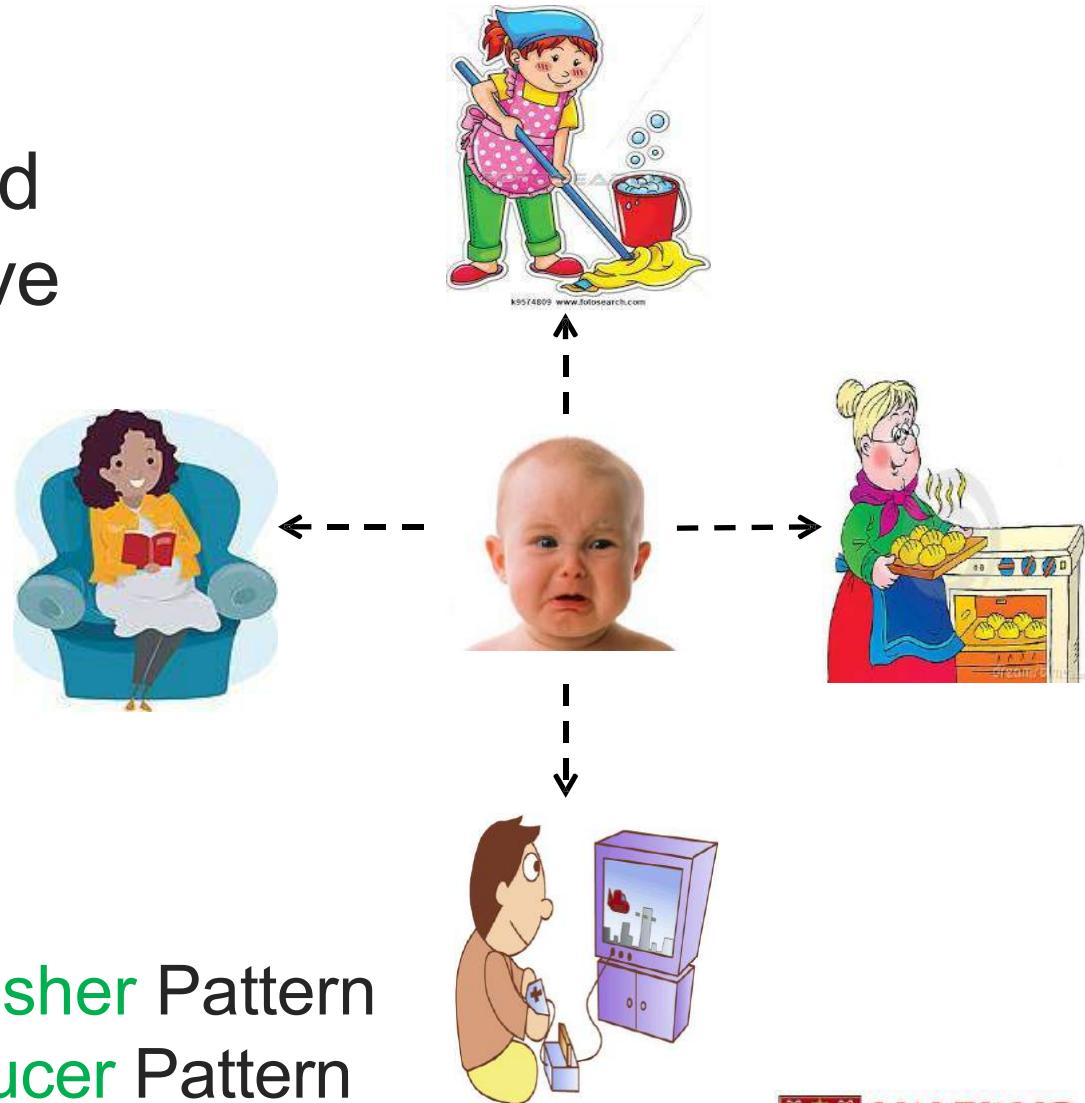


Strategy

- Pros
 - Provides encapsulation
 - Hides implementation
 - Allows behavior change at runtime
- Cons
 - Results in complex, hard to understand code if overused

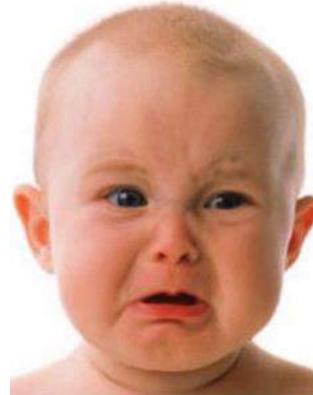
When to Use the Observer Pattern

- ▶ When you need many objects (called **observers**) to receive an **update** when another object (called **subject**) **changes**



a.k.a. **Subscriber – Publisher** Pattern
Consumer – Producer Pattern

Why to use the Observer Pattern?



Subject

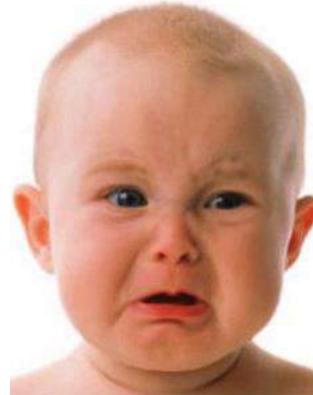
- I do not know (or care) who are looking after me and how many are looking after me
- I want to let whoever are looking after me know when I cry
- It is up to caregivers to decide what to do with me crying



Observer

- Sometimes I want to look after the baby, sometimes I do not
- I want to know the incident once the baby cries, but I do not know when he will cry
- I cannot constantly check the baby's status because I have to do something else important

Why to use the Observer Pattern?



Subject

- I do not know (or care) who are looking after me and how many are looking after me
- I want to let whoever are looking after me know when I cry
- It is up to caregivers to decide what to do with me crying

What kind of coupling (tight or loose) is preferred? Why?



Observer

- Sometimes I want to look after the baby, sometimes I do not
- I want to know the incident once the baby cries, but I do not know when he will cry
- I cannot constantly check the baby's status because I have to do something else important

Why to use the Observer Pattern?



Subject

- *I do not know (or care) who are looking after me and how many are looking after me*
- I want to let whoever are looking after me know when I cry
- It is up to caregivers to decide what to do with me crying

Loose coupling is a benefit for both sides!

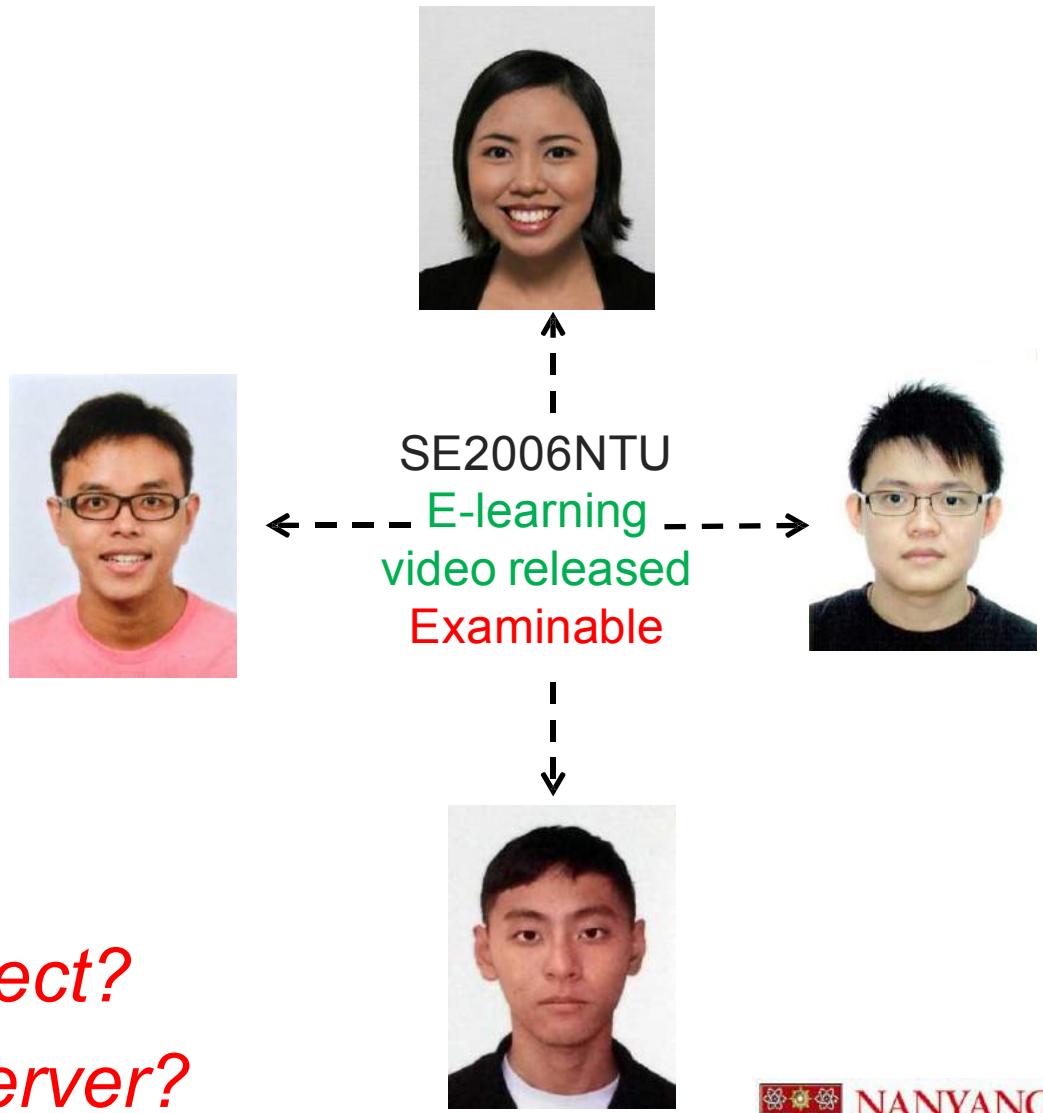


Observer

- Sometimes I want to look after the baby, sometimes I do not
- I want to know the incident once the baby cries, but I do not know when he will cry
- *I cannot constantly check* the baby's status because I have to do something else important

When to Use the Observer Pattern

- ▶ When you need many objects (called **observers**) to receive an **update** when another object (called **subject**) **changes**



- ▶ *Who's the Subject?*
- ▶ *Who's the Observer?*

Why to use the Observer Pattern?



School of Computer Engineering

Subject

- It does not matter who register the course and how many register the course
- I want to let whoever register the course know the latest announcement
- It is up to course takers to decide what to do with the change



Observer

- I want to be able to register/dropoff the course
- I want to know the latest announcement once it occurs, but I do not know when it will occur
- I cannot constantly check the announcement because I have to do something else important

Why to use the Observer Pattern?



School of Computer Engineering

Subject

- It does not matter who register the course and how many register the course
- I want to let whoever register the course know the latest announcement
- It is up to course takers to decide what to do with the change

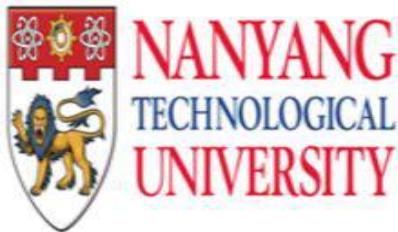
What kind of coupling (tight or loose) is preferred?



Observer

- I want to be able to register/dropoff the course
- I want to know the latest announcement once it occurs, but I do not know when it will occur
- I cannot constantly check the announcement because I have to do something else important

Why to use the Observer Pattern?



School of Computer Engineering

Subject

- *It does not matter who register the course and how many register the course*
- I want to let whoever register the course know the latest announcement
- It is up to course takers to decide what to do with the change

Loose coupling is a benefit for both sides!

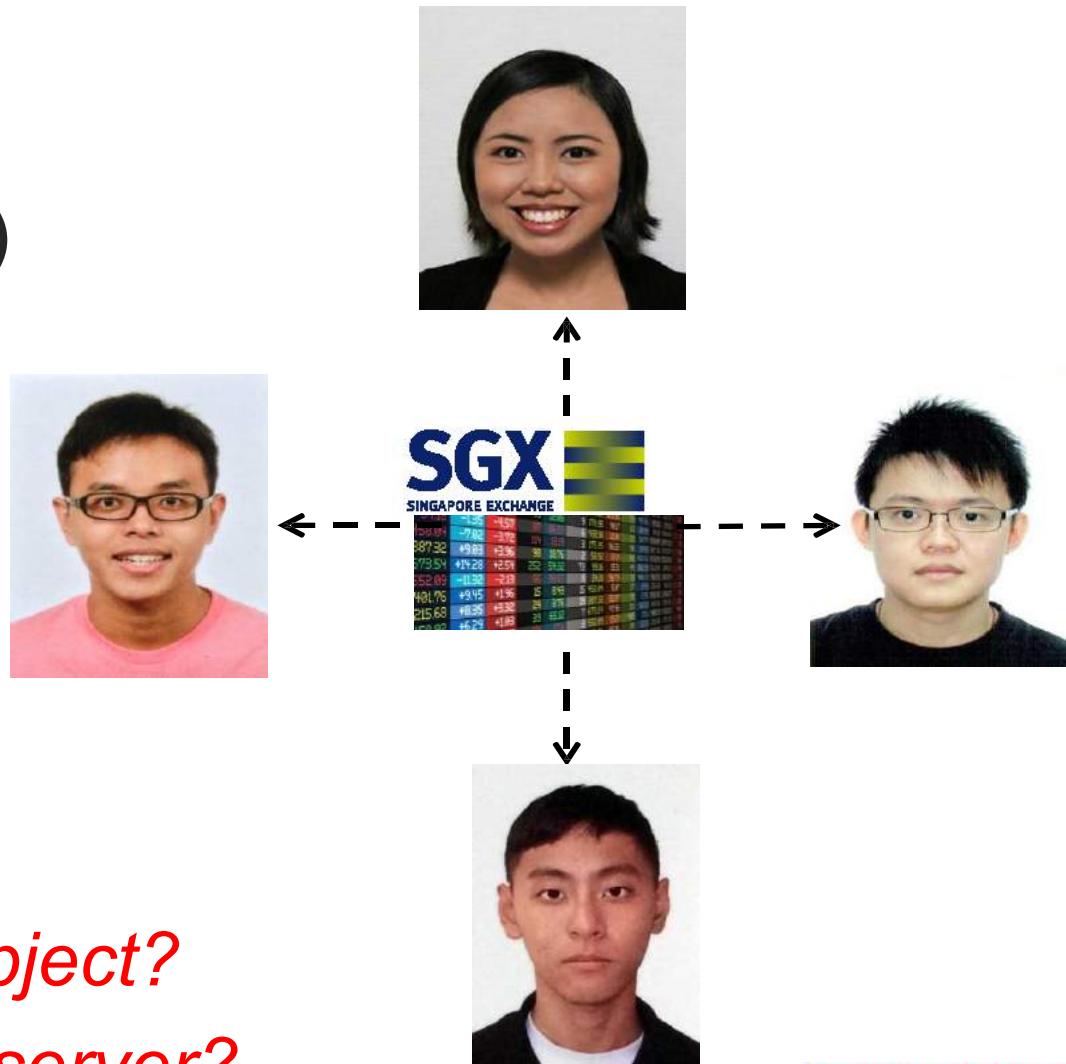


Observer

- I want to be able to register/dropoff the course
- I want to know the latest announcement once it occurs, but I do not know when it will occur
- *I cannot constantly check the announcement* because I have to do something else important

When to Use the Observer Pattern

- ▶ When you need many objects (called **observers**) to receive an **update** when another object (called **subject**) **changes**



- ▶ *Who's the Subject?*
- ▶ *Who's the Observer?*

Why to use the Observer Pattern?

57	13	42.08	9	17
87	86.53	6	49	
72	114	13.19	3	17
96	98	18.76	2	5
54	252	54.32	73	9
13	86	98.65	8	3
96	15	8.43	15	45
32	24	3.76	19	38
03	39	65.12	7	67
54	126	3	55	

Subject

- I do not care who invest in the market and how many invest in the market
- I want to let whoever invest in the market know the latest stock change
- It is up to investors to decide what to do with the change



Observer

- I want to be able to invest in or withdraw from the market freely
- I want to know the latest stock change once it occurs, but I do not know when it will occur
- I cannot constantly check the change because I have to do something else important

Why to use the Observer Pattern?

57	13	42.08	9	17
87	87	86.53	6	43
72	114	13.19	3	17
96	98	18.76	2	5
54	252	54.32	73	9
13	86	98.65	8	3
96	15	8.43	15	45
32	24	3.76	19	38
03	39	65.12	7	67
44	126	3	55	

Subject

- I do not care who invest in the market and how many invest in the market*
- I want to let whoever invest in the market know the latest stock change
- It is up to investors to decide what to do with the change

Loose coupling is a benefit for both sides!



Observer

- I want to be able to invest in or withdraw from the market freely
- I want to know the latest stock change once it occurs, but I do not know when it will occur
- I cannot constantly check the change* because I have to do something else important

Observer Pattern – Design Problems

- ▶ Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

*Easy Subscription and Automatic
Notification*

How to Resolve these Problems?

- ▶ **Subscription** mechanism
 - ▶ Observers freely register/unregister their interests in Subject
- ▶ **Notification** mechanism
 - ▶ Subject propagates the change to Observer when the change occurs

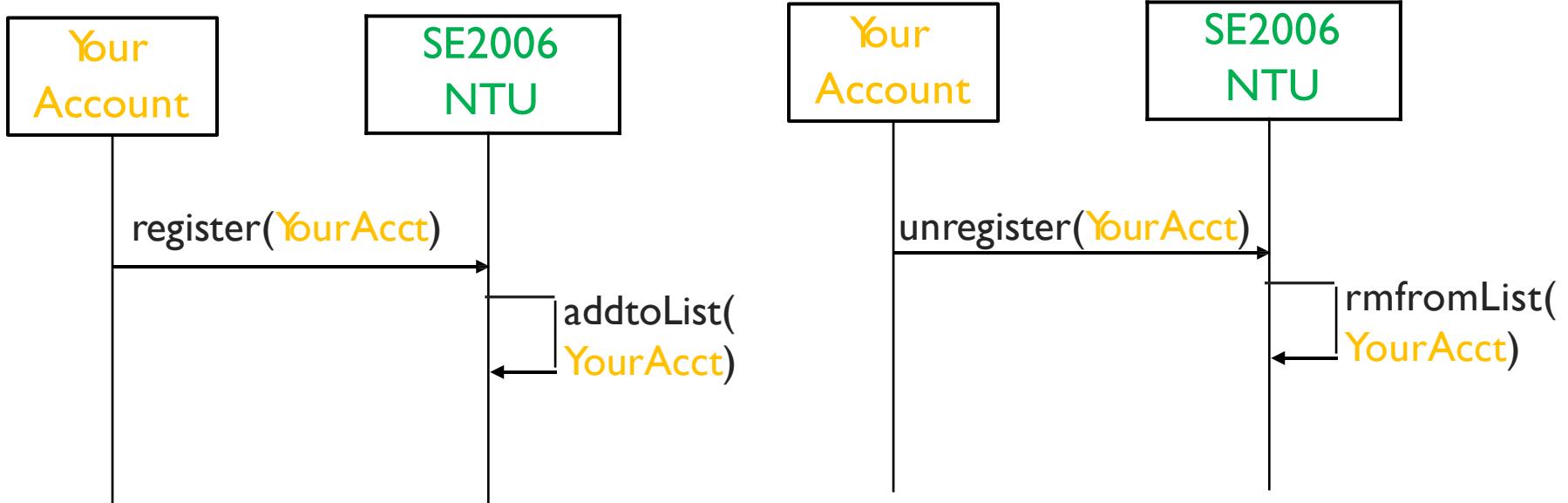
Understand How it Works: NTULearn Case Study

- ▶ In NTULearn system
- ▶ **Subject:** SE2006NTU
- ▶ **Observers:** Course takers (your NTU account)



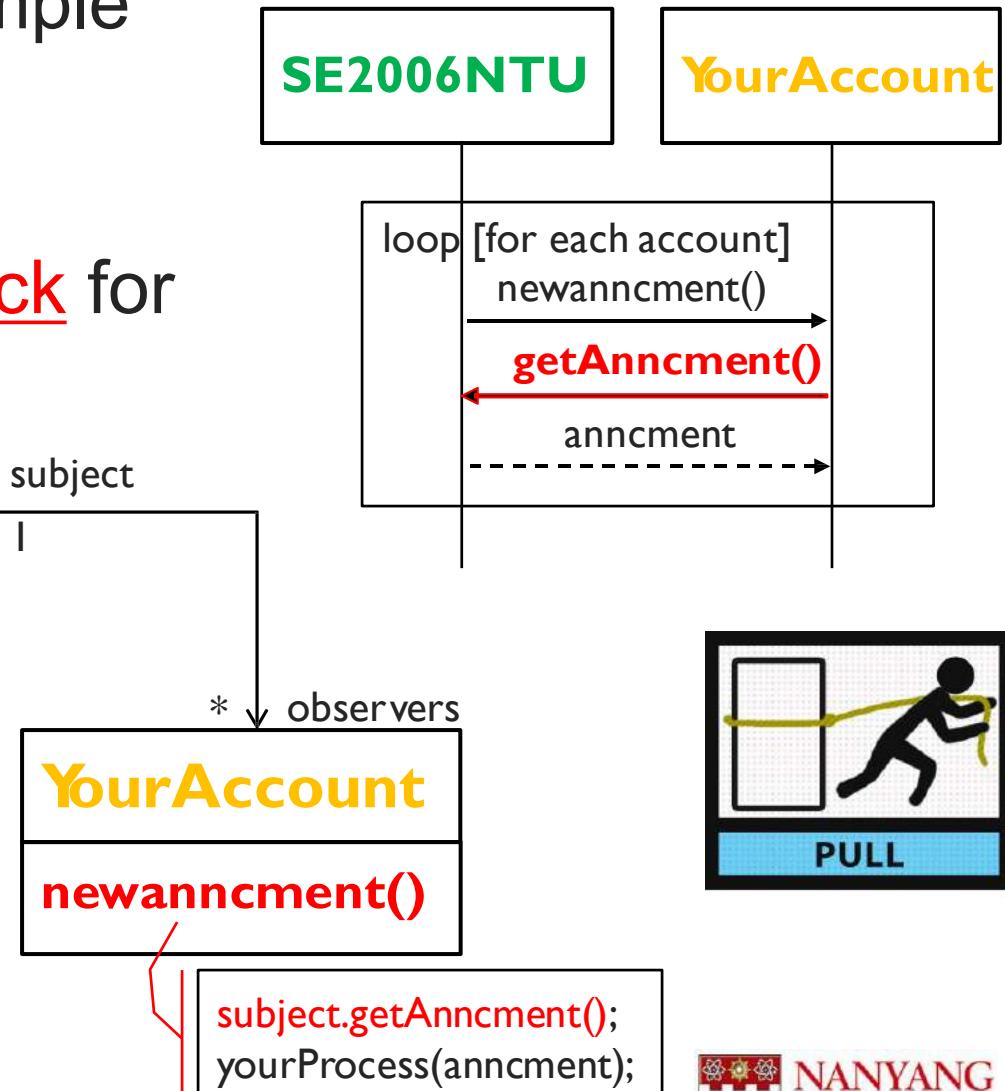
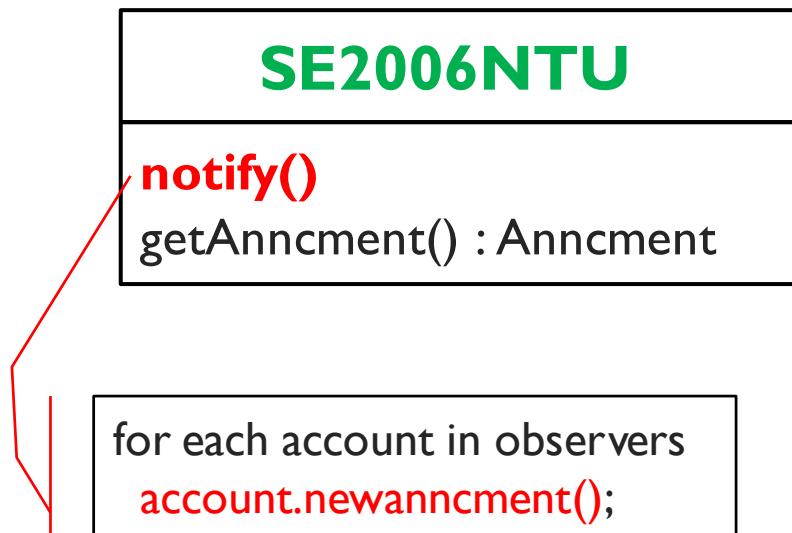
Subscription Mechanism

- ▶ Add to course
(i.e., register)
- ▶ Remove from course
(i.e., unregister)

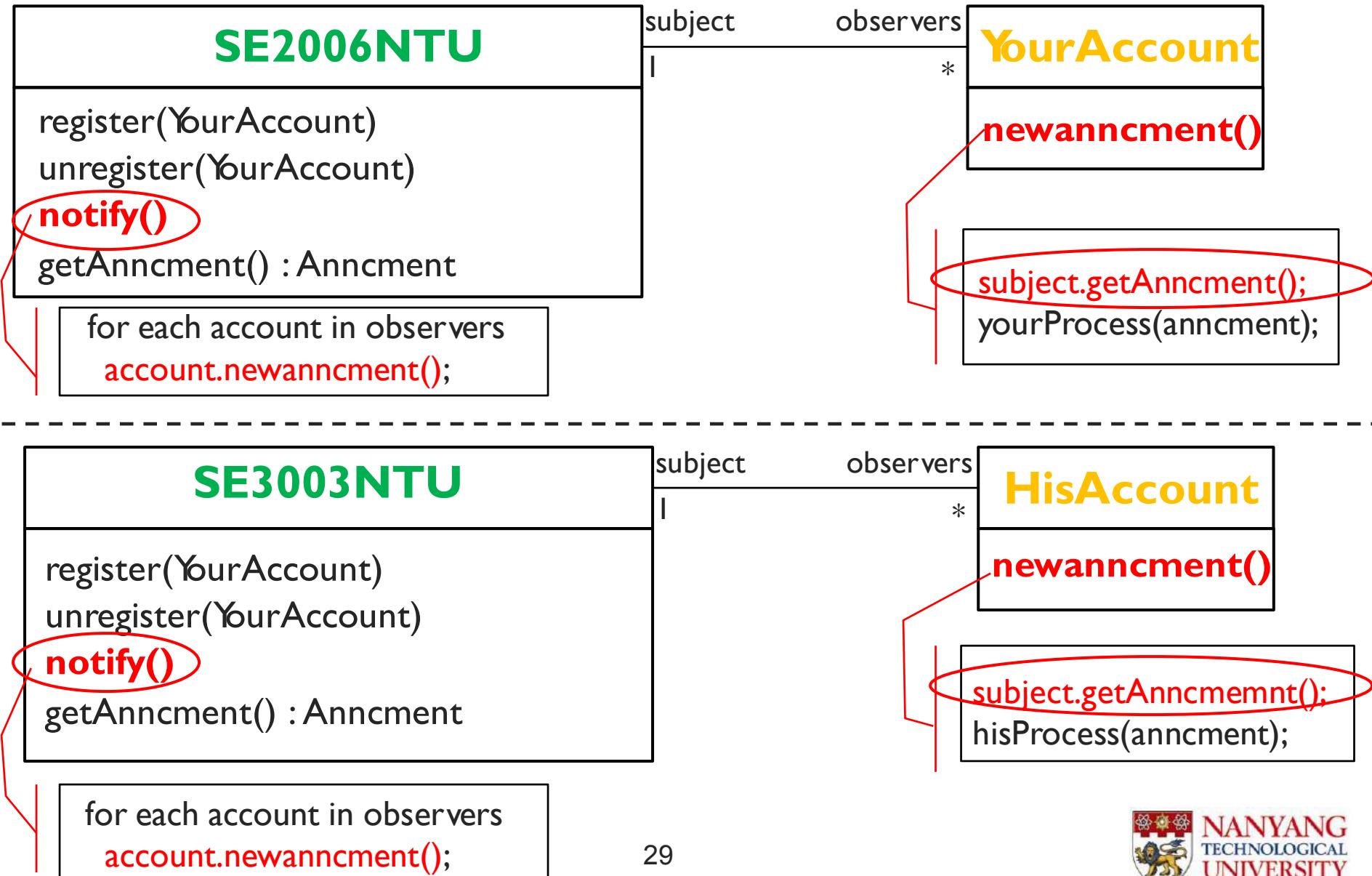


Notification Mechanism – *Pull Update*

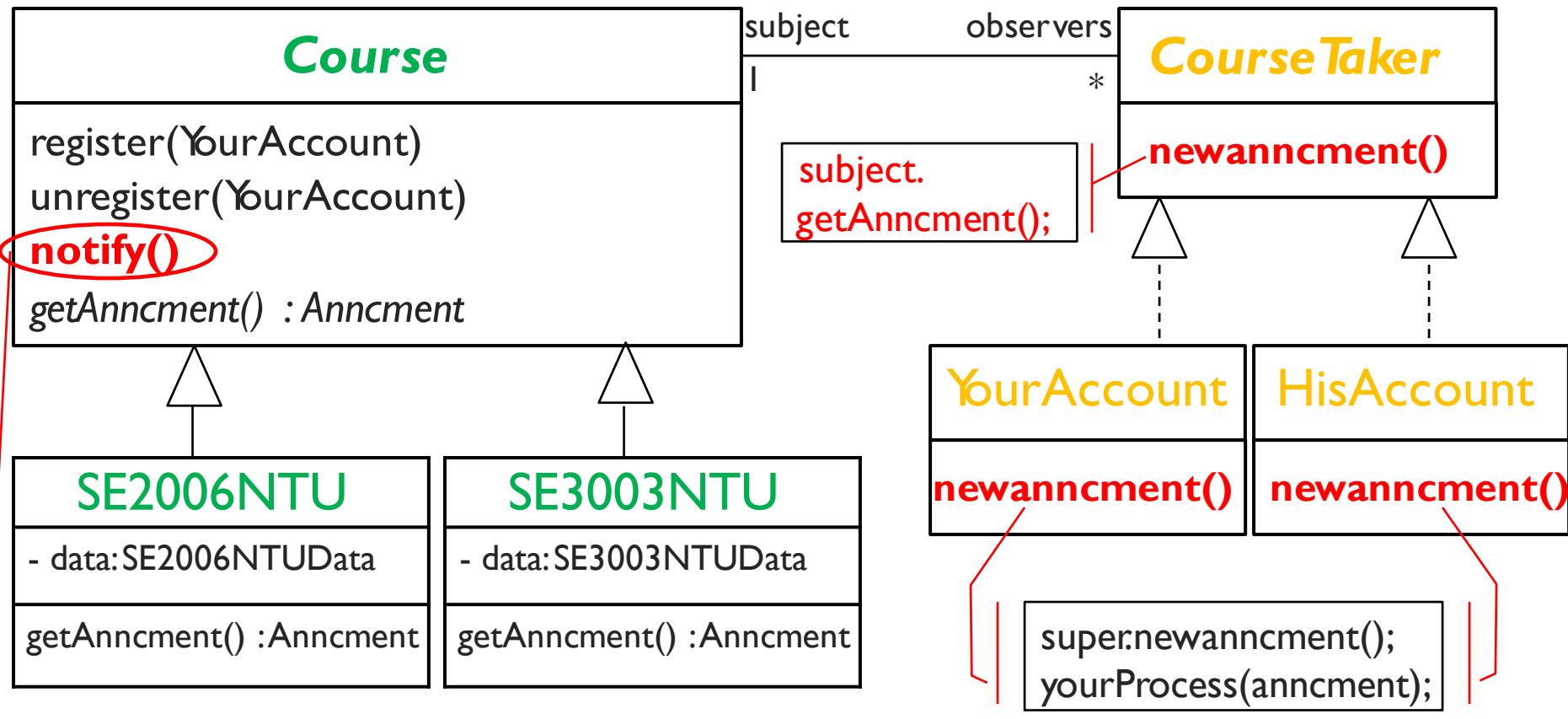
- ▶ **SE2006NTU** sends simple notification (e.g., new announcement)
- ▶ **YourAccount** calls back for details



Pull it Together in a Class Diagram

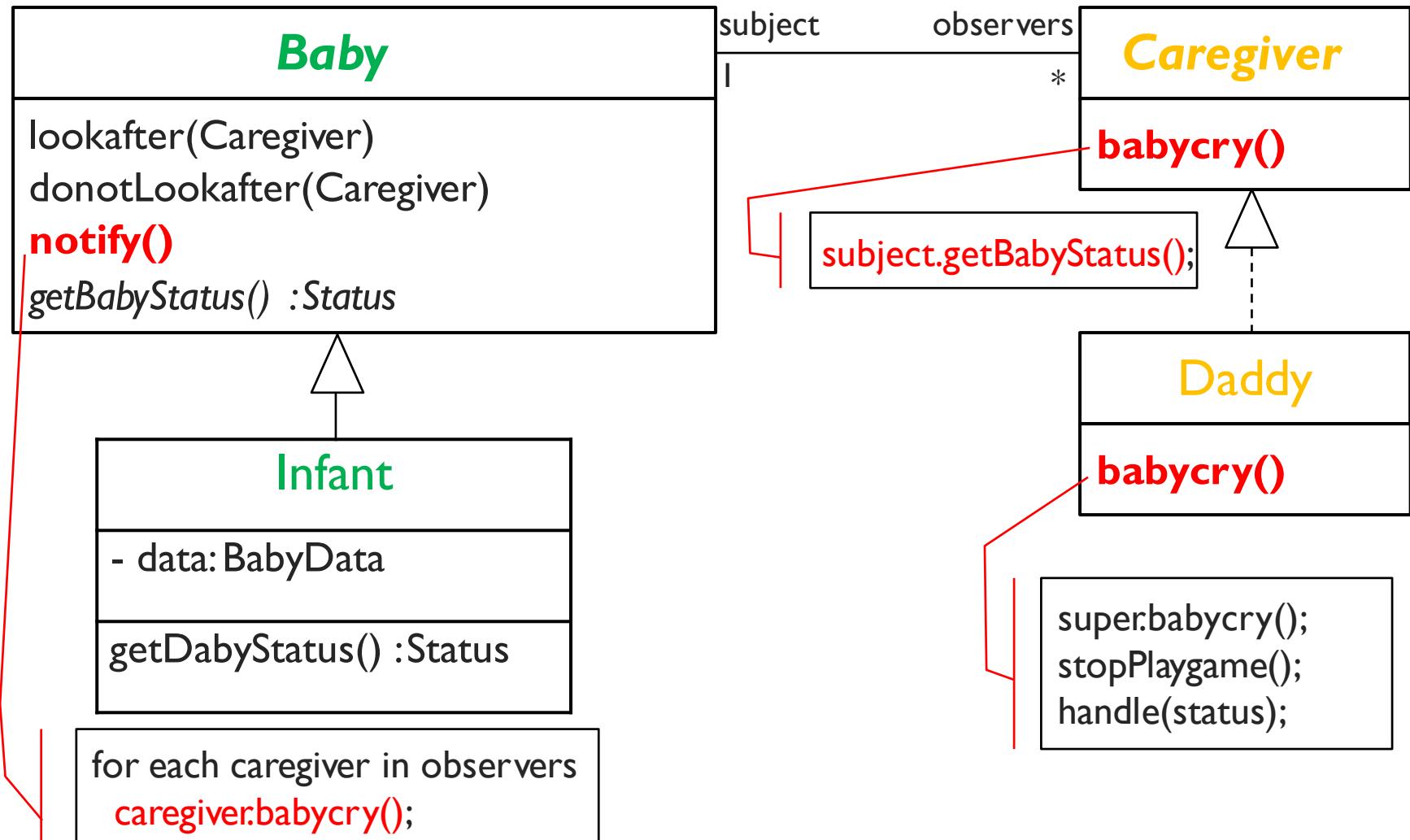


Abstract Commonalities into Superclasses – CourseTaker – Course Design

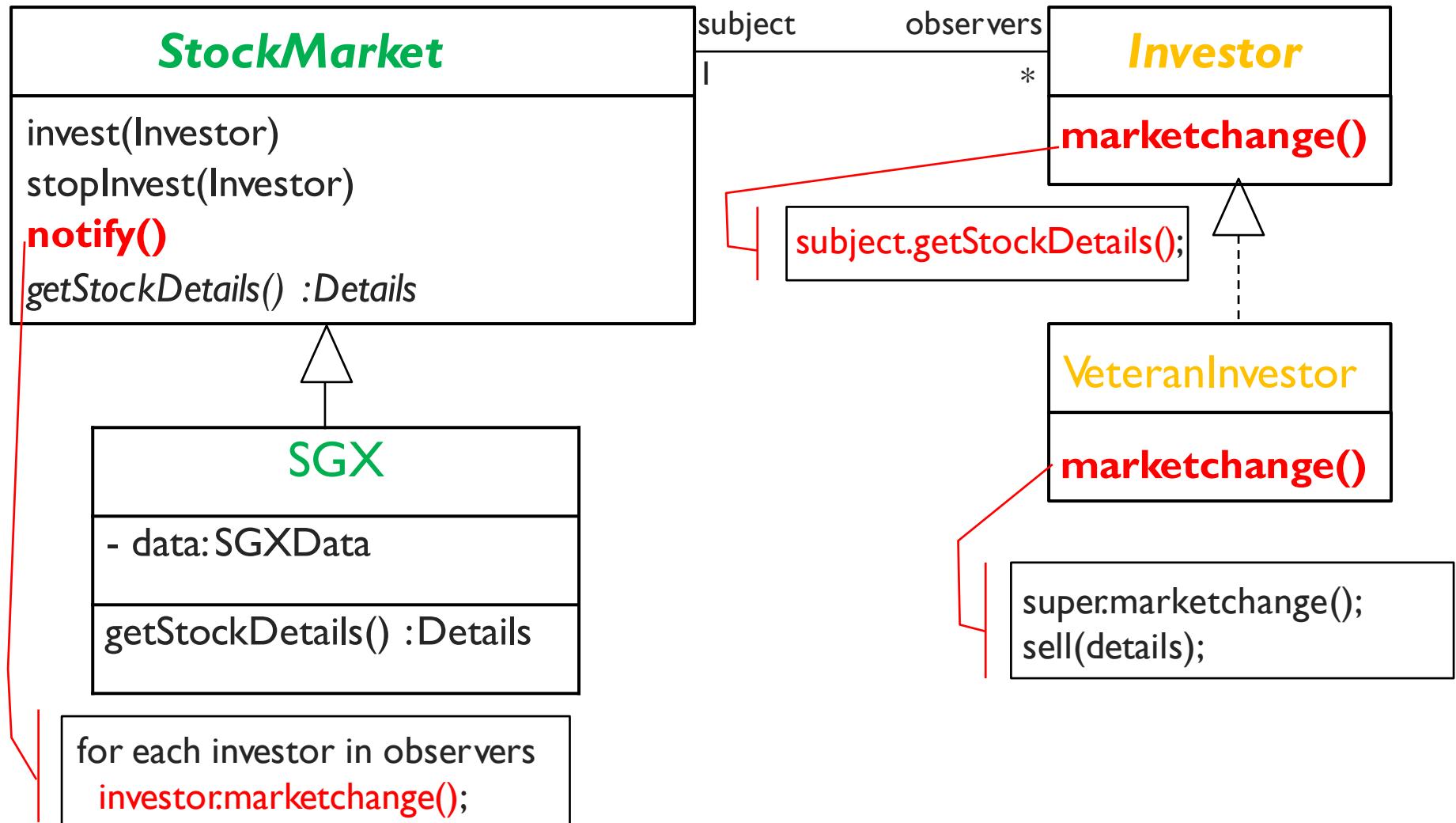


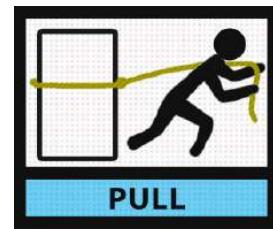
```
for each account in observers  
    account.newannncment();
```

Caregiver – Baby Design

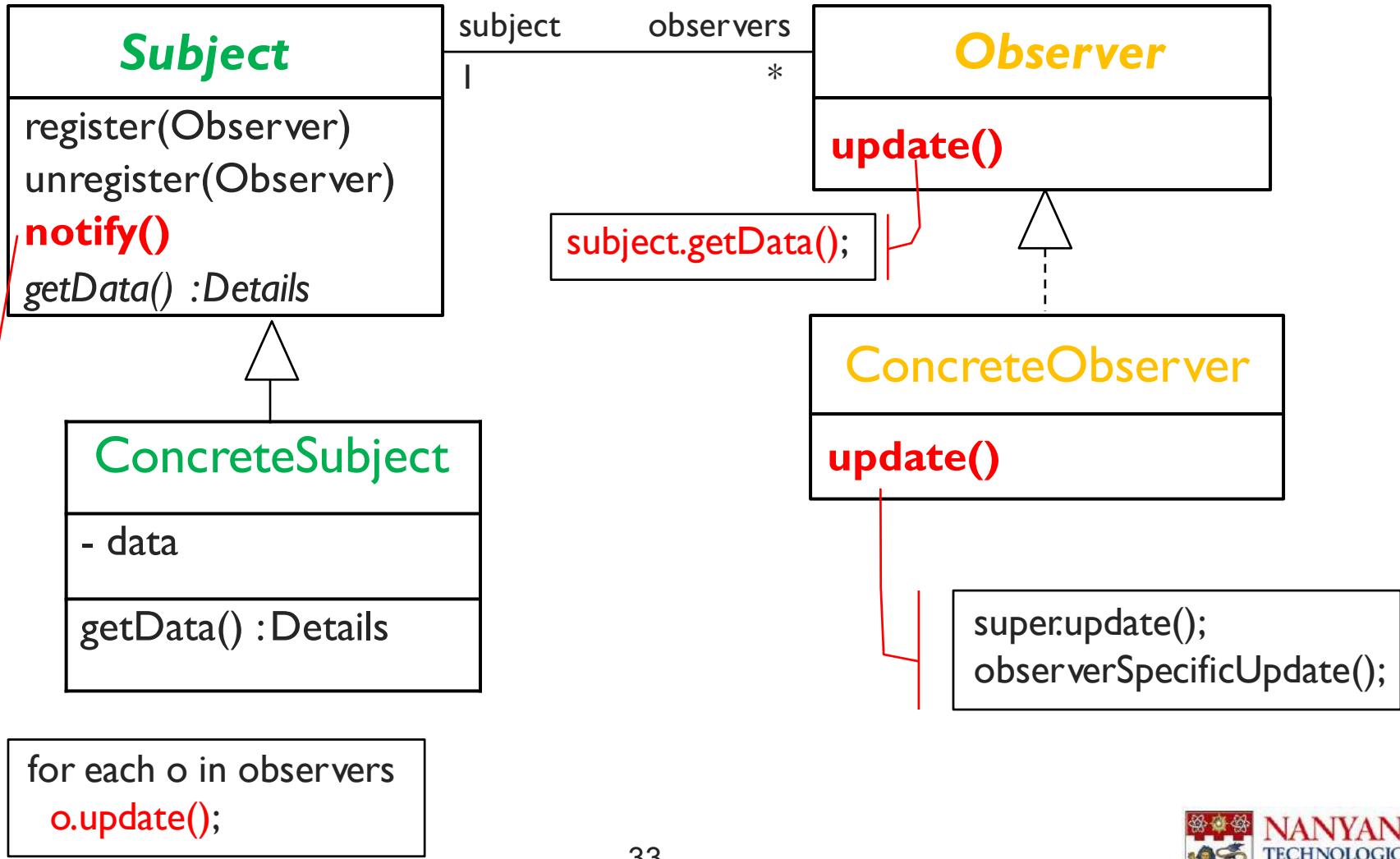


Investor – StockMarket Design



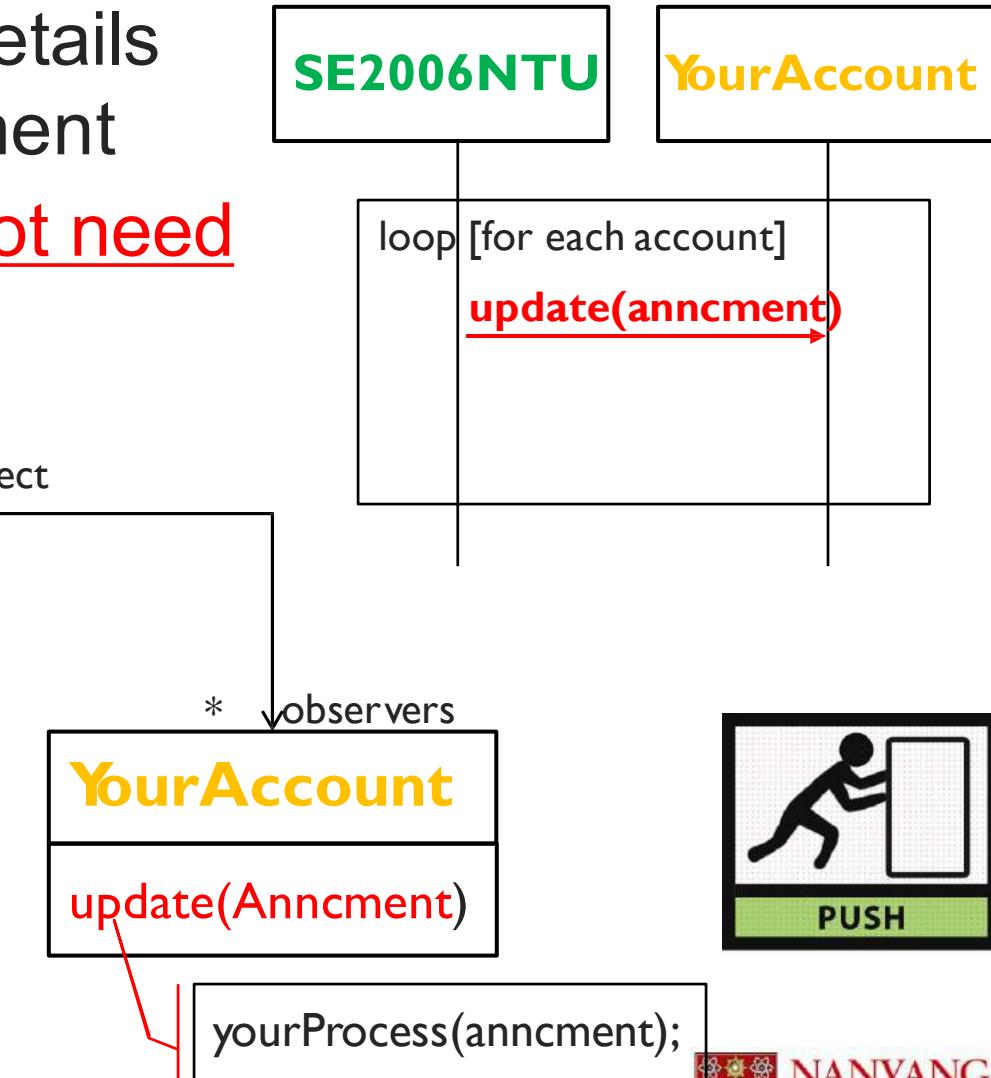
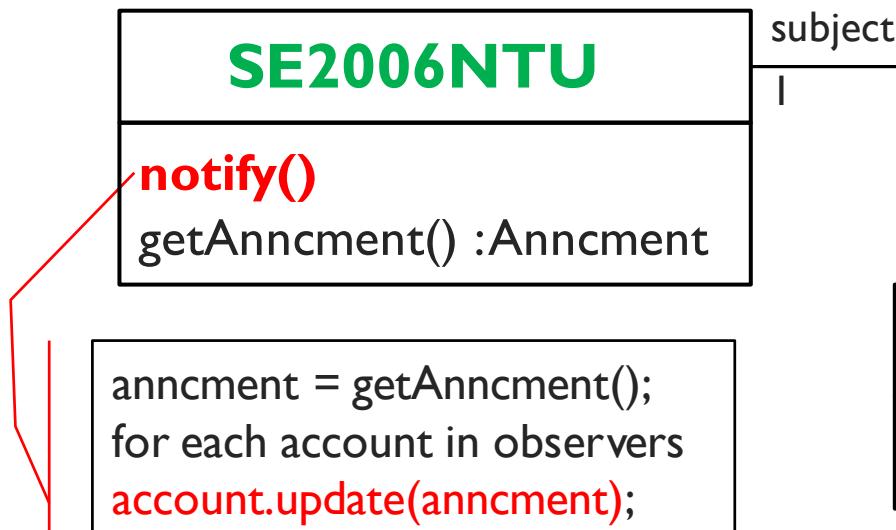


Observer Pattern Template



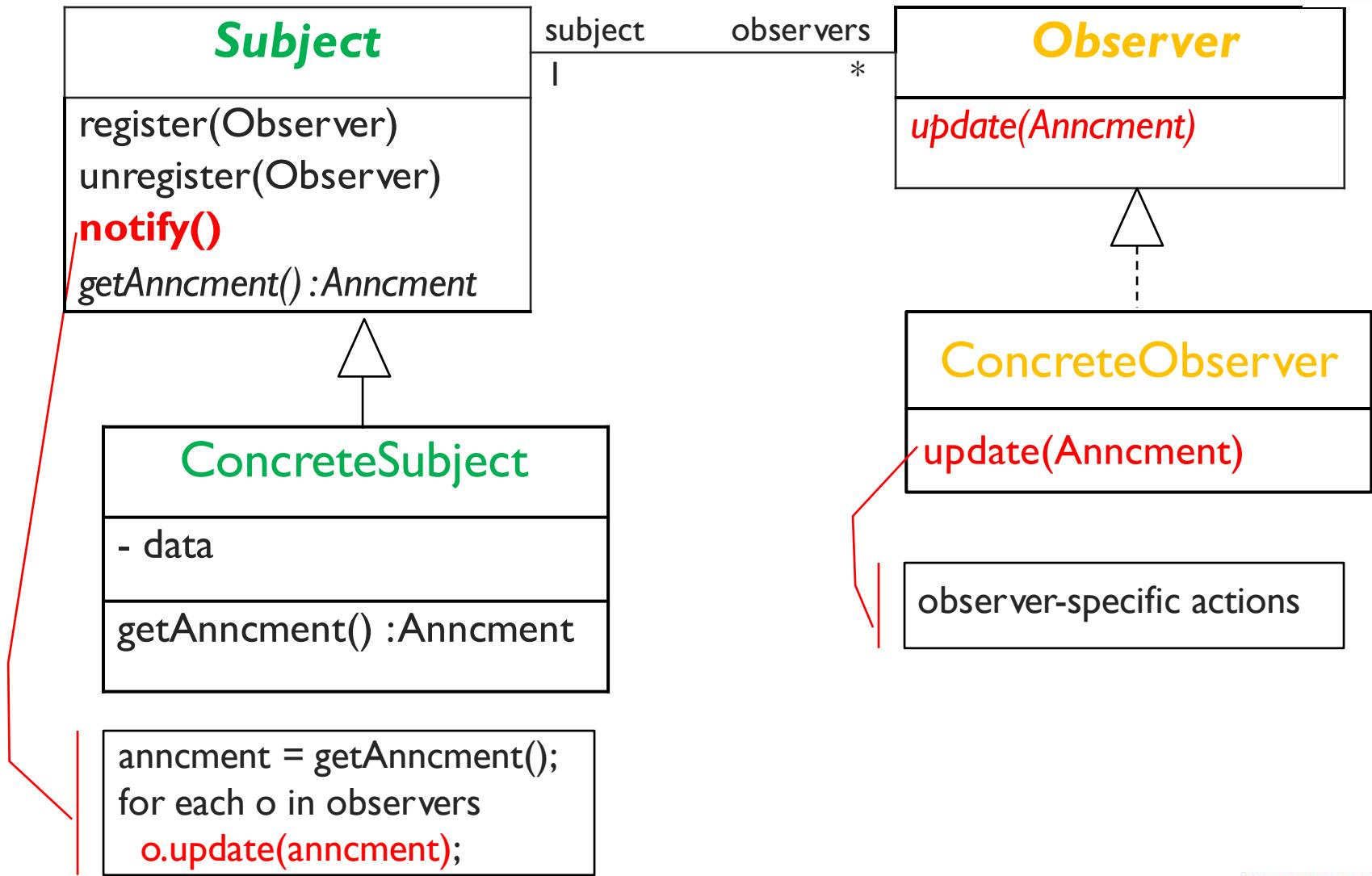
Notification Mechanism – Push Update

- ▶ **SE2006NTU** sends details about the announcement
- ▶ **YourAccount** does not need to call back





Observer Pattern Template

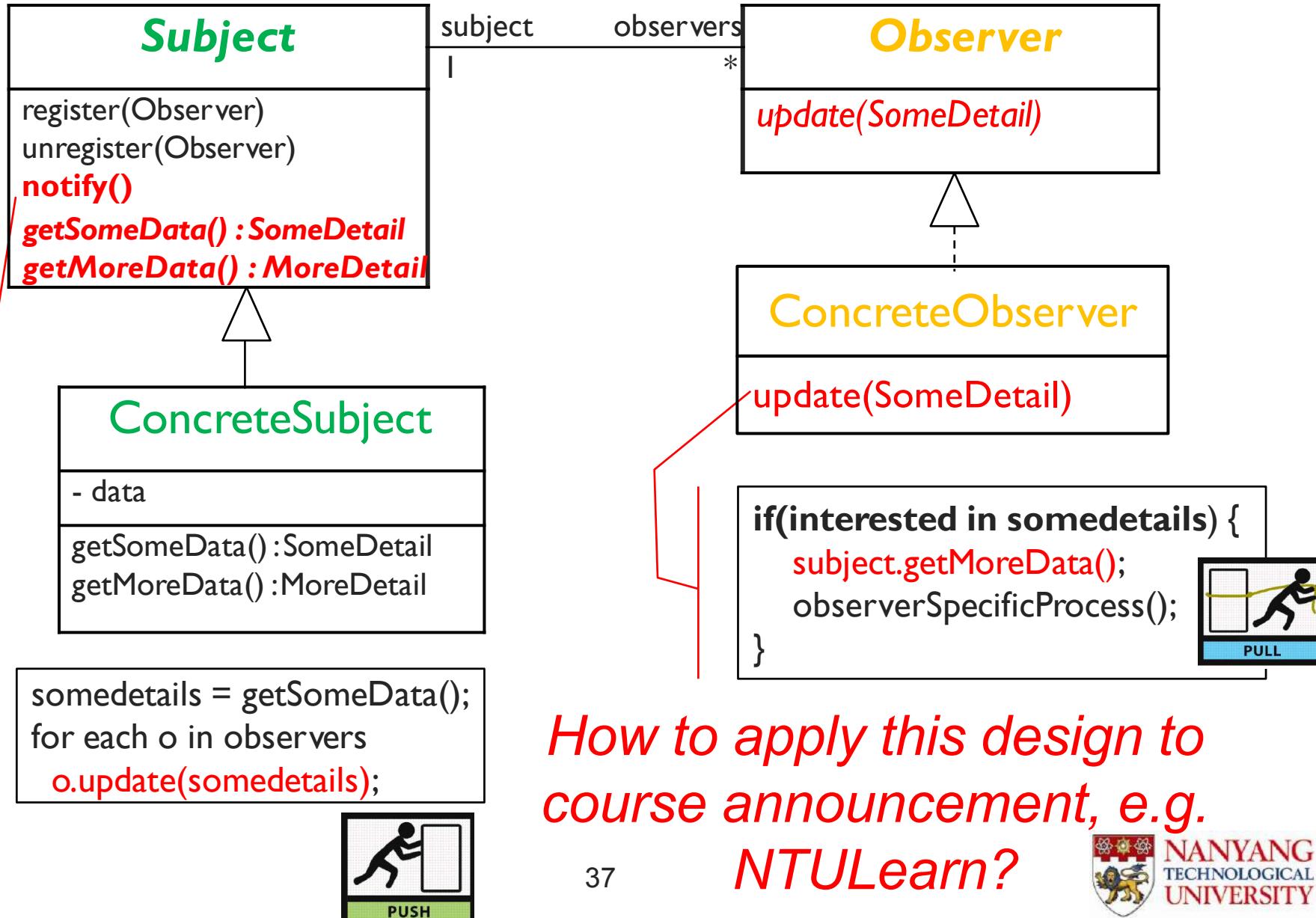


Update Protocol: Pull or Push?

Which one is better?

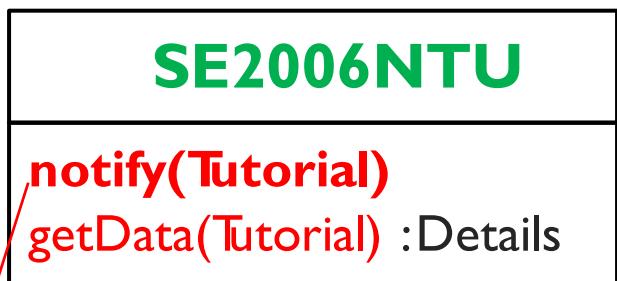
- ▶ **Pull update**
 - ▶ Subject sends simple notification (via `Observer.update()`).
 - ▶ The Observer **calls back** (via `Subject.getData()`) for details explicitly (if Observer is interested) – *two way communication.*
- ▶ **Push update**
 - ▶ Subject sends detailed information about the change (or update), whether Observer wants it (interested) or not – Observer **does not need** to call back – *one way communication.*

Observer Pattern – Push + Pull Update

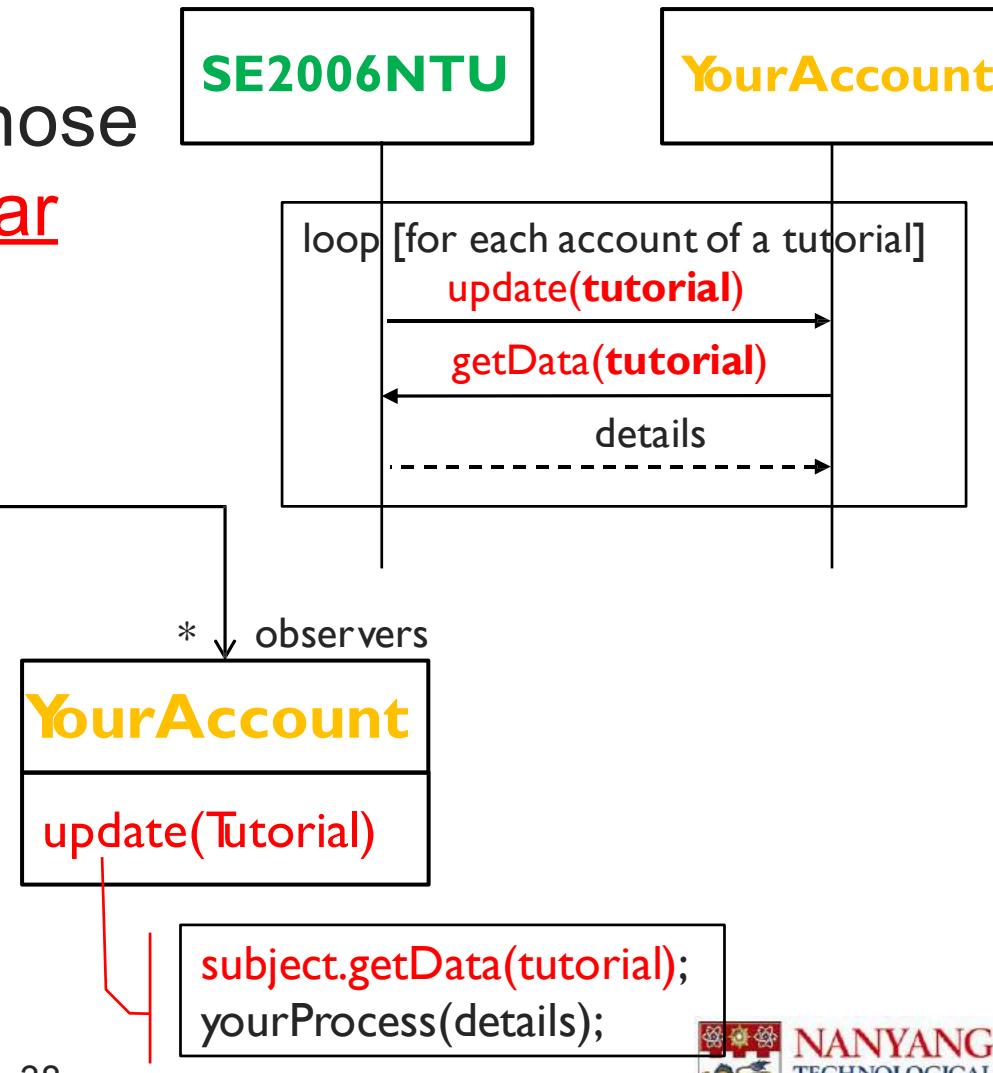


Notification Mechanism – Selective Notification

- ▶ **SE2006NTU** notifies those who register a particular tutorial (i.e. interests)

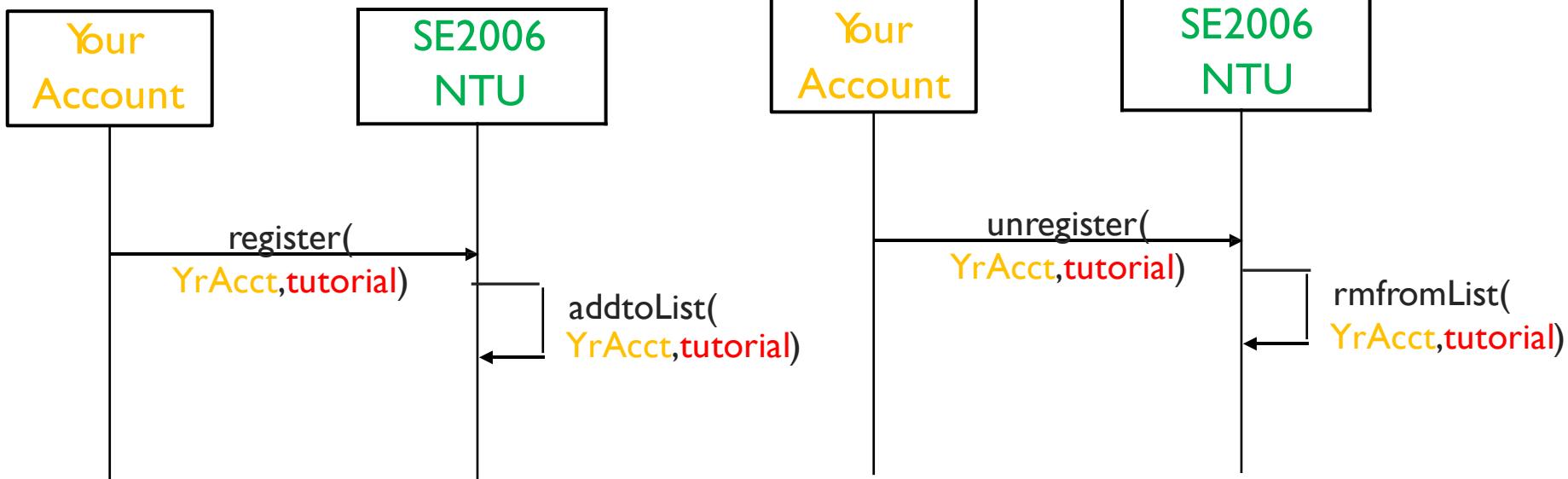


for each account in observers
of a **particular tutorial**
`account.update(tutorial);`



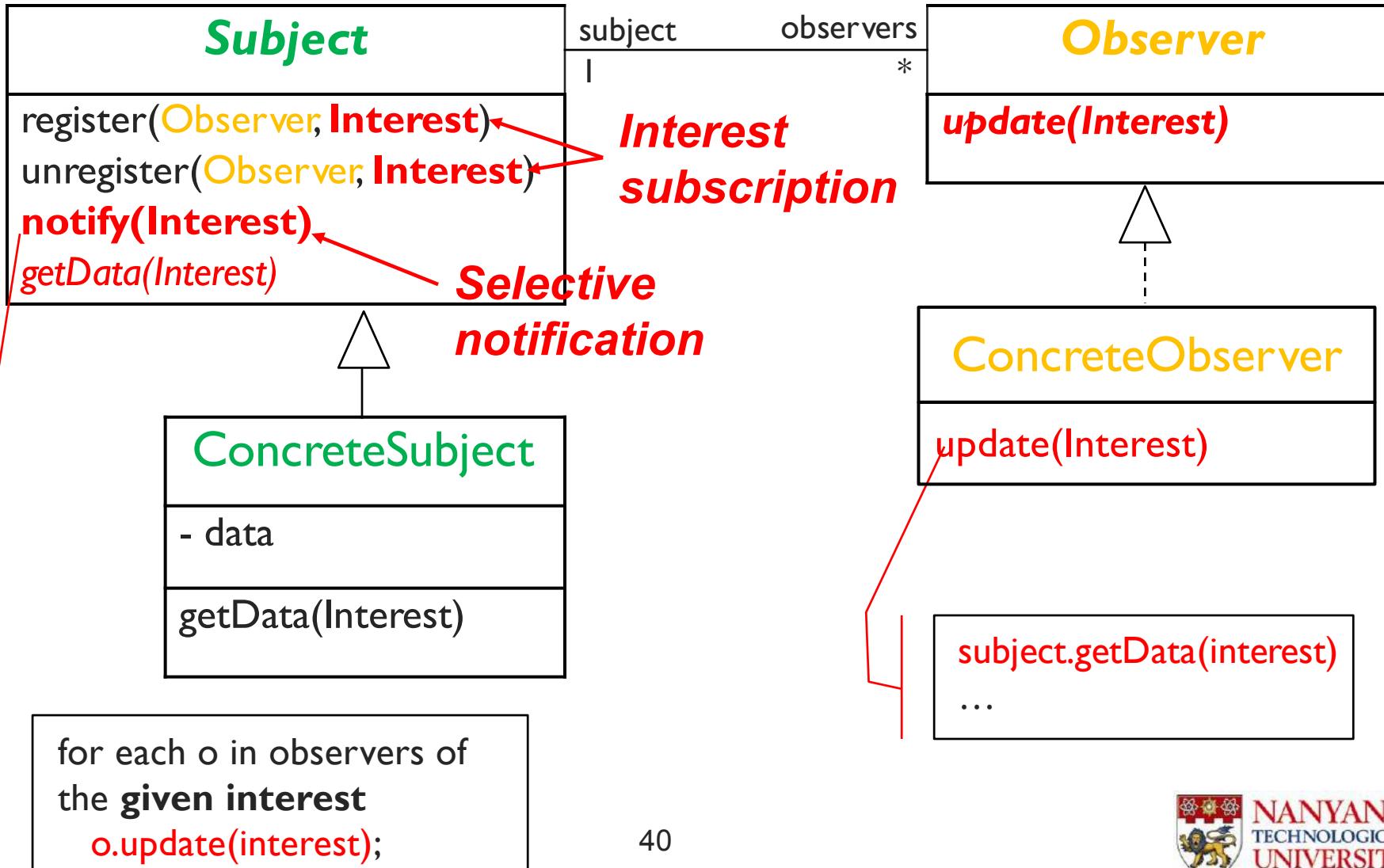
Subscription Mechanism – *Interests Subscription*

- ▶ Add to course
(i.e., register)
- ▶ Remove from course
(i.e., unregister)



Indicate your tutorial session when registering the course

Observer Pattern – *Interest Subscription + Selective Notification*



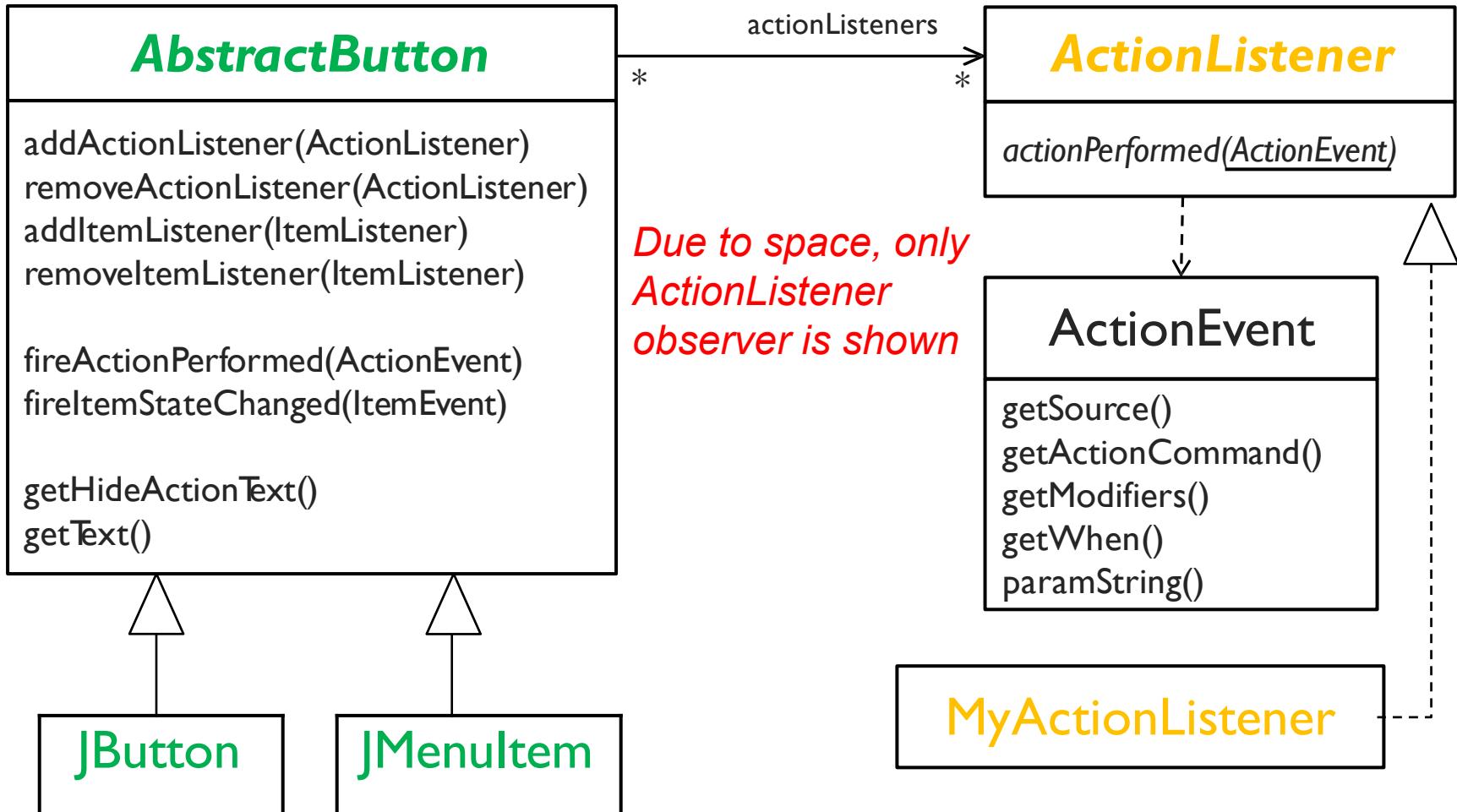
Event Handling – Design Problems

- ▶ **Loose coupling** between an **event source** (**subject**) and its dependent **event handlers** (**observers**)
- ▶ **Event source** wants to notify **event handlers** the event once it occurs, but even handlers can change their interests in event source freely
- ▶ **Event handlers** want to know the event once it occurs, but they do not know when the event will occur in event source

Event Handling & Observer Pattern – Design Problems

- ▶ Loose coupling between an event source (subject) and its dependent event handlers (observers)
 - ▶ Event source wants to notify event handlers the event once it occurs, but even handlers can change their interests in event source freely
 - ▶ Event handlers want to know the event once it occurs, but they do not know when the event will occur in event source
- ▶ Loose coupling between an object (subject) and its dependent objects (observers)
 - ▶ Subject wants to notify observers the change once it occurs, but observers can change their interests freely and constantly
 - ▶ Observers want to know the change once it occurs, but they do not know when the change will occur

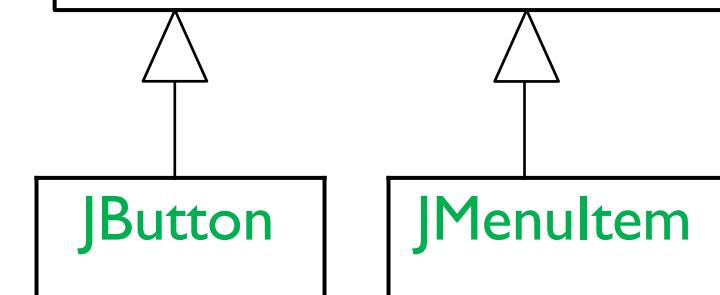
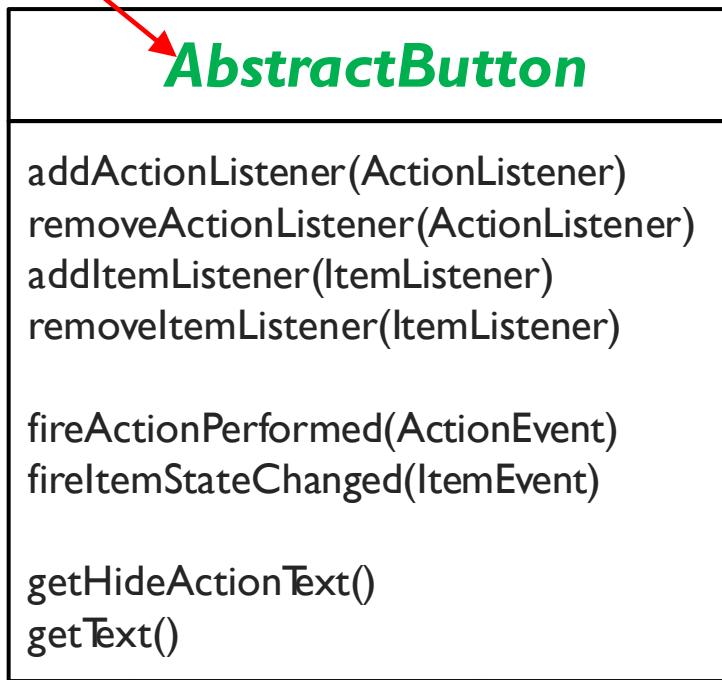
Java Swing Event Handling



What are the **event sources**?
What are the **event handlers**?

Java Swing Event Handling

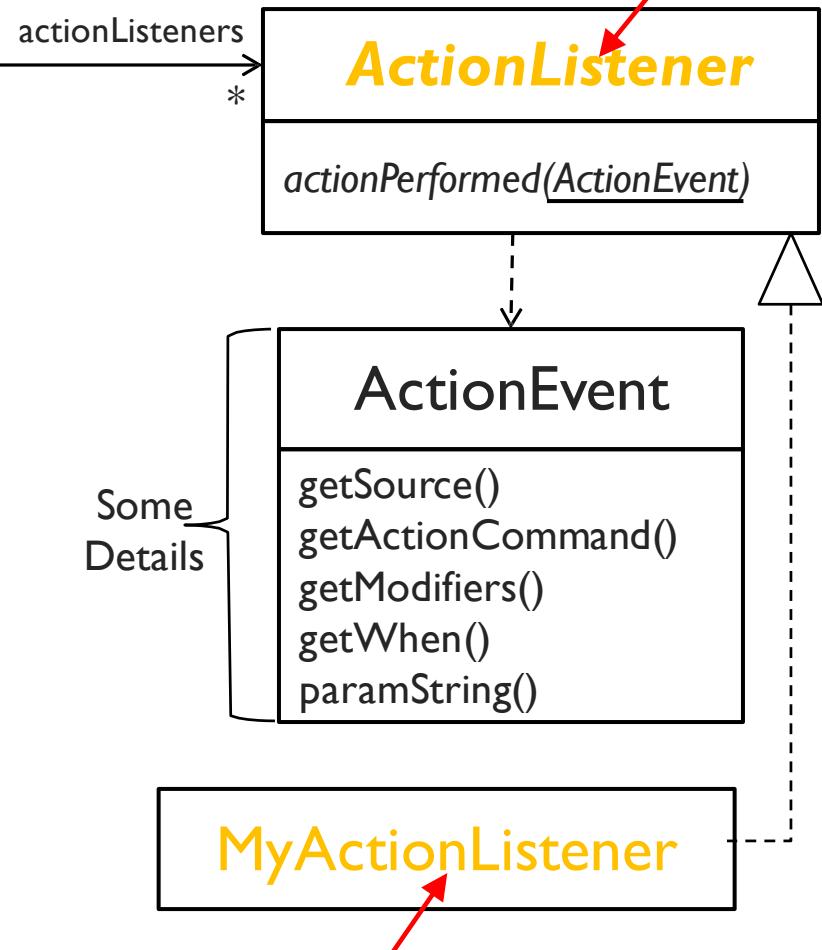
Subject



Concrete subjects

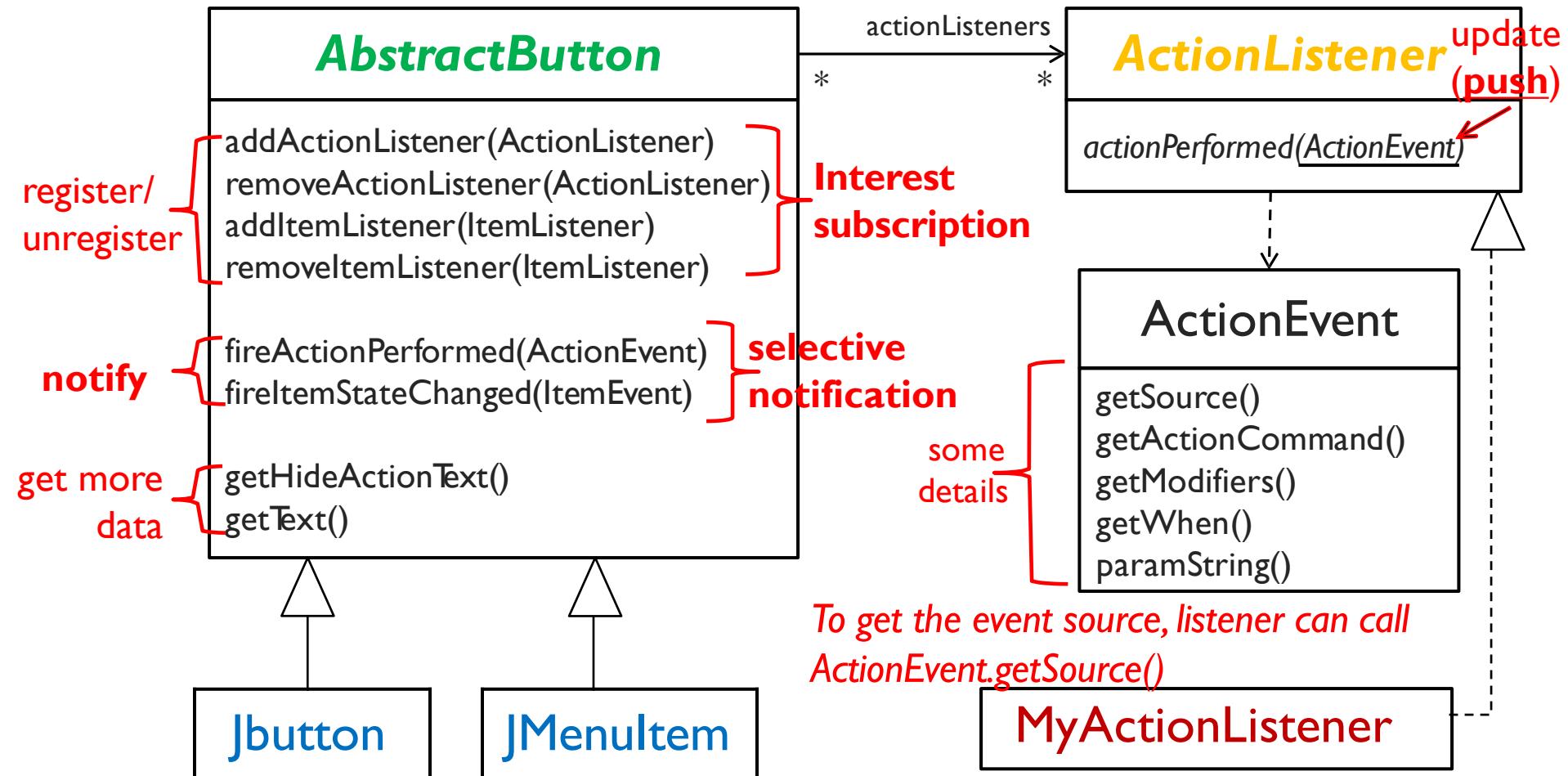
44

Observer



**Concrete
observer**

Java Swing Event Handling



- Register different interests
- Selective notification
- Push + Pull update

To get more information, listener can pull more information, e.g. `ActionEvent.getSource().getText()`

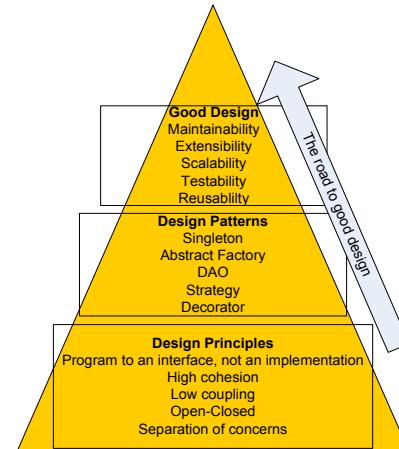
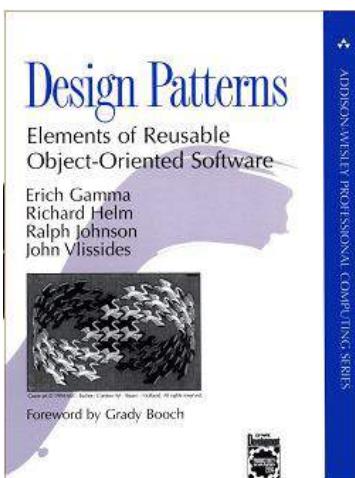
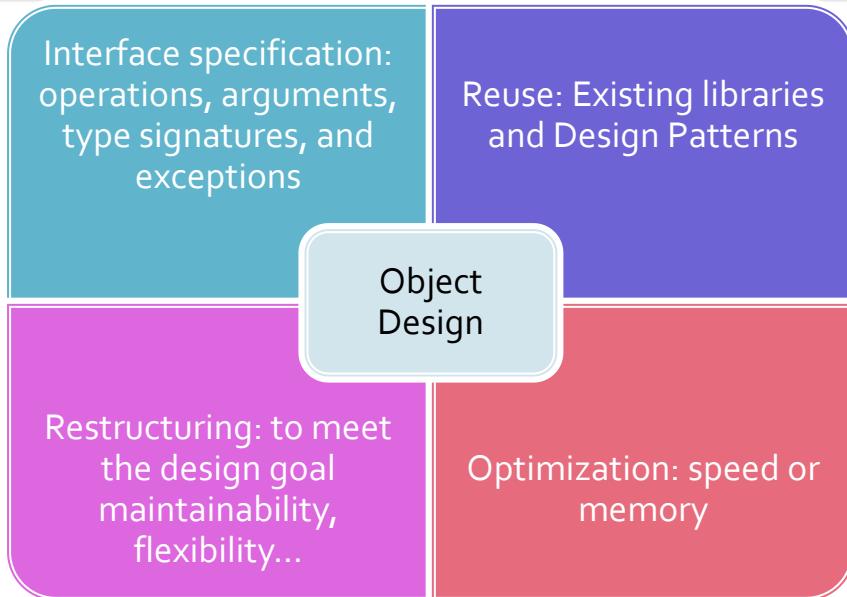
Observer

- Pros
 - Abstracts coupling between Subject and Observer
 - Supports broadcast communication
 - Enables reusability of subjects and observers independently of each other

Loose coupling is a benefit for both sides!

- Cons
 - Slower performance
 - If not used carefully the observer pattern can add unnecessary complexity

Summary of Object Design



4 Elements of a Design Pattern

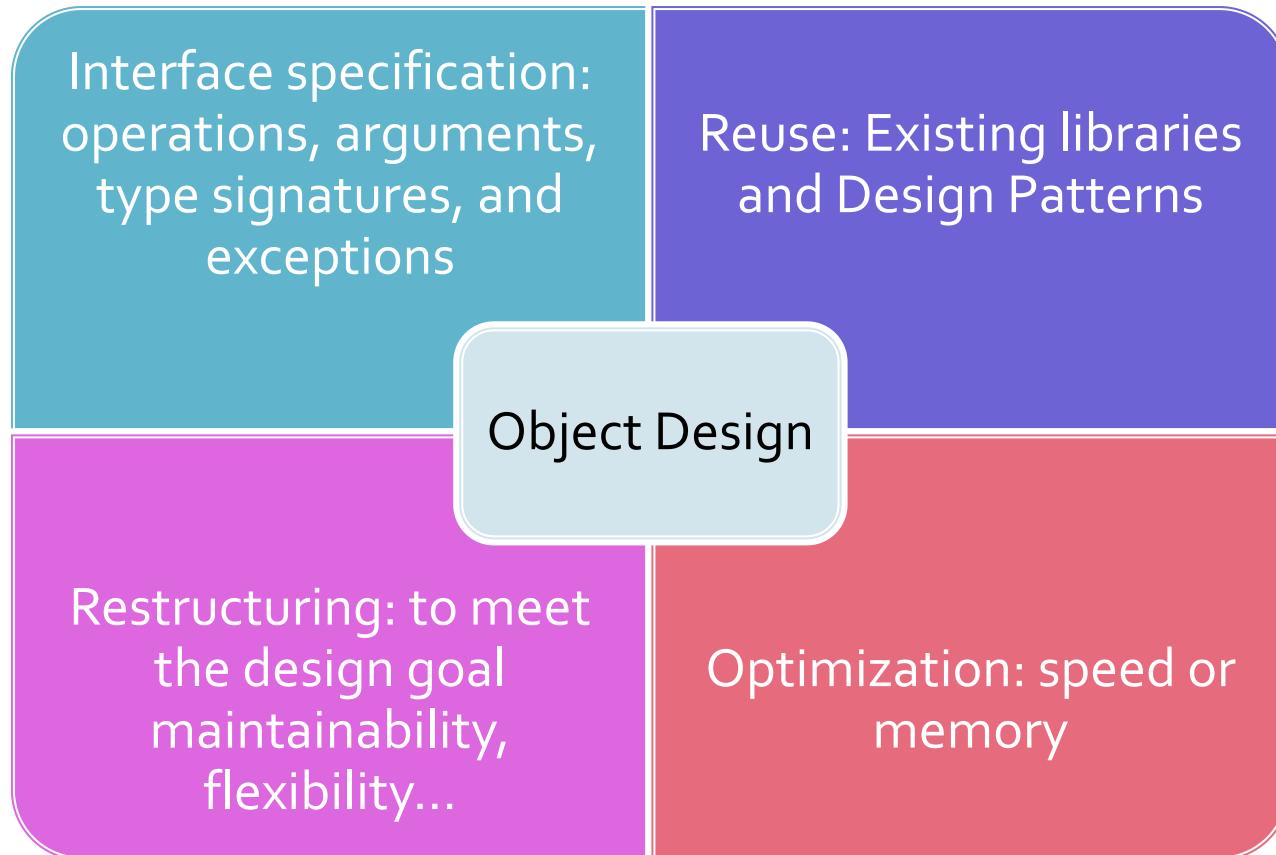
- **Name**
 - Describes the pattern
 - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
- **Problem**
 - Describes when to apply the pattern
 - Answers - What is the pattern trying to solve?
- **Solution**
 - Describes elements, relationships, responsibilities, and collaborations which make up the design
- **Consequences**
 - Results of applying the pattern
 - Benefits and Costs
 - Subjective depending on concrete scenarios

CZ2006/CE2006 Software Engineering

Lecture 14: Design Patterns

Liu Yang

Recap of Object Design



GoF Patterns (23)

– *Creational Patterns*

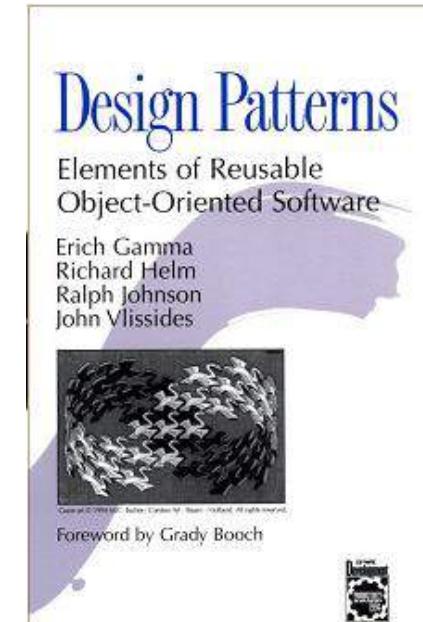
- Abstract Factory
- Builder
- **Factory Method**
- Prototype
- Singleton

– *Structural Patterns*

- Adapter
- Bridge
- Composite
- Decorator
- **Façade**
- Flyweight
- Proxy

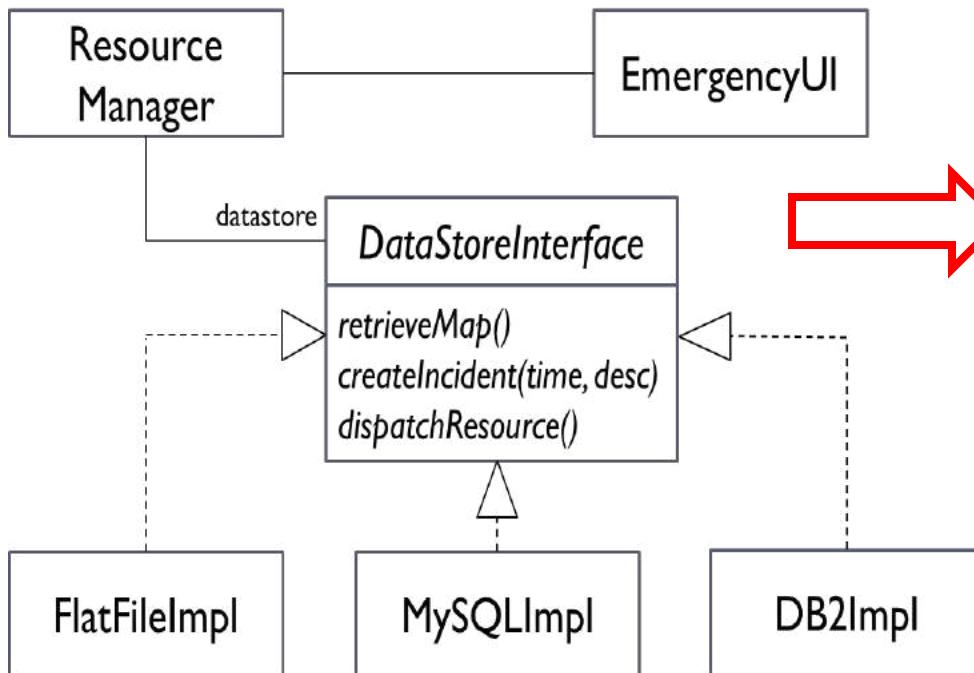
– *Behavioral Patterns*

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- **Observer**
- State
- **Strategy**
- Template Method
- Visitor



What To Do When System Starts Up

- ▶ Create objects, and associate them up according to the class diagram



```
// create data store object user chooses
DataStoreInterface datastore = null;
if(datastoreOption.equals("M")) {
    datastore = new MySQLImpl();
} else if(datastoreOption.equals("D")) {
    datastore = new DB2Impl();
} else if(datastoreOption.equals("F")) {
    datastore = new FlatFileImpl();
} else {
    datastore = new FlatFileImpl();
}
```

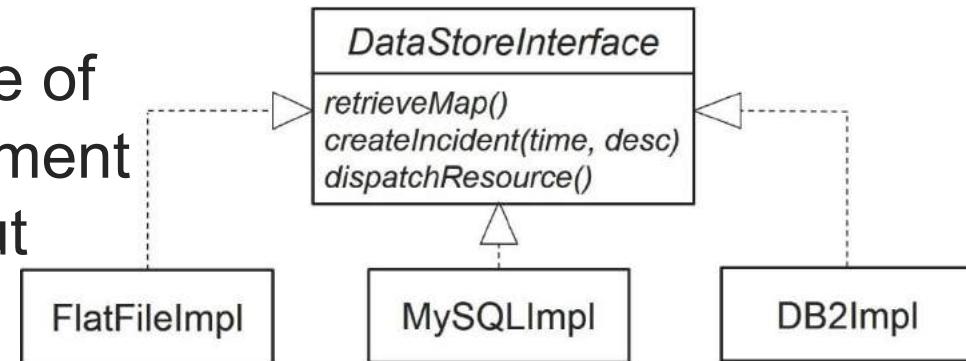
```
// create resource manager object
ResourceManager resourceManager =
    new ResourceManager(datastore);
```

```
// create emergency ui object
EmergencyUI ui = new EmergencyUI();
ui.setResourceManager(resourceManager);
```

See [EMSNoFactory.zip](#) for implementation!

When To Use Factory Pattern?

- ▶ When you've a set of classes but don't know exactly which one you'll need to instantiate until runtime.
- ▶ When you need to create one of several classes that implement a common superclass without exposing the creation logic to the client.
- ▶ When you want to localize the logic to instantiate objects.



Solution: Factory Pattern

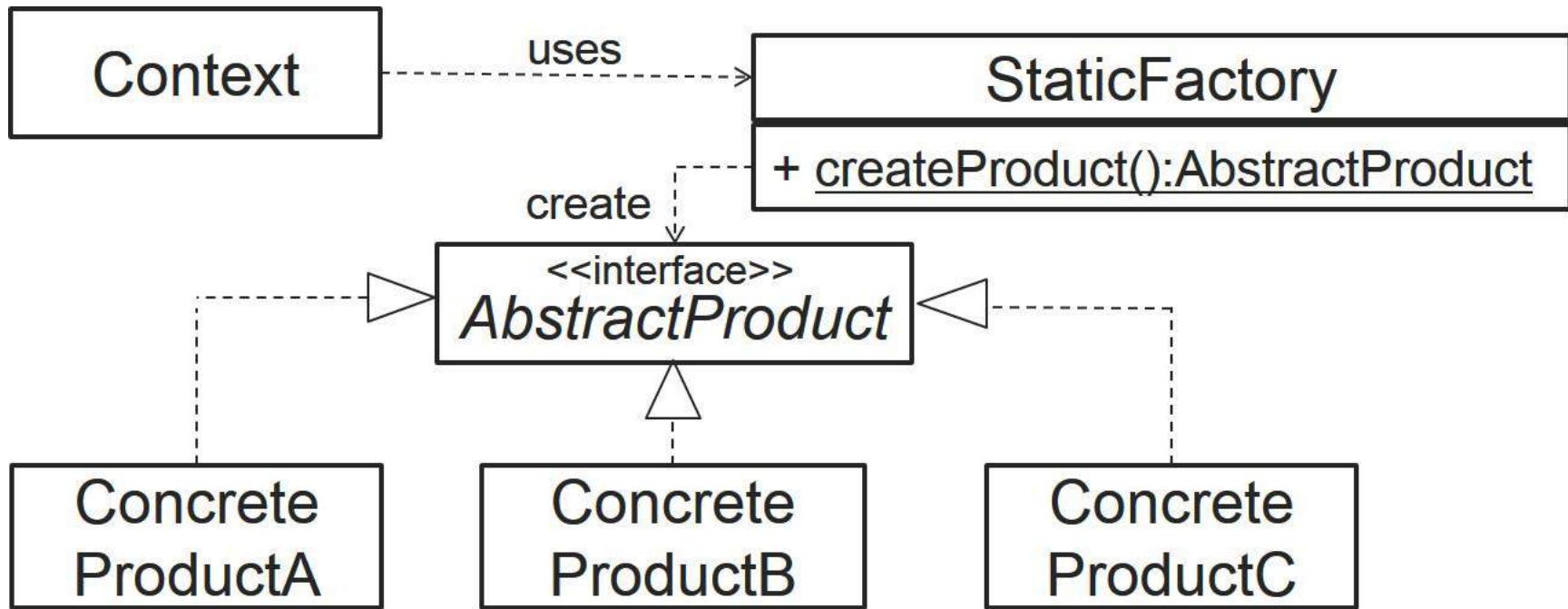
- ▶ Defines an interface for creating different objects, without knowing beforehand what sort of objects it needs to create or how the object is created – lets subclasses decide which class to instantiate.

What Are The Design Problems?

- ▶ **Decouple class selection and object creation** (abstract instantiation process) from the client where the object is used allowing greater flexibility in object creation
- ▶ Existing subclasses can be replaced, or new subclasses can be added
- ▶ Process of selecting subclass to use and creating object can be complex
- ▶ Client does not know ahead of time which class will be used, and the class can be chosen at runtime

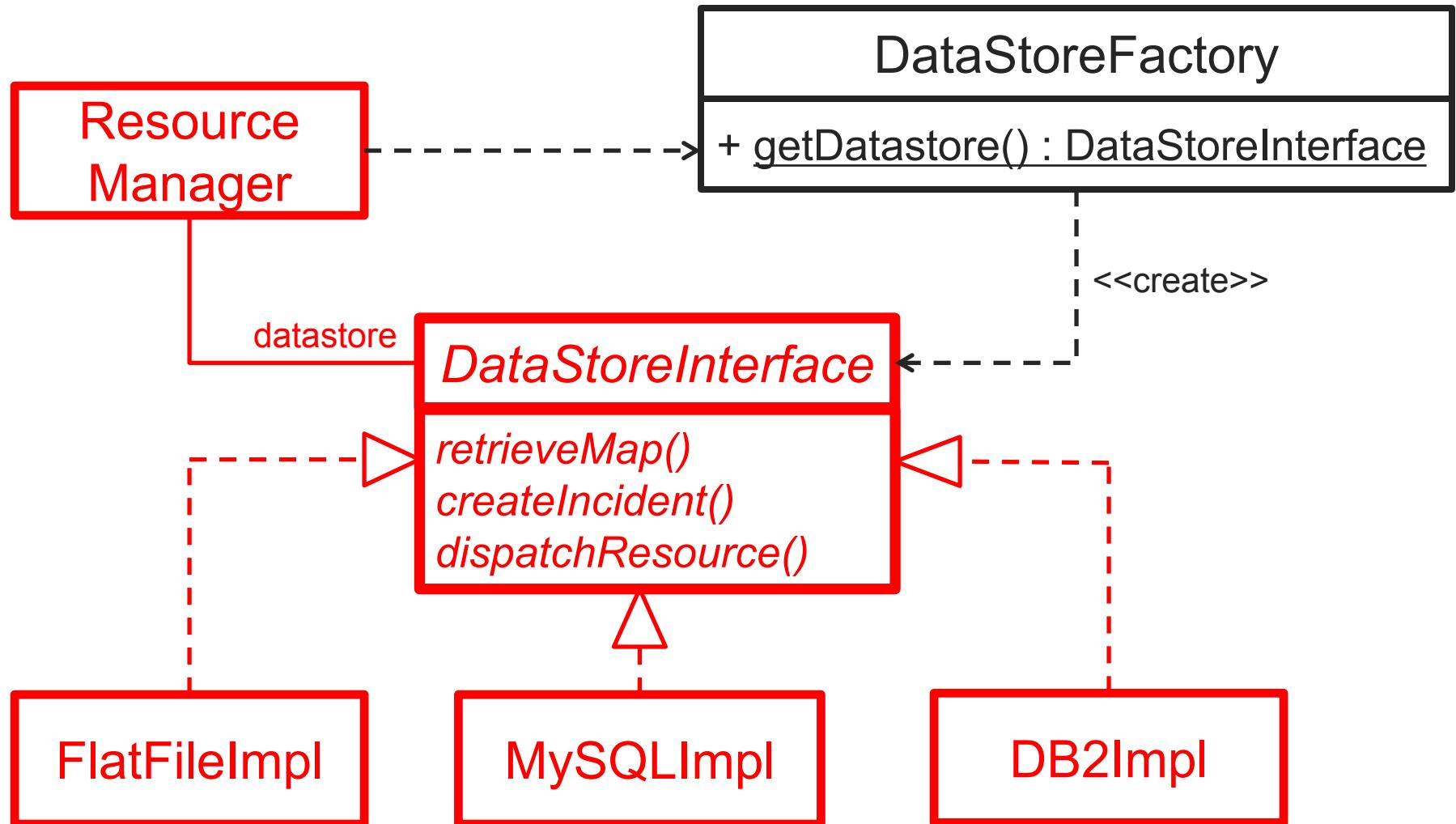
Factory Pattern (a.k.a. Static Factory Pattern)

– “Standard” Version (Template)



- ▶ **Context** uses **StaticFactory** to obtain an object that implements **AbstractProduct** interface.

Data Access: Strategy Design + Factory Design



See [*EMSBasicFactory.zip*](#) for implementation

What Are The Benefits?

▶ Encapsulate object creation

- ▶ Remove duplicate object creation code from clients
(see *EMSNoFactory.zip* and *EMSBasicFactory.zip*)
- ▶ Centralize class selection and object creation code

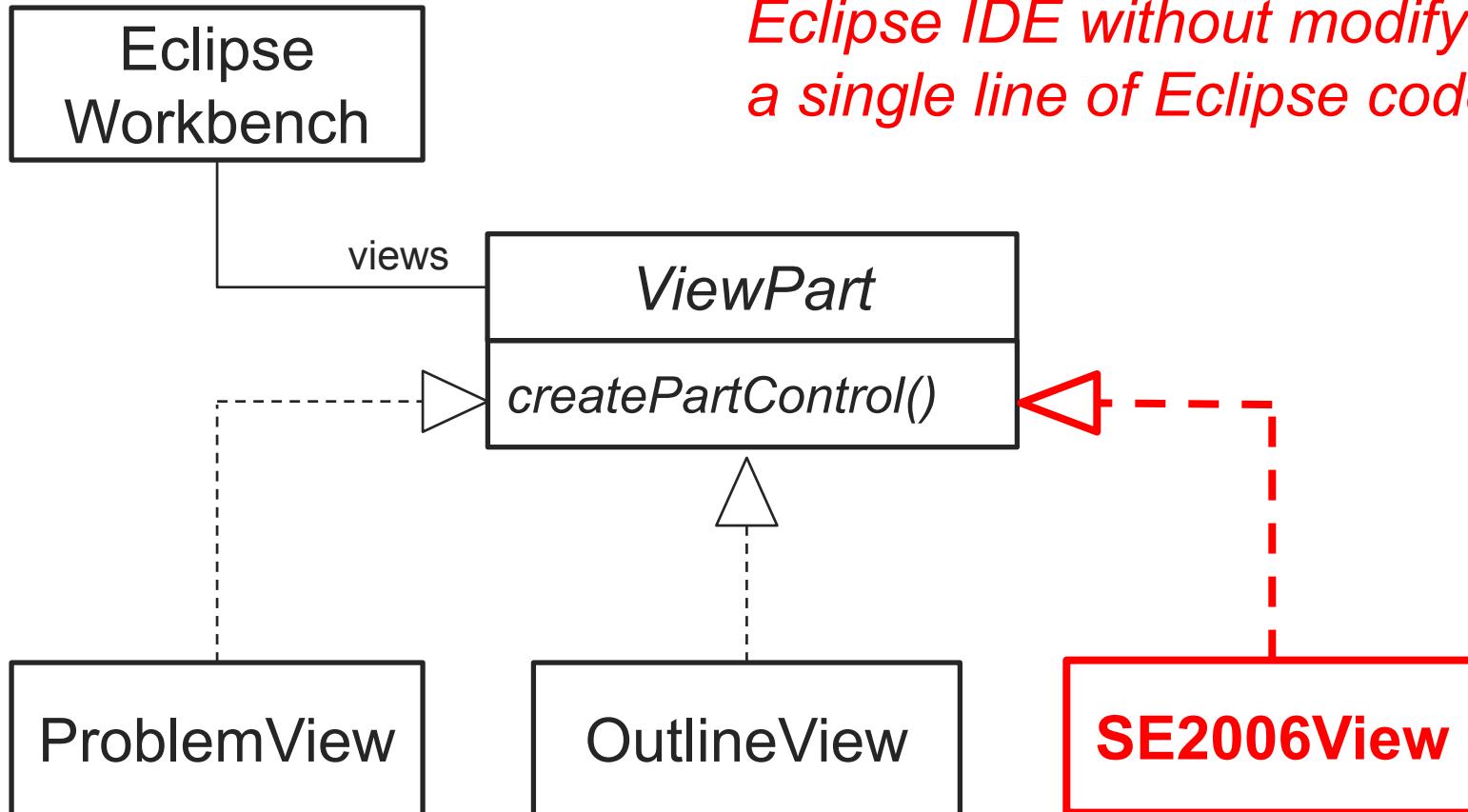
▶ Easy to extend, replace, or add new subclasses

- ▶ Replace MySQLImpl1 with MySQLImpl2
- ▶ Support new OracleImpl

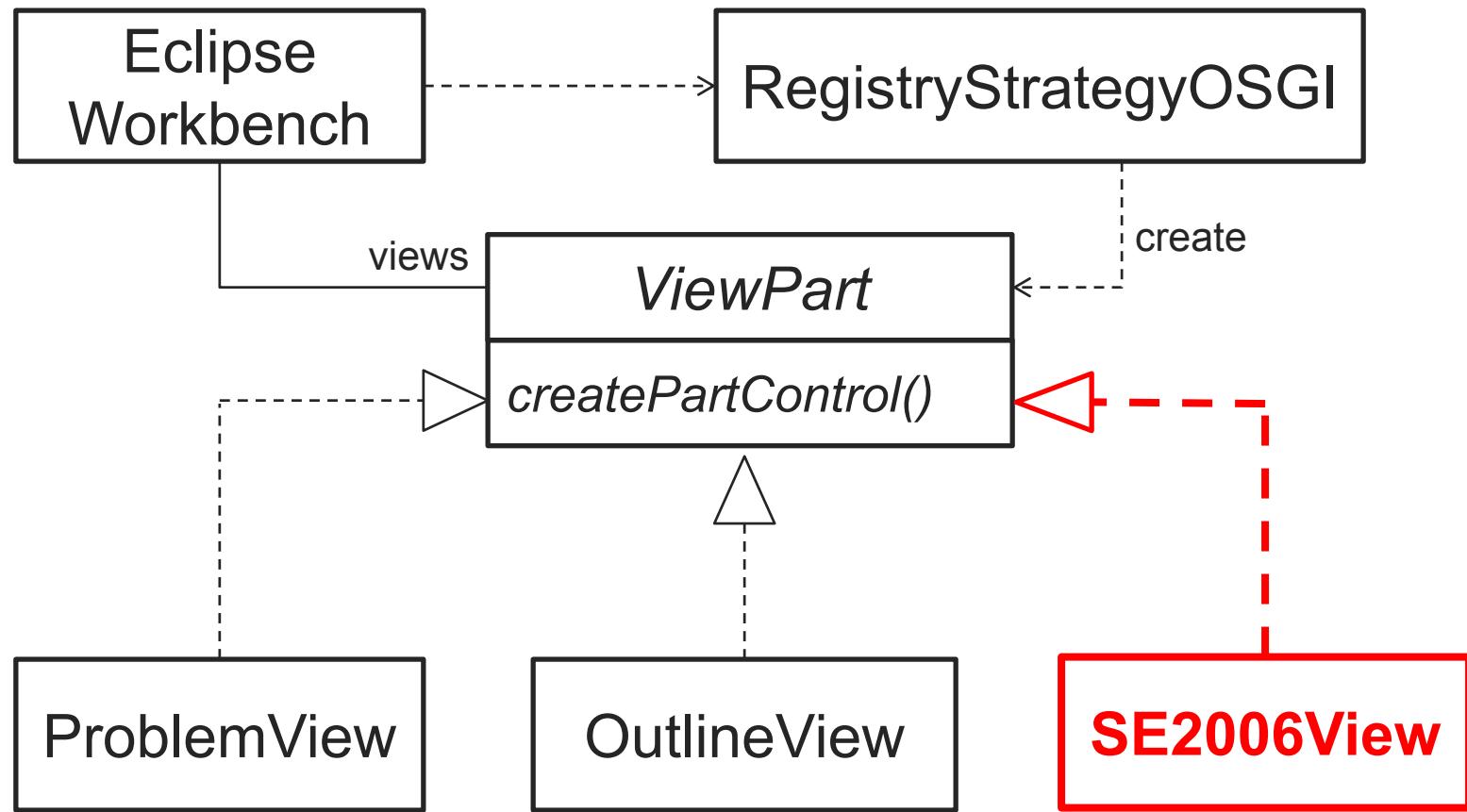
▶ Easy to change object creation logic

- ▶ Support Singleton object (see *EMSFactoryUsingSingleton.zip*)
- ▶ Support lazy initialization (see *EMSFactoryLazyInitialization.zip*)

Can You Extend Eclipse IDE?

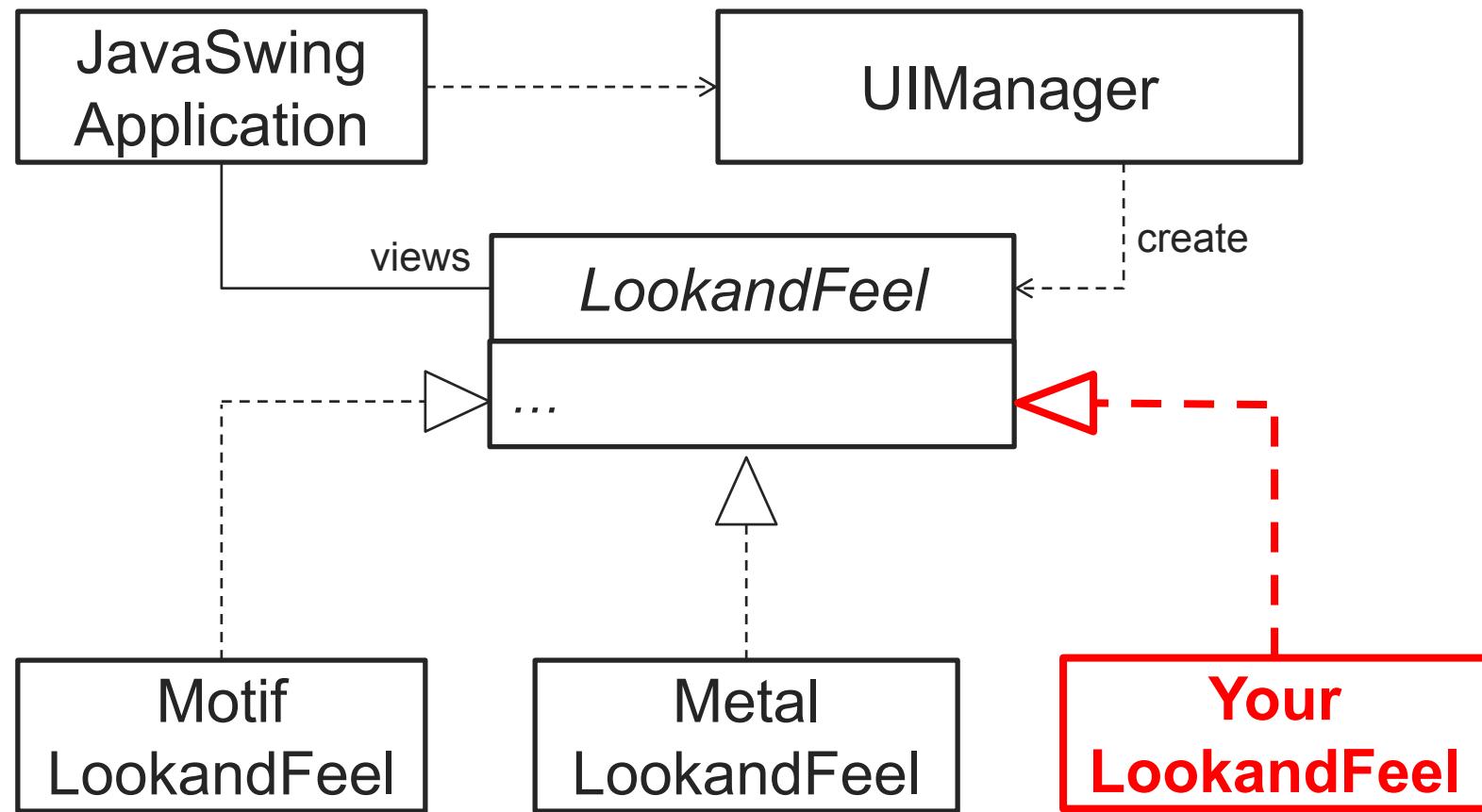


The Magic Power: Strategy Design + Factory Design + Dynamic Loading



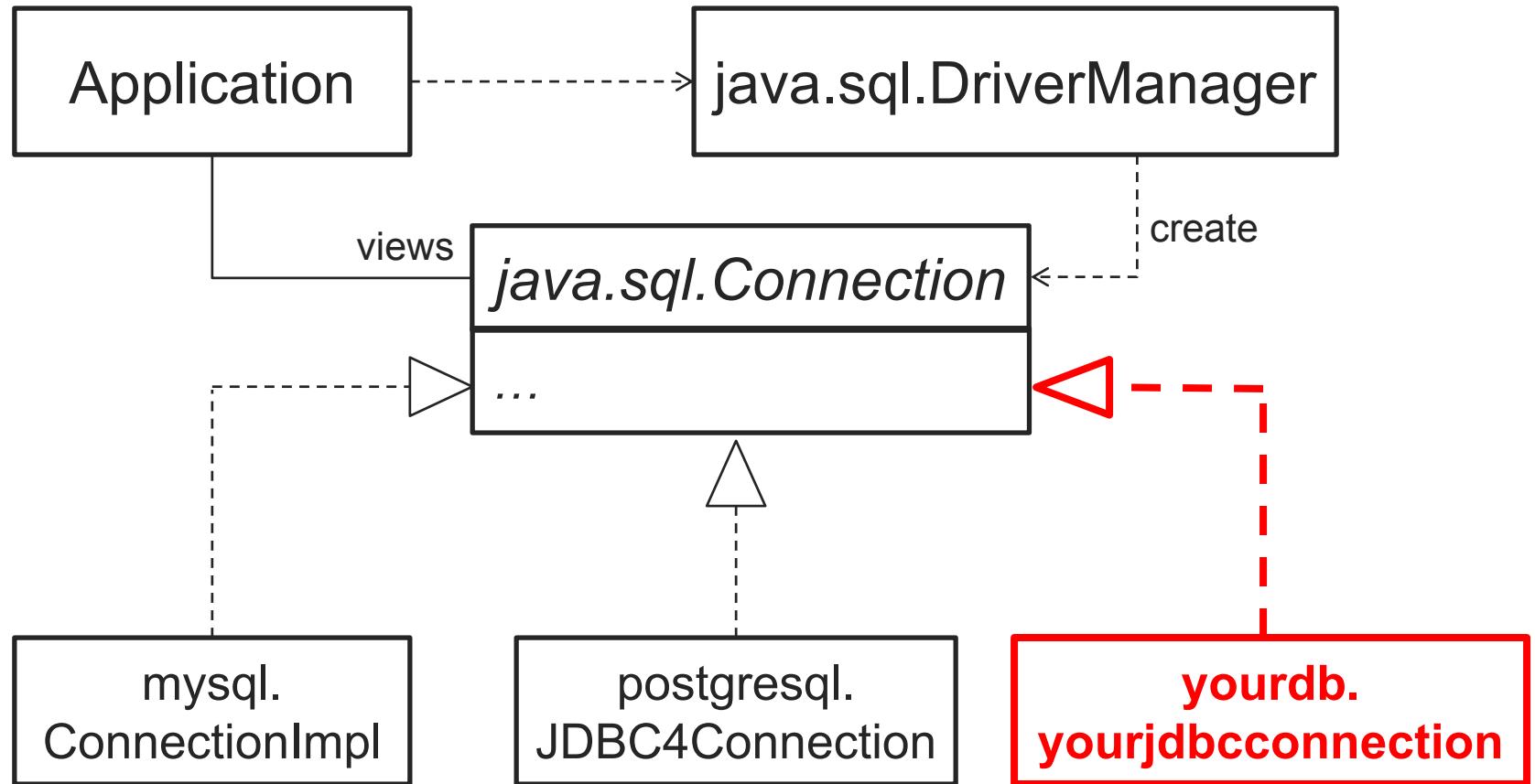
See [YouExtendEclipse.zip](#) for code example

Java Swing Framework Look & Feel



Strategy Design + Factory Design + Dynamic Loading

JDBC Database Driver



Strategy Design + Factory Design + Dynamic Loading

Strategy + Factory + Dynamic Loading

- ▶ Create any subclass object without specifying the class name in the code
- ▶ No need to change a single line of code to use new classes
- ▶ Provide the foundation for modern software frameworks to load application-specific classes

See [*EMSFactoryDynamicLoading.zip*](#) for code example

Façade Pattern Definition

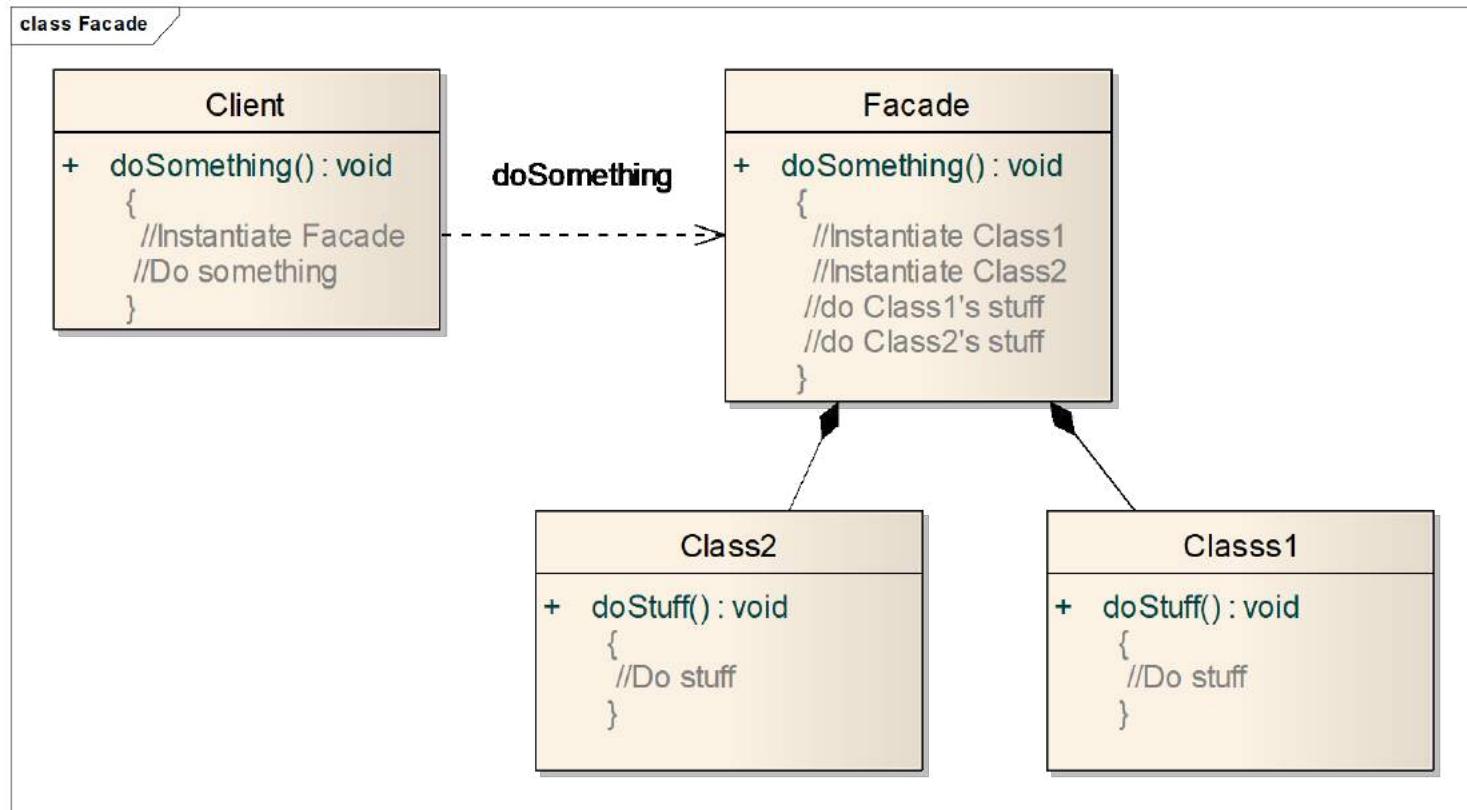
Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher level interface that makes the subsystem easier to use.



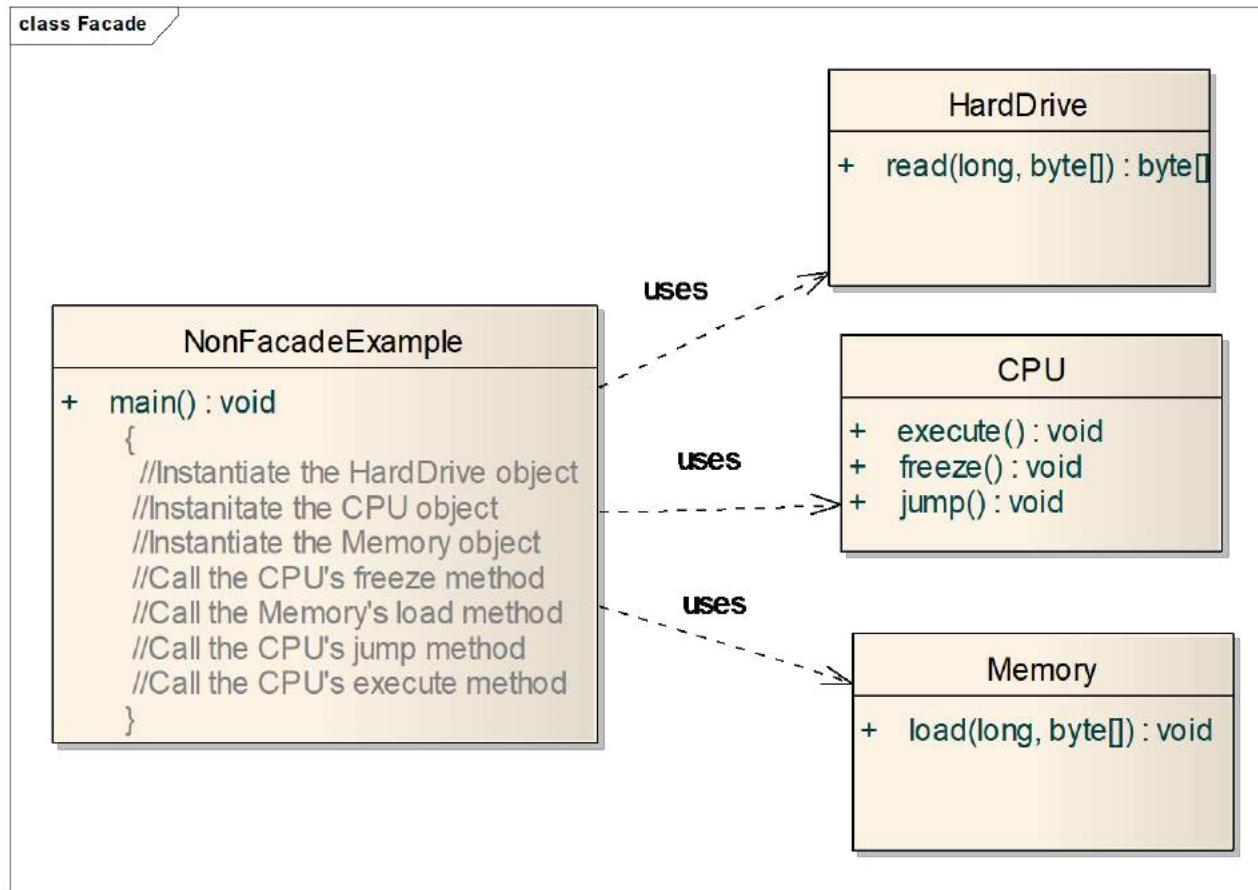
Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Principle of least knowledge – talk only to your immediate friends

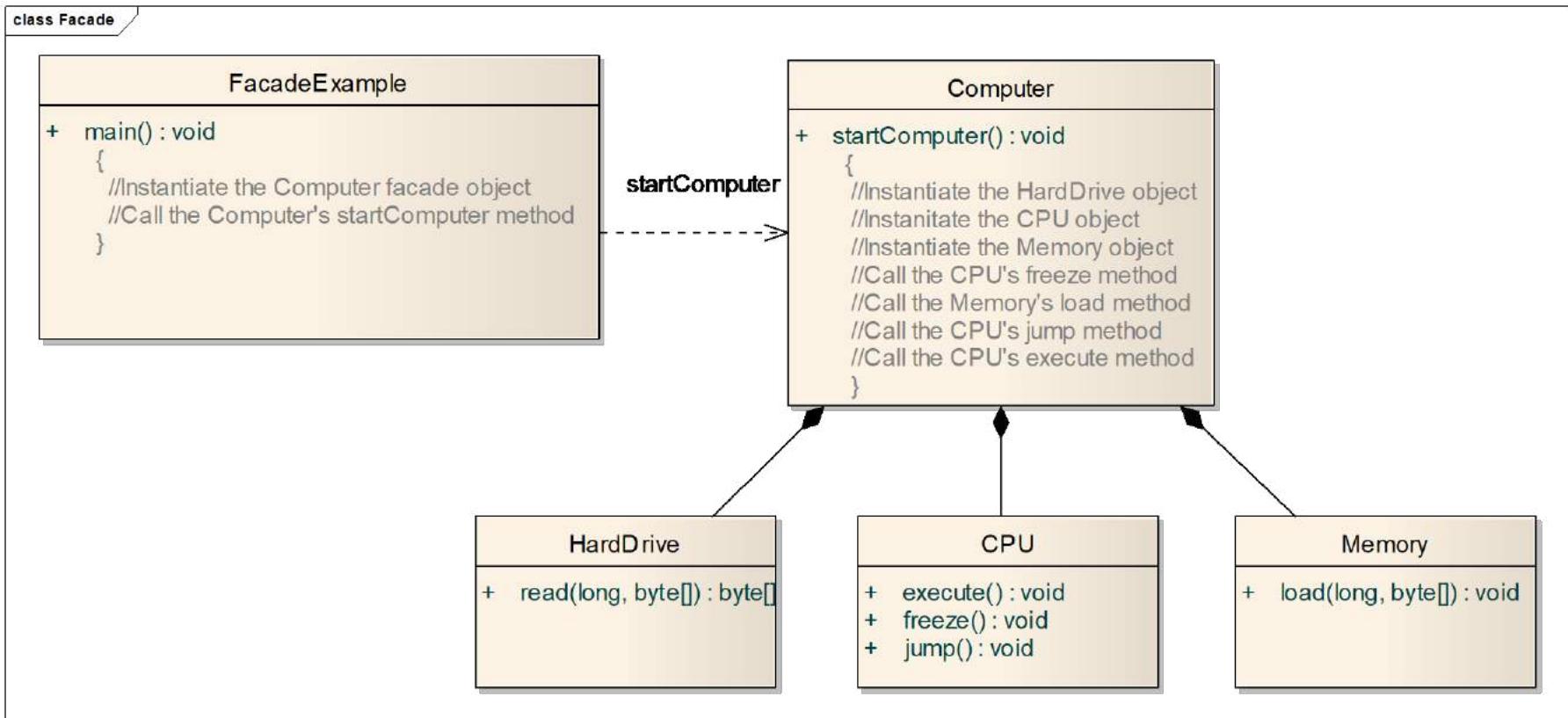
Façade – Class diagram



Façade - Problem



Façade - Solution

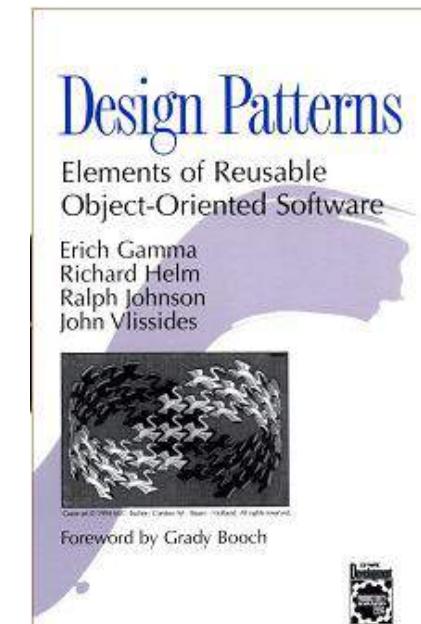


Facade

- Pros
 - Makes code easier to use and understand
 - Reduces dependencies on classes
 - Decouples a client from a complex system
- Cons
 - Results in more rework for improperly designed Façade class
 - Increases complexity and decreases runtime performance for large number of Façade classes

Summary of Patterns

- *Creational Patterns*
 - Abstract Factory
 - Builder
 - **Factory Method**
 - Prototype
 - Singleton
- *Structural Patterns*
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - **Façade**
 - Flyweight
 - Proxy
- *Behavioral Patterns*
 - Chain of Responsibility
 - Command
 - Interpreter
 - Iterator
 - Mediator
 - Memento
 - **Observer**
 - State
 - **Strategy**
 - Template Method
 - Visitor



CZ2006/CE2006 Software Engineering

Lecture 15: MVC Design Patterns

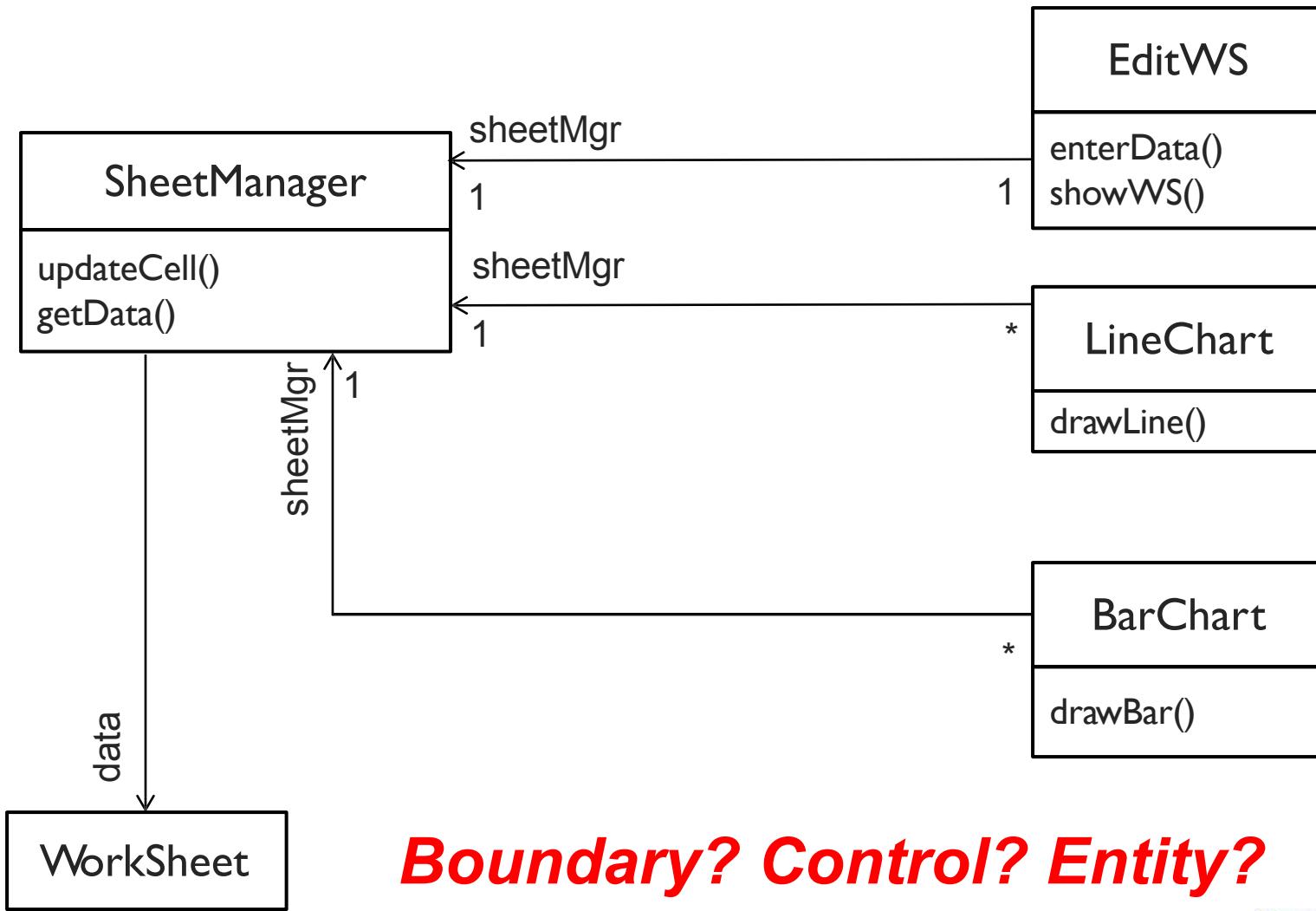
Liu Yang

Interactive Event-Driven Systems

- ▶ UIs are prone to change
 - ▶ The same data can be presented differently
 - ▶ Changes to UI must be easy and even possible at runtime
 - ▶ The application display and behavior must reflect data changes immediately
 - ▶ The same presentation can have different look and feel
 - ▶ The application reacts to user input differently
- ▶ UI should **NOT** be tightly coupled with core functionality and data in order to allow for the above changes

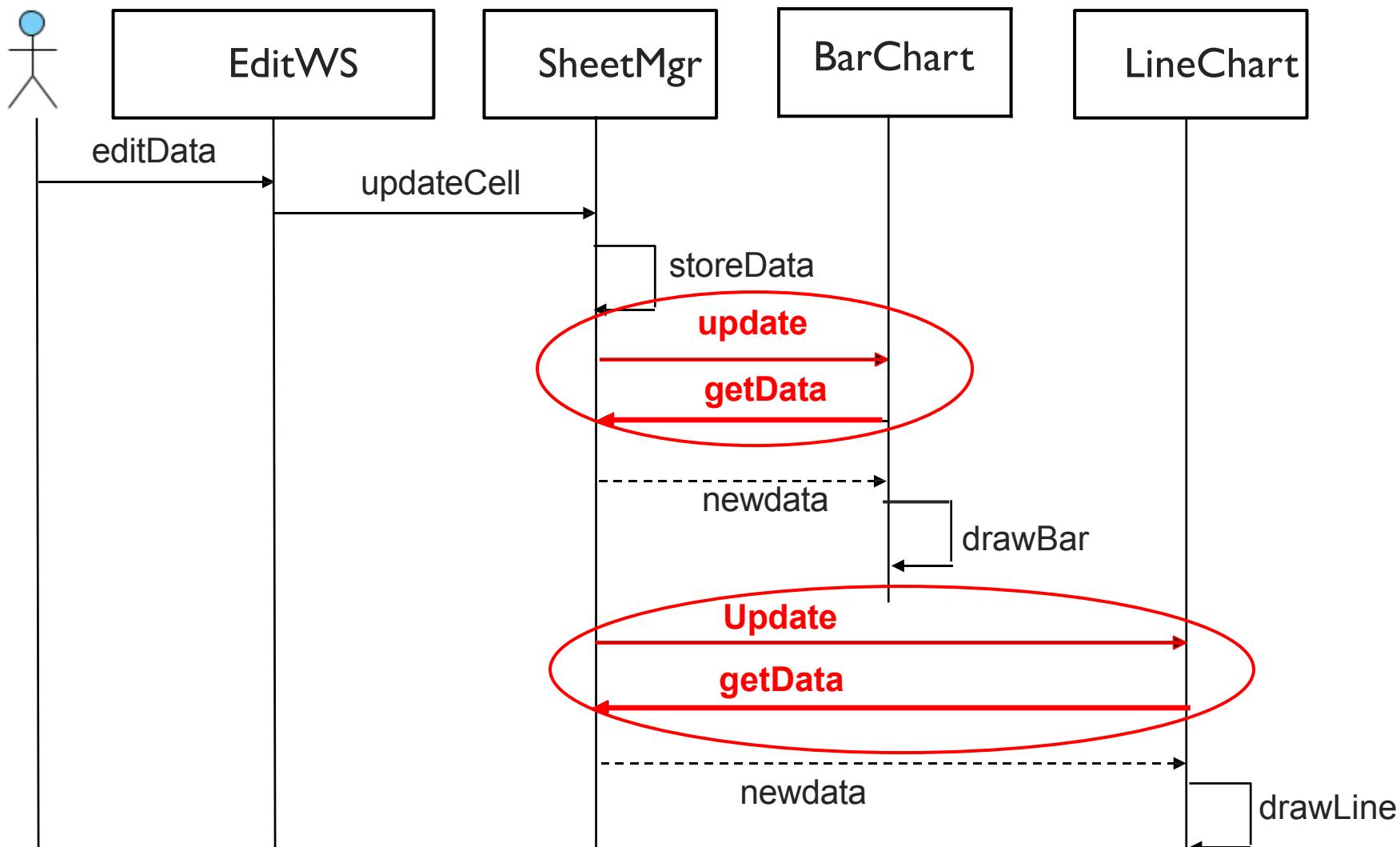
*This is not a complete list of design challenges
for interactive event-driven systems!*

Excel Conceptual Model

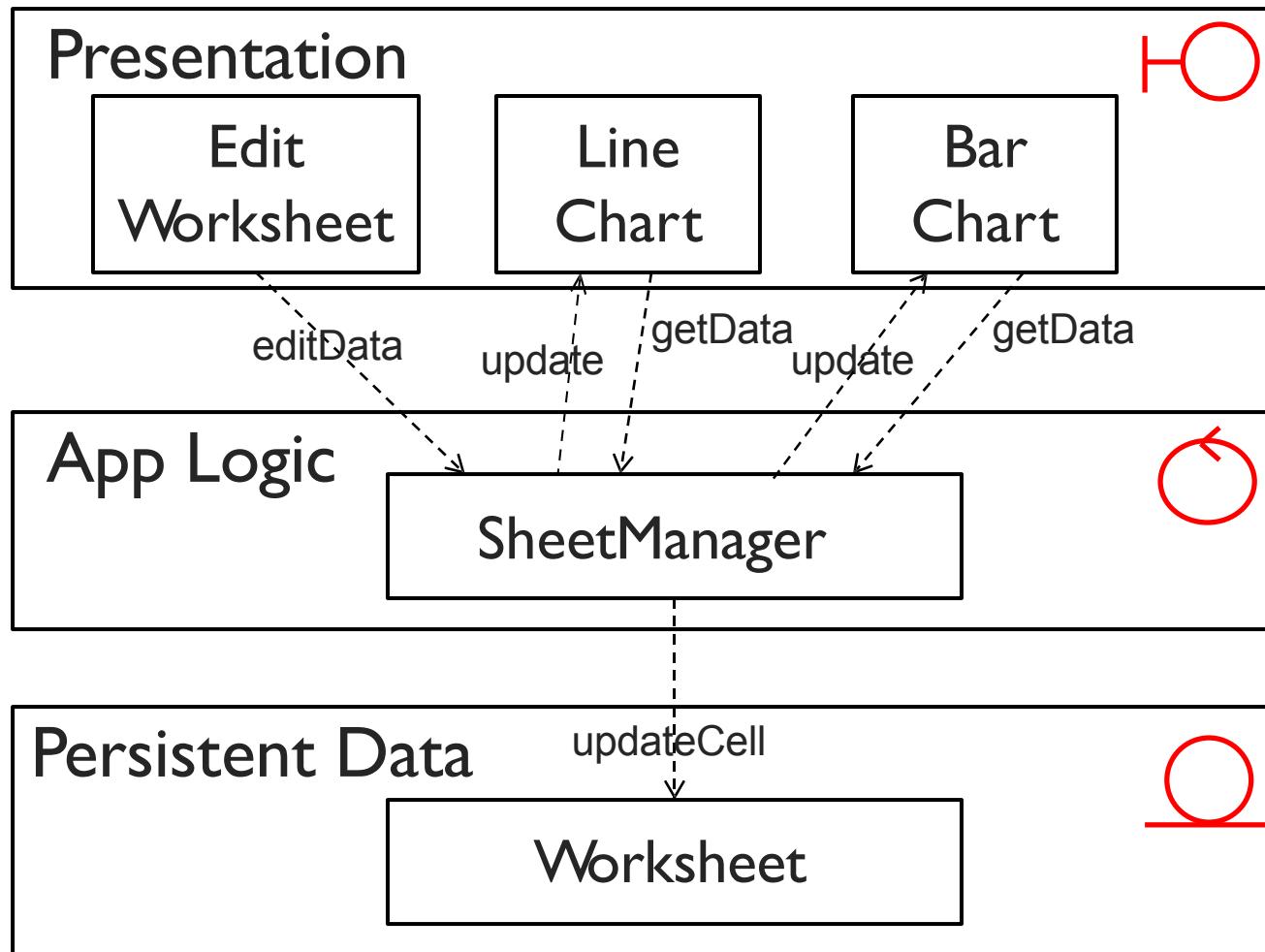


Boundary? Control? Entity?

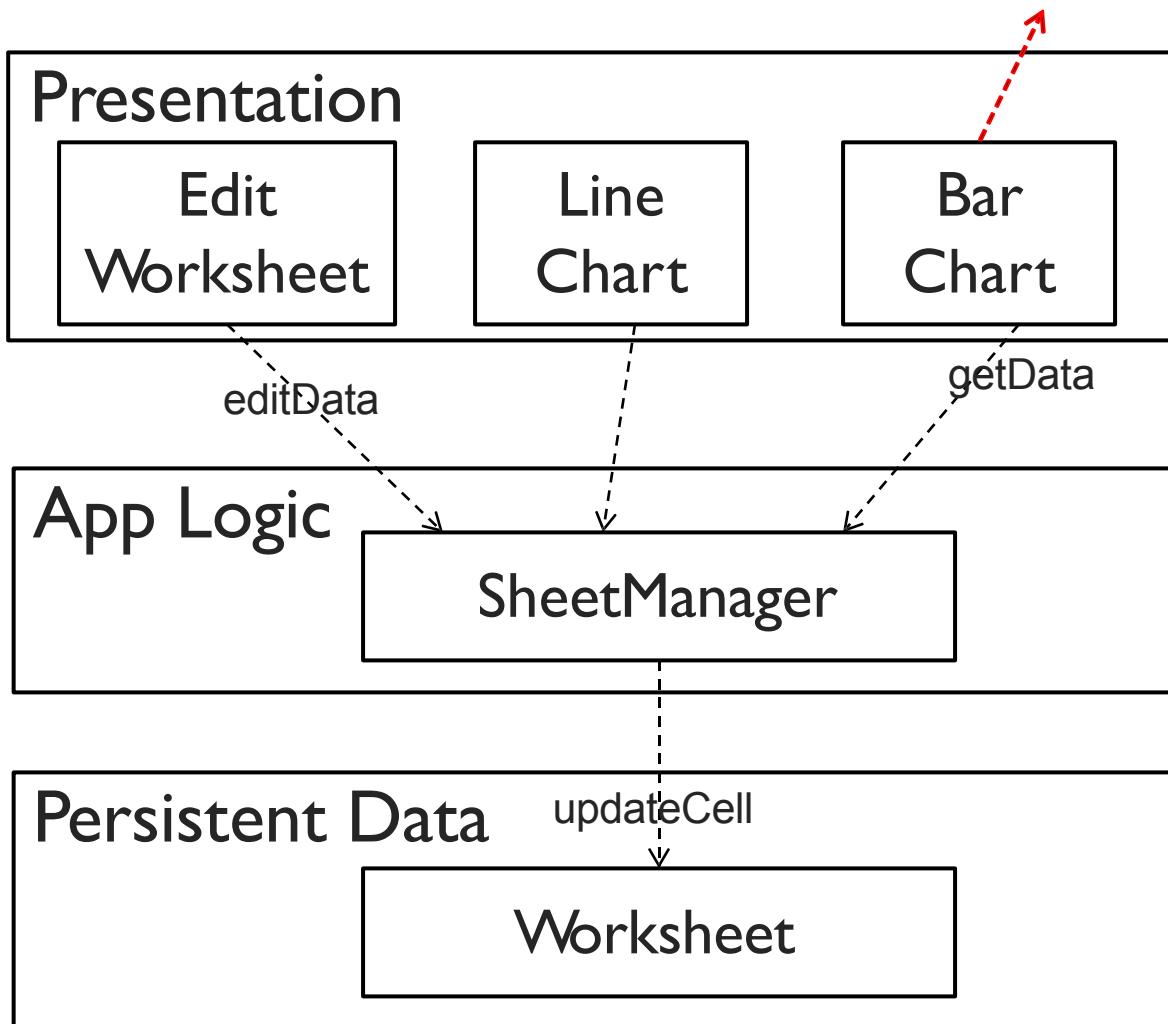
Update Worksheet – Sequence Diagram



Excel Layered Architecture



Excel Layered Architecture



***What's wrong
with this
Layered
Architecture?***

***What kind of
coupling between
Presentation and
App Logic?***

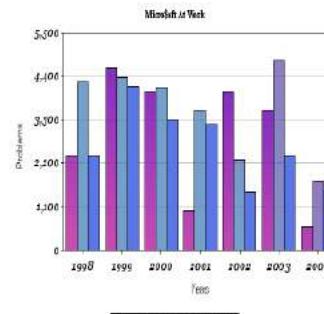
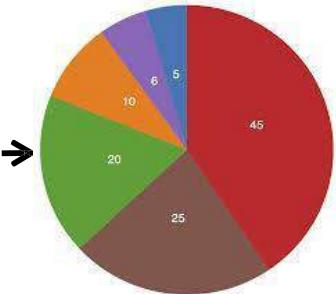
What do We Really Need?

- ▶ We need many **views** to receive an **update** when **worksheet changes**



Name	Thread pitch (mm)	Minor diameter tolerance	Nominal diameter (mm)	Head shape	Price for 50 screws	Available at factory outlet?	Number in stock	Flat or Phillips head?
M4	0.7	4g	4	Pan	\$10.08	Yes	776	Flat
M5	0.8	4g	5	Round	\$13.89	Yes	183	Both
M6	1	5g	6	Button	\$10.42	Yes	1043	Flat
M8	1.25	5g	8	Pan	\$11.98	No	298	Phillips
M10	1.5	6g	10	Round	\$16.74	Yes	488	Phillips
M12	1.75	7g	12	Pan	\$18.26	No	588	Flat
M14	2	7g	14	Round	\$21.19	No	235	Phillips
M16	2	8g	16	Button	\$23.57	Yes	292	Both
M18	2.1	8g	18	Button	\$25.87	No	664	Both
M20	2.4	8g	20	Pan	\$28.09	Yes	486	Both
M24	2.55	9g	24	Round	\$33.01	Yes	982	Phillips
M28	2.7	10g	28	Button	\$35.66	No	1067	Phillips
M36	3.2	13g	36	Pan	\$41.32	No	434	Both
M50	4.5	13g	50	Pan	\$44.72	No	740	Flat

Worksheet
Cell A10
changed



What are the Design Problems?

- ▶ Loose coupling between an **worksheet** (**subject**) and its dependent **views** (**observers**)
- ▶ **Worksheet** wants to notify **views** the data change once it occurs, but views of a worksheet can be created/deleted freely and constantly.
- ▶ **Views** want to show the data change immediately once it occurs, but they do not know when the data change will occur.

What design pattern can help?

Identify the Design Pattern

Worksheet

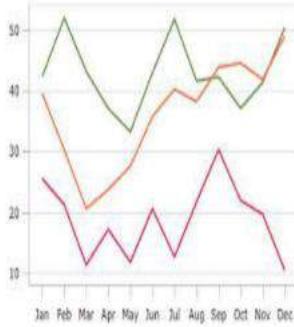
Cell A10
changed

Subject

- I do not care in what format the data is visualized and how many views are visualizing the data
- I want to let whoever is visualizing the data know the latest data change when it occurs
- It is up to views to decide what to do with the data change

Loose coupling is a benefit for both sides!

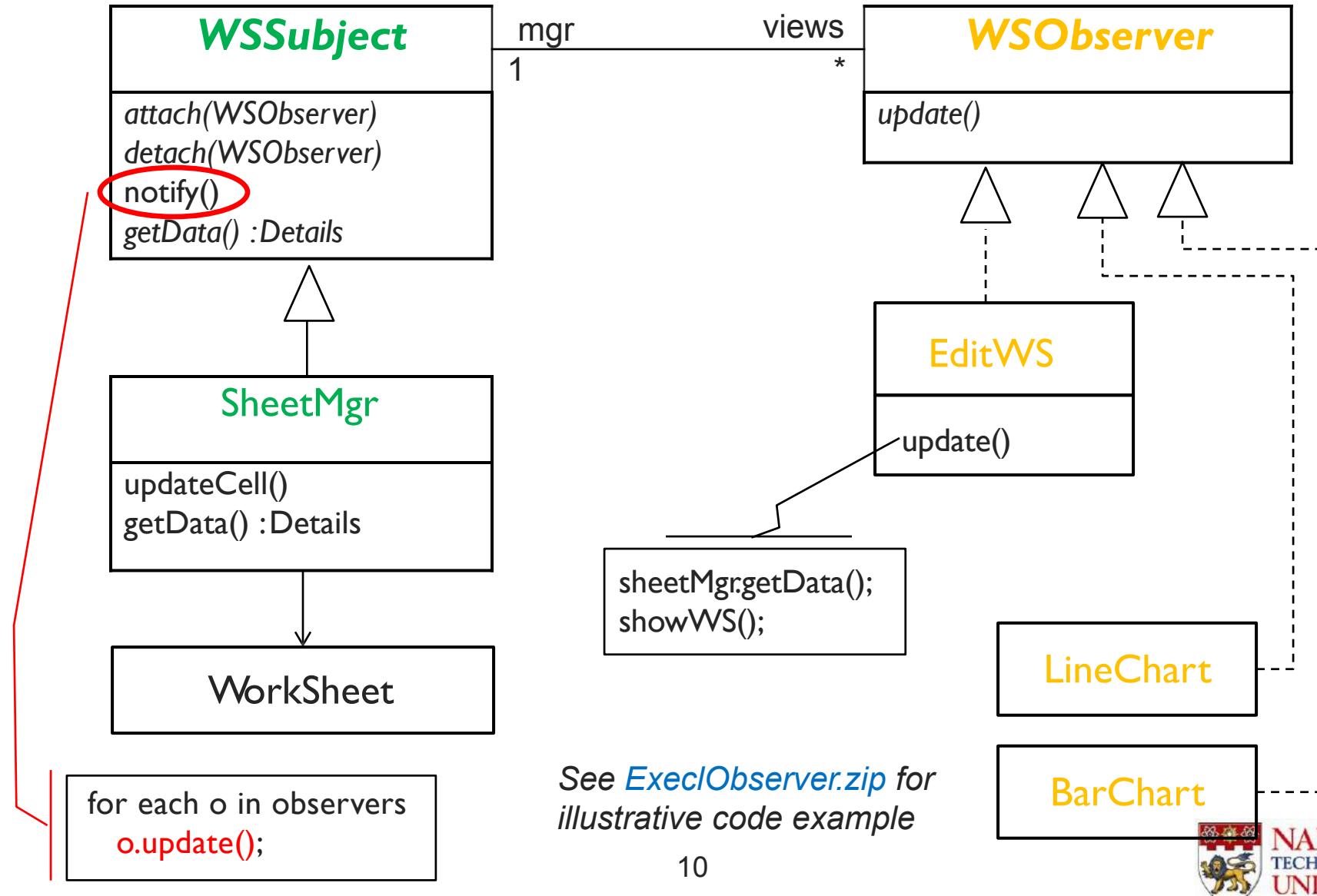
**Observer
Pattern!**



Observer

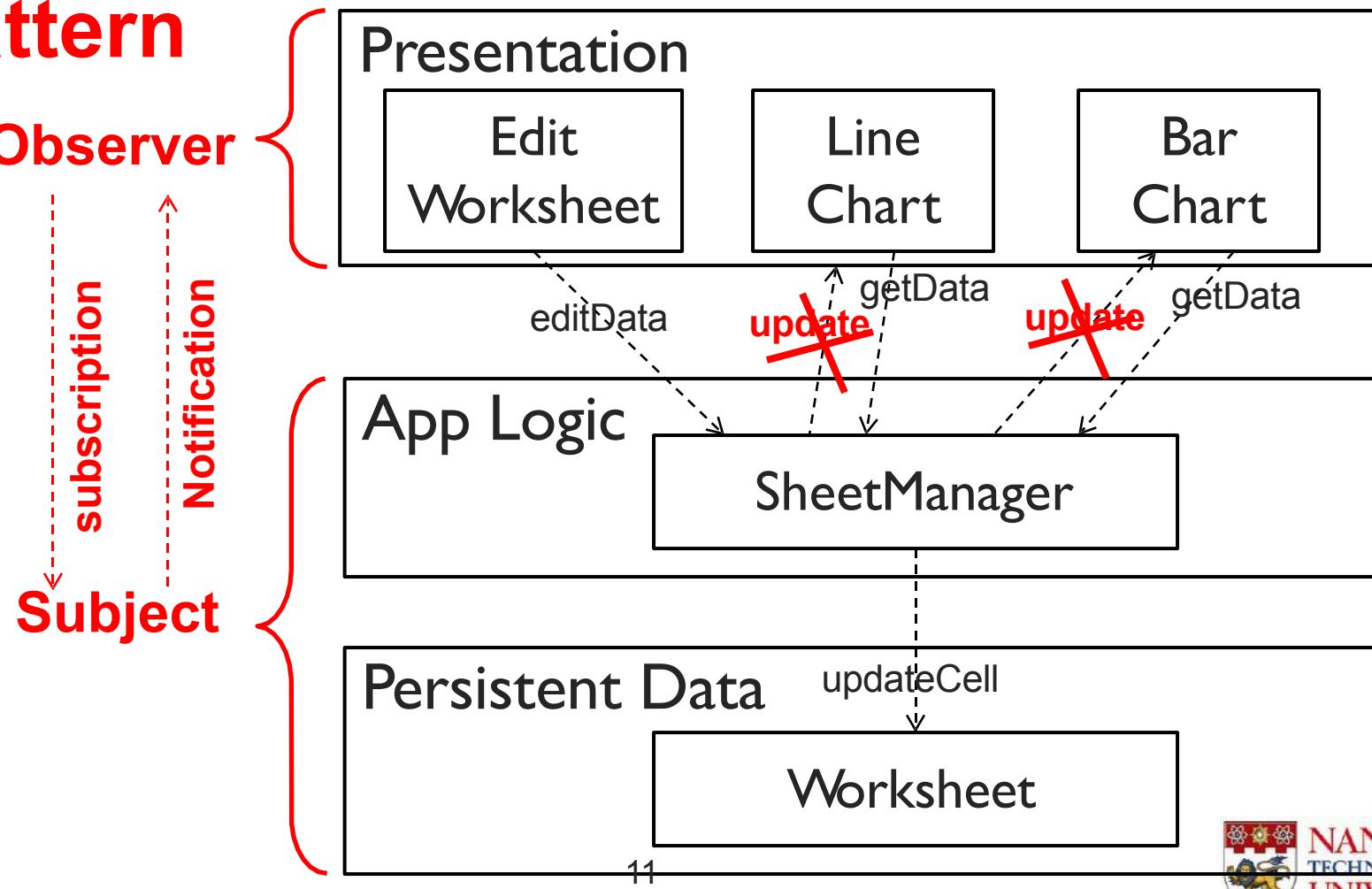
- I can be created and deleted for data visualization freely
- I want to update the chart with the latest data change, but I do not know when the data change will occur
- I cannot constantly check the data change because I do not know how frequent I should check

Observer Pattern – Excel Worksheet – Views

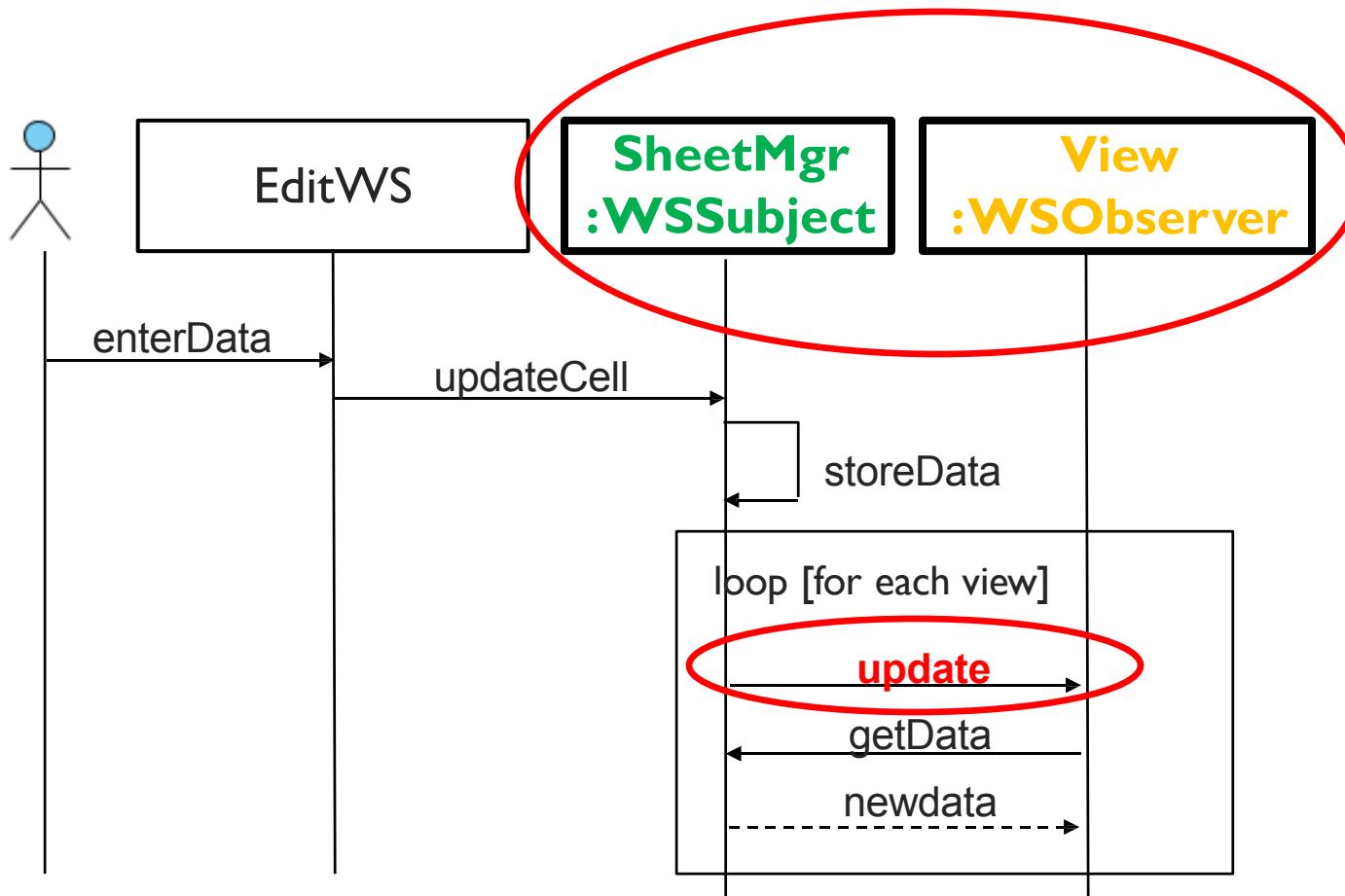


Excel Architecture

Apply
Observer
Pattern



Observer Pattern – Update Worksheet



SheetManager (or other manager classes) and *views* communicate as **WSSubject** and **WSObserver** objects, not specific manager classes or views!

All Problems Solved?

- ▶ UIs are prone to change
- ✓ ▶ The same data can be presented differently
- ✓ ▶ Changes to UI must be easy and even possible at runtime
- ✓ ▶ The application display and behavior must reflect data changes immediately
- ? ▶ The same presentation can have ***different look and feel*** – ***what pattern can be used here?***
- ? ▶ The application ***reacts to user input differently*** – ***what pattern can be used here?***

All Problems Solved?

- ▶ UIs are prone to change
- ✓ ▶ The same data can be presented differently
- ✓ ▶ Changes to UI must be easy and even possible at runtime
- ✓ ▶ The application display and behavior must reflect data changes immediately
- ▶ The same presentation can have ***different look and feel – Strategy Pattern (interchangeability)***
- ▶ The application ***reacts to user input differently – Strategy Pattern (interchangeability)***

What We Have Discovered

- ▶ One problem with layering (layered architecture) is that many situations require up-calls from lower to higher layers – *dependency*, e.g. Excel worksheet – *update*
- ▶ How do we allow up-calls without creating dependencies between lower and higher layers?
 - ▶ ***Apply design pattern – observer pattern (loose coupling between two layers)***
- ▶ **Observer pattern** enables loose coupling between Excel worksheet and views, but it DOES NOT solve all problems – may need more than one design pattern!
E.g. observer pattern + strategy pattern

Model-View-Controller (MVC) Architecture

- ▶ **Main Goal**
 - ▶ To facilitate and optimize the implementation of **interactive intensive systems**, particularly those that use multiple synchronized presentations of shared information.
- ▶ **Key Idea**
 - ▶ The separation of the **Model** from **View** and **Controller** components allows multiple **Views** of the same **Model** – It separates presentation and interaction from the data. If the user changes the **Model** via the **Controller** of one **View**, all other **Views** dependent on this data should reflect the changes.

Model-View-Controller (MVC) Architecture

- ▶ Separation of UI from the core data model
 - ▶ Need to support multiple different user interface (e.g. desktop GUI, web browser, PDA, cell phone, etc.) – Core functionality should be reusable across all different interfaces.
 - ▶ Interface details changes frequently – Changing UI details should not affect core functionality.
 - ▶ It should be easy to add a new user interface.

Model-View-Controller (MVC) architecture model describes separation of UI from core data model

Model-View-Controller (MVC) Architecture

▶ MVC Components

Boundary? Control? Entity?

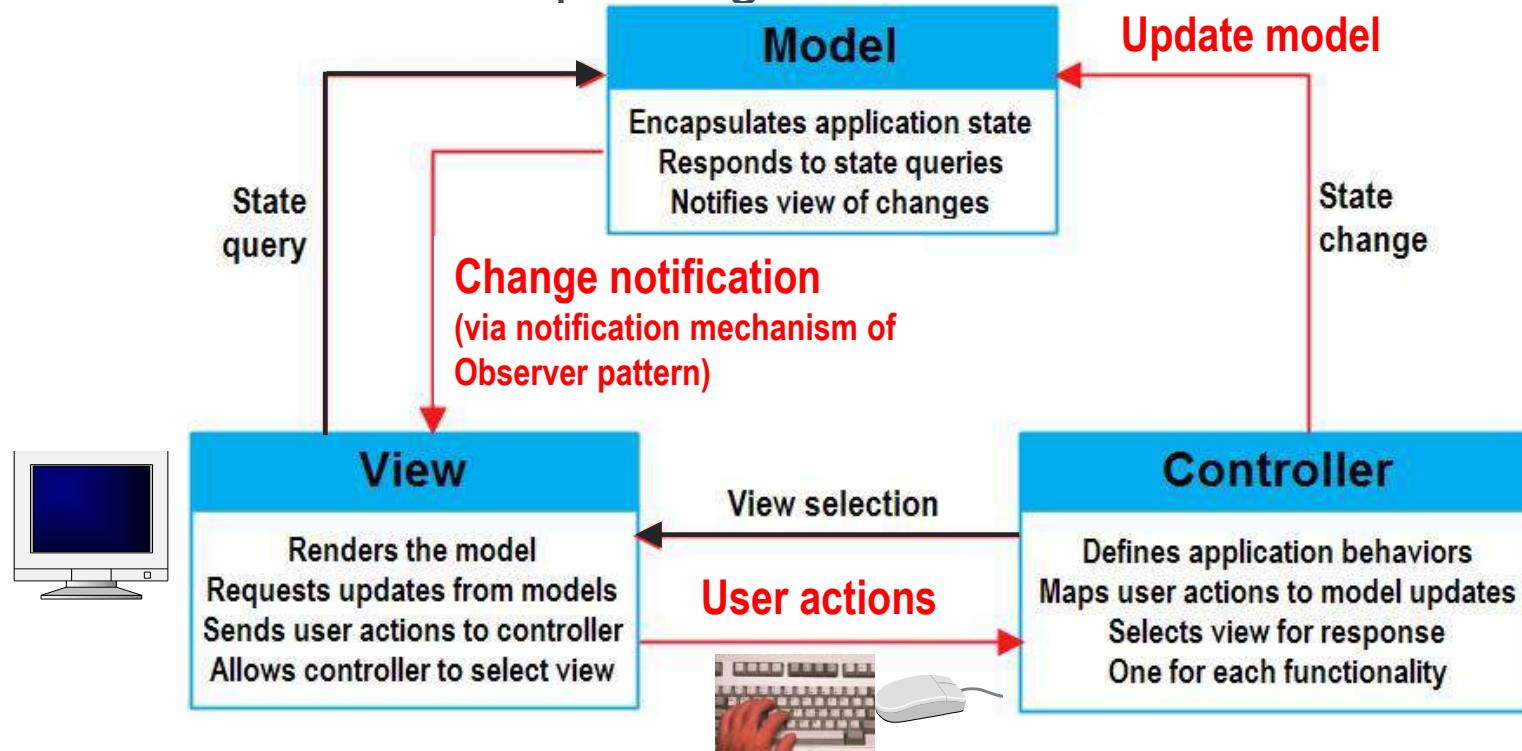
View, Controller) **(Model)**

- ▶ Encapsulate core Functionality (App Logic) + Data in **Model**.
 - ▶ Implement UI in **View** (present data) + **Controller** (react to user input).

Model	Contains the processing (operations) and the data involved.
View	Presents the output – Defines and manages how the data is presented to the user; each View provides specific presentation of the same Model . Each View “observes” the Model – Whenever the data Model changes, all Views immediately notified and so they can update their graphical presentation.
Controller	Manages user interaction – Captures user input (events-> mouse clicks, key presses, etc.) and passes these interactions to the View and the Model . Each View is associated to a Controller that captures and processes user input and modifies the data Model . The user interacts with the system solely through Controllers .

Model-View-Controller (MVC) Architecture

- ▶ **Controllers** typically implement event-handling mechanisms that are executed when corresponding events occur.



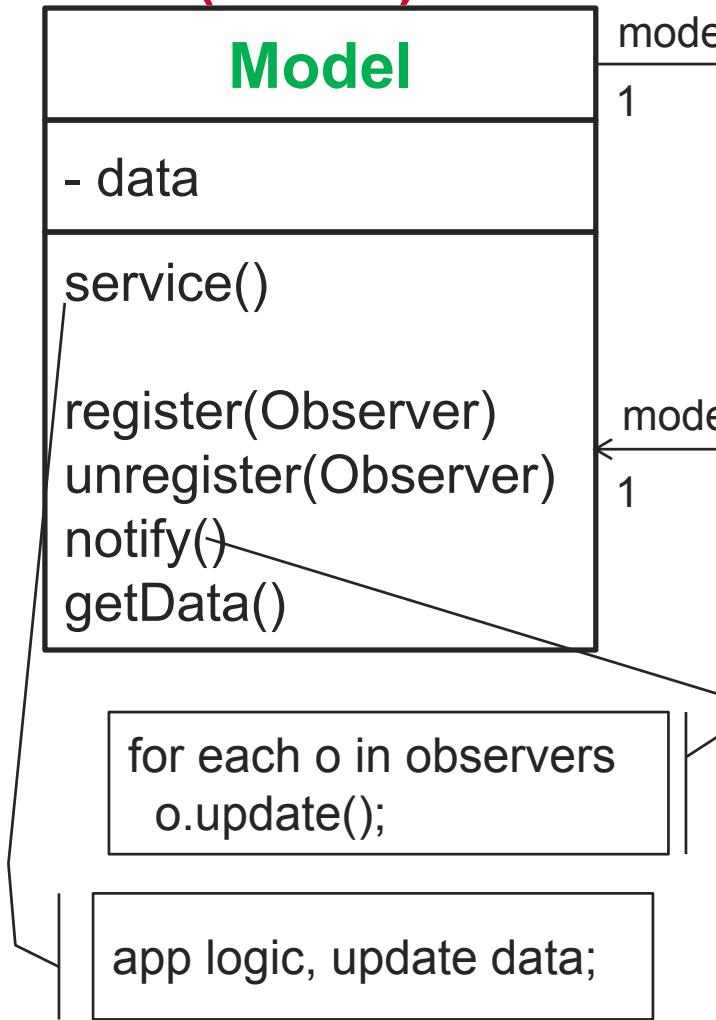
- ▶ Changes made to the **Model** by the user via **Controllers** are directly propagated to the corresponding **Views**. The change propagation mechanism can be implemented using the **observer pattern** (via subscribe/notify protocol).

Model-View-Controller (MVC) Architecture

- ▶ **MVC architecture style is non-hierarchical (triangular)**
- ▶ **View** subscribes for changes to the **Model** (via Observer subscription mechanism)
- ▶ **Controller** gathers input (or change) from the users and updates the **Model**.
- ▶ **Model** updates the change and **notifies** all subscribers (the **Views**) of the change (via Observer notification mechanism)
- ▶ **View** is notified and updates its display; the user sees the change.

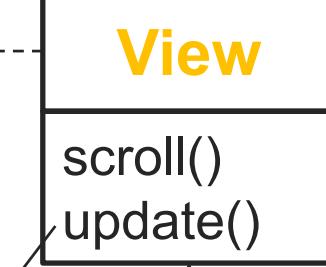
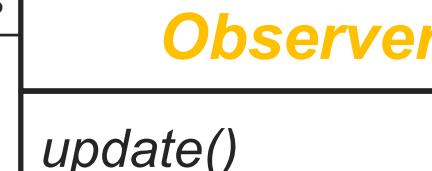
MVC Architecture

**AppLogic + Data
(Model)**



Presentation

(View + Controller)



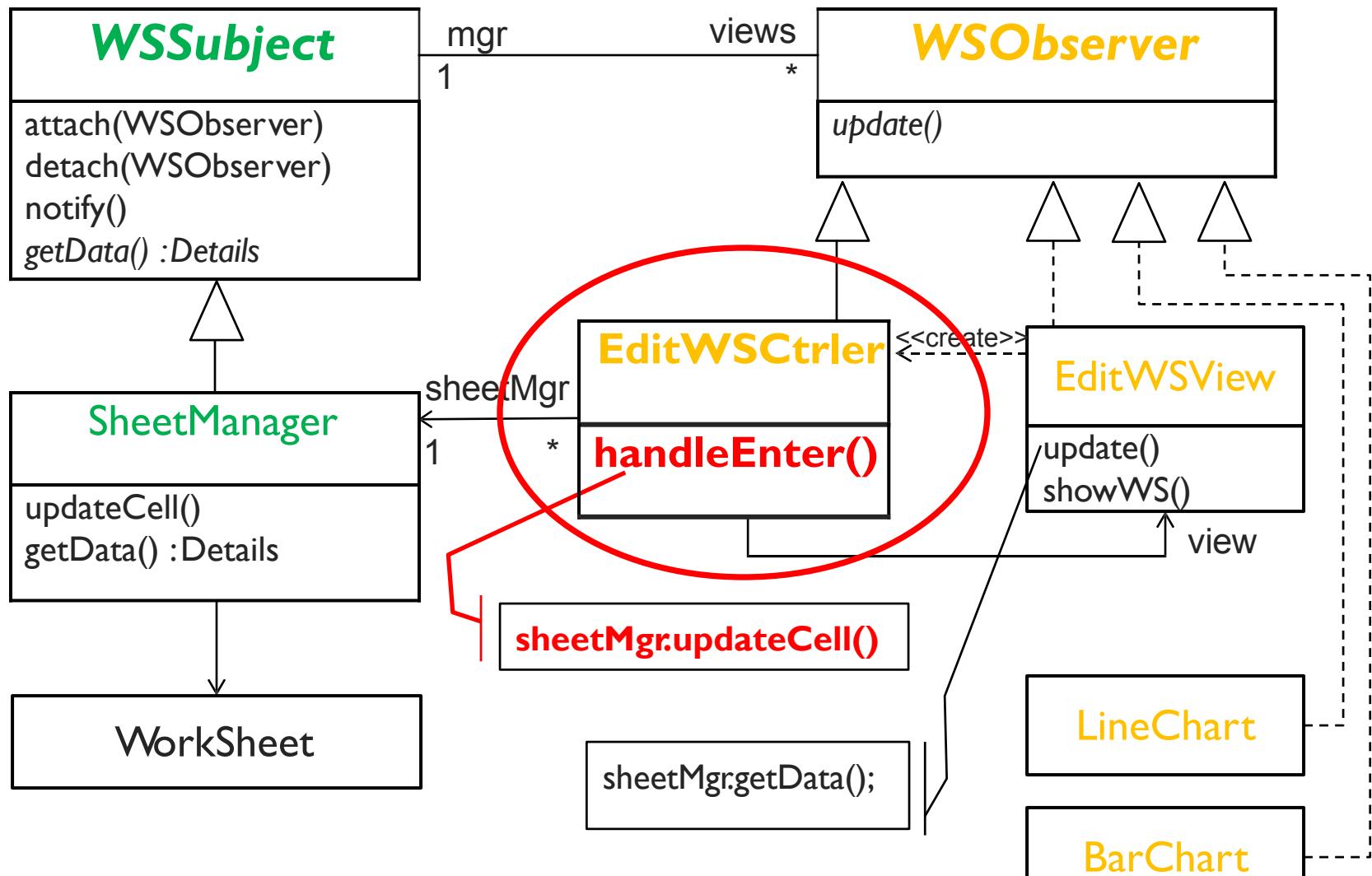
model.service()

model.getData();
redraw view;

Observer + Strategy Patterns in MVC

- ▶ **Observer Pattern (View–Model relationship):**
 - ▶ **Model** uses the Observer Pattern to keep the **View(s)** updated.
 - ▶ When **Model** (Subject) data is changed, it updates all the **Views** (Observers, via notification mechanism) – Allows multiple **Views** to the same **Model**.
- ▶ **Strategy Pattern (View–Controller relationship):**
 - ▶ **View** and **Controller** use the Strategy Pattern as the **View** is concerned with the visual aspects (display data) and delegates the interface behaviors to the **Controller**.
 - ▶ The **View** (Context) uses the **Controller** (Strategy Interface) to implement a specific type of response (or strategy objects). The **Controller** can be changed to allow **View** to respond differently to user input.

Excel MVC



Model-View-Controller (MVC) Architecture

► Design patterns in MVC

► **Observer**

- ▶ View – Model; Controller - Model

► **Strategy**

- ▶ View – Controller; View – Look & Feel

► **Abstract Factory**

- ▶ Create controller or look & feel for a view

► **Composite**

- ▶ Nested views (e.g. Panel contains Button; Nested Panels)

► **Decorator**

- ▶ Attach additional UI functionalities (e.g., scrolling) to a view

► **Command**

- ▶ Support undo/redo user actions

*Observer, Strategy
and Abstract
Factory are the key
patterns in MVC*

*Other patterns are
also important for
the design of
interactive systems.*

MVC

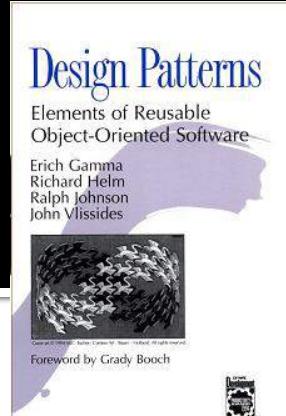
■ Pros

- Simultaneous development
- Multiple views for a model – Models can have multiple views
- High cohesion
 - MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Low coupling
 - The very nature of the MVC framework is such that there is low coupling among models, views or controllers

■ Cons

- Code navigability and pronounced learning curve
 - The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Multi-artifact consistency
 - Decomposing a feature into three artifacts causes scattering. Thus, requiring developers to maintain the consistency of multiple representations at once.

Summary of MVC



- Problem
 - Interactive Event-Driven Systems
- Solution (Pattern Composition)
 - Observer Pattern + Strategy Pattern + ...
- Popular patterns in many real frameworks
 - Django, Reils, .NET MVC...

CZ2006/CE2006 Software Engineering

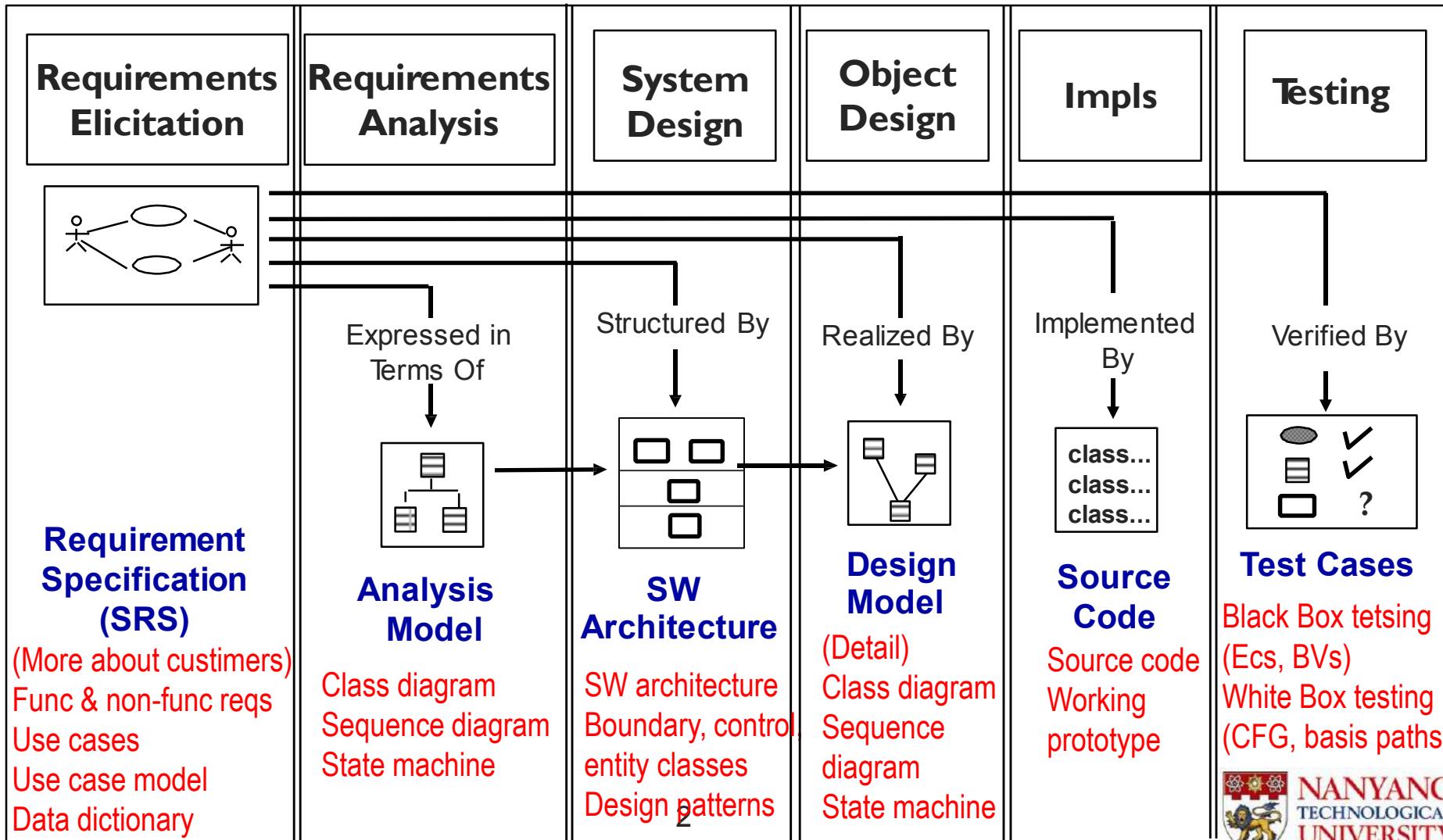
Lecture 16: Test Fundamentals

Liu Yang

SDLC Activities

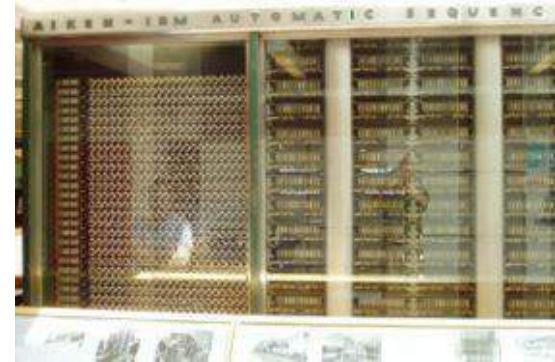
Problem

Solution



What is a computer bug?

- In 1947 Harvard University was operating a room-sized computer Called the Mark II.
 - mechanical relays
 - glowing vacuum tubes
 - technicians program the computer by reconfiguring it
 - Technicians had to change the occasional vacuum tube.
- A moth flew into the computer and was zapped by the high voltage when it landed on a relay.
- Hence, the first computer bug!



When I launch my app after several hours of development

Unexpected things happen!



A **software bug** is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.

Test Fundamentals

Can We Test Everything?

- ▶ A function is supposed to
 - ▶ *Take an integer input value, add 1 to the input, divide it by 30000 and return the value*
- ▶ How many possible input values for 16-bits integer?
 - ▶ $2^{16} = 65536$

A buggy program!

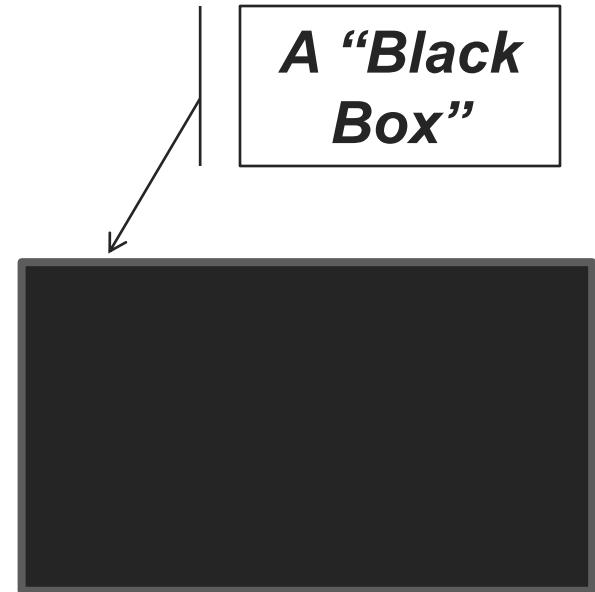
```
int blech(int input) {  
    input = input - 1;  
    return input/30000;  
}
```

Can we test all the possible inputs?

Yes, but TOO COSTLY!

Can We Test Everything?

- ▶ A function is supposed to
 - ▶ *Take an integer input value, add 1 to the input, divide it by 30000 and return the value*
- ▶ Which input values do you choose to test the program?
 - ▶ 1, 42, 32000, -31218, ...



*Are these good selection of inputs?
Can we detect the bug of the program?*

Equivalence Class and Boundary Value Testing

- Add1Divide30000 ($-32768 \leq \text{input} \leq -30001$) [return -1]
 - Lower boundary -32768: ~~32769~~, -32768, ~~32767~~
 - Upper boundary -30001: ~~30002~~, -30001, ~~30000~~
- Add1Divide30000 ($-30000 \leq \text{input} \leq 29998$) [return 0]
 - Lower boundary -30000: ~~30001~~, -30000, ~~29999~~
 - Upper boundary 29998: ~~29997~~, 29998, ~~29999~~
- Add1Divide30000 ($29999 \leq \text{input} \leq 32767$) [return 1]
 - Lower boundary 29999: ~~29998~~, 29999, ~~30000~~
 - Upper boundary 32766: ~~32766~~, 32767, ~~32768~~

Can We Test Everything?

- ▶ How many possible input values for 16-bits integer n?
 - ▶ $2^{16} = 65536$
- ▶ Which input values do you choose to test the program?
 - ▶ 1, 42, 32000, -31218, ...
- ▶ Can we cover all statements, all branches, or all execution paths?

A “White Box”

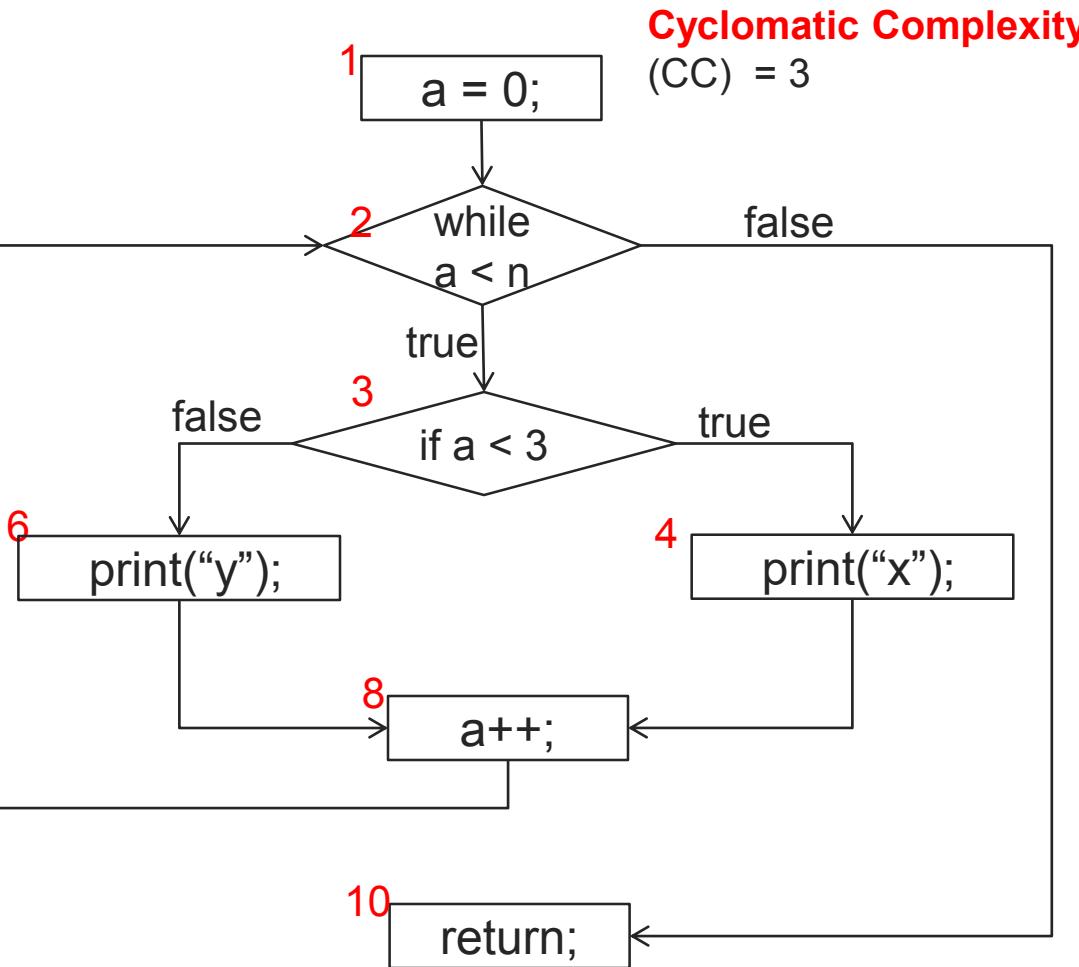
```
doSomething(int n) {  
    1. int a = 0;  
    2. while(a < n) {  
    3.     if(a < 3) {  
    4.         print("X");  
    5.     } else {  
    6.         print("Y");  
    7.     }  
    8.     a++;  
    9. }  
10. return;  
}
```

Basis Path Testing

- ▶ **Basis Path Testing** (or structured testing) is a white box testing method used for designing test cases intended to examine all possible paths of execution in a program at least once.
 - ▶ Analyzes Control flow graph (CFG)
 - ▶ Determines Cyclomatic Complexity (CC) value
 - ▶ Obtains linearly independent paths
 - ▶ Generates test cases for each path

Basis Path Testing

Control Flow Graph (CFG)



- Three basis paths
- I. 1, 2, 10
- II. 1, 2, 3, 4, 8, 2, 10
- III. 1, 2, 3, 6, 8, 2, 10

- Three test cases
- I. $n = 0$
- II. $n = 1$
- III. $n = 4$

- Real execution paths
- I. 1, 2, 10
- II. 1, 2, 3, 4, 8, 2, 10
- III. 1, 2, 3, 4, 8, 2, 3, 4, 8, 2, 3, 4, 8, 2, 3, 6, 8, 2, 10

What is Testing?

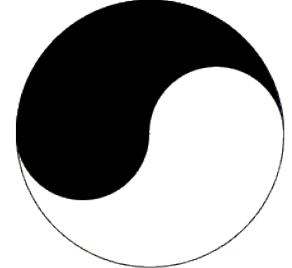
- ▶ ~~Testing = Debugging ?~~
- ▶ ~~To show that software works ?~~
- ▶ To show that software **doesn't** work ?
- ▶ To **reduce** the perceived risk of system not working to an **acceptable value**
- ▶ To result in low-risk software **without much testing effort**

Testing:

Software *testing* can be used to *show the presence* of bugs, but never to *show their absence*.

-- Edsger Dijkstra

Types Of Testing



▶ Black Box Testing

- ▶ What to analyze: Requirements and specifications
 - E.g. "*The income of an applicant must be between \$1500 per month and \$8000 per month*"
- ▶ Type of black box testing
 - **Equivalence class testing and boundary value testing**
 - State-based testing

▶ White Box Testing

- ▶ What to analyze: Implementation details, internal paths and structure
- ▶ Type of white box testing
 - **Control flow testing**
 - Data flow testing

Levels Of Testing

- ▶ Unit testing (we learn in this course)
 - ▶ Test an unit **individually**
- ▶ Integration testing
 - ▶ Test units in **combination**
- ▶ System testing
 - ▶ Test **everything** (functionality, usability, reliability, performance, portability, security ...)
- ▶ Acceptance testing (UAT User Acceptance Testing)
 - ▶ **Customer** accepts the software and give us their **money**

Ingredients Of Test Case

Ingredient	Description
Name	Distinct name of test case
Location	Path to test program and its inputs
Input	Input data or commands
Oracle	Expected test output
Log	Actual output produced by the test

Test Cases for Mortgage Application

Each row is a test case

Input				Oracle	Log
Income	Number of mortgage	Applicant Type	Property Type	Expected Result	Test Result
\$5000	2	Person	Cando	Approval	Approval
\$30000	3	Person	HDB	Approval	Approval
\$15000	4	Person	HDB	Approval	Approval
\$100	1	Person	HDB	Reject	Reject
\$1501	8	Person	Condo	Reject	Approval
\$19845	3	Company	Condo	Reject	Reject
\$79999	2	Person	Mobilehome	Reject	Reject

An Android Unit Test Case

```
package mortgageapplication.test; ← Location  
public class TestMortgageApplicationActity {  
    private MortgateProcessor processor;  
    public void testQualifiedApplication() {  
        int income = 5000; ← Name  
        int nummortage = 2;  
        AppcantType applicant = ApplicantType.Person; ← Log  
        PropertyType property = PropertyType.Cando;  
        ApplicationResult result = ← Log  
            processor.analyzeApplication( income,  
                nummortage, applicant, property);  
        assertEquals(result, ApplicationResult.Approval); ← Oracle  
    }  
    public void testLowIncome() {  
        int income = 100; ← Name  
        int nummortage = 1; ← Name  
        AppcantType applicant = ApplicantType.Person; ← Log  
        PropertyType property = PropertyType.HDB; ← Log  
        ApplicationResult results= ← Log  
            processor.analyzeApplication( income,  
                nummortage, applicant, property);  
        assertEquals(result, ApplicationResult.Reject); ← Oracle  
    }  
}
```

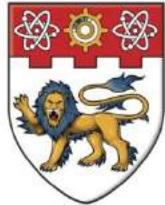
Google
“Android Testing Fundamentals” and
“Android Activity Testing Tutorial”

Order Of Test Case Execution

- ▶ Cascading test cases, e.g.,
 1. Create an order
 2. Read the order
 3. Update the order
 4. Delete the order
 5. Read the deleted order
 - ➔ Smaller and simpler test cases
 - ➔ One fails, the subsequent test may be invalid
- ▶ Independent test cases, e.g.,
 - ▶ Self-contained test cases for the above features
 - ➔ Parallel test execution
 - ➔ Larger and more complex test cases

Summary of Testing

- Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test.
 - Dynamic execution
 - To find bug, rather than to prove no bug
- Type of testing
 - Whitebox vs Blackbox
- Level of testing
 - Unit, integration, system and acceptance
- Test Case
 - 5 elements, and execution order



NANYANG
TECHNOLOGICAL
UNIVERSITY

CZ2006/CE2006 **Software Engineering**

Lecture 16-17 **Equivalence Class Testing**

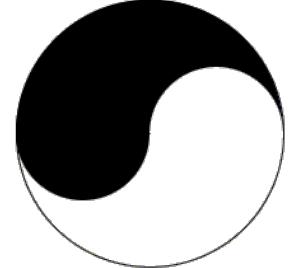
When I launch my app after several hours of development

Unexpected things happen!



A **software bug** is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.

Types Of Testing



▶ Black Box Testing

- ▶ What to analyze: Requirements and specifications
 - E.g. "*The income of an applicant must be between \$1500 per month and \$8000 per month*"
- ▶ Type of black box testing
 - **Equivalence class testing and boundary value testing**
 - State-based testing

▶ White Box Testing

- ▶ What to analyze: Implementation details, internal paths and structure
- ▶ Type of white box testing
 - **Control flow testing**
 - Data flow testing

Levels Of Testing

- ▶ Unit testing (we learn in this course)
 - ▶ Test an unit **individually**
- ▶ Integration testing
 - ▶ Test units in **combination**
- ▶ System testing
 - ▶ Test **everything** (functionality, usability, reliability, performance, portability, security ...)
- ▶ Acceptance testing (UAT User Acceptance Testing)
 - ▶ **Customer** accepts the software and give us their **money**

Ingredients Of Test Case

Ingredient	Description
Name	Distinct name of test case
Location	Path to test program and its inputs
Input	Input data or commands
Oracle	Expected test output
Log	Actual output produced by the test

Black Box Testing Process

► The tester

1. Analyzes **requirements** or **specifications**
 - ▶ equivalence classes and boundary values
2. Chooses **valid** inputs and **invalid** inputs
3. Determines expected outputs (oracle) for chosen inputs
4. Constructs tests with the chosen inputs (e.g., write JUnit program)

The most difficult steps

► The testing engine (e.g., JUnit)

1. Executes the tests
2. Compares actual outputs (log) with the expected outputs (oracle)
3. Determines whether the SUT (System Under Test) functions properly

Black Box Testing Pros and Cons

- ▶ **Applicability**
 - ▶ All levels of system development (Unit, Integration, System, Acceptance)
- ▶ **Cons**
 - ▶ Cannot know how much of the implementation have been tested
 - ▶ No notion of testing coverage like in white box testing
- ▶ **Pros**
 - ▶ Directs the tester to a very small subset of test inputs that can highly likely find the bugs

We do Black Box Testing Often...



<https://www.dpreview.com/galleries/6998361880/photos/1163297>

Assumption #1

- ▶ **Verifiable requirements** and API specification exists
 - ▶ Make software more testable from its inception!
- ▶ A requirement like this is **NOT** verifiable!
 - ▶ The company must not hire **too-young** or **too-old** people
 - ▶ The company can hire **juniors** on part-time basis
 - ▶ The company can hire **adult** as full-time employees

*Compare this with verifiable
requirements on Hire or Not slide!*

Assumption #2

- ▶ **Code must be testable!**
- ▶ Examples that make black-box testing less effective

```
if (age==0) hire=NO;  
if (age==1) hire=NO;  
...  
if (age==15) hire=NO;  
if (age==16) hire=PART;  
if (age==17) hire=PART;  
if (age==18) hire=FULL;  
...  
if (age==54) hire=FULL;  
if (age==55) hire=NO;  
if (age==56) hire=NO;  
...  
if (age==98) hire=NO;  
if (age==99) hire=NO;
```

```
if (age>=0 && age <= 15) hire=NO;  
if (age>=16 && age <= 18) hire=PART;  
if (age>=18 && age <= 34) hire=FULL;  
  
if (age==35 && name=="xingze")  
hire=CEO;  
if (age==35 && name!="xingzc")  
hire=FULL;  
  
if (age>=36 && age<=54) hire=FULL;  
if (age>=55 && age<=99) hire=NO;
```

Motivating Example: Hire or Not?

- ▶ Test a module implementing the following hiring requirements
 - ▶ 0 – 15, 55 – 99 Do not hire
 - ▶ 16 – 17 Part-time
 - ▶ 18 – 54 Full-time

~~Test ..., -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100, 101, 102, ...?~~

Test only one value for each range, one invalid value below 0, and one invalid value above 99

Equivalence Class Testing – Partition Input Values into Equivalence Classes

- ▶ A set of input values forms an **equivalence class (EC)** if
 - ▶ They all are supposed to produce the **same output**
 - ▶ If one value catches a bug, the others probably will too
 - ▶ If one value does not catch a bug, the others probably will not either

A metaphor



<http://web.1.c2.audiovideoweb.com/1c2web3536/lowcalrice.jpg>

Equivalence Class Testing Process

► The tester

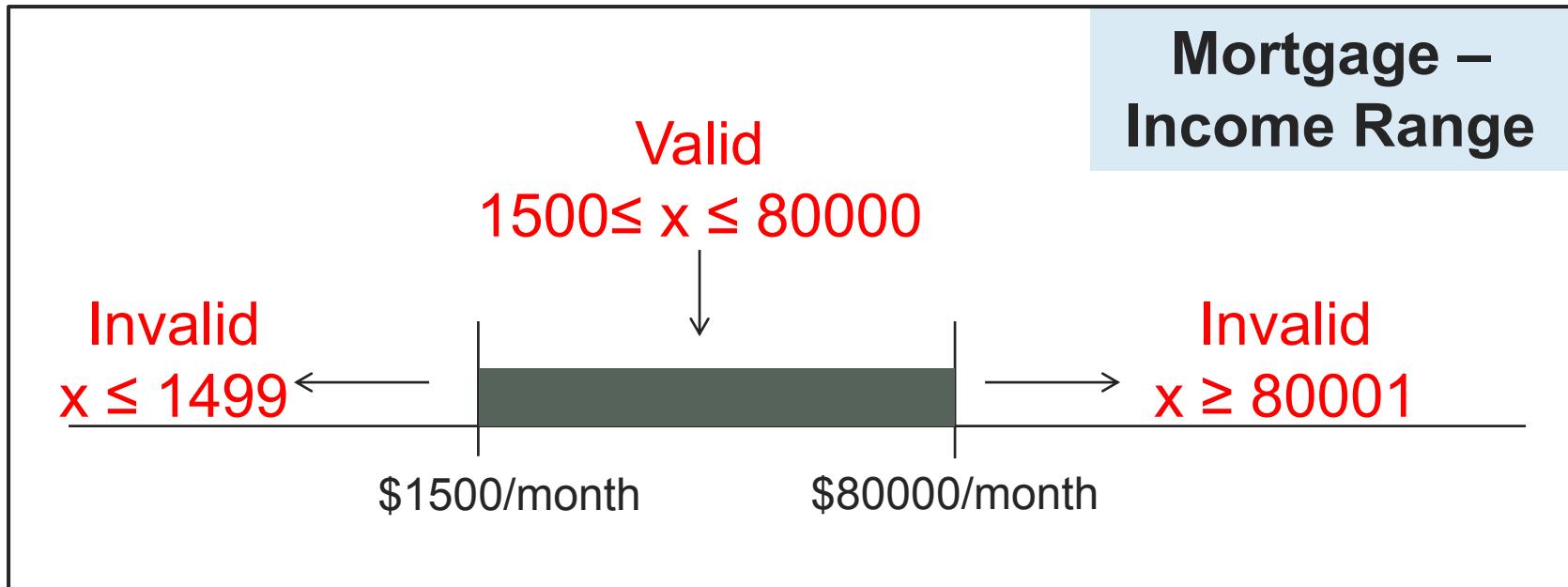
1. Identify the **valid and invalid equivalence classes** that partition the input values
2. Choose **at least one input value** for each equivalence class of each input parameter
3. Determines expected outputs (oracle) for chosen inputs
4. Constructs tests with the chosen inputs (e.g., write JUnit program)

► The testing engine (e.g., JUnit)

1. Executes the tests
2. Compares actual outputs (log) with the expected outputs (oracle)
3. Determines whether the SUT (System Under Test) functions properly

Range of Values

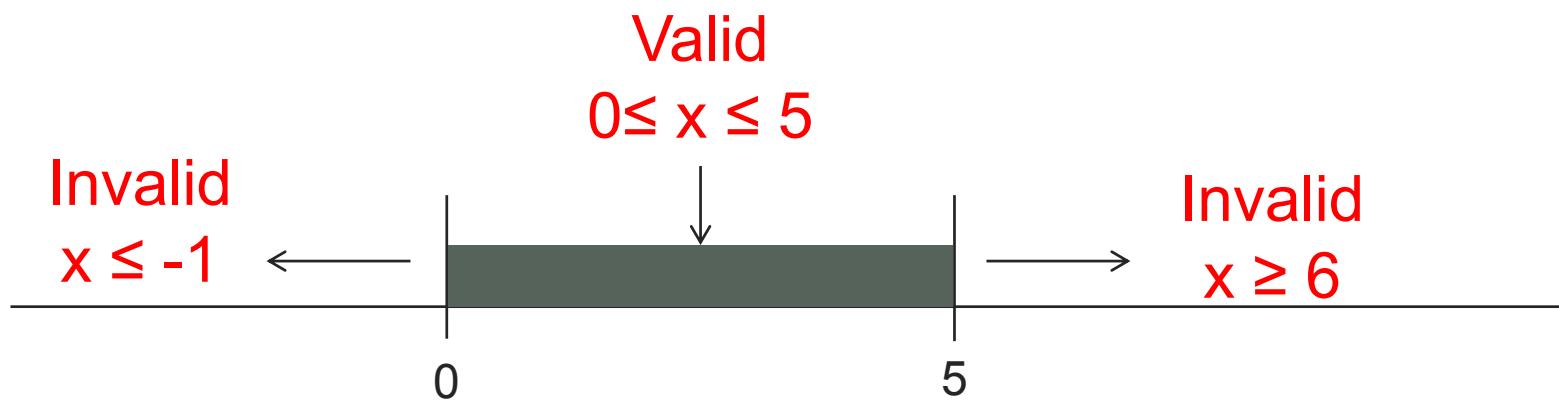
- ▶ **ONE** equivalence class of **VALID** values
- ▶ **TWO** equivalence classes of **INVALID** values



Requirement: The system approves mortgage application if income is between \$1500/month and \$80000/month

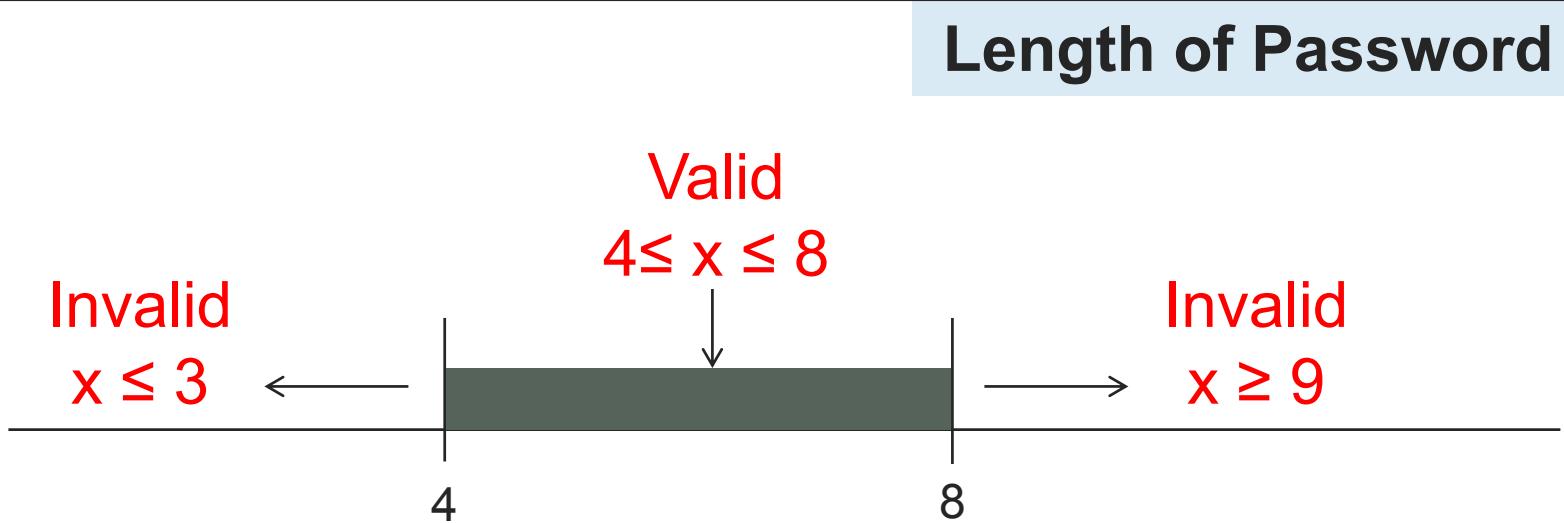
Range of Values – Exercise

Mortgage –
Number of Mortgages



Requirement: The system approves application if #mortgage per applicant is between 0 and 5

Range of Values – Exercise



Requirement: The password must contain 4-8 chars and digits

Range of Values – Exercise

Hiring Rules

Valid equivalence classes

$0 \leq \text{age} \leq 15$ (do not hire)

$16 \leq \text{age} \leq 17$ (hire part-time)

$18 \leq \text{age} \leq 54$ (hire full-time)

$55 \leq \text{age} \leq 99$ (do not hire)

Invalid equivalence classes

$\text{age} \leq -1$

$\text{age} \geq 100$

Requirements: $0 - 15, 55 - 99$

$16 - 17$

$18 - 54$

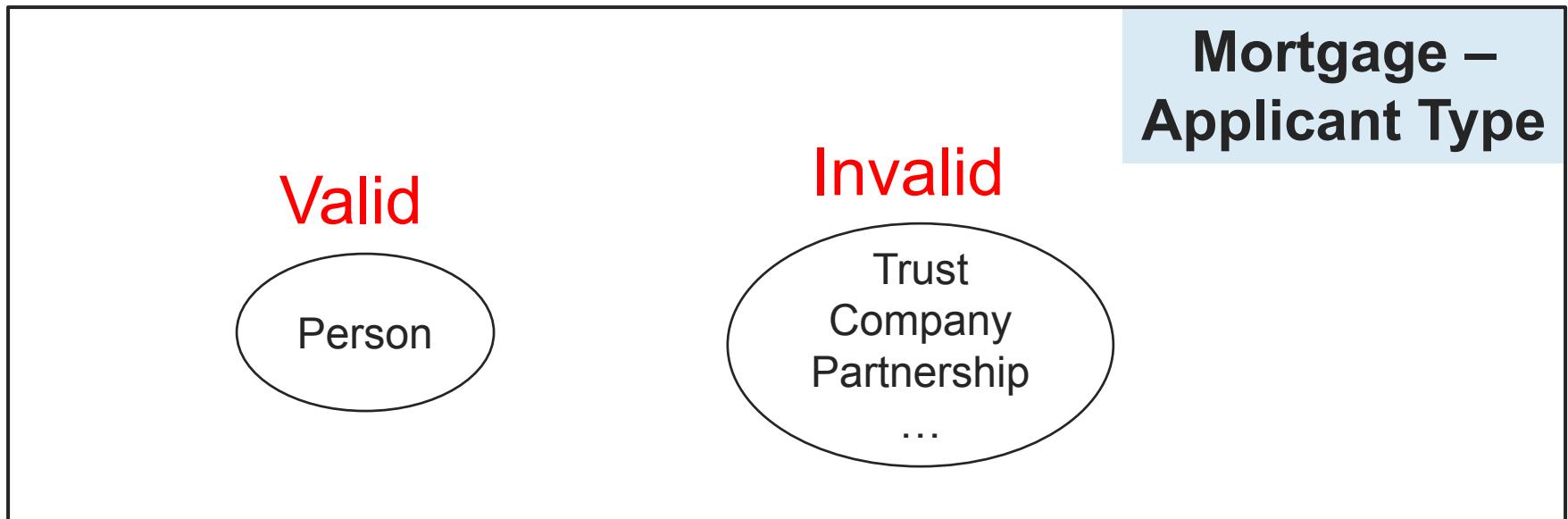
Do not hire

Part-time

Full-time

Discrete Values

- ▶ **ONE** equivalence class of **VALID** values
- ▶ **ONE** equivalence class of **INVALID** values



Requirement: The system approves only personal mortgage applications

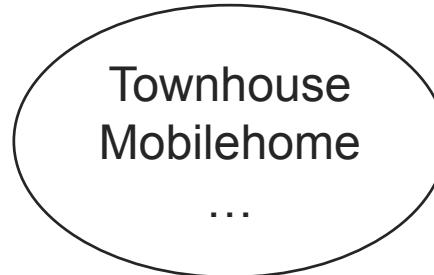
Discrete Values – More Examples

Mortgage –
Property Type

Valid



Invalid



Requirement: The system approves only HDB and Condo
mortgage applications

Discrete Values – More Examples

Character of Password

Valid

a b ... z
A B ... Z
0 1 2 ... 9

Invalid

+ - () ...

Requirement: The password must be characters a-z, A-Z, and 0-9

Design Test Cases Using ECs

1. Test **VALID** inputs of several parameters

- ▶ Test valid inputs of several parameters at the same time (i.e. per test case)
- ▶ Choose one valid input for each input parameter from an equivalence class

2. Test **INVALID** inputs of several parameters

- ▶ Test **ONLY ONE INVALID** input from an equivalence class of one input parameter at the same time (i.e. per test case)
- ▶ Choose valid inputs from an equivalence class of all other input parameters

Test Valid Inputs Using ECs

- ▶ You may test **valid** input values of **several** input parameters at the same time
- ▶ You may create test cases for **all the values** in an valid equivalence class if the set of valid values in the equivalence class is **small**
 - e.g. The number of mortgages (0, 1, 2, 3, 4, 5)
 - Property type (HDB, Condo)

but not for income range, or characters of password

Test Valid Inputs – Example

Each row is a test case.
For each test case, choose one **valid** input value from a valid equivalence class of each parameter

As each parameter has only one valid EC, this single test case satisfies **EC testing principle**

Income	Number of mortgage	Applicant Type	Property Type	Expected Result
\$1500	0	Person	Condo	Approval
\$2000	1	Person	Condo	Approval
\$5000	2	Person	Condo	Approval
\$12345	3	Person	HDB	Approval
\$76543	4	Person	HDB	Approval
\$80000	5	Person	HDB	Approval

Try to cover a few more valid income value when enumerating #mortgage and PropertyType values

As #mortgage and PropertyType have **only 6 (0-5)** and **2 (Condo, HDB)** valid values, we create several more test cases to cover **all** these values

Should We Test Invalid Inputs?

- ▶ Testing by Contract
 - ▶ Create test cases **only** for **valid** inputs (i.e., the input values that satisfy the **pre-conditions**)
- ▶ Defensive Testing
 - ▶ Create test cases for **BOTH VALID** and **INVALID** inputs
 - ▶ Do not assume that users will use your code the way it is supposed to be used
 - ▶ Test the module to see **whether it behaves as expected for invalid inputs**, i.e. do not cause unforeseen errors or abnormal system behavior (sudden program termination)

Which one does the testers prefer?

- ▶ If it does not work, who will get the blame? – the testers
- ▶ Testers prefer defensive testing

Test Invalid Inputs Using ECs

Why?

- You must test **ONLY ONE INVALID** input value from an equivalence class of one input parameter per test case i.e. each test case should have no more than one invalid input

Income	Number of mortgage	Applicant Type	Property Type	Expected Result
\$100	8	Company	Mobilehome	Reject

Invalid input Invalid input Invalid input Invalid input

If we test only one invalid input for each test case, then we can learn the error handling or behavior for each type of invalid input

Test Invalid Inputs Using ECs

- ▶ You may create test cases for **all the values** in an invalid equivalence class if the set of invalid values in the equivalence class is **small**

Test Invalid Inputs – Example

Each row is a test case.
Test **ONLY ONE INVALID**
input value **per test case**

Income	Number of mortgage	Applicant Type	Property Type	Expected Result
\$100	1	Person	Condo	Reject
\$81000	2	Person	HDB	Reject
\$5000	-1	Person	Condo	Reject
\$10000	8	Person	HDB	Reject
\$12345	3	Company	HDB	Reject
\$76543	4	Person	Mobilehome	Reject

Choose valid input values
for all other parameters

Quick Summary

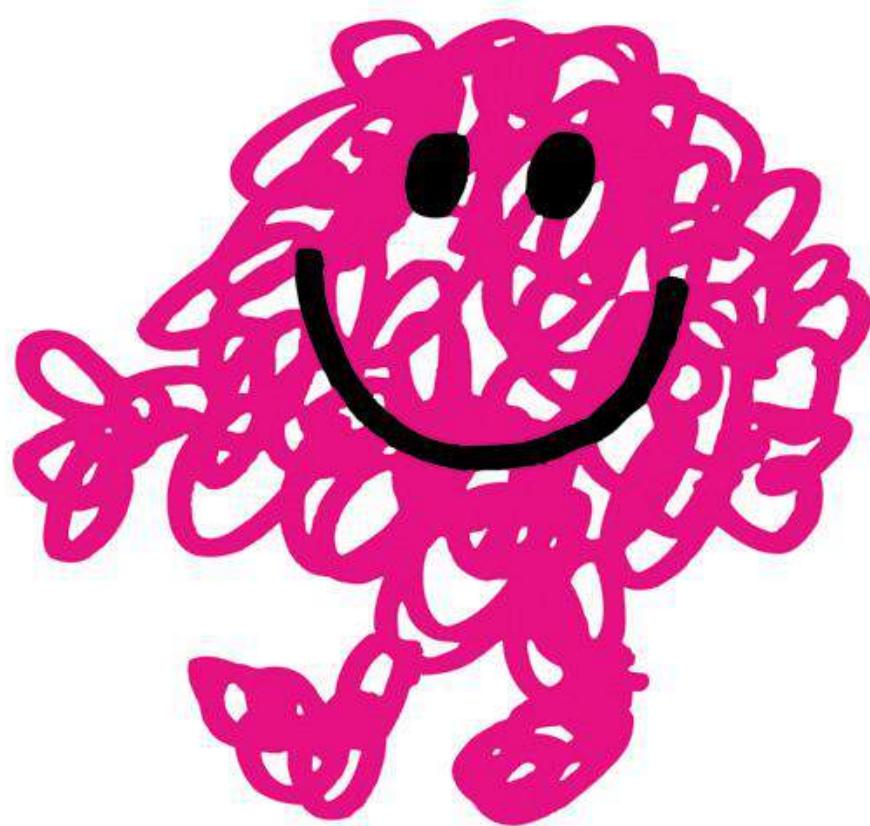
A key aspect of black box testing is identifying Equivalent Classes (EC).

Identifying ECs helps us test effectively with a minimal set of inputs.

For equivalence classes (ECs), range of values –

- **ONE VALID EC of VALID values**
- **TWO INVALID ECs of INVALID values**

But, the Real World is Messy



http://mrmen.wikia.com/wiki/File:MR_MESSY_4A.jpg

Real World Requirements are Messy Too 😊

Requirement ID	KBO_Req_01
Requirement Description	<p>Administrator shall manually access the transaction information for a particular account on a periodic basis from the existing Master Ledger system. (Automation of this access mechanism is outside the scope of KBO's current release and may become a Requirement for a future release.) Administrator shall record the transaction information for a particular account accessed from the ML system, in the KBO system. Only successful transactions need to be available at KBO for viewing by users within 48 hours after they have been initiated by the user. Users will be notified of failed transactions through procedures outside the scope of KBO's current release.</p>

Functional part	Non-functional part
Recording transaction details in KBO, by administrator	<ul style="list-style-type: none">Ensuring transactions are recorded in a format that makes it reasonably easy to be understood by users.Ensuring transactions are correctly recorded for the appropriate user accounts.Ensuring successful transactions are recorded at KBO within 48 hours of their initiation.Ensuring large volumes of transaction for a particular account within a particular period of time can be handled.

A Key Skill is to Derive Verifiable Requirements for Effective Black Box Testing

Requirement ID	KBO_Req_01	Functional part	Non-functional part
Requirement Description	<p>Administrator shall manually access the transaction information for a particular account on a periodic basis from the existing Master Ledger system. (Automation of this access mechanism is outside the scope of KBO's current release and may become a Requirement for a future release.) Administrator shall record the transaction information for a particular account accessed from the ML system, in the KBO system. Only successful transactions need to be available at KBO for viewing by users within 48 hours after they have been initiated by the user. Users will be notified of failed transactions through procedures outside the scope of KBO's current release.</p>	Recording transaction details in KBO, by administrator	<ul style="list-style-type: none">Ensuring transactions are recorded in a format that makes it reasonably easy to be understood by users.Ensuring transactions are correctly recorded for the appropriate user accounts.Ensuring successful transactions are recorded at KBO within 48 hours of their initiation.Ensuring large volumes of transaction for a particular account within a particular period of time can be handled.

Assumption #1

- ▶ **Verifiable requirements** and API specification exists
 - ▶ Make software more testable from its inception!

CZ2006/CE2006 Software Engineering

Lecture 18: Boundary Value Testing

Liu Yang

Equivalence Class Testing – Partition Input Values into Equivalence Classes

- ▶ A set of input values forms an **equivalence class (EC)** if
 - ▶ They all are supposed to produce the **same output**
 - ▶ If one value catches a bug, the others probably will too
 - ▶ If one value does not catch a bug, the others probably will not either



Equivalence Class Testing Process

► The tester

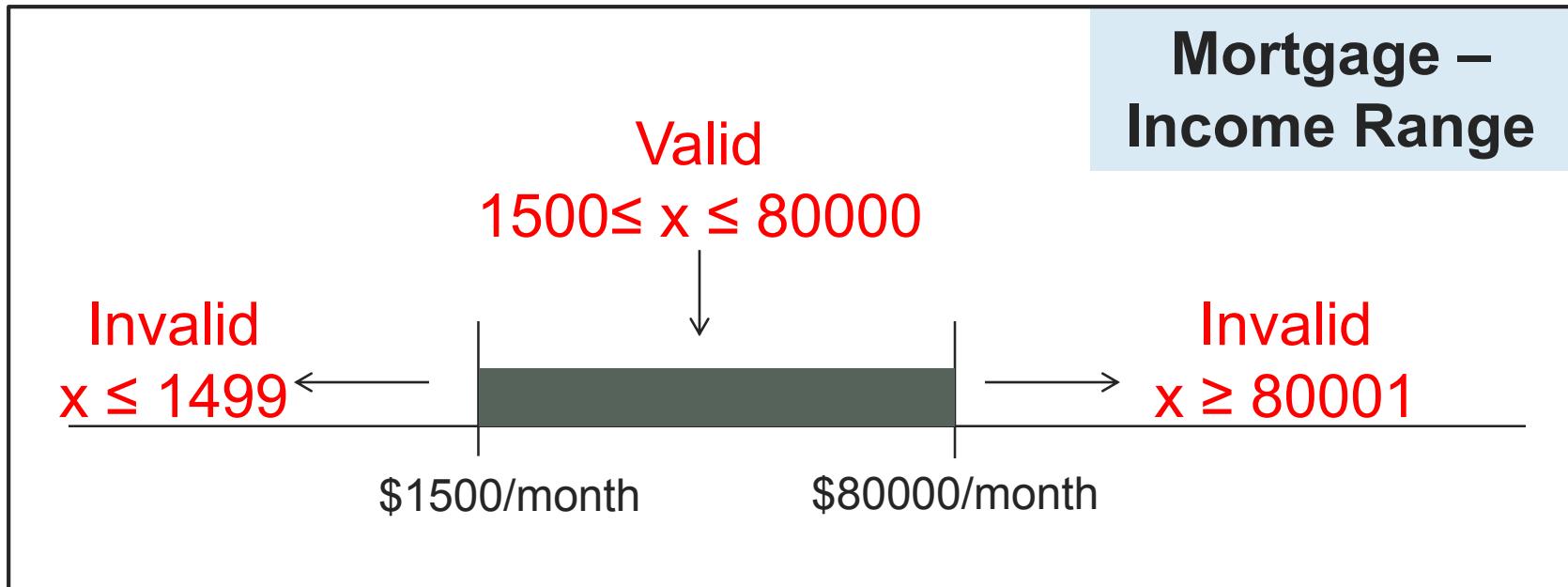
1. Identify the **valid and invalid equivalence classes** that partition the input values
2. Choose **at least one input value** for each equivalence class of each input parameter
3. Determines expected outputs (oracle) for chosen inputs
4. Constructs tests with the chosen inputs (e.g., write JUnit program)

► The testing engine (e.g., JUnit)

1. Executes the tests
2. Compares actual outputs (log) with the expected outputs (oracle)
3. Determines whether the SUT (System Under Test) functions properly

Range of Values

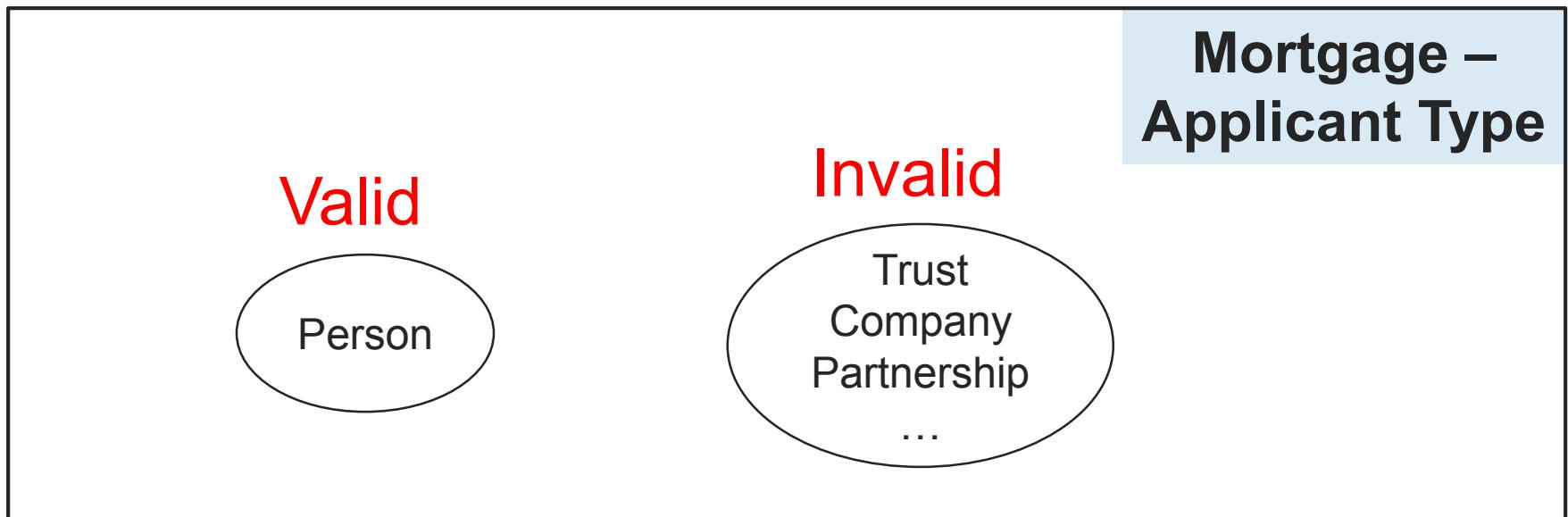
- ▶ **ONE** equivalence class of **VALID** values
- ▶ **TWO** equivalence classes of **INVALID** values



Requirement: The system approves mortgage application if income is between \$1500/month and \$80000/month

Discrete Values

- ▶ **ONE** equivalence class of **VALID** values
- ▶ **ONE** equivalence class of **INVALID** values



Requirement: The system approves only personal mortgage applications

Design Test Cases Using ECs

1. Test **VALID** inputs of several parameters

- ▶ Test valid inputs of several parameters at the same time (i.e. per test case)
- ▶ Choose one valid input for each input parameter from an equivalence class

2. Test **INVALID** inputs of several parameters

- ▶ Test **ONLY ONE INVALID** input from an equivalence class of one input parameter at the same time (i.e. per test case)
- ▶ Choose valid inputs from an equivalence class of all other input parameters

Test Valid Inputs Using ECs

- ▶ You may test **valid** input values of **several** input parameters at the same time
- ▶ You may create test cases for **all the values** in an valid equivalence class if the set of valid values in the equivalence class is **small**
 - e.g. The number of mortgages (0, 1, 2, 3, 4, 5)
 - Property type (HDB, Condo)

but not for income range, or characters of password

Test Valid Inputs – Example

Each row is a test case.
For each test case, choose one **valid** input value from a valid equivalence class of each parameter

As each parameter has only one valid EC, this single test case satisfies **EC testing principle**

Income	Number of mortgage	Applicant Type	Property Type	Expected Result
\$1500	0	Person	Condo	Approval
\$2000	1	Person	Condo	Approval
\$5000	2	Person	Condo	Approval
\$12345	3	Person	HDB	Approval
\$76543	4	Person	HDB	Approval
\$80000	5	Person	HDB	Approval

Try to cover a few more valid income value when enumerating #mortgage and PropertyType values

As #mortgage and PropertyType have **only 6 (0-5)** and **2 (Condo, HDB)** valid values, we create several more test cases to cover **all** these values

Test Invalid Inputs Using ECs

Why?

- You must test **ONLY ONE INVALID** input value from an equivalence class of one input parameter per test case i.e. each test case should have no more than one invalid input

Income	Number of mortgage	Applicant Type	Property Type	Expected Result
\$100	8	Company	Mobilehome	Reject

Invalid input Invalid input Invalid input Invalid input

If we test only one invalid input for each test case, then we can learn the error handling or behavior for each type of invalid input

Test Invalid Inputs – Example

Each row is a test case.
Test **ONLY ONE INVALID**
input value **per test case**

Income	Number of mortgage	Applicant Type	Property Type	Expected Result
\$100	1	Person	Condo	Reject
\$81000	2	Person	HDB	Reject
\$5000	-1	Person	Condo	Reject
\$10000	8	Person	HDB	Reject
\$12345	3	Company	HDB	Reject
\$76543	4	Person	Mobilehome	Reject

Choose valid input values
for all other parameters

Boundary Value Testing (BVT) – Which Values to Test?

- ▶ BVT is **only applicable** to **range of values**
 - ▶ Focus on **boundaries** of the equivalence classes of range of values
 - ▶ Select a **minimal** set of **valid** and **invalid inputs** (values **on-the-boundary**, **just-below**, or **just-above** boundaries) to test
- ▶ What is the boundary values of parameter Property type?
 - ▶ Do **discrete values**, such as property type, have boundary values? **NO**

Boundary Value Testing (BVT) Process

► The tester

1. Identify the **lower and upper boundaries** of equivalence classes of the range-of-values parameters
2. Choose three input values for each boundary
 - ▶ One value **on-the-boundary**, one value **just-below**, and one value **just-above**
 - ▶ Just-below and just-above value depends on the value's unit
 - ▶ No need to duplicate the test cases if the just-below or just-above values fall into other ECs (including current EC)
3. Determines expected outputs (oracle) for chosen inputs
4. Constructs tests with the chosen inputs (e.g., write JUnit program)

Determine Boundary Values (BVs) – Example

- ▶ Income range valid equivalence class ($1500 \leq x \leq 80000$)
 - Lower boundary 1500: ~~1499, 1500, 1501~~
 - Upper boundary 80000: ~~79999, 80000, 80001~~
 - Valid boundary values: {1500, 80000}
 - Invalid boundary values: {1499, 80001}
- ▶ Number of mortgage valid equivalence class ($0 \leq x \leq 5$)
 - Lower boundary 0: ~~-1, 0, 1~~
 - Upper boundary 5: ~~4, 5, 6~~
 - Valid boundary values: {0, 5}
 - Invalid boundary values: {-1, 6}
- ▶ Length of password valid equivalence class ($4 \leq x \leq 8$)
 - Lower boundary 4: ~~3, 4, 5~~
 - Upper boundary 8: ~~7, 8, 9~~
 - Valid boundary values: {4, 8}
 - Invalid boundary values: {3, 9}

Determine Boundary Values (BVs) – Example

- ▶ Hiring rules ($0 \leq \text{age} \leq 15$) (Do not hire)
 - Lower boundary 0: ~~-1~~, ~~0~~, ~~1~~
 - Upper boundary 15: ~~14~~, ~~15~~, ~~16~~
- ▶ Hiring rules ($16 \leq \text{age} \leq 17$) (Part time)
 - Lower boundary 16: ~~15~~, ~~16~~, ~~17~~
 - Upper boundary 17: ~~16~~, ~~17~~, ~~18~~
- ▶ Hiring rules ($18 \leq \text{age} \leq 54$) (Full time)
 - Lower boundary 18: ~~17~~, ~~18~~, ~~19~~
 - Upper boundary 54: ~~53~~, ~~54~~, ~~55~~
- ▶ Hiring rules ($55 \leq \text{age} \leq 99$) (Do not hire)
 - Lower boundary 55: ~~54~~, ~~55~~, ~~56~~
 - Upper boundary 99: ~~98~~, ~~99~~, ~~100~~

Valid
boundary
values:

{0, 15,
16, 17,
18, 54,
55, 00}

Invalid
boundary
values:
{-1, 100}



NANYANG
TECHNOLOGICAL
UNIVERSITY

Determine Boundary Values (BVs) – Example

- ▶ Add1Divide30000 ($-32768 \leq \text{input} \leq -30001$) (return -1)
 - Lower boundary -32768: ~~-32769~~, ~~-32768~~, ~~-32767~~
 - Upper boundary -30001: ~~-30002~~, ~~-30001~~, ~~-30000~~
- ▶ Add1Divide30000 ($-30000 \leq \text{input} \leq 29998$) (return 0)
 - Lower boundary -30000: ~~-30001~~, ~~-30000~~, ~~-29999~~
 - Upper boundary 29998: ~~29997~~, ~~29998~~, ~~29999~~
- ▶ Add1Divide30000 ($29999 \leq \text{input} \leq 32767$) (return 1)
 - Lower boundary 29999: ~~29998~~, ~~29999~~, ~~30000~~
 - Upper boundary 32766: ~~32766~~, ~~32767~~, ~~32768~~

Design Test Cases Using BVs

1. Test **VALID** inputs of several parameters
 - ▶ Test valid inputs of several parameters at the same time
 - ▶ Choose one valid input from an equivalence class of each input parameter
 2. Test **INVALID** inputs of several parameters
 - ▶ Test **ONLY ONE INVALID** input from an equivalence class of one input parameter **per test case**
 - ▶ Choose **VALID** inputs from an equivalence class of **all other** input parameters
- ▶ **Use the boundary values of the equivalence classes of range-of-values parameters**
- ▶ Value **on-the-boundary** as **VALID** input
 - ▶ **Just-below** or **just-above** values (if not in other ECs, including the current EC) as **INVALID** inputs

Black Box Testing – ECs and BVs

	Range of Values	Discrete Values
Equivalence Classes (ECs)	ONE Valid EC TWO Invalid ECs	ONE valid EC ONE Invalid EC

Range of values	Lower Boundary	Upper Boundary
Boundary Values (BVs) <i>(for each EC)</i>	ONE value <i>on-the-boundary</i> ONE value <i>just-below</i> ONE value <i>just-above</i>	ONE value <i>on-the-boundary</i> ONE value <i>just-below</i> ONE value <i>just-above</i>

Discrete Values Boundary Values (BVs) <i>(for each EC)</i>	NO Boundary Values
---	---------------------------

Black Box Testing – ECs and BVs

For test cases (use boundary values) –

- Test combination of **VALID inputs** (i.e. value **on-the-boundary**), one from each EC
- Test **ONE INVALID** input from an EC per test case, all others are VALID inputs => ***One test case can contain AT MOST ONE invalid input***
- **Test combination of boundary values**
Example: (two parameters, A and B of range of values)
 1. Valid A + Valid B → *combination of VALID inputs*
 2. Valid A + Invalid B → } *combination of boundary values*
 3. Invalid A + Valid B → }
- Note: For VALID EC: **On-the-boundary**: valid
Just-below and **just-above**: invalid
- Remove **just-below** or **just-above** values from the boundary values of each EC (no need to duplicate the test cases) if the **just-below** or **just-above** values fall into other ECs (including current EC)

CZ2006/CE2006 Software Engineering

Lecture 19: Control Flow Testing

Liu Yang

Black Box Testing Pros and Cons

- ▶ **Applicability**
 - ▶ All levels of system development (Unit, Integration, System, Acceptance)
- ▶ **Cons**
 - ▶ Cannot know how much of the implementation have been tested
 - ▶ No notion of testing coverage like in white box testing
- ▶ **Pros**
 - ▶ Directs the tester to a very small subset of test inputs that can highly likely find the bugs

White Box Testing Process

► The tester

1. Analyzes SUTs (System Under Test) implementation
 - ▶ Control Flow Graph or Data Flow Graph
2. Identify execution paths through the SUT
3. Chooses inputs to cause the SUT to execute selected paths
4. Determines expected outputs (oracle) for chosen inputs
5. Constructs tests with the chosen inputs (e.g., write JUnit program)

The most difficult steps

► The testing engine (e.g., JUnit)

1. Executes the tests
2. Compares actual outputs (log) with the expected outputs (oracle)
3. Determines whether the SUT (System Under Test) functions properly

White Box Testing Pros and Cons

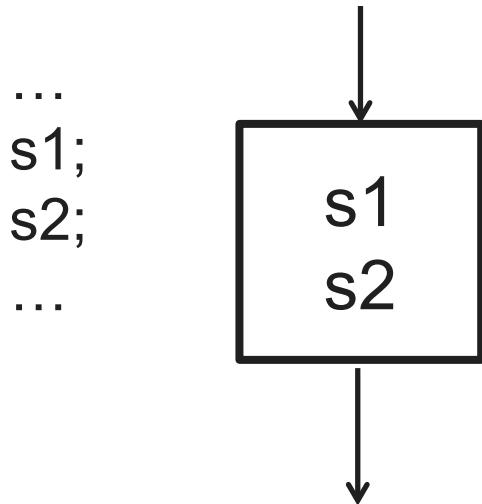
- ▶ **Applicability**
 - ▶ All levels of system development (Unit, Integration, System, Acceptance)
- ▶ **Pros**
 - ▶ Ensure that every path through the SUT has been identified and tested
- ▶ **Cons**
 - ▶ Testing all execution paths is generally infeasible
 - ▶ e.g., loop, #decision points
 - ▶ May not detect data and arithmetic bugs
 - ▶ e.g., $a = a - 1$ // should be $a = a + 1$; or
 - ▶ a/b // b cannot be zero
 - ▶ Never find paths that are not implemented
 - ▶ e.g., paths in requirements are missing in implementation

Control Flow Graph (CFG)

- ▶ A directed graph, consists of nodes and edges, where:
 - ▶ Each node represents a process block or a decision point
 - ▶ Each edge represents control flow (i.e., what happens after what)

Process Block

Process Block

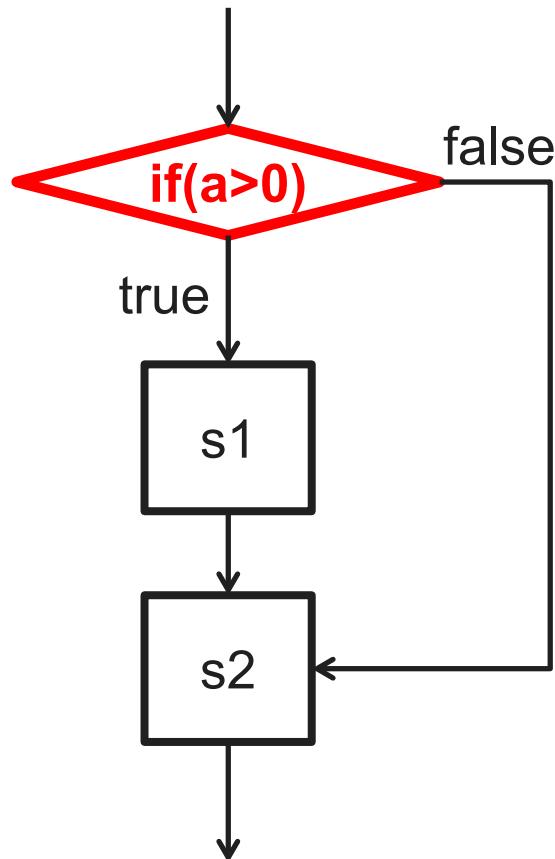


- ▶ A sequence of statements **execute sequentially**
- ▶ Does not contain if/while/for/switch/goto statements
- ▶ Contains any number of sequentially executed statements in the process block
- ▶ One control flow edge into the process block
- ▶ One control flow edge out of the process block

Binary Decision Point – if statement

Binary Decision Point

```
...  
if(a>0) {  
    s1;  
}  
s2;  
...
```

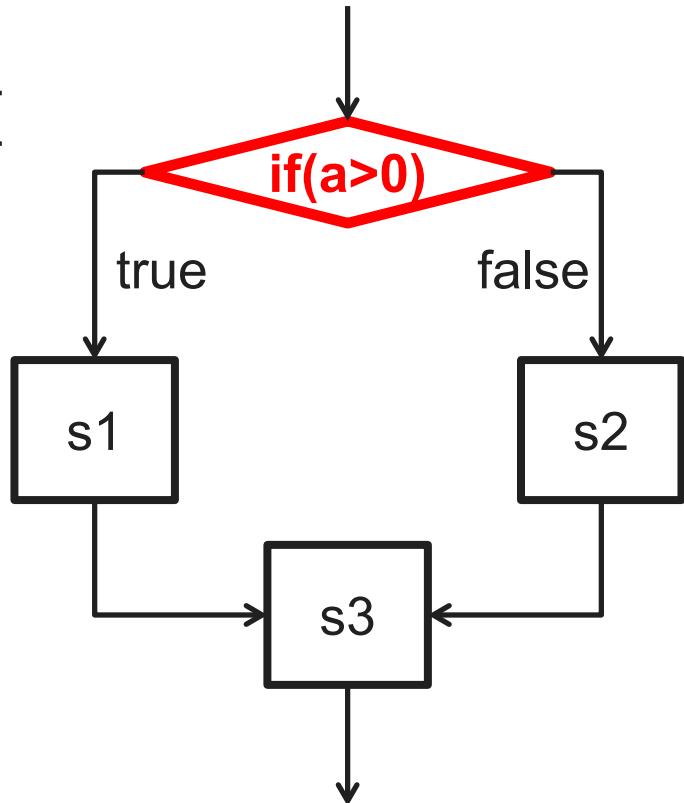


- ▶ Control flow can change at the decision point
- ▶ One control flow edge into the decision point
- ▶ **Two** control flow edges out of the decision point
 - ▶ True branch
 - ▶ False branch

Binary Decision Point – if-else statement

Binary Decision Point

```
...  
if(a>0) {  
    s1;  
} else {  
    s2;  
}  
s3;  
...
```

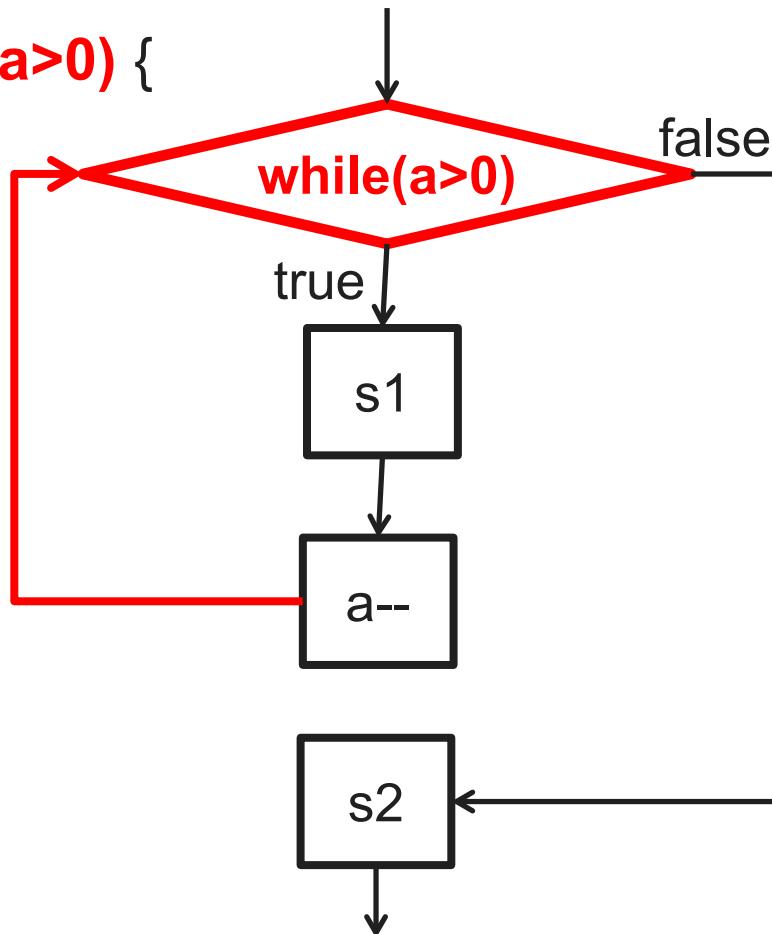


- ▶ Control flow can change at the decision point
- ▶ One control flow edge into the decision point
- ▶ **Two** control flow edges out of the decision point
 - ▶ True branch
 - ▶ False branch

Binary Decision Point – while statement

Binary Decision Point

```
...  
while(a>0) {  
    s1;  
    a--;  
}  
s2;  
...
```



- ▶ Control flow can change at the decision point
- ▶ One control flow edge into the decision point
- ▶ **Two** control flow edges out of the decision point
 - ▶ True branch
 - ▶ False branch
- ▶ **Loop back** to the decision point from the last statement

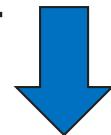
Binary Decision Point – for statement

Binary Decision Point

```
...  
for(a=100;a>0;a--) {
```

```
    s1;  
}
```

```
s2;  
...
```



transform
into

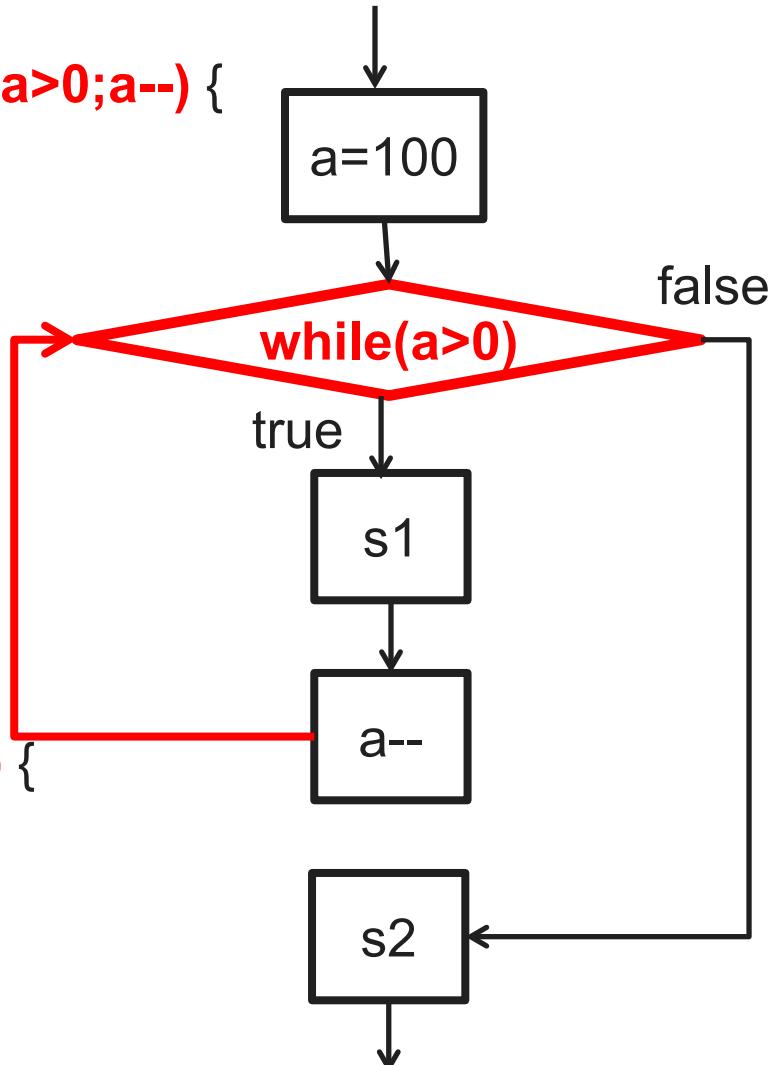
```
...
```

```
a=100;
```

```
while(a>0) {
```

```
    s1;  
    a--;  
}
```

```
s2;  
...
```

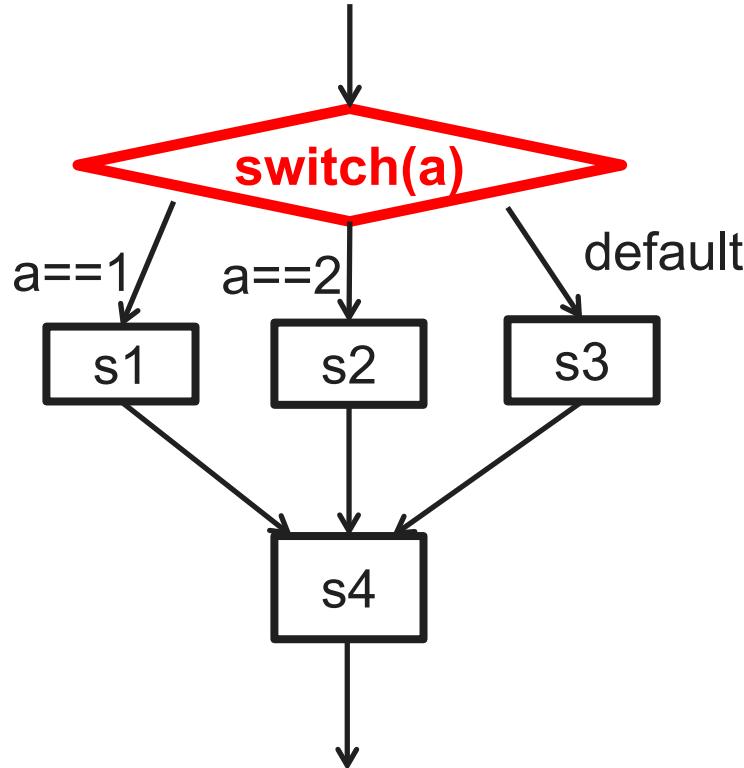


- ▶ Control flow can change at the decision point
- ▶ One control flow edge into the decision point
- ▶ Two control flow edges out of the decision point
 - ▶ True branch
 - ▶ False branch
- ▶ Loop back to the decision point from the last statement

N-ary Decision Point – switch statement

N-ary Decision Point

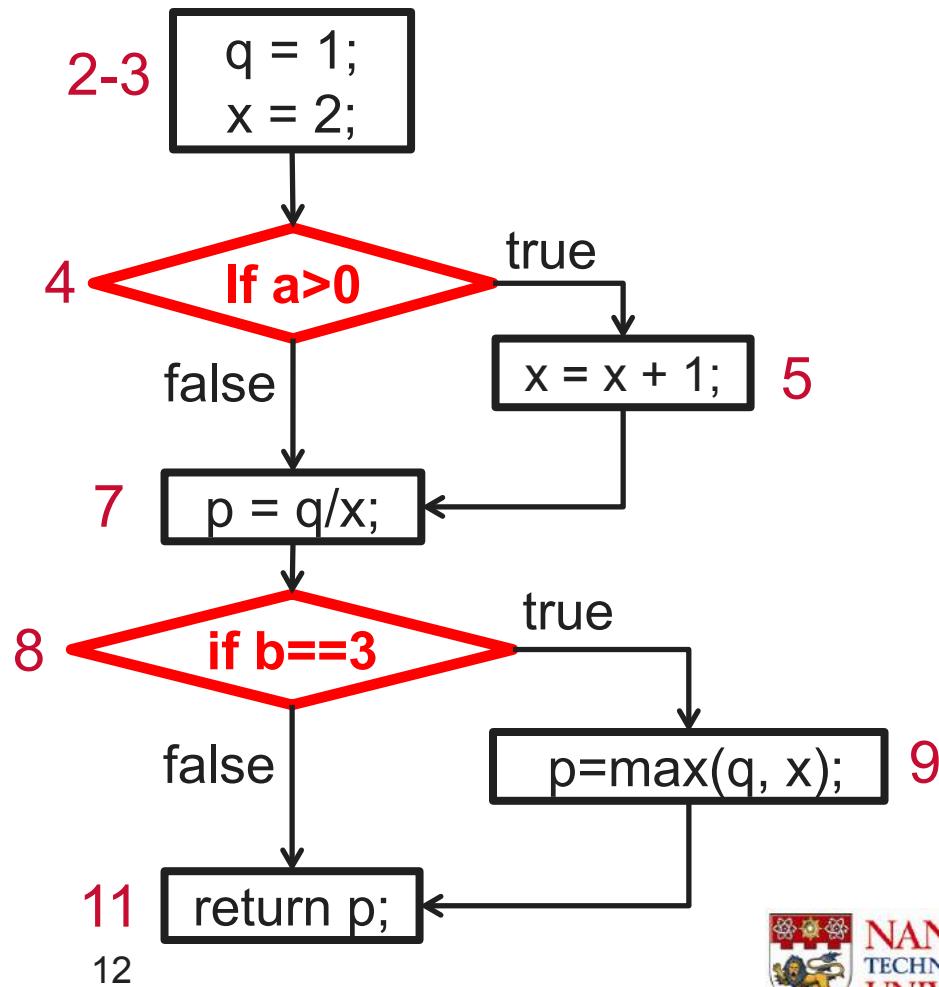
```
...  
switch(a) {  
    case 1:  
        s1;  
        break;  
    case 2:  
        s2;  
        break;  
    default:  
        s3;  
}  
s4;  
...
```



- ▶ Control flow can change at the decision point
- ▶ One control flow edge into the decision point
- ▶ **N** control flow edges out of the decision point

Control Flow Graph - Example

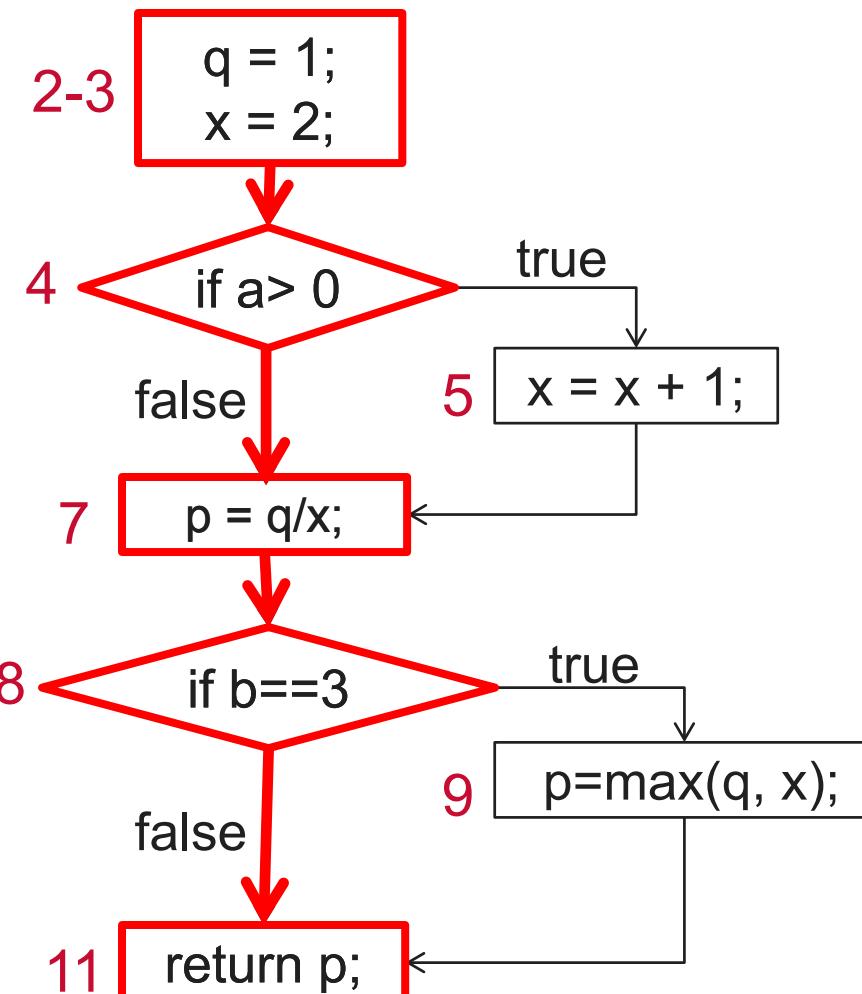
```
int computeP(int a, int b) {  
    1     int q, x, p;  
    2     q = 1;  
    3     x = 2;  
    4     if(a > 0) {  
    5         x = x + 1;  
    6     }  
    7     p = q/x;  
    8     if(b == 3) {  
    9         p = max(q, x);  
   10    }  
   11    return p;  
   12}
```



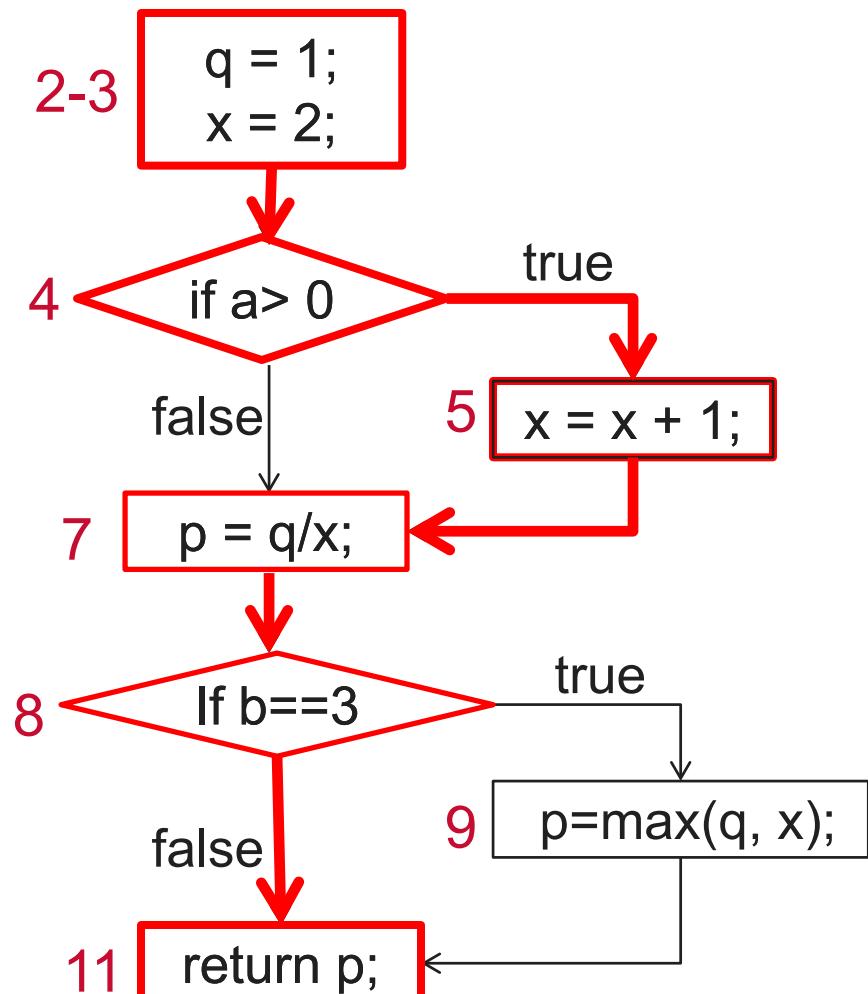
Execution Path Through the CFG

- ▶ A sequence of process blocks and decision points through the CFG of the program

Execution Path – computeP(int,int)



Path#1: **2-3, 4, 7, 8, 11**

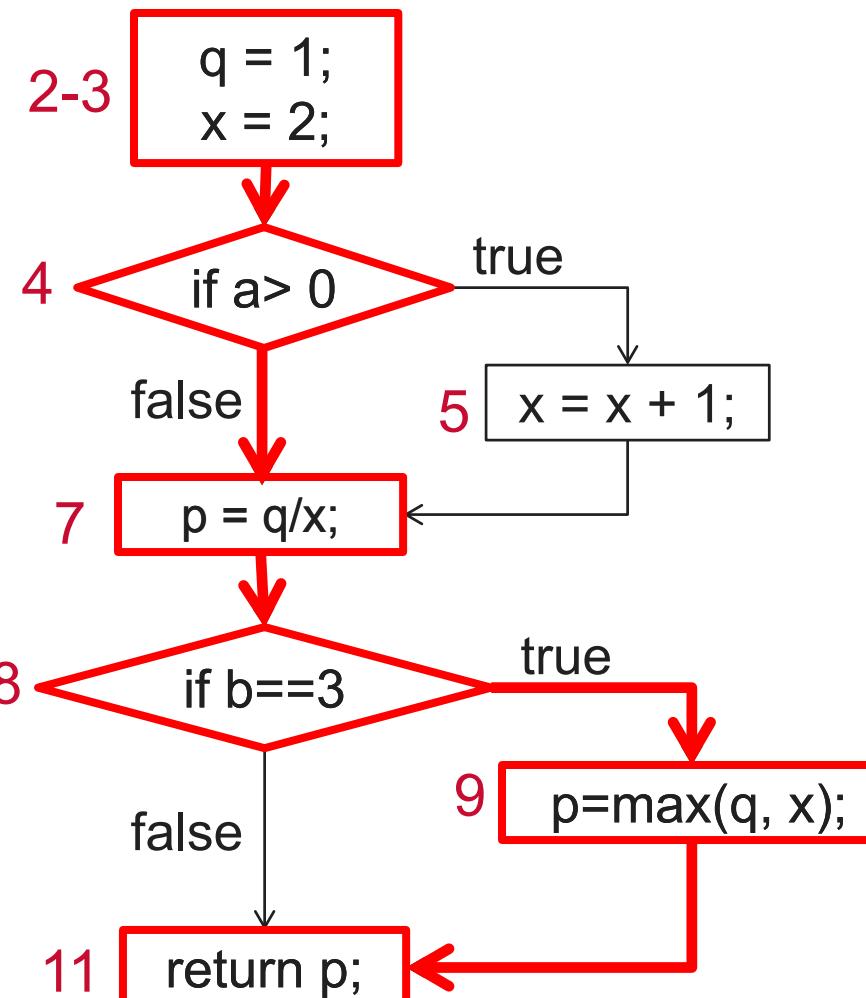


Path#2: **2-3, 4, 5, 7, 8, 11**

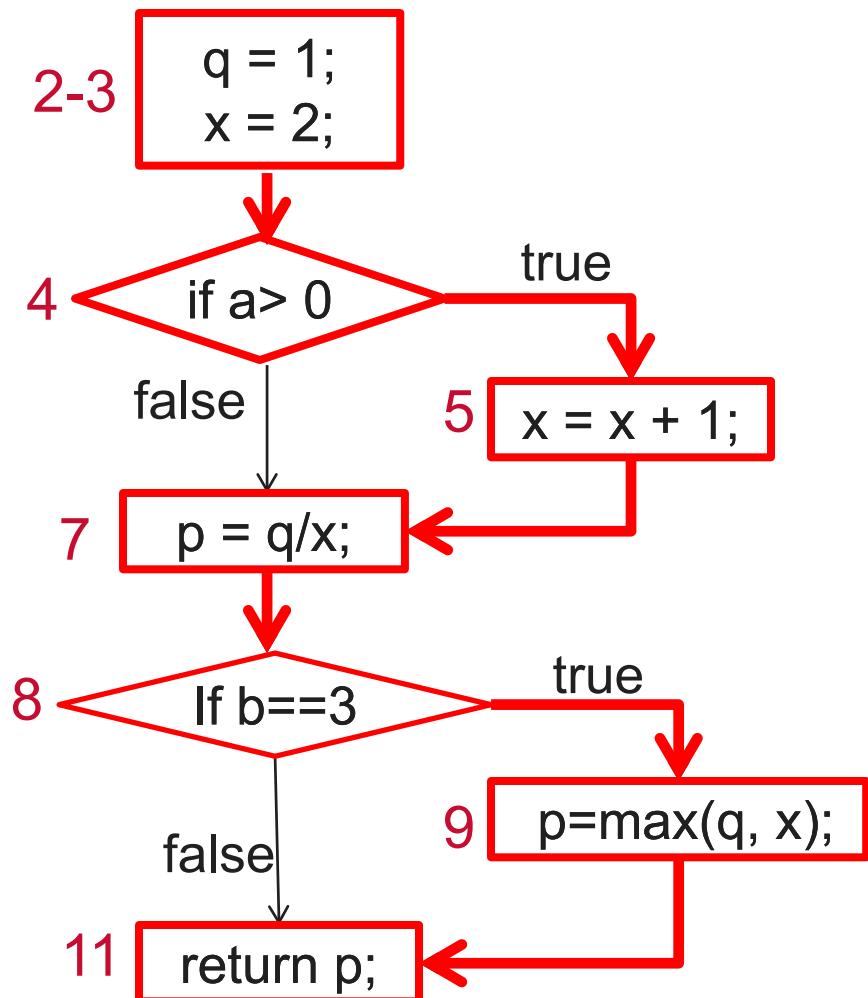
Other execution paths for computeP(int,int)?



Execution Path – computeP(int,int)



Path#3: 2-3, 4, 7, 8, 9, 11



Path#4: 2-3, 4, 5, 7, 8, 9, 11

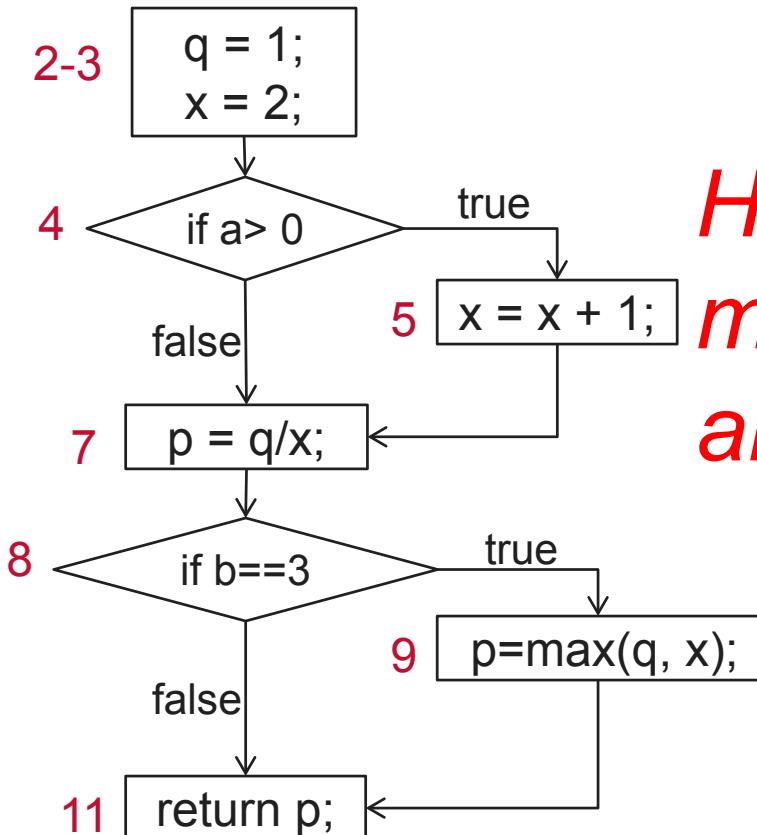
Two decision points → Four execution paths ($2^2 = 4$)

Level 1 - 100% Statement Coverage

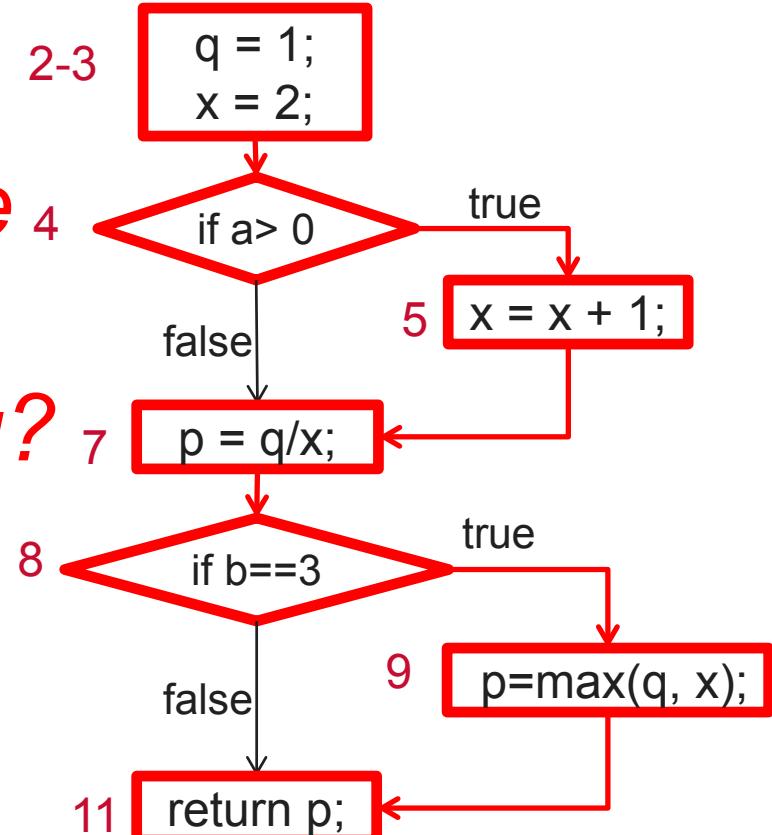
- ▶ Create test cases such that all statements are executed at least once under test
 - ▶ Select execution path(s) to cover all the CFG nodes at least once

Level 1 Coverage – computeP(a, b)

- Select execution path(s) to cover all the CFG nodes at least once



Have we missed anything?



Path: 2-3, 4, 5, 7, 8, 9, 11

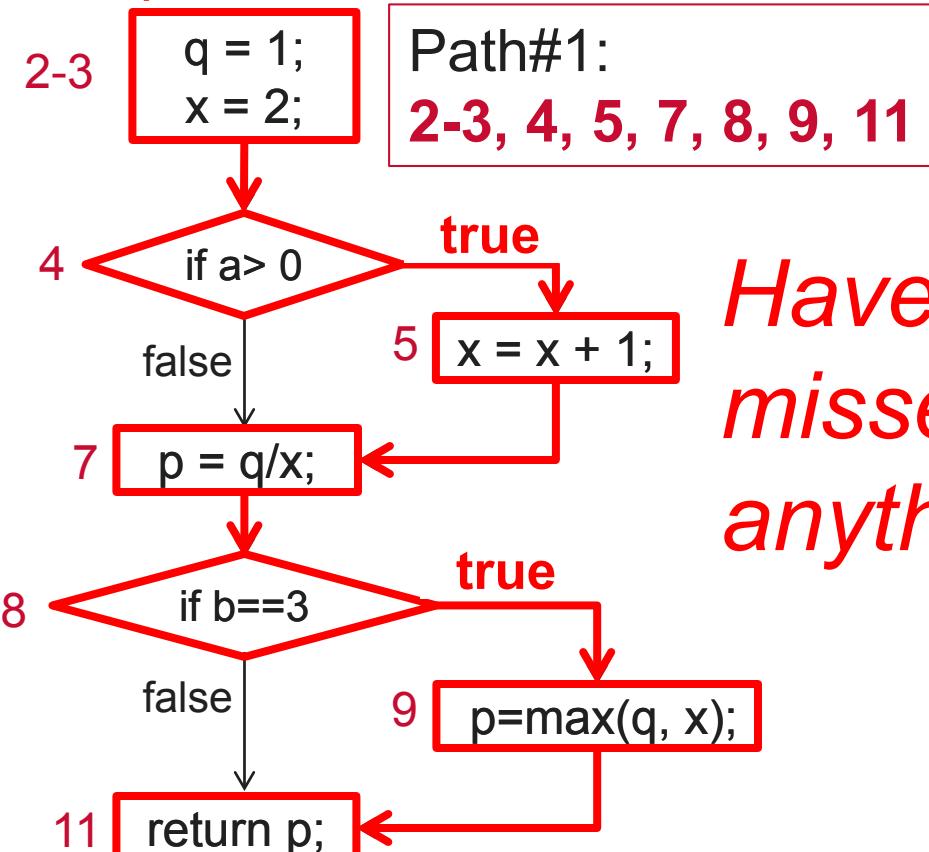
- Choose input values to execute the selected path
e.g., $a=1$ and $b=3$, or $a=10$ and $b=3$, etc

Level 2 – 100% Branch Coverage

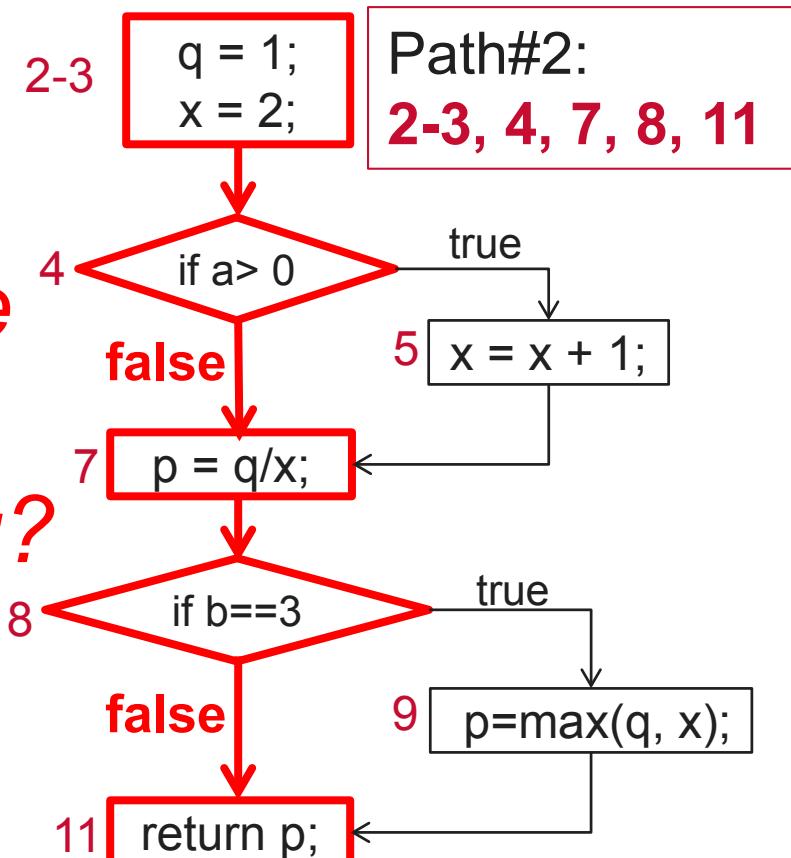
- ▶ Create test cases such that all branches out of the decision points are executed at least once under test
- ▶ Binary decision point evaluates to TRUE and FALSE outcome at least once

Level 2 Coverage – computeP(a, b)

- Select execution paths to execute all branches of the decision points at least once



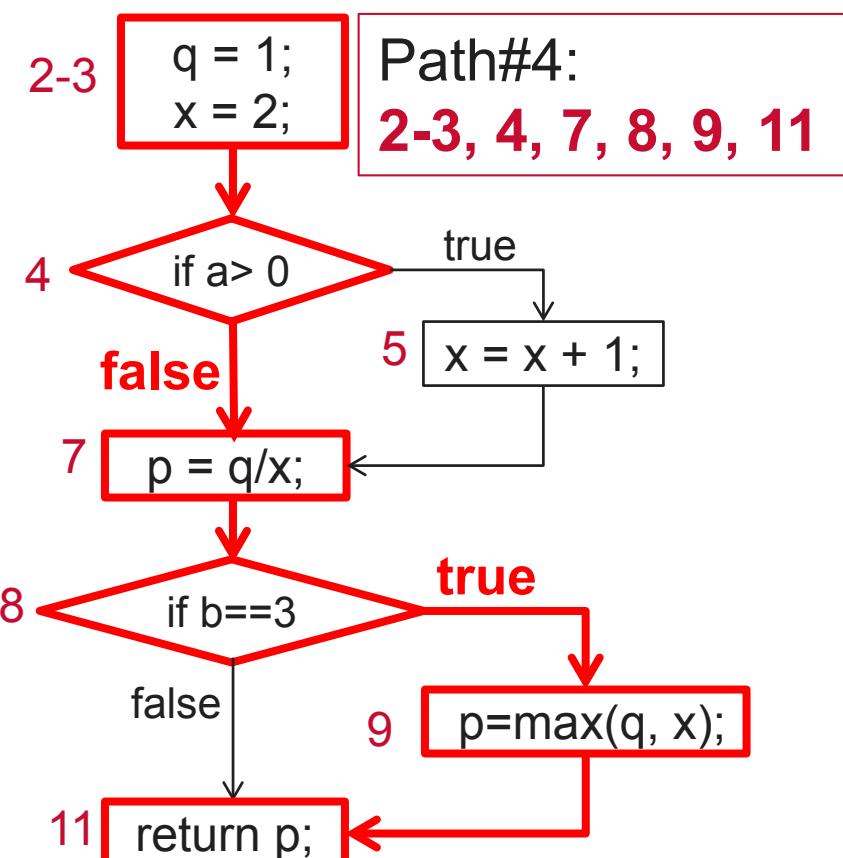
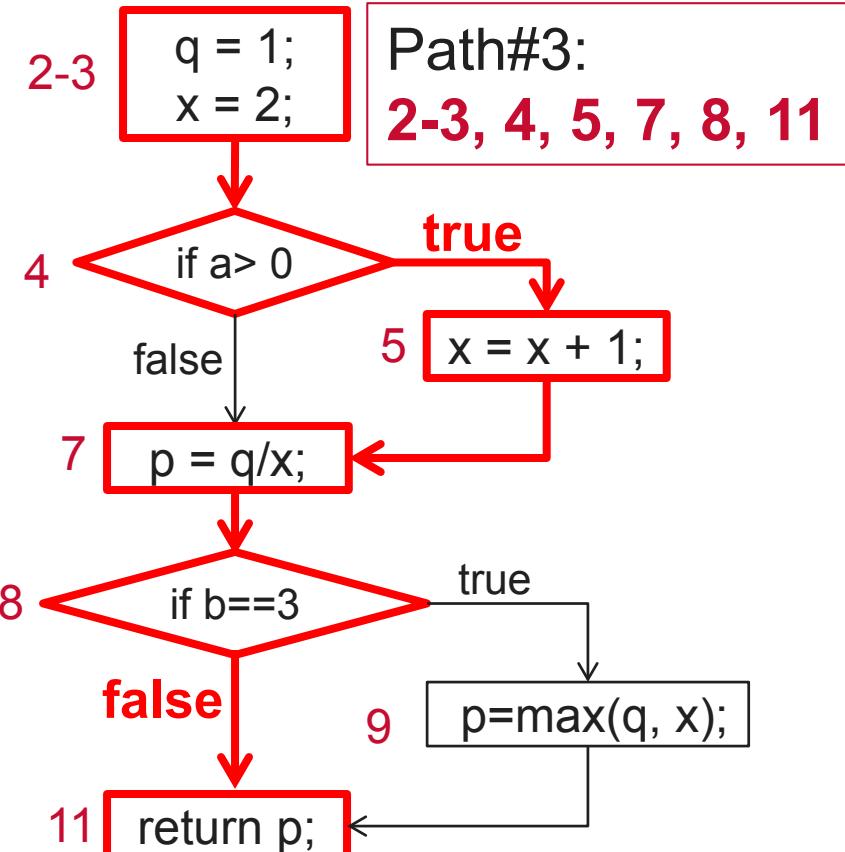
Have we missed anything?



- Choose input values to execute the selected path
e.g., $a=1$ (true), $b=3$ (true); and $a=0$ (false), $b=2$ (false)

Level 4 – 100% Path Coverage

- Two more execution paths (from level 2) to achieve 100% path coverage of all branches of the decision points at least once



- Choose input values to execute the selected path
e.g., a=1 (true), b=2 (false); and a=0 (false), b=3 (true)

Level 4 – 100% Path Coverage

- ▶ Create test cases such that **all execution paths are executed at least once** under test
- ▶ May **not** be **feasible** in general (*we shall see example later*)
 - ▶ Every **decision point** doubles the number of paths (i.e., $2^{|decision|}$)
 - ▶ Every **loop** multiplies the paths by the number of iterations

Level 3 – Basis Path Coverage

- ▶ Create test cases such that the **minimum number of basis paths are executed at least once** under test.
- ▶ What is basis path (a.k.a. Linear Independent Path)?
 - ▶ Each basis path traverses **at least one new control flow edge (i.e. at least one new node)** that **has not** been traversed before the path is defined.
 - ▶ A set of basis paths (or basis set) is a set of linearly independent paths.
 - ▶ The basis set is not unique (*we shall visit this point later*). *Basis Path testing is an effective mechanism to deal with exponential combination of branches.*

Basis Path Testing

From Test Fundamentals lecture slide#11

- ▶ **Basis Path Testing** (or structured testing) is a white box testing method used for designing test cases intended to examine all possible paths of execution in a program at least once.
 - ▶ Analyzes Control flow graph (CFG)
 - ▶ Determines Cyclomatic Complexity (CC) value
 - ▶ Obtains linearly independent paths
 - ▶ Generates test cases for each path

Basis Path Testing Process

► The tester

1. Construct the **control flow graph (CFG)** of the SUT (System Under Test)
2. Identifies **execution paths** through the CFG of the SUT
 - 2.1 computes the CFG's **Cyclomatic Complexity (CC)**
 - 2.2 selects a set of **#CC basis paths**
3. ... *The rest of steps are the same as the slide #3*
4. ...
5. ...

► The test engine

1. ...
2. ...
3. ...

***Basis Path testing guarantees
100% statement and branch
coverage***

Cyclomatic Complexity of a CFG

- ▶ CC measures how complex your program is in terms of the number of decision points (i.e., if/while/for/switch).
- ▶ The more decision points (i.e., if/while/for/switch) a program has → The higher the CC → The more basis paths you need to test.

E.g. if your program has CC value 6, then you've to identify 6 basic paths to test.

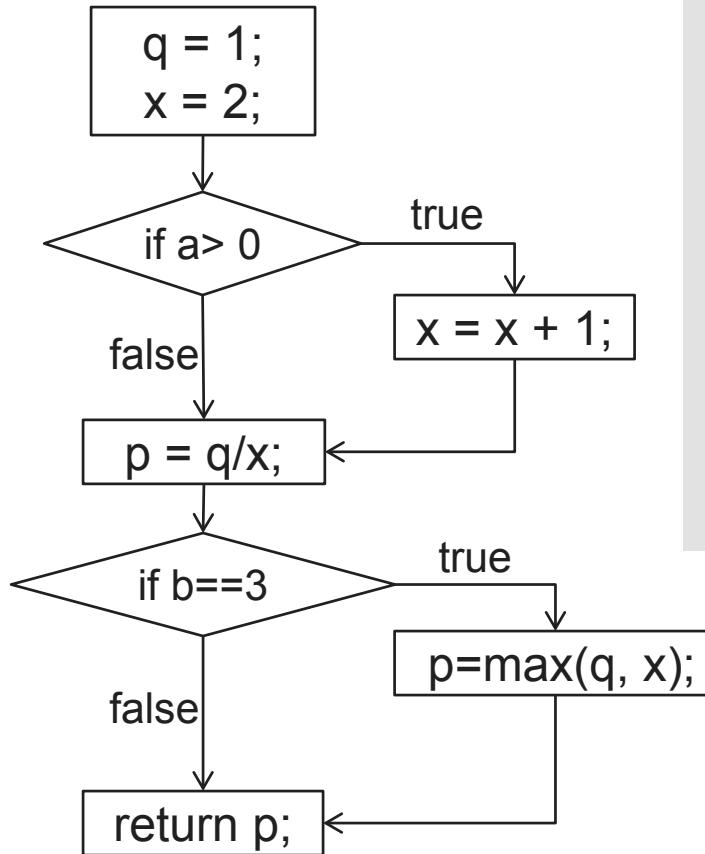
Cyclomatic Complexity of a CFG

- ▶ Use the following two equations to compute CC
 - ▶ $CC = |\text{edges}| - |\text{nodes}| + 2$
 - ▶ $CC = |\text{decisionpoint}| + 1$, if all decision points are binary, i.e., one true branch + one false branch

The two equations compute the same value if all decision points are binary

Cyclomatic Complexity – Example

```
int computeP(int a, int b) {  
    1    int q, x, p;  
    2    q = 1;  
    3    x = 2;  
    4    if(a > 0) {  
    5        x = x + 1;  
    6    }  
    7    p = q/x;  
    8    if(b == 3) {  
    9        p = max(q, x);  
    10   }  
    11   return p;  
}
```



$$\begin{aligned}|\text{edges}| &= 8 \\|\text{nodes}| &= 7 \\ \text{CC} &= 8 - 7 + 2 = 3\end{aligned}$$

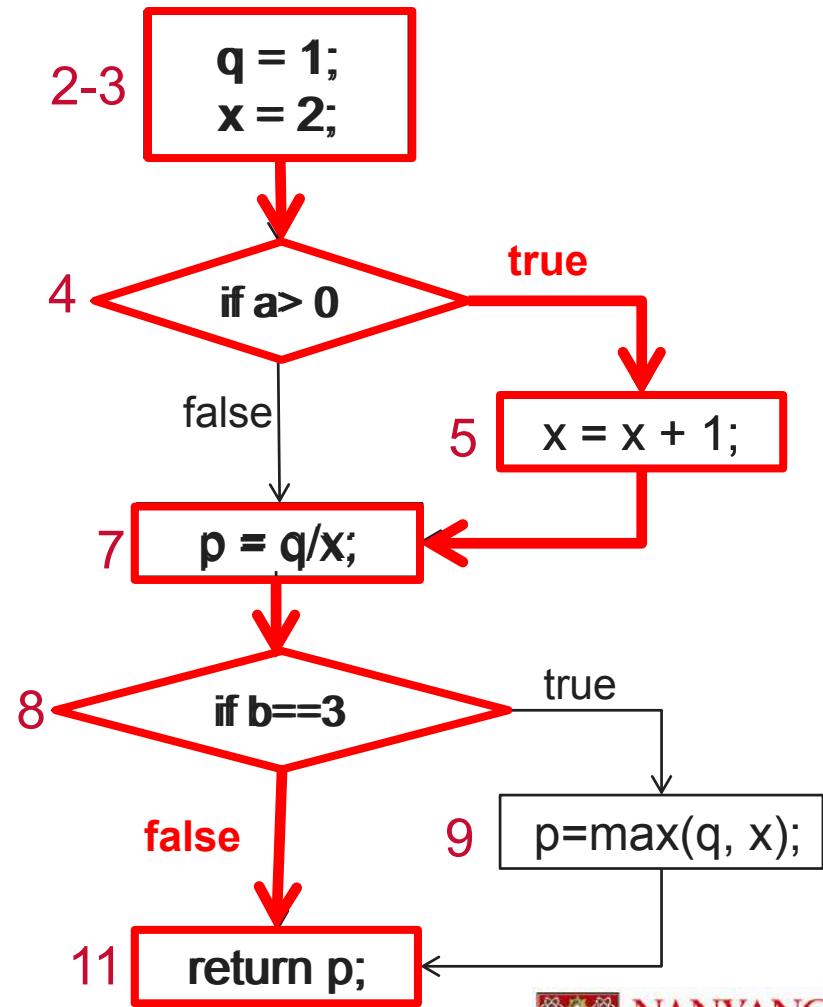
Or,

$$\begin{aligned}|\text{decisionpoint}| &= 2 \\ \text{CC} &= 2 + 1 = 3\end{aligned}$$

Select a Set of Basis Paths

- ▶ Pick a “**baseline**” path
- ▶ Reasonably “typical” path of execution
- ▶ Most important path from the tester’s view

Basis path#1 (baseline):
2-3, 4, 5, 7, 8, 11

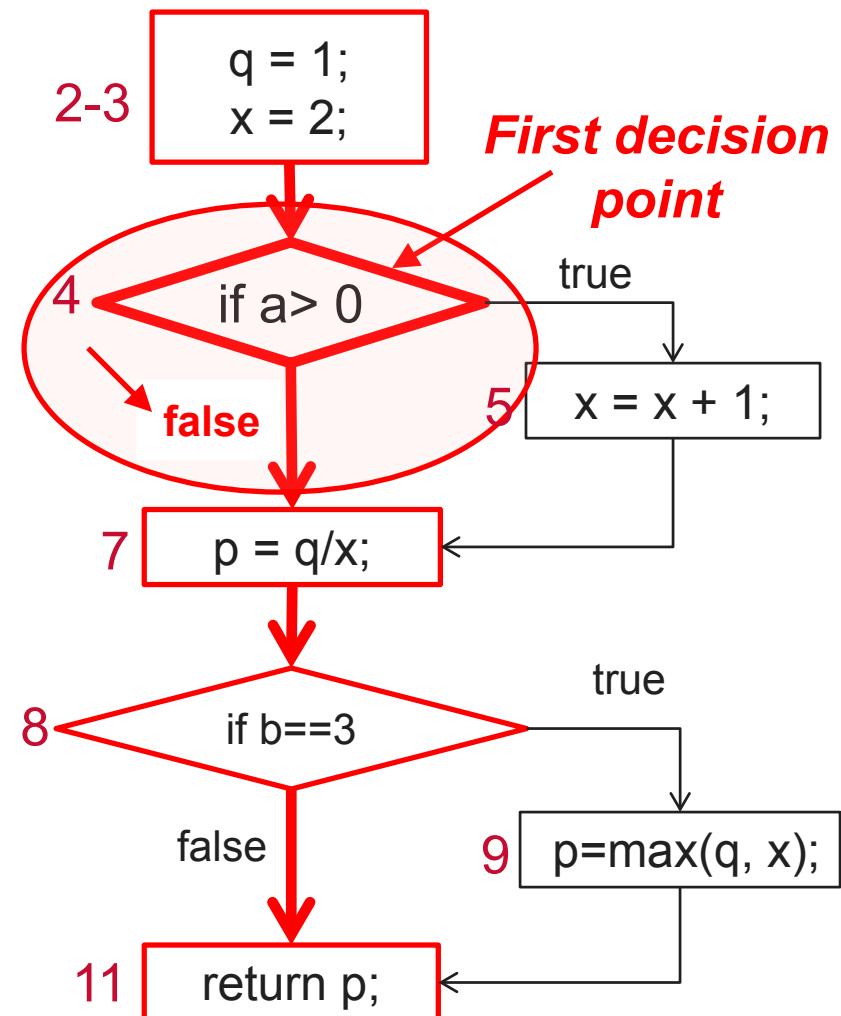


Select a Set of Basis Paths

- ▶ Change the outcome of the first decision point

- ▶ Keep the maximum number of other decision points the same

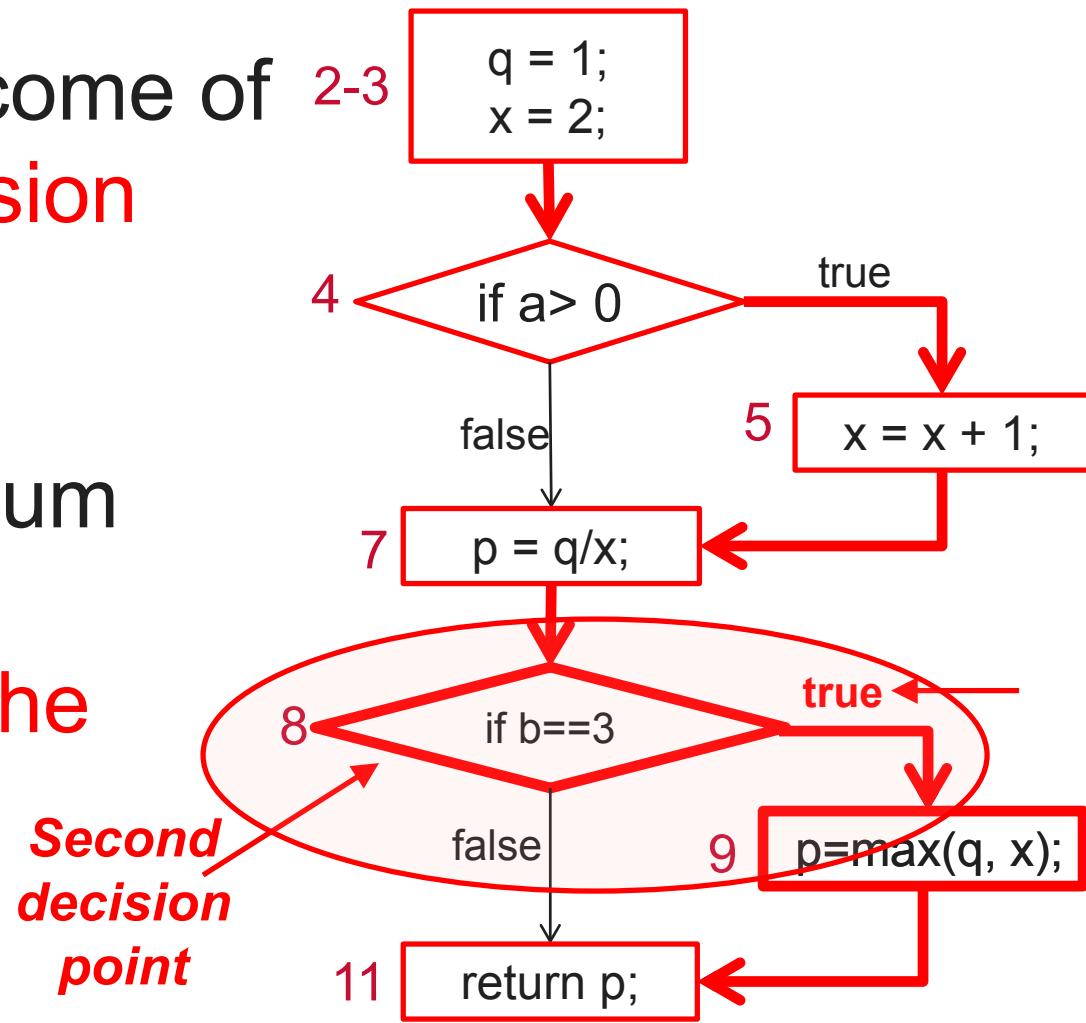
Basis path#2:
2-3, 4, 7, 8, 11



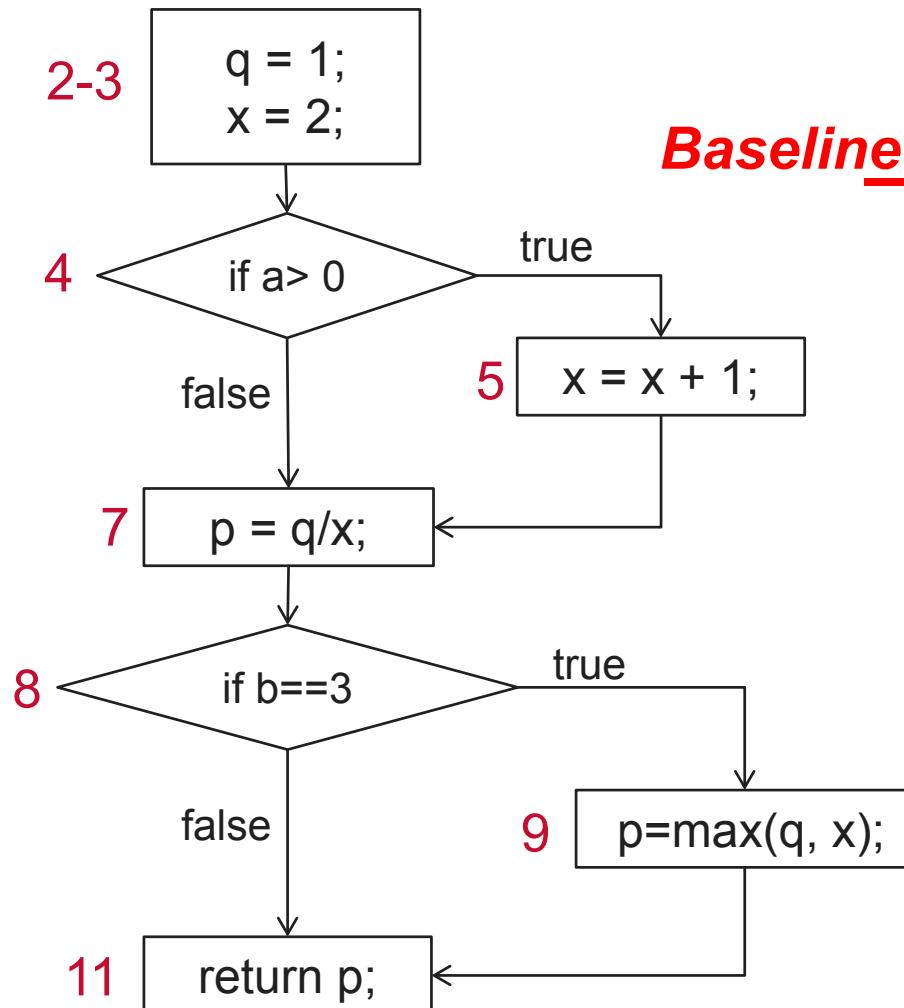
Select a Set of Basis Paths

- ▶ Change the outcome of the second decision point
- ▶ Keep the maximum number of other decision points the same

Basis path#3 :
2-3, 4, 5, 7, 8, 9, 11



Create a Test Case for Each Basis Path



Baseline

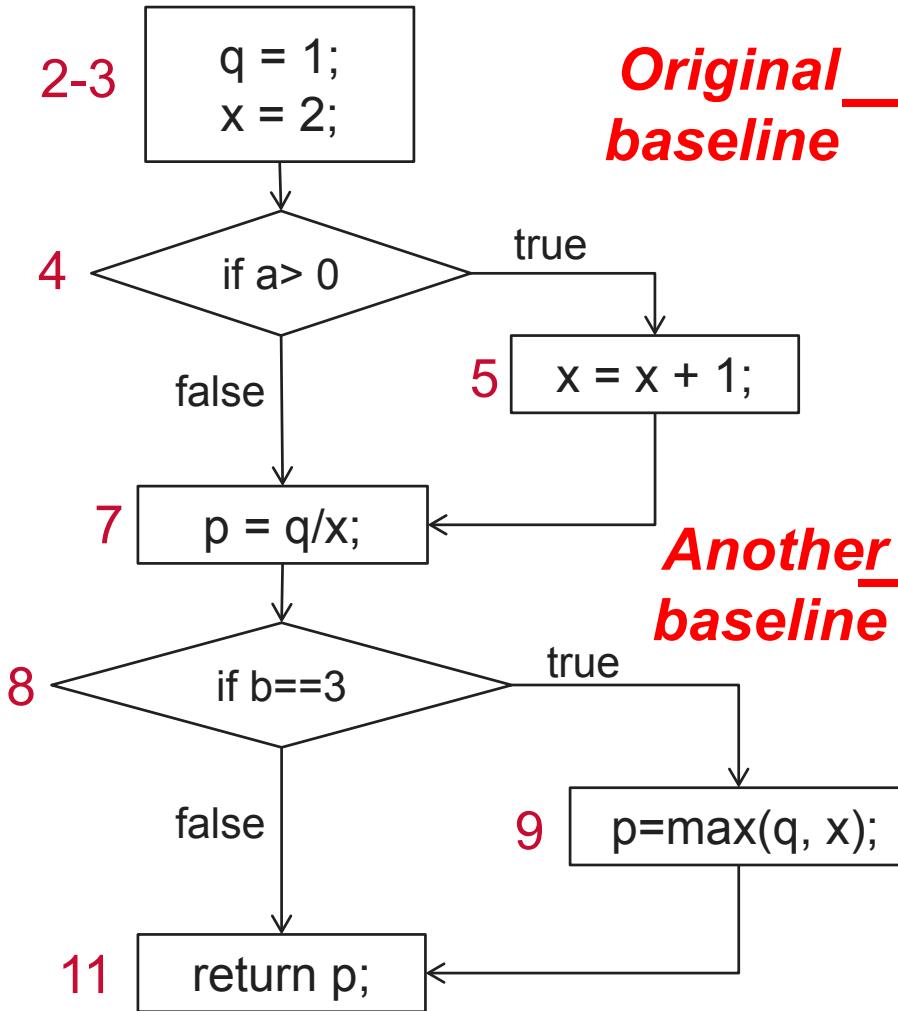
► One set of basis paths

- I. 2-3, 4, 5, 7, 8, 11
- II. 2-3, 4, 7, 8, 11
- III. 2-3, 4, 5, 7, 8, 9, 11

► Three test cases

- I. $a = 4; b = 2$
- II. $a = 0; b = 5$
- III. $a = 7; b = 3$

Basis Paths for computeP()



Original baseline

► One set of basis paths

I. 2-3, 4, 5, 7, 8, 11

II. 2-3, 4, 7, 8, 11

III. 2-3, 4, 5, 7, 8, 9, 11

Another baseline

► Another set of basis paths

I. 2-3, 4, 7, 8, 11

II. 2-3, 4, 5, 7, 8, 11

III. 2-3, 4, 7, 8, 9, 11

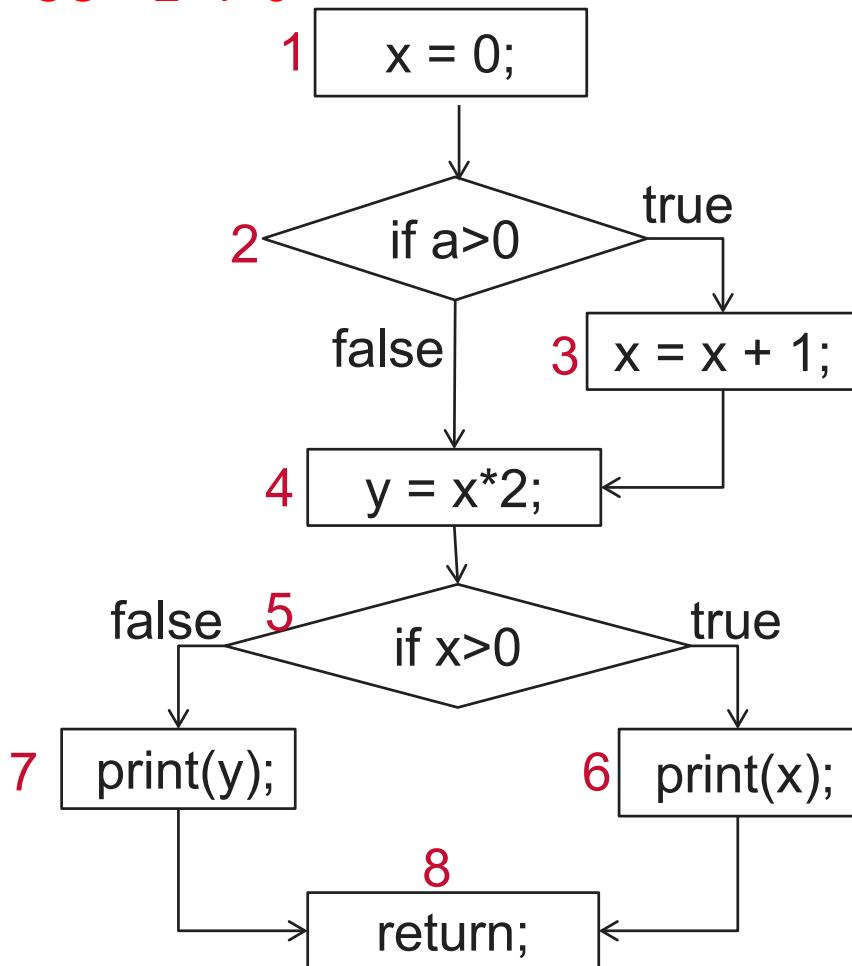
The third paths in the two sets are different

The basis set is not unique; but each basis set has the same number of basis paths (i.e. 3 in this case)

Not All Basis Paths are Feasible

Consider another program logic

$$CC = 2+1=3$$



- ▶ One set of basis paths

I. 1, 2, 3, 4, 5, 6, 8 **Baseline**

II. 1, 2, 4, 5, 6, 8 (infeasible)

III. 1, 2, 3, 4, 5, 7, 8 (infeasible)

- *Two infeasible paths*

- ▶ One test case

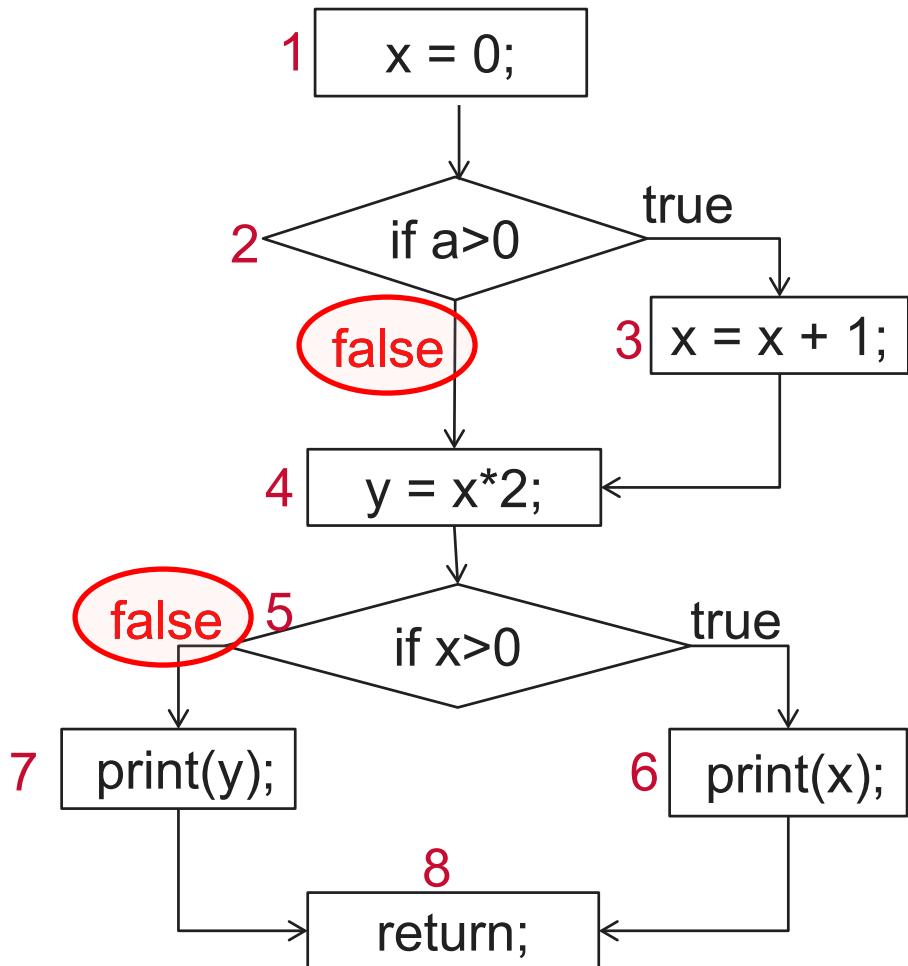
I. `a = 4`

II. Infeasible basis path

III. Infeasible basis path

*Fail to test 2, 4, 5, 6, 8, and
5, 7, 8, branches*

Minimize Infeasible Basis Paths



- ▶ Another set of basis paths
 - I. 1, 2, 3, 4, 5, 6, 8
 - II. 1, 2, 4, 5, 6, 8 (infeasible)
 - III. 1, 2, 4, 5, 7, 8 (change both decision points at the same time)
 - One infeasible path

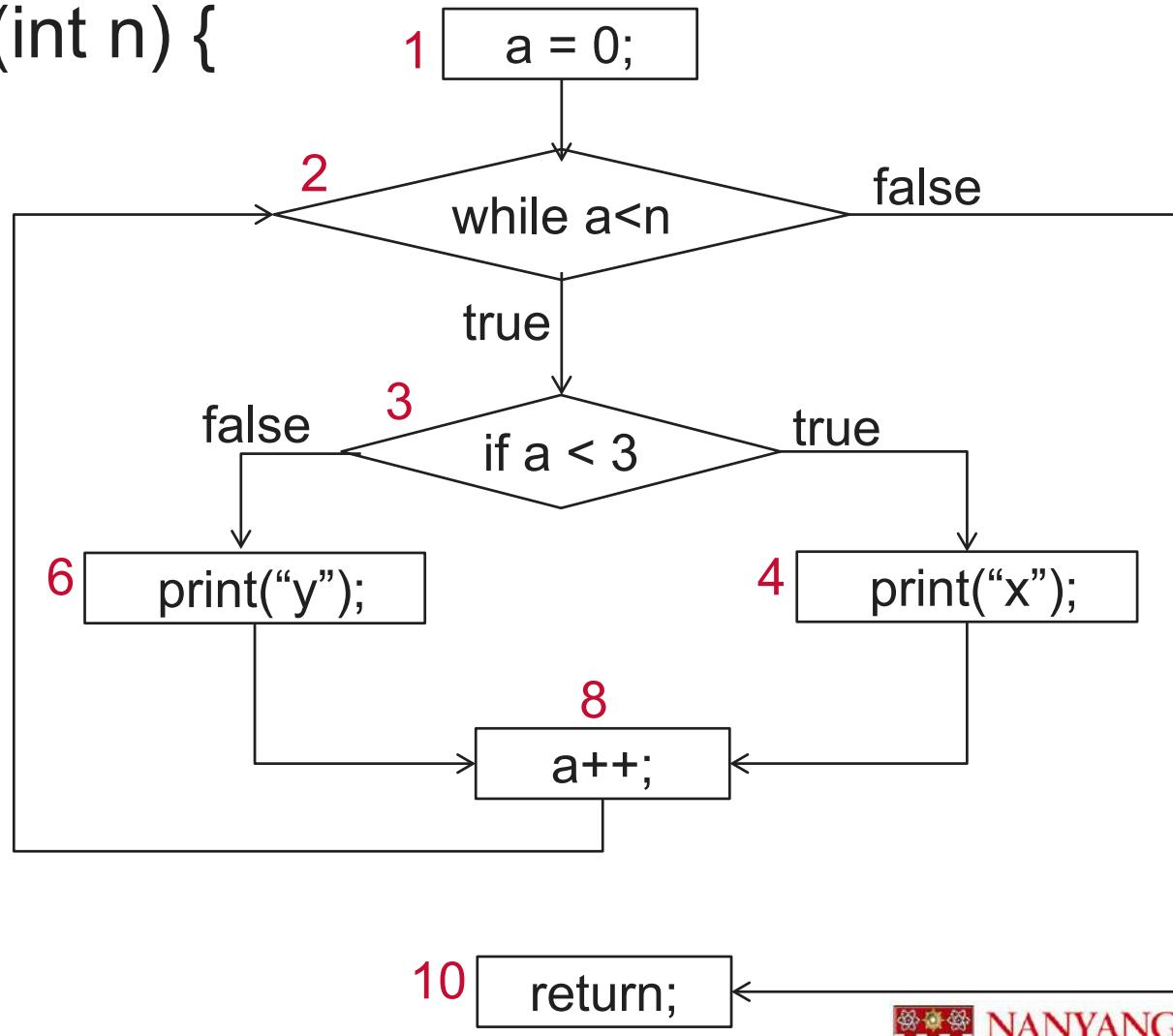
- ▶ Two test case
 - I. a = 4
 - II. Infeasible basis path
 - III. a = 0

How to Deal with Loop?

- ▶ When selecting basis paths, select the loop **only zero and once** (no need to consider iteration)
- ▶ When **selecting baseline paths**, select **false branch first** at loop decision point, i.e., ***do not enter the loop***
- ▶ When choosing input values to execute the path, the real execution path may execute loop several times (see *example next slide*).

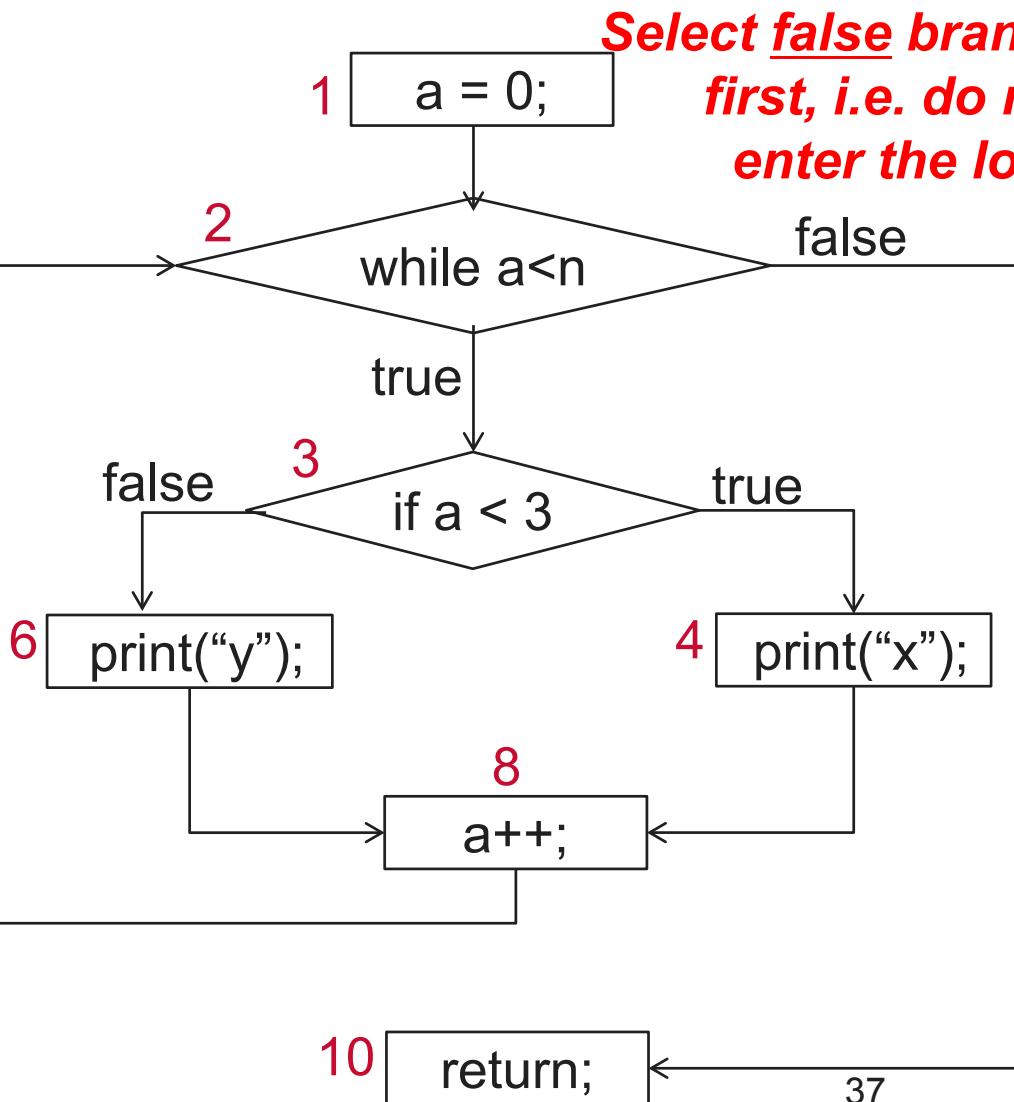
Exercise – Basis Path Testing

```
doSomething(int n) {  
1   int a = 0;  
2   while(a < n) {  
3       if(a < 3) {  
4           print("X");  
5       } else {  
6           print("Y");  
7       }  
8       a++;  
9   }  
10  return;  
11 }
```



Exercise – Basis Path Testing

$$CC = 2+1 = 3$$



Select false branch first, i.e. do not enter the loop

- ▶ Three basis paths
- I. 1, 2, 10 Baseline
- II. 1, 2, 3, 4, 8, 2, 10
- III. 1, 2, 3, 6, 8, 2, 10

- ▶ Three test cases
- I. $n = 0$
- II. $n = 1$
- III. $n = 4$

- ▶ Real execution paths
- I. 1, 2, 10
- II. 1, 2, 3, 4, 8, 2, 10
- III. 1, 2, 3, 4, 8, 2, 3, 4, 8, 2, 3, 4, 8, 2, 3, 6, 8, 2, 10

Testing – Both Testings are Important!

▶ Black Box Testing

- ▶ Equivalence class testing (range of values, discrete values)
- ▶ Boundary value testing (only applicable to range of values)

▶ White Box Testing

- ▶ Control flow graph (CFG)
- ▶ Statement, branch, basis path, path coverage
- ▶ Basis path testing
 - ▶ Cyclomatic complexity (how many basis paths you need to test)
 - ▶ Select basis paths

CZ2006/CE2006 Software Engineering

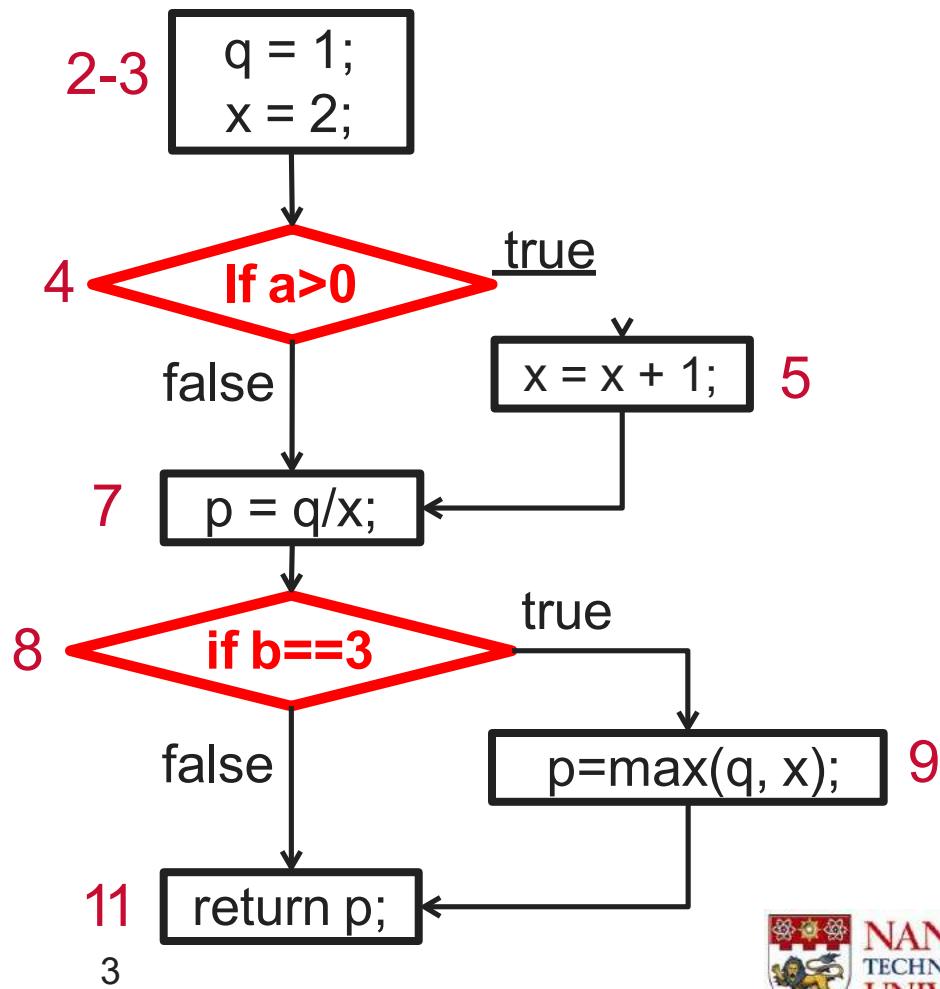
Lecture 21: Software Maintenance

Liu Yang

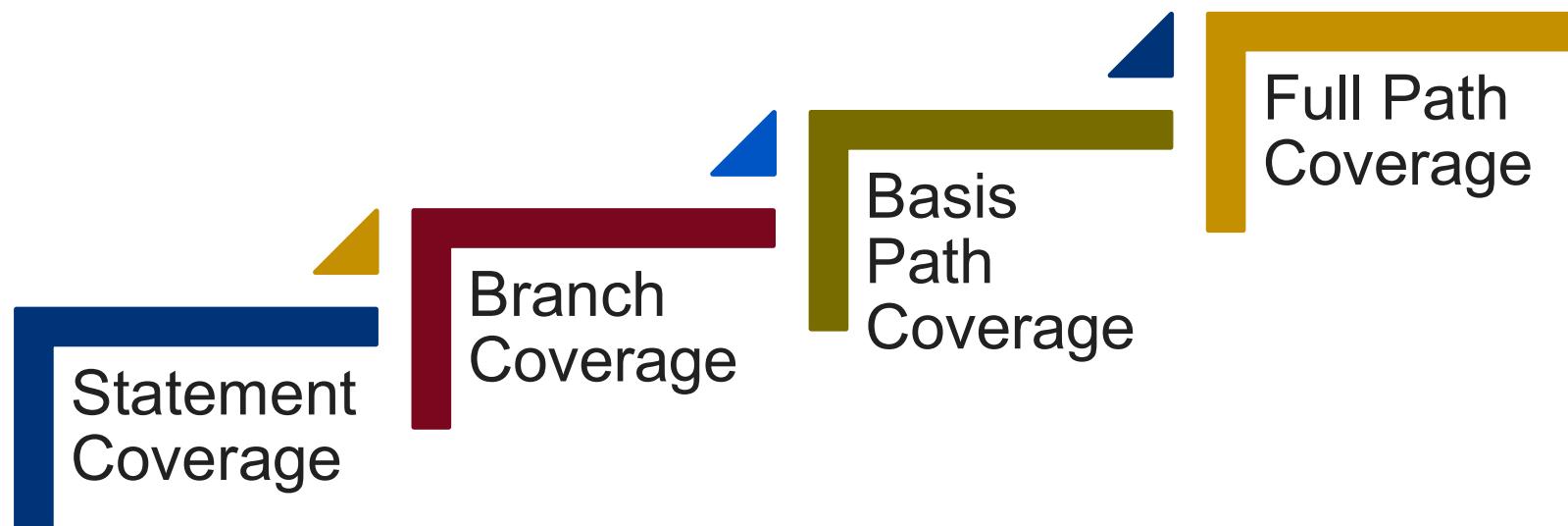
White Box Testing Summary: Control Flow Testing

Control Flow Graph - Example

```
int computeP(int a, int b) {  
    1     int q, x, p;  
    2     q = 1;  
    3     x = 2;  
    4     if(a > 0) {  
    5         x = x + 1;  
    6     }  
    7     p = q/x;  
    8     if(b == 3) {  
    9         p = max(q, x);  
   10    }  
   11    return p;  
   12 }
```



Coverage Criteria



White Box Testing: Basis Path Testing Process

- ▶ Construct **control flow graph (CFG)** from the program (code fragment) of the SUT.
- ▶ Identifies **execution paths** through the CFG.
 - ▶ Computes the CFG's **Cyclomatic Complexity (CC)**
 - ▶ Selects a set of **#CC basis paths**
- ▶ Choose **input values** to execute the paths.
- ▶ The **CC** value tells us the upper bound on the size of the basis set, i.e. gives us the minimum number of linearly independent paths (basis paths) we need to find. E.g. if your program has CC value 6, then you've to identify six basic paths to test.

Basis Path testing guarantees 100% statement and branch coverage

White Box Testing: Cyclomatic Complexity (CC) of a CFG

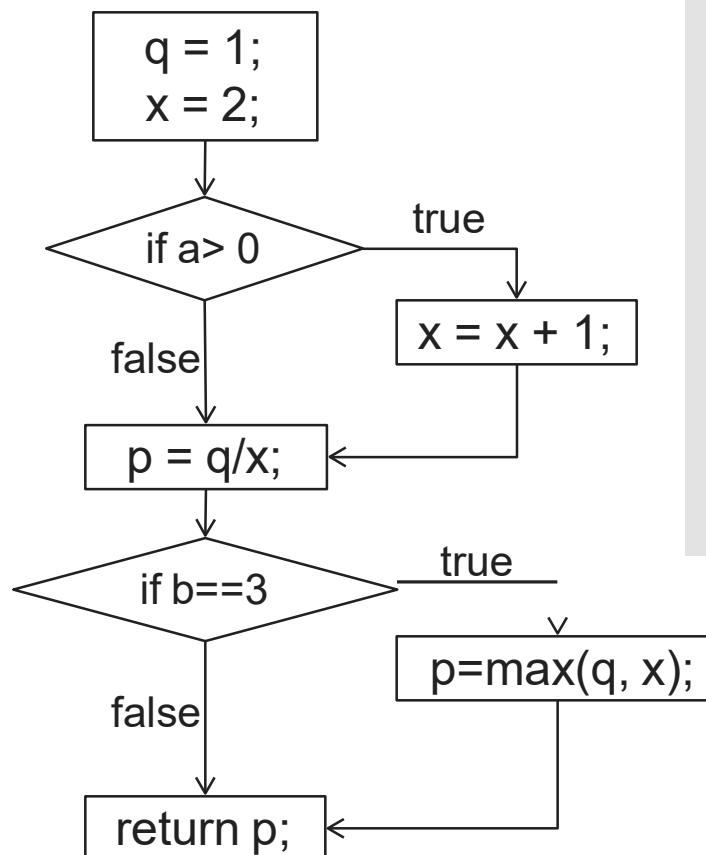
- ▶ Use the following two equations to compute CC
 - ▶ $CC = |\text{edges}| - |\text{nodes}| + 2$
 - ▶ $CC = |\text{decisionpoint}| + 1$, if all decision points are binary, i.e., one true branch + one false branch

The two equations compute the same value if all decision points are binary

White Box Testing: Cyclomatic Complexity (CC) – Example

```
int computeP(int a, int b) {
```

```
1   int q, x, p;  
2   q = 1;  
3   x = 2;  
4   if(a > 0) {  
5       x = x + 1;  
6   }  
7   p = q/x;  
8   if(b == 3) {  
9       p = max(q, x);  
10  }  
11  return p;  
}
```



$|\text{edges}| = 8$

$|\text{nodes}| = 7$

$$\text{CC} = 8 - 7 + 2 = 3$$

Or,

$|\text{decisionpoint}| = 2$

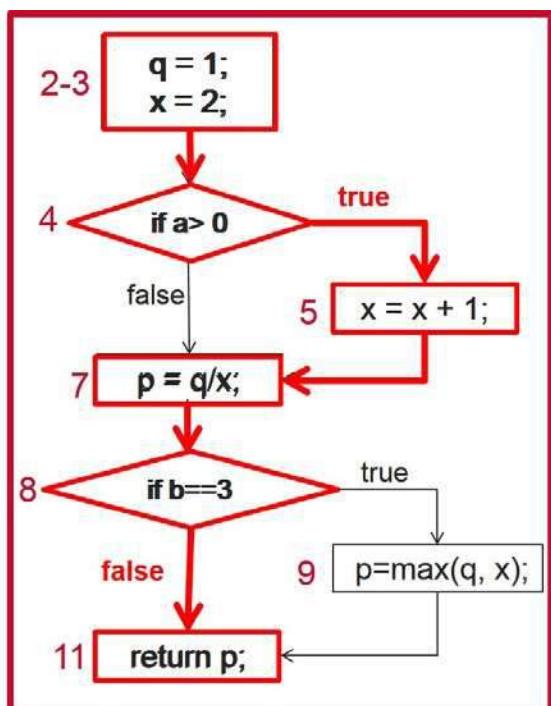
$$\text{CC} = 2 + 1 = 3$$

White Box Testing: Select a Set of Basis Paths

- ▶ Choose a “**baseline**” path
 - ▶ Reasonably “typical” path of execution
 - ▶ Most important path from the tester’s view
- ▶ **Change** the outcome of the **first decision point**, keep the maximum number of other decision points the same
- ▶ **Change** the outcome of the **second decision point**, keep the maximum number of other decision points the same
- ▶ **Change** the outcome of the **third decision point**, and so forth

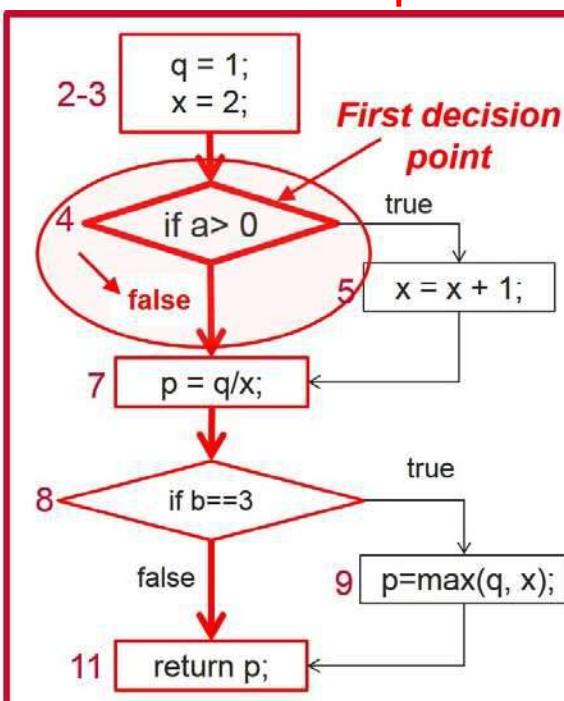
White Box Testing: Select a Set of Basis Paths

Select baseline path



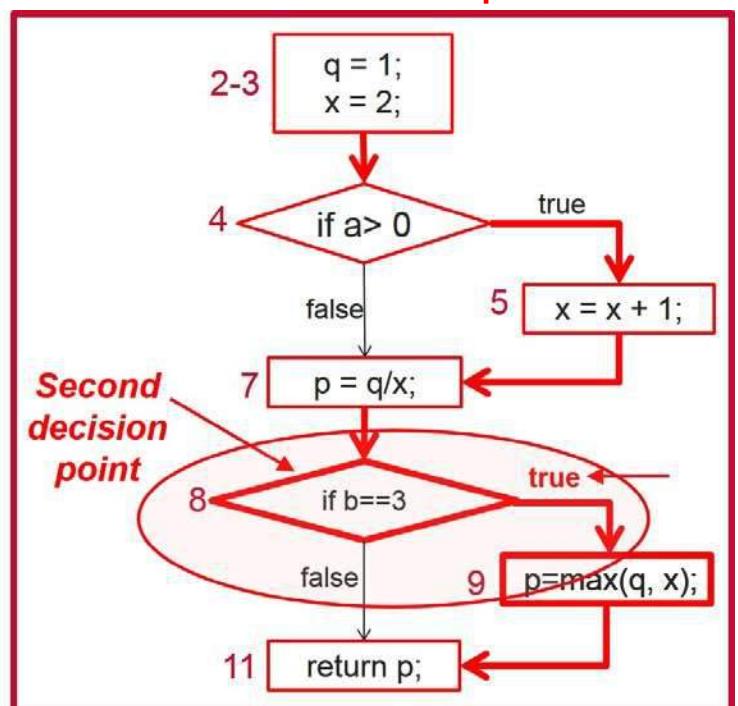
Basis path#1 (baseline):
2-3, 4, 5, 7, 8, 11

Change outcome of first decision point



Basis path#2:
2-3, 4, 7, 8, 11

Change outcome of second decision point



Basis path#3 :
2-3, 4, 5, 7, 8, 9, 11

White Box Testing: Basis Path Testing

For independent paths (Basis Paths) –

- The basis set is **not unique**
- There **may be several different basis sets for the given algorithm, based on the baseline path you've chosen**. You may have derived a different basis set
- The basis set **“covers” all the nodes and edges** in the CFG (Basis Path testing guarantees 100% statement and branch coverage)

For test cases –

- Prepare test cases that will force execution of each path in the basis set, i.e. **choose input parameters (values) to execute each path in the basis set.**
- A basis path **may not be feasible** regardless of input parameters.

White Box Testing – Dealing with Loop

1. To choose basis path (for loop):

- ▶ When selecting basis paths, always execute the loop **zero and once** (i.e. no need to consider iteration).
- ▶ When selecting baseline paths, select false branch first at loop decision point, i.e., **do not enter the loop**.
- ▶ Then execute the loop **ONE time** (i.e. the TRUE branch next, for one time).

1. For basis path testing (for loop), the test cases may execute the loop **multiple iterations**.

Testing – Both Testings are Important!

▶ Black Box Testing

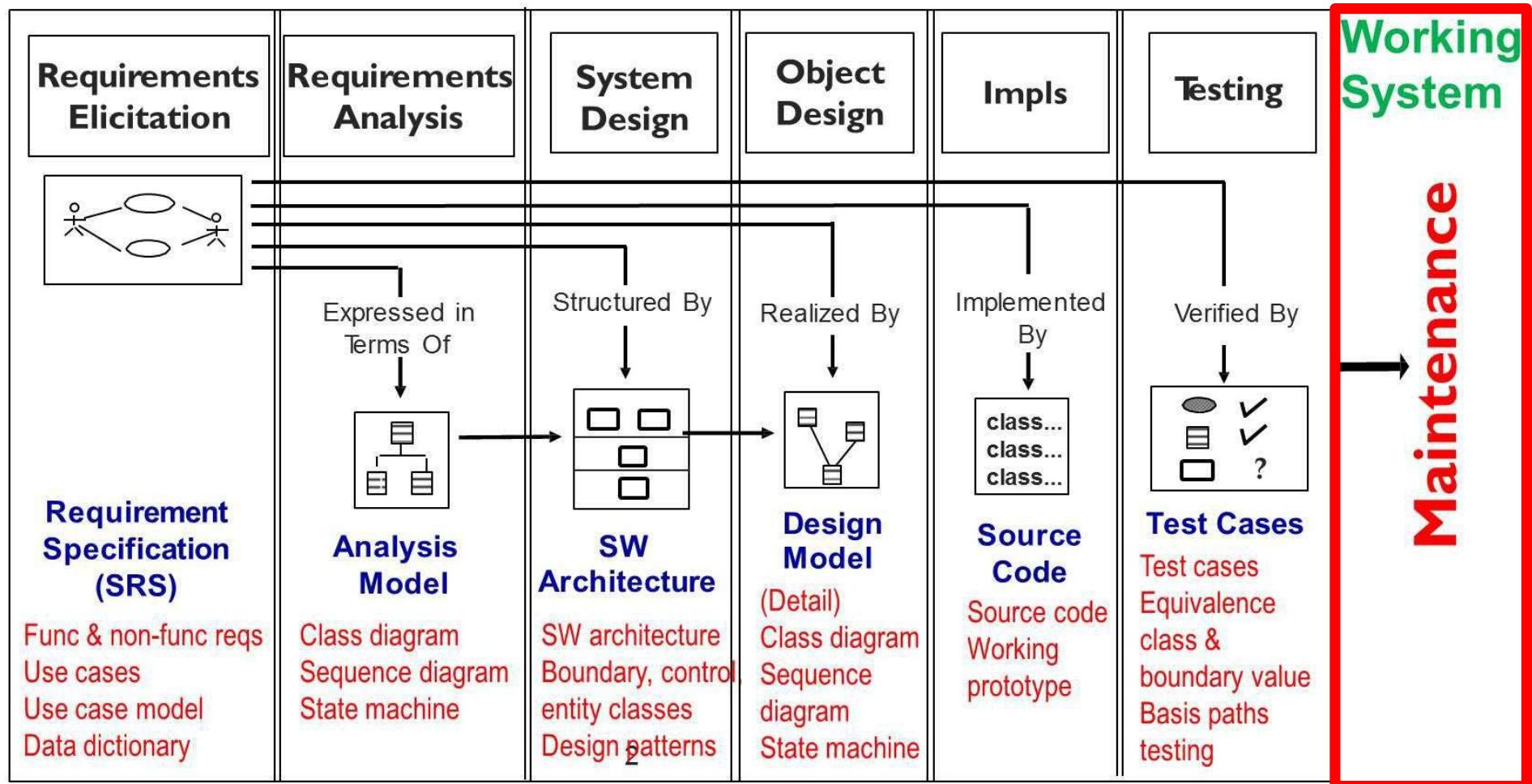
- ▶ Equivalence class testing (range of values, discrete values)
- ▶ Boundary value testing (only applicable to range of values)

▶ White Box Testing

- ▶ Control flow graph (CFG)
- ▶ Statement, branch, basis path, path coverage
- ▶ Basis path testing
 - ▶ Cyclomatic complexity (how many basis paths you need to test)
 - ▶ Select **basis** paths
 - ▶ Choose **input values** to execute the paths

*Lecture#16 – Lecture#19
Tutorial#9 Q1, Tutorial#10 Q1*

Software Maintenance



Causes of Maintenance Problems

- ▶ It is estimated that there are more than 100 billion lines of code in production in the world.*
- ▶ As much as 80% of it is *unstructured, patched, and badly documented.**

* Hans van Vliet “Software Engineering Principles and Practices” 3rd Edition

Causes of Maintenance Problems

▶ Unstructured code

- ▶ E.g. Numerous GOTO statements – *hard to read, understand, and maintain!* (Spaghetti code)
 - Improve by using structured programming such as IF-ELSE statements.
- ▶ E.g. Poor and inconsistent naming, long procedures, strong coupling, weak cohesion, deeply-nested IF statements, etc.

▶ Insufficient domain knowledge

- ▶ To gain a sufficient understanding of a system from its source code.
- ▶ The more spaghetti-like this code is, the less easy it becomes to disentangle it.

▶ Insufficient documentation

- ▶ No documentation, out-of-date documentation, or insufficient documentation.

What is Software Maintenance?

- ▶ **Software maintenance** is defined as (IEEE610, 1990)*:

*“The process of **modifying** a software system after delivery to **correct faults, improve performance or other attributes, or adapt to a changed environment.**”*

* IEEE610 (1990). IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12.

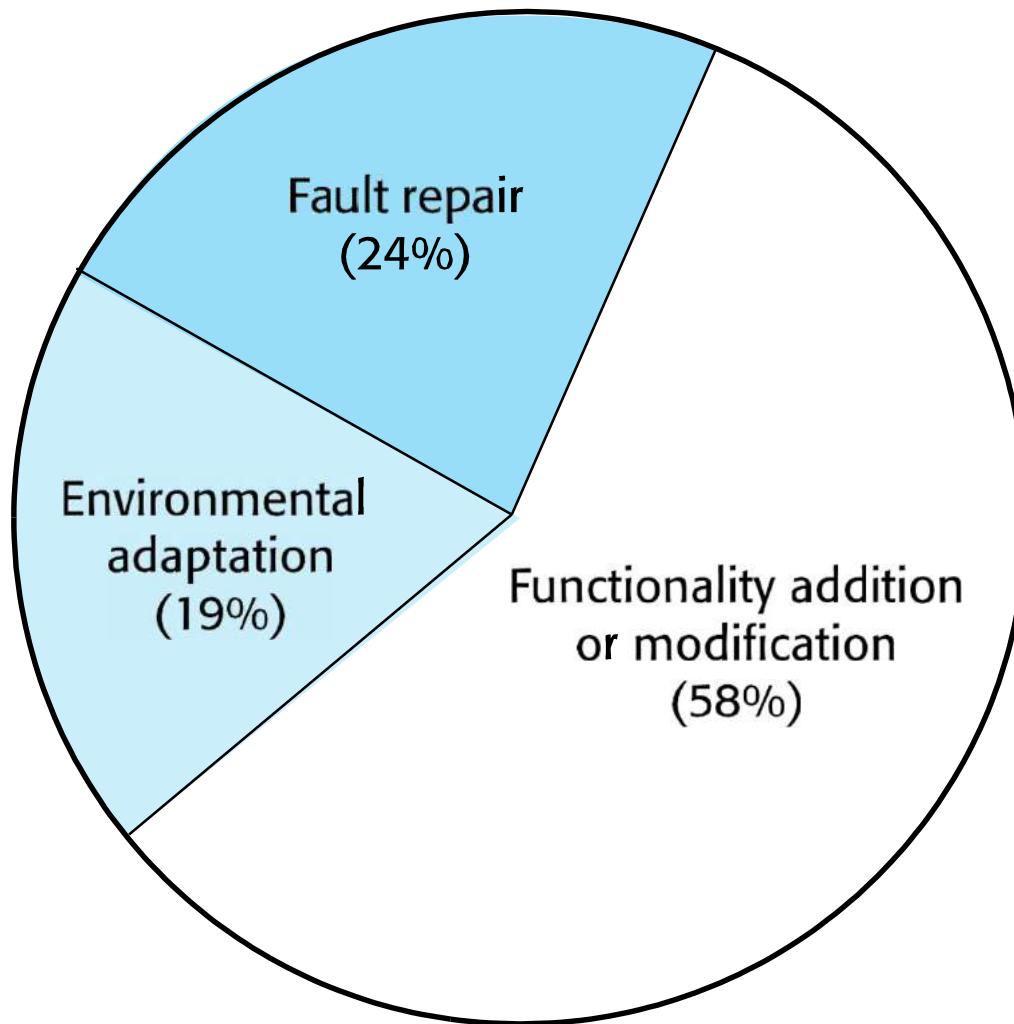
What is Software Maintenance?

- ▶ **Software maintenance is thus concerned with:**
 - ▶ Correcting errors found after the software has been delivered.
 - ▶ Adapting the software to changing requirements, changing environments, etc.
- ▶ Changes are implemented by modifying existing system components and/or adding new components to the system.

Types of Software Maintenance

- ▶ **Fault repairs:**
 - ▶ Changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements.
- ▶ **Environmental adaptation:**
 - ▶ Maintenance to adapt software to a different operating environment.
 - ▶ Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- ▶ **Functionality addition/modification:**
 - ▶ Modifying the system to satisfy new requirements.

Maintenance Effort Distribution



Adapted from Ian Sommerville "Software Engineering" 9th Edition

Software Refactoring

- ▶ **Refactoring** is the process of making improvements to a program to slow down degradation through change. It means modifying (but not adding functionality) a program to:
 - ▶ Improve its structure
 - ▶ Reduce its complexity
 - ▶ Make it easier to understand
- ▶ Refactoring does not change the behavior of the software

Software Refactoring

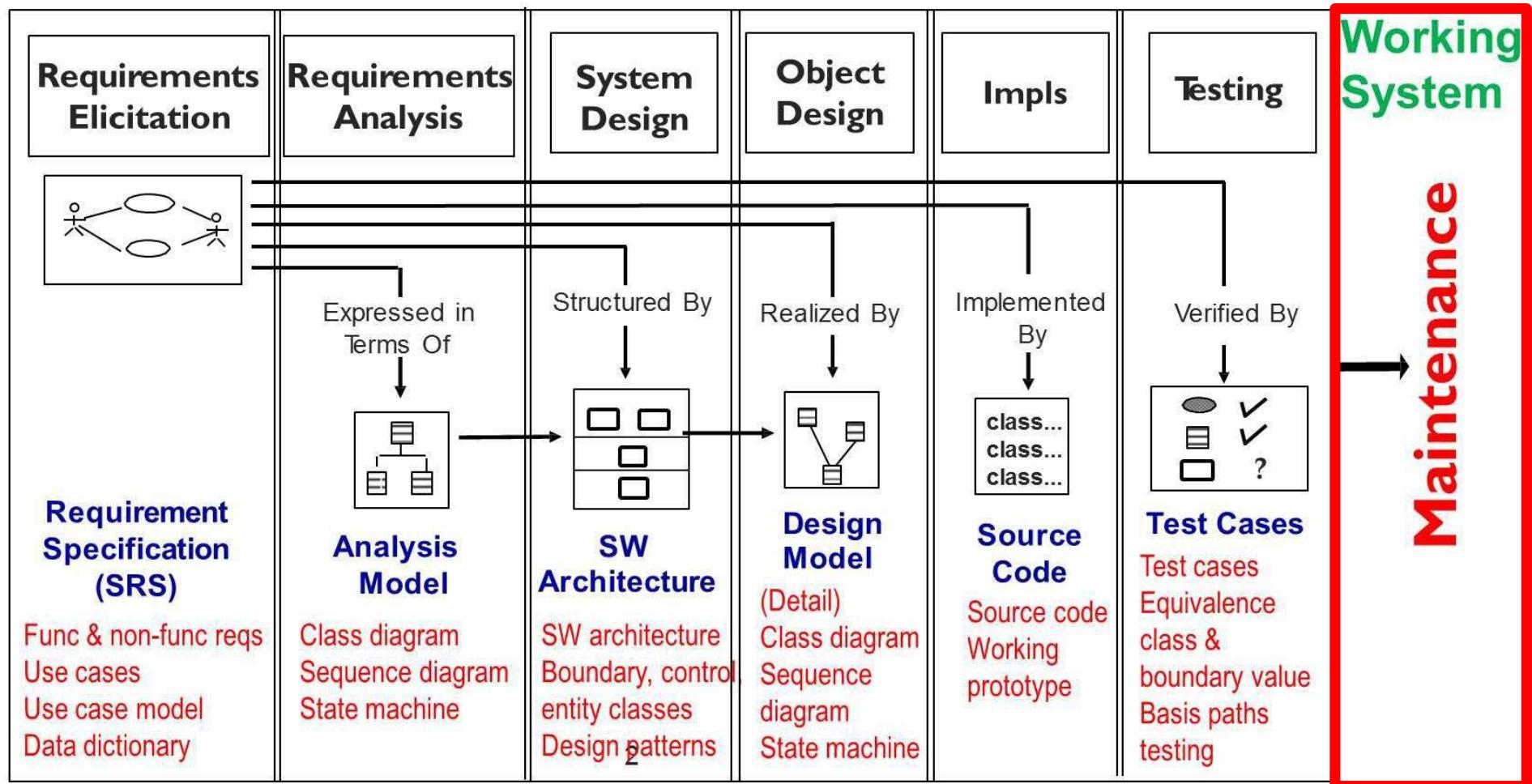
- ▶ Refactoring is a continuous process of improvement throughout the development and evolution process.
- ▶ Refactoring is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Program Code Improvements

- ▶ Situations (“*bad smells*”*) in which the code of a program can be improved:
 - ▶ Duplicate code
 - ▶ Long methods
 - ▶ Large class
 - ▶ Temporary field
 - ▶ Switch (case) statements
 - ▶ Lazy class
 - ▶ Data clumping (same group of data reoccur in several places)
 - ▶ Tight coupling of two classes

* Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

Software Maintenance



Quiz

- ▶ Quiz format:
 - ▶ Date: 18 Nov 2021 (Thursday)
 - ▶ Time: 12 noon
 - ▶ Duration: 1 hour
 - ▶ Answer **ALL** questions
 - ▶ Five (5) MCQ
 - ▶ Five (5) short-answer questions
 - ▶ Questions carry different marks
 - ▶ Questions may have sub-questions

Quiz

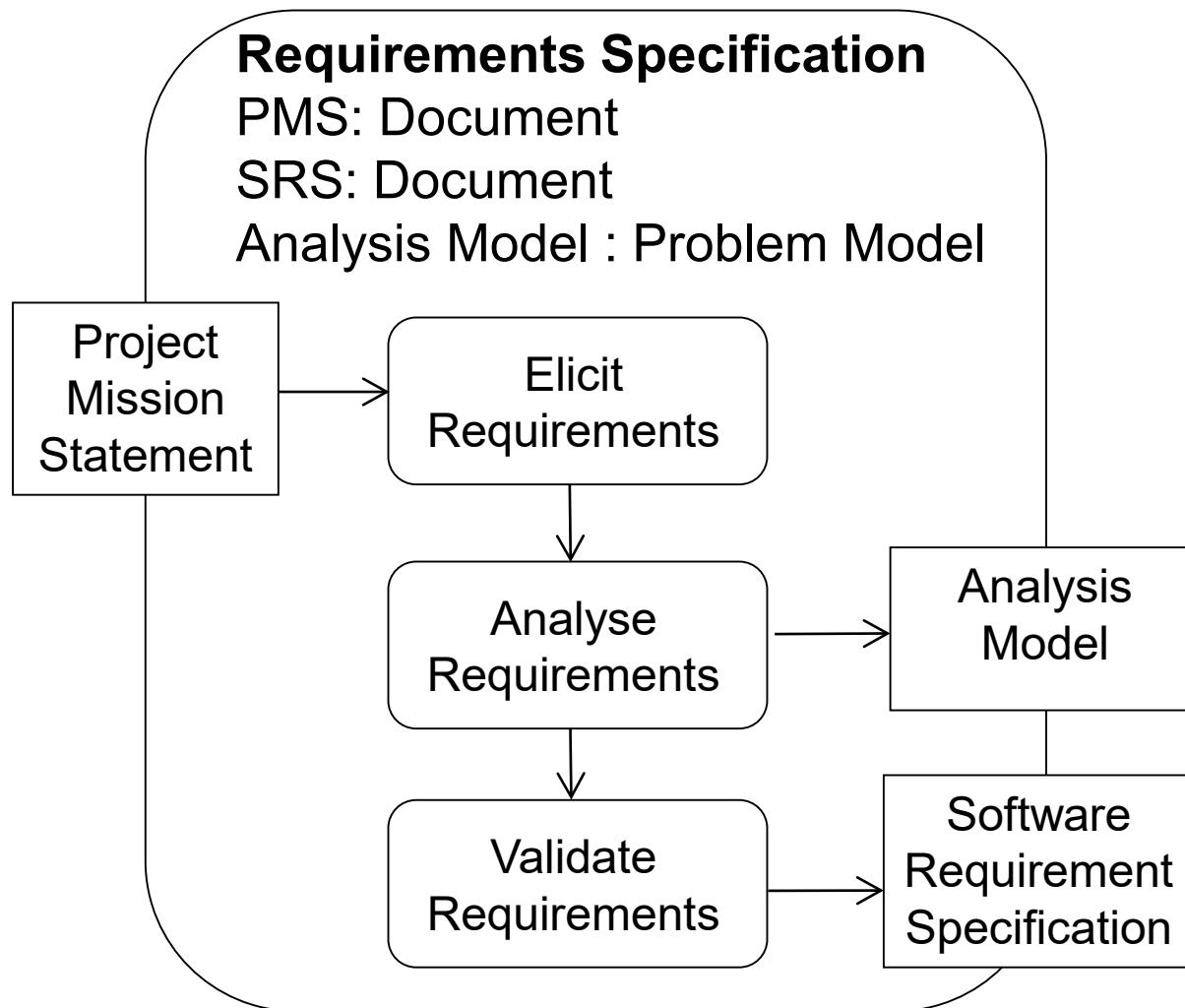
- ▶ Open Book
 - ▶ ONLY paper materials (not limited to cheat sheet and textbook)
 - ▶ NO electronic devices allowed

Software Engineering

CE2006/CZ2006

Software Requirements Specification

Requirements Specification Activities



Software Requirements Specification is

1. The output of the Requirements Elicitation and Requirements Analysis.
2. An organization's understanding (in writing) of a customer or potential client's system requirements and dependencies *at a particular point in time* (usually) prior to any actual design or development work.
3. A two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.
4. A statement in precise and clear language describing those functions and capabilities a software system must provide, as well as stating any required constraints by which the system must abide.
5. A blueprint for completing a project with as little cost growth as possible.
6. Detailed functional and non-functional requirements. it DOES NOT contain any design suggestions.

Purpose of the SRS is to

1. Give the customer assurance that the development organization understands the issues or problems to be solved and the software behaviour necessary to address those problems.
2. Decomposes the problem into component parts.
3. Serve as an input to the design specification so must contain sufficient detail in the functional requirements so that a design solution can be devised.
4. Serve as the parent or master document for testing and validation strategies that will be applied to the requirements for verification in the testing stage.

How and Who should write the SRS?

1. It should be written in **natural language**, in an **unambiguous** manner that may also include diagrams as necessary.
2. At the same time the detailed functional and non-functional **requirements need to be defined and understood** by the development team.
3. **Development team** members (programmers and managers) are **seldom the best writers of formal documents**. However, they often have to write it!
4. It is good practice to involve a **technical writer** with the development team in the drafting of the SRS because they are usually better at assessing and planning documentation projects and better meet customer document needs.

Contents of the SRS

1. Each organisation will have its own template for an SRS.
2. There are key content that must be addressed in an SRS
3. There is an IEEE standard
830-1998 - IEEE Recommended Practice for Software Requirements Specifications.

This is a document that you can download from IEEEXplore.

Contents of the SRS

- 1) Product Description**
 - a) Purpose of the System (mission statement)**
 - b) Scope of the System**
 - c) Users and Stakeholders**
 - d) Assumptions and Constraints**
- 2) Functional Requirements**
- 3) Non-Functional Requirements**
- 4) Interface Requirements**
 - a) User**
 - b) Hardware**
 - c) Software**
- 5) Data Dictionary**

See SRS Template on NTULearn course website.

Contents of the SRS

Give Project Mission Statement and state the purpose of the system in terms of what it does from a high level perspective. This should not be a long statement. 100-200 words should be sufficient.

- 1) **Product Description**
 - a) **Purpose of the System**
 - b) **Scope of the System**
 - c) **Users and Stakeholders**
 - d) **Assumptions and Constraints**
-
- The diagram consists of four red arrows originating from the text descriptions on the right and pointing to the corresponding items in the SRS contents list on the left. The first arrow points from the 'Identify the system...' text to 'a) Purpose of the System'. The second arrow points from the 'Identify all the people...' text to 'c) Users and Stakeholders'. The third arrow points from the 'Identify the system...' text to 'd) Assumptions and Constraints'. The fourth arrow points from the 'Identify the system...' text to 'b) Scope of the System'.

Include any information that may affect the SRS.

E.g. assuming that any external interfaces are fully specified and will not change, regulations that have to be met.....

Contents of the SRS

2) Functional Requirements

State the functional requirements in precise numbered statements and use

- *Activity Diagrams*
- *Use Case Diagrams with Use Case Descriptions*
- *Class Diagrams*
- *Sequence Diagrams*
- *Communication Diagrams*
- *State Machine Diagrams and Prototype UI as appropriate*

Contents of the SRS

3) Non-Functional Requirements



State the non-functional requirements in precise numbered statements covering any necessary

- *Performance requirements*
- *Any standards requirements*
- *Reliability*
- *Availability*
- *Security*
- *Maintainability*
- *Portability*

Contents of the SRS

4) Interface Requirements

- a) User
- b) Hardware
- c) Software

e.g. required screen formats, page or window layouts, content of any reports or menus, or availability of programmable function keys

e.g. number of ports, instruction sets, etc.

e.g., interfaces to a data management system, an operating system, a mathematical package

Contents of the SRS

5) Data Dictionary



Unambiguously defines all terms, phrases and abbreviations used in the SRS

Language quality characteristics of an SRS

1. **Correct.** Each requirement must accurately describe the functionality to be delivered. Only user representatives can determine the correctness of user requirements, which is why it is essential to include them in inspections of the requirements.
2. **Feasible.** It must be possible to implement each requirement within the known capabilities and limitations of the system and its environment.
3. **Necessary.** Each requirement should document something the customers really need or something that is required for conformance to an external requirement, an external interface, or a standard. If you cannot identify the origin of the requirement, perhaps the requirement is not really necessary.
4. **Unambiguous.** The reader of a requirement statement must be able to draw only one interpretation of it. Also, multiple readers of a requirement must arrive at the same interpretation.
5. **Verifiable.** See whether you can devise tests or use other verification approaches, such as inspection or demonstration, to determine whether each requirement is properly implemented in the product.

Language quality characteristics of an SRS

6. **Complete.** No requirements or necessary information should be missing. It is hard to spot missing requirements because they aren't there. Organize the requirements hierarchically in the SRS to help reviewers understand the structure of the functionality described, so it will be easier for them to tell if something is missing.
7. **Consistent.** Consistent requirements do not conflict with other software requirements or with higher level (system or business) requirements. Disagreements among requirements must be resolved before development can proceed.
8. **Modifiable.** You must be able to revise the SRS when necessary and maintain a history of changes made to each requirement. This means that each requirement be uniquely labeled and expressed separately from other requirements so you can refer to it unambiguously.
9. **Traceable.** You should be able to link each software requirement to its source, which could be a higher-level system requirement, a use case, or a voice-of-the-customer statement.

Good words/phrases to use in an SRS

Shall

Used to dictate the provision of a functional capability.

Must or Must not

Most often used to establish performance requirement or constraints.

Is required to

Used as an imperative in SRS statements.

Are applicable

Used to include, by reference, standards, or other documentation as an addition to the requirement being specified.

Responsible for

Used as an imperative in SRSs that are written for systems with pre-defined architectures.

Will

Used to cite things that the operational or development environment is to provide to the capability being specified.

Words/phrases NOT to use in an SRS

Avoid these words/phrases like the following as they can create uncertainty

- adequate**
- as a minimum**
- be capable of...**
- as applicable**
- as appropriate**
- effective**
- if possible**
- if practical**
- timely**
- normal**

Summary: Requirements Specification Activities

