

**Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering**

**OBJECT ORIENTED SOFTWARE ENGINEERING
Chapter One**

Introduction to software and software engineering

**by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.**

Course Outline → 5 hours, 9 Marks

- 1.1. Introduction to software and software engineering*
- 1.2. Software process and software process model*
- 1.3. Agile software development*
- 1.4. Requirement engineering process*
- 1.5. System modeling*
- 1.6. Software prototyping*
- 1.7. Object oriented software development*

*Software and
software
engineering*

Software and Software engineering

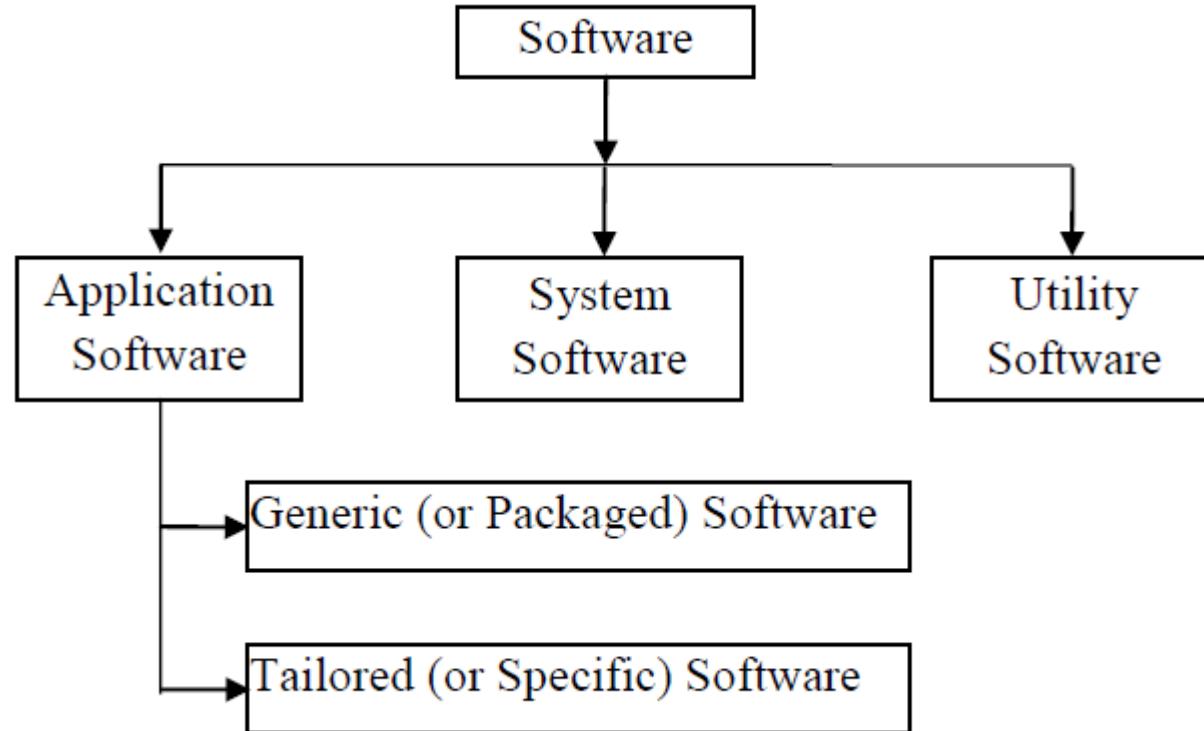
- ❖ The term '**Software**' is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product**.

- ❖ The other term '**Engineering**' is all about developing some quality products by using some well-defined, scientific principles and standard within the specified budget and time.

Software and Software engineering

- ❖ Hence, Software engineering is defined as: It is a branch of computer science which is associated with development of quality software product using well-defined scientific principles, methods and procedures.
- ❖ The outcome of software engineering is an efficient and reliable software product that support over the long term.

Software types



1. Application Software

- ❖ It can be divided into Generic and Specific

a. Generic (or Packaged) Software

- ❖ The application software which is designed to fulfill the needs of large group of users is known as generic or packaged software.

Example: MS-Word, Adobe Reader, MS-Excel.

b. Tailored (or Specific) Software

The application software which is designed to fulfill the needs of a particular user/company/organization is known as tailored or specific software.

Example: Software used in department stores, hospitals, schools etc.

2. System Software:-

- ❖ The software which can directly control the hardware of the computer are known as system software.

Example: Video driver, audio driver.

3. Utility Software:-

- ❖ Small software that usually performs some useful tasks is known as utility software.

Example: Win Zip, JPEG Compressor, PDF Merger, PDF to Word Converter etc.

Software process & software process model

1.2. Software process and software process model

- ❖ Since the **prime objective** of **software engineering** is to develop methods for large systems, which produce **high quality** software at **low cost** and in **reasonable time**.
- ❖ So it is essential to perform software development **in phases**. This **phased development of software** is often referred to as software development life cycle (**SDLC**) or software process.
- ❖ And the models used to achieve these goals are termed as **Software Process Models**.

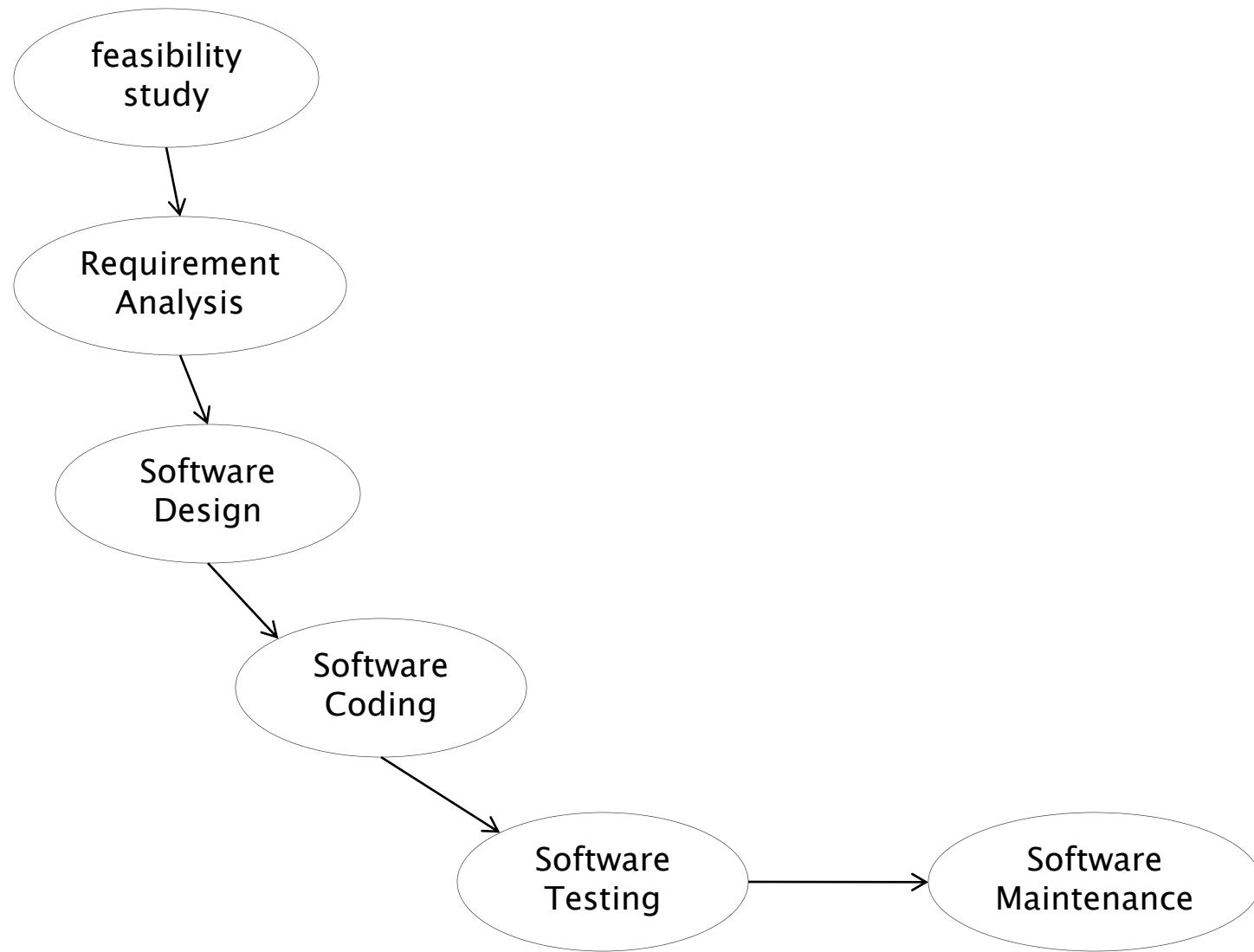


Fig 1 software development process

In **fig 1**,

- ❖ These phases **work in top to bottom** approach.
- ❖ The phases take inputs from the previous phases, add features, and then produce outputs.

1. Feasibility study/ Preliminary investigation:

- ❖ Feasibility study decides whether the new system should be developed or not.
- ❖ There are three constraints, which decides the go or no-go decision.
 - a. **Technical:**
 - ❖ determines whether technology needed for proposed system is available or not.
 - ❖ determines whether the existing system can be upgraded to use new technology
 - ❖ whether the organization has the expertise to use it or not.

b. Time:

- ❖ determines the time needed to complete a project.
- ❖ Time is an important issue as cost increases with an increase in the time period of a project.

c. Budget:

- ❖ This evaluation looks at the financial aspect of the project.
- ❖ determines whether the investment needed to implement the system will be recovered at later stages or not.

2. Requirement Analysis/Software Analysis:

- ❖ Studies the problem or requirements of software in detail.
- ❖ After analyzing and elicitations of the requirements of the user, a requirement statement known as **software requirement specification (SRS)** is developed.

3. Software Design:

- ❖ Most crucial phase in the development of a system. The SDLC process continues to move from the what questions of the analysis phase to the how.
- ❖ Logical design is turned into a physical design.
- ❖ Based on the user requirements and the detailed analysis the system must be designed.
- ❖ Input, output, databases, forms, processing specifications etc. are drawn up in detail.
- ❖ Tools and techniques used for describing the system design are: Flowchart, ERD, Data flow diagram (DFD), UML diagrams like Use case, Activity, Sequence etc.

4. Software Coding:

- ❖ Physical design into software code.**
- ❖** Writing a software code requires a prior knowledge of programming language and its tools. Therefore, it is important to choose the appropriate programming language according to the user requirements.
- ❖** A program code is efficient if it makes optimal use of resources and contains minimum errors.

5. Software Testing:

- ❖ **Software testing is performed to ensure that software produces the correct outputs i.e. free from errors.** This implies that outputs produced should be according to user requirements.
- ❖ Efficient testing improves the quality of software.
- ❖ Test plan is created to test software in a planned and systematic manner.

6. Software Maintenance:

- ❖ This phase comprises of a set of software engineering activities that occur after software is delivered to the user.
- ❖ After the software is developed and delivered, it may require changes. Sometimes, changes are made in software system when user requirements are not completely met.
- ❖ To make changes in software system, software maintenance process evaluates, controls, and implements changes.

Class Work

Mention different phases of Software Development life cycle(SDLC), if you are under the project of **Library Management system.**

What is Software process?

When you work to build a product or system, it's important to go through a series of predictable steps—a **road map** that helps you to create a timely, high-quality result. The road map that you follow is called a “software process.”

Who does it?

Software engineers and their managers adapt the process to their needs and then follow it. In addition, the people who have requested the software have in the process of defining, building, and testing it.

Why is it important?

Because it provides path, stability, control over your project.

What are the steps?

At a detailed level, **the process that you adopt depends on the software that you're building.** One process might be appropriate for creating software for an aircraft avionics system, while an entirely different process would be indicated for the creation of a website.

from a technical point of view:

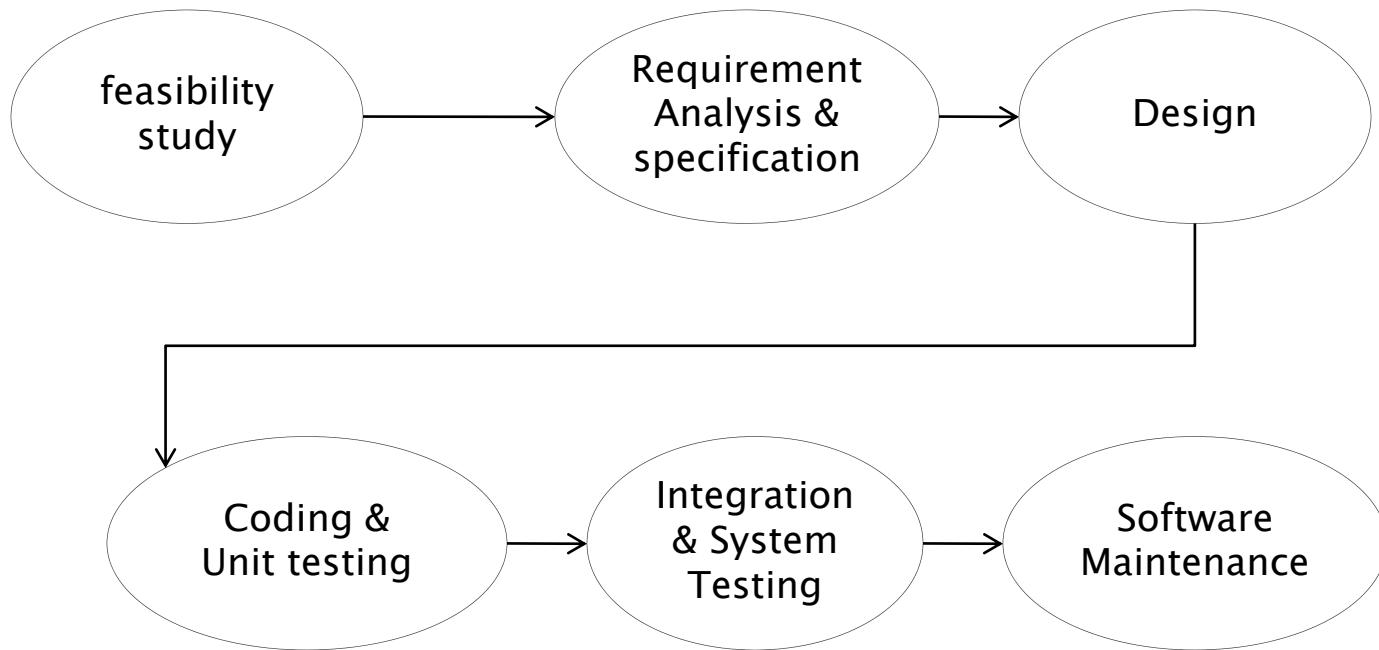
A **software process** is a **framework** for the activities, actions, and tasks that are required to build high-quality software-

Roger S. Pressman

*software process
model cont...*

1. Waterfall model

Waterfall Model



i. Feasibility study

- ✓ Technical
- ✓ Financial
- ✓ Time etc.

ii. Requirement specification

To specify the requirements' users specification should be clearly understood and the requirements should be analyzed. This phase involves **interaction between user and software engineer**, and produces a document known as software requirement specification (SRS).

iii. Design

Determines the detailed process of developing software after the requirements are analyzed. It utilizes software requirements defined by the user and translates them into a software representation. In this phase, the emphasis is on finding a solution to the problems defined in the requirement analysis phase. The software engineer, in this phase is mainly concerned with the data structure, algorithmic detail, and interface representations.

iv. Coding

Emphasizes on translation of design into a programming language using the coding style and guidelines. The programs created should be easy to read and understand. All the programs written are documented according to the specification.

v. Testing:

Ensures that the product is developed according to the requirements of the user. Testing is performed to verify that the product is functioning efficiently with minimum errors. It focuses on the internal logics and external functions of the software

vi. Implementation and maintenance

Delivers fully functioning operational software to the user. Once the software is accepted and deployed at the user's end, various changes occur due to changes in external environment (these include upgrading new operating system or addition of a new peripheral device). The changes also occur due to changing requirements of the user and the changes occurring in the field of technology. This phase focuses on modifying software, correcting errors, and improving the performance of the software.

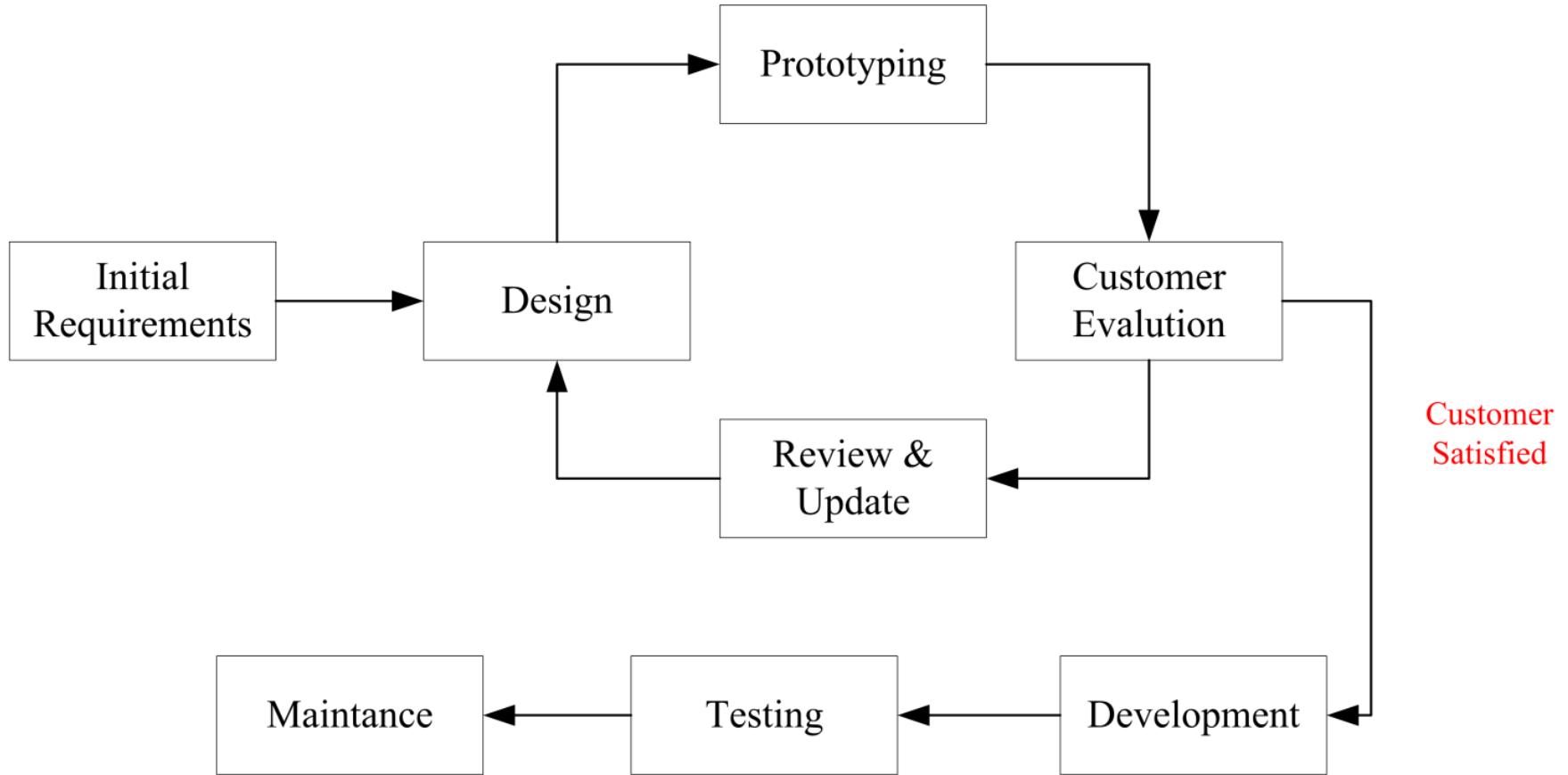
Waterfall model

Advantages	Disadvantages
<ul style="list-style-type: none">▪ Relatively simple to understand.▪ Each phase of development proceeds sequentially.▪ Allows managerial control where a schedule with deadlines is set for each stage of development.▪ Helps in controlling schedules, budgets, and documentation.	<ul style="list-style-type: none">▪ Requirements need to be specified before the development proceeds.▪ The changes of requirements in later phases of the waterfall model cannot be done. This implies that once an application is in the testing phase, it is difficult to incorporate changes at such a late phase.▪ No user involvement and working version of the software is available when the software is developed.▪ Does not involve risk management.▪ Assumes that requirements are stable and are frozen across the project span.

2. Prototype model

b. prototype model

- ❖ The prototyping model is applied when there is an absence of detailed information regarding input and output requirements in the software.
- ❖ It is used if the requirements are **not** previously specified.
- ❖ Prototyping model **increases flexibility** of the development process by allowing the user to interact and experiment with a **working representation of the product** known as **prototype**. A prototype gives the user an actual feel of the system.



Prototype model

i. Requirements gathering and analysis

Prototyping model begins with requirements analysis, and the requirements of the system are defined in detail. The **user** is interviewed to know the requirements of the system.

ii. Quick design

When requirements are known, a preliminary design or a quick design for the system is created. It is **not a detailed design**, however, it includes the important aspects of the system, which gives an idea of the system to the user. Quick design helps in developing the prototype.

iii. Build prototype

Information gathered from quick design is modified to form a prototype. The first prototype of the required system is developed from quick design. It **represents a ‘rough’ design of the required system.**

iv. User evaluation

Next, the proposed system is presented to the user for consideration as a part of development process. The users thoroughly evaluate the prototype and recognize its strengths and weaknesses, such as what is to be added or removed. **Comments and suggestions are collected from the users and are provided to the developer.**

v. Prototype refinement

Once the user evaluates the prototype, it is refined according to the requirements. The developer revises the prototype to make it more effective and efficient according to the user requirement. If the user is not satisfied with the developed prototype, a new prototype is developed with the additional information provided by the user. The new prototype is evaluated in the same manner as the previous prototype, process continues until all the requirements specified by the user are met. Once the user is satisfied a final system is developed.

vi. Engineer product

Once the requirements are completely known, user accepts the final prototype. The **final system is thoroughly evaluated and tested followed by routine maintenance on continuing basis to prevent large-scale failures and to minimize downtime.**

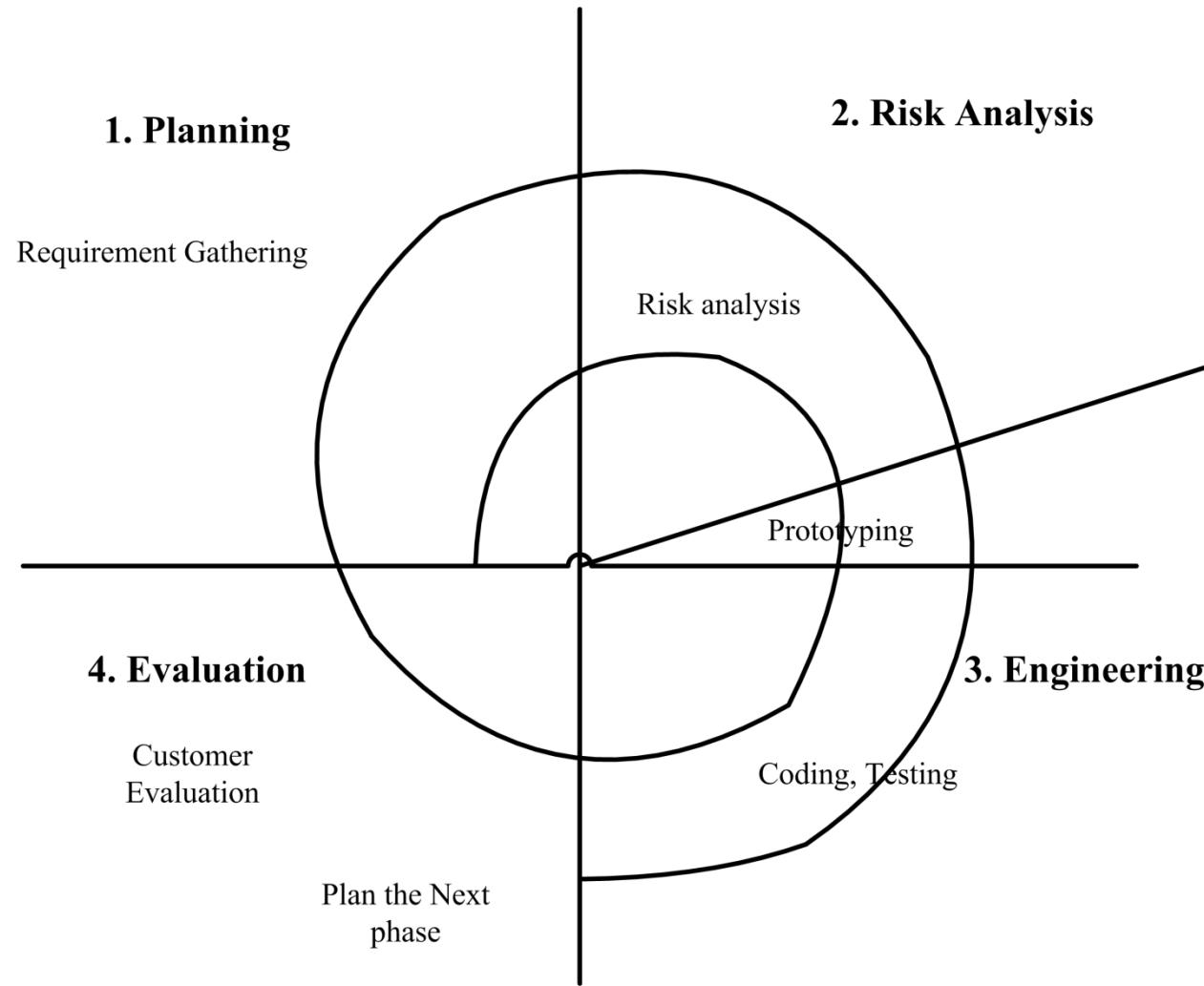
Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Provides a working model to the user early in the process, enabling early assessment and increasing user confidence. ▪ Developer gains experience and insight by developing a prototype, thereby resulting in better implementation of requirements. ▪ Prototyping model serves to clarify requirements, which are not clear, hence reducing ambiguity and improving communication between developer and user. ▪ There is a great involvement of users in software development. Hence, the requirements of the users are met to the greatest extent. ▪ Helps in reducing risks associated with the project. 	<ul style="list-style-type: none"> ▪ If the user is not satisfied by the developed prototype, then a new prototype is developed. This process goes on until a perfect prototype is developed. Thus, this model is time consuming and expensive. ▪ Developer loses focus of the real purpose of prototype and compromise with the quality of the product. For example, they apply some of the inefficient algorithms or inappropriate programming languages used in developing the prototype. ▪ Prototyping can lead to false expectations. It often creates a situation where user believes that the development of the system is finished when it is not. ▪ The primary goal of prototyping is rapid development, thus, the design of system can suffer as it is built in a series of layers without considering integration of all the other components.

3. Spiral Model

C. Spiral Model

In 1980's Boehm introduced a process model known as spiral model. The spiral model comprises of activities organized in a spiral, which has many cycles. This model combines the features of prototyping model and waterfall model and is advantageous for large, complex and expensive projects which involves high risk.

C. Spiral Model



1. Each cycle of the **first quadrant** commences with identifying the goals for that cycle. In addition, it **determines other alternatives, which are possible in accomplishing those goals.**
2. Next step in the cycle evaluates alternatives based on objectives and constraints. This process **identifies the project risks.** Risk signifies that there is a possibility that the objectives of the project cannot be accomplished. If so the formulation of a cost effective **strategy for resolving risks is followed.** the strategy, which includes prototyping, simulation, benchmarking..

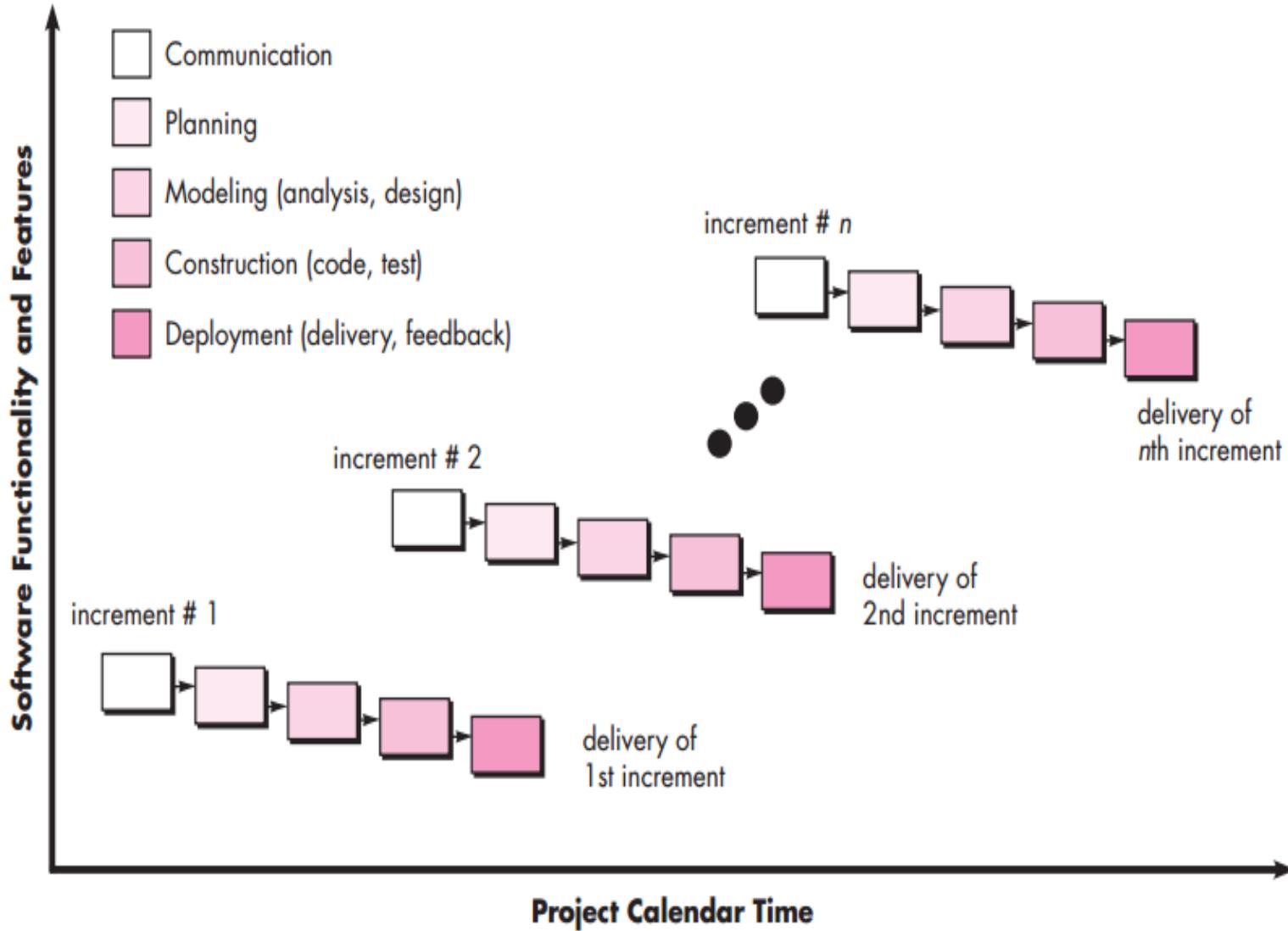
3. The development of the software depends on remaining risks. The third quadrant **develops the final software while considering the risks that can occur**. Risk management considers the time and effort to be devoted to each project activity, such as planning, configuration management, quality assurance, verification, and testing.
4. The last quadrant plans the next step, and includes planning for the next prototype and thus,comprises of requirements plan, development plan, integration plan, and test plan

Advantages	Disadvantages
<ul style="list-style-type: none"> ▪ Avoids the problems resulting in risk-driven approach in the software. ▪ Specifies a mechanism for software quality assurance activities. ▪ Spiral model is utilised by complex and dynamic projects. ▪ Re-evaluation after each step allows changes in user perspectives, technology advances or financial perspectives. ▪ Estimation of budget and schedule gets realistic as the work progresses. 	<ul style="list-style-type: none"> ▪ Assessment of project risks and its resolution is not an easy task. ▪ Difficult to estimate budget and schedule in the beginning, as some of the analysis is not done until the design of the software is developed.

4. Evolutionary model

d. Evolutionary model

- ❖ An Evolutionary model **breaks up an overall solution into increments** of functionality and develops each increment individually.
- ❖ The evolution model **divides** the development cycle into **smaller, "Incremental Waterfall Model"** in which users are able to get access to the product at the end of each cycle.
- ❖ The users provide feedback on the product for planning stage of the next cycle and the development team responds, often by changing the product, plans or process.



5. RAD model

RAD model

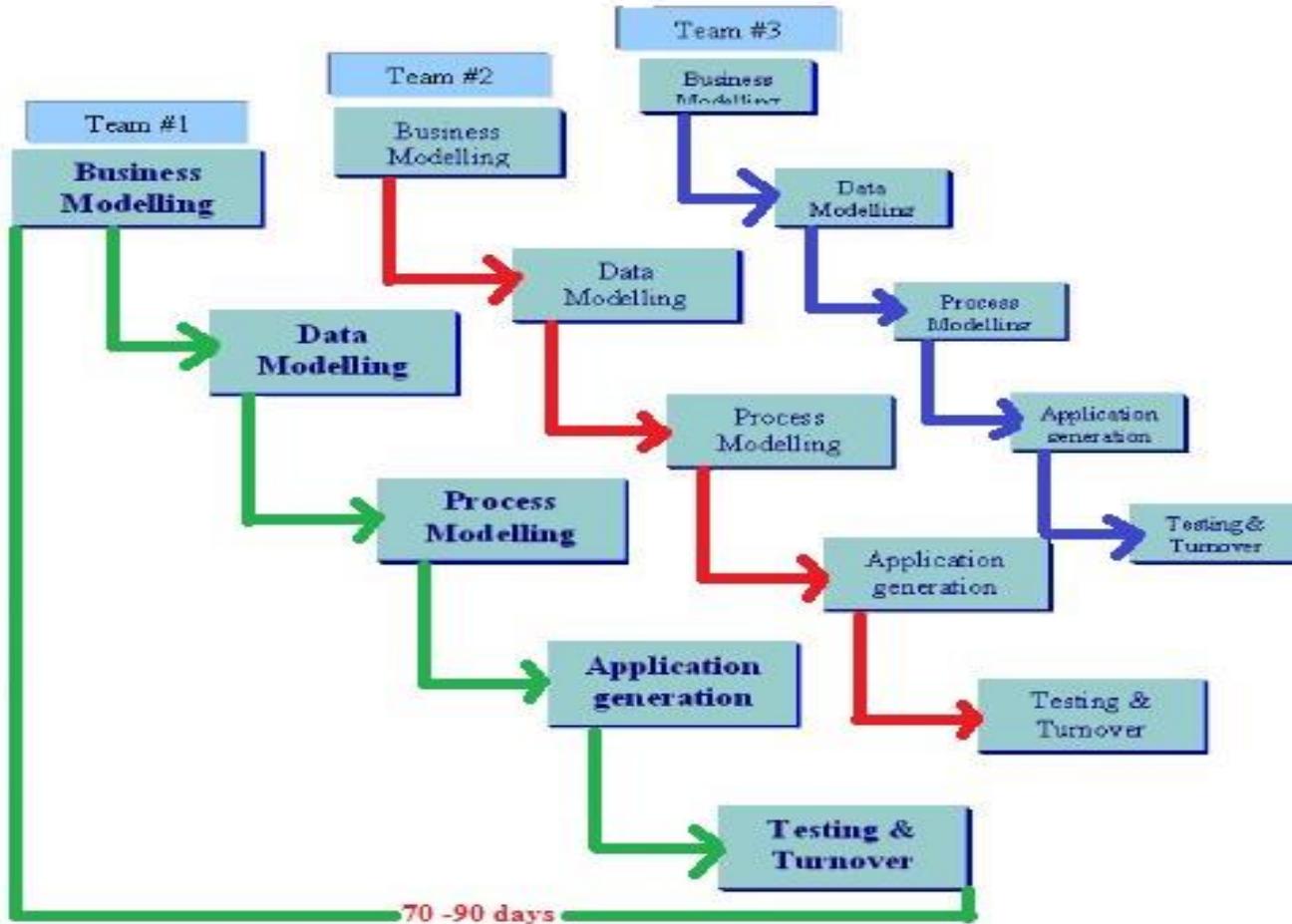
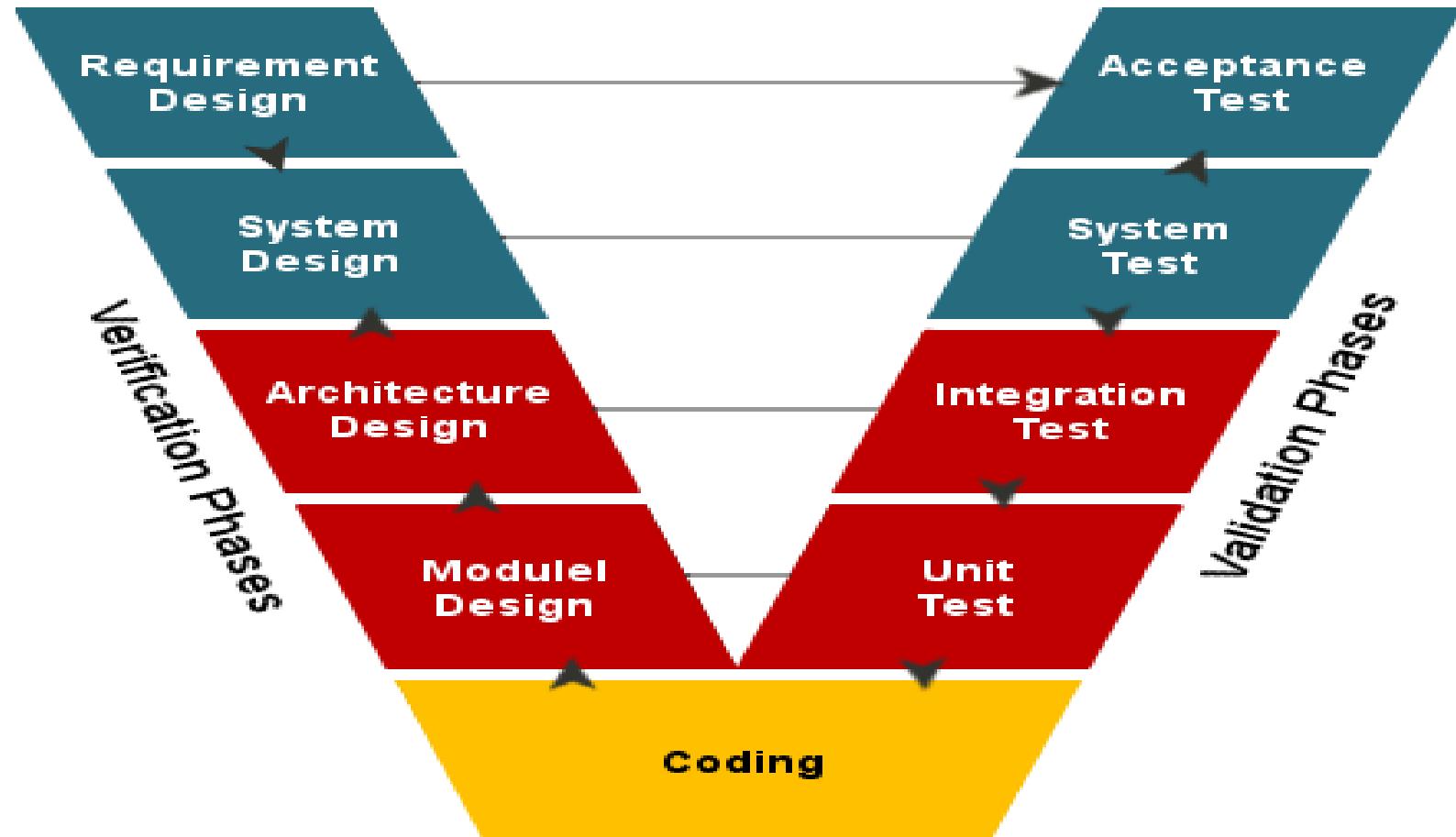


Fig:- RAD (Rapid Application Development) Model

6. V model

V model



5. Agile Software development

Presentation topic → Group 1

Requirement Engineering Process

Requirements Engineering Process

- ❖ The requirements engineering (RE) process is a series of activities that are performed in requirements phase in order to express requirements in **software requirements specification (SRS) document**.
- ❖ These **steps include** feasibility study, requirements elicitation, requirements analysis, requirements specification, requirements validation, and requirement management.

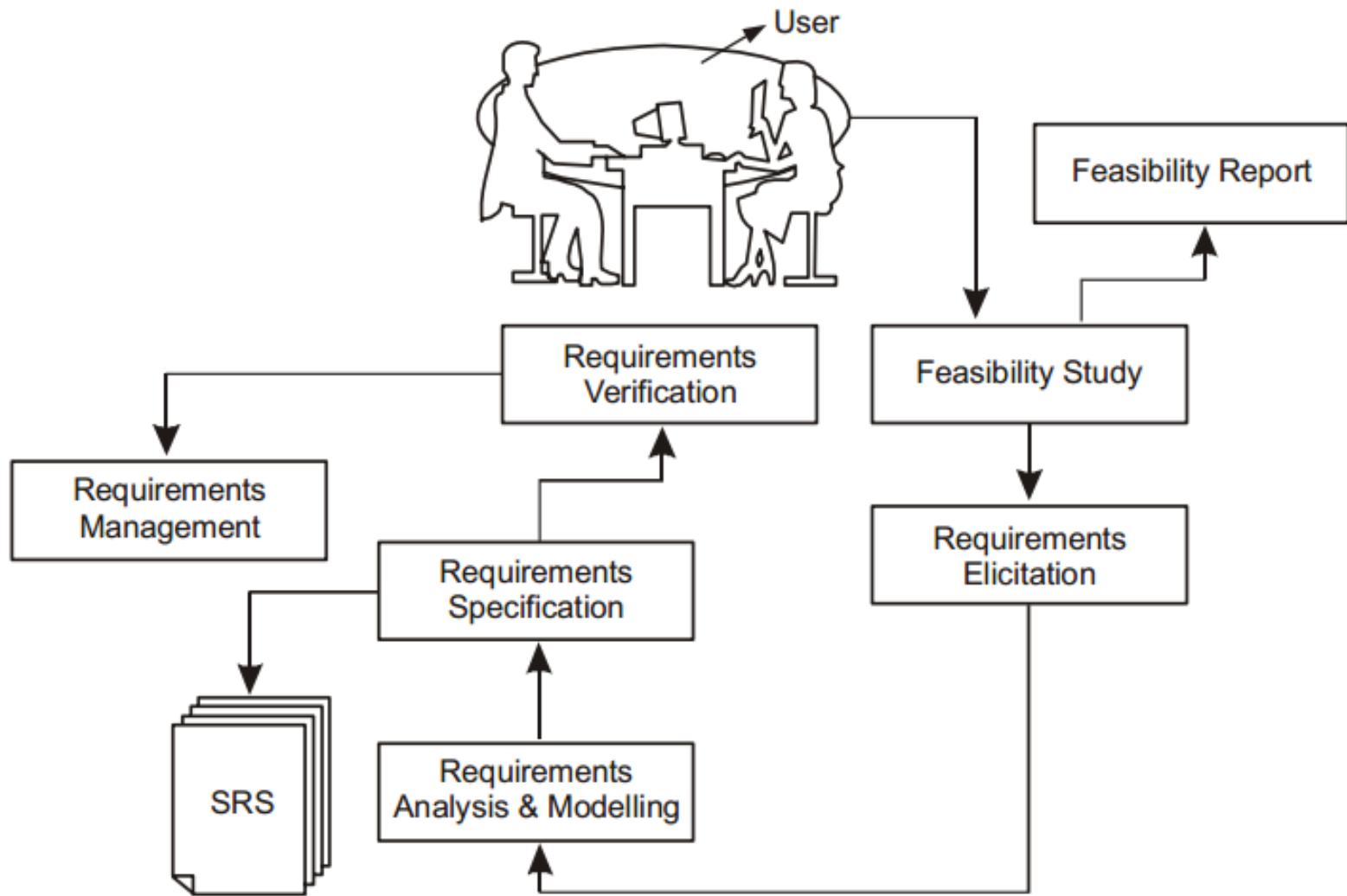


Fig: Requirement engineering process

STEP 1: FEASIBILITY STUDY

Objectives of feasibility study:

- ❖ To determine whether the software can be implemented using current technology and within the specified budget and schedule or not.
- ❖ To determine whether the software can be integrated with other existing software or not.
- ❖ To minimizes project failure.

Types of feasibility study

Technical

- ✓ technical skills and capabilities of development team.
- ✓ Assure that the technology chosen, has large number of users so that they can be consulted when problems arise.

Operational

- ✓ solution suggested by software development team is acceptable or not.
- ✓ whether users will adapt to new software or not.

Economic feasibility/ Budget

- ✓ whether the required software is capable of generating financial gains for an organization or not.
- ✓ cost incurred on software development team
- ✓ estimated cost of hardware and software.
- ✓ cost of performing feasibility study.

Time

- ✓ Whether the project will be completed on pre-specified time or not.

Feasibility Study Process

1. Information assessment:

- ❖ verifies that the system can be implemented using new technology and within the budget.

2. Information collection:

- ❖ **Specifies the sources** from where information about software can be obtained.
- ❖ **Sources:**
 - ✓ users (who will operate the software)
 - ✓ organization (where the software will be used).
 - ✓ software development team (who understands user requirements and knows how to fulfill them in software).

3. Report writing:

- ❖ Information about changes in software scope, budget, schedule, and suggestion of any requirements in the system.

Step 2: Requirements Elicitation

- ❖ **Process of collecting information** about software requirements from different stakeholders (users, developer, project manager etc.)

Issues in Requirement Elicitation

1. Problems of understanding

- ✓ Users are not certain about their requirements and thus are unable to express what they require in software and which requirements are feasible.
- ✓ This problem also arises when users have no or little knowledge of the problem domain and are unable to understand the limitations of computing environment of software.

2. Problems of volatility

- ✓ This problem arises when **requirements change over time**.

Elicitation Techniques

The commonly followed elicitation techniques are listed below:

1. Interviews:

- ❖ Ways for eliciting requirements, it helps software engineer, users, & development team to understand the problem and suggest solution for the problem.
- ❖ **An effective interview should have characteristics listed below:**
 - ✓ Individuals involved in interviews should be able to accept new ideas, focus on listening to the views of stakeholders & avoid biased views. *f*
 - ✓ Interviews should be conducted in defined context to requirements rather than in general terms. E.g. a set of a questions or a *requirements proposal*.

2.Scenarios:

- ❖ Helps to determine possible future outcome before implementation.
- ❖ In Generally, a scenario comprises of:
 - ✓ Description of **what** users expect when scenario starts.
 - ✓ Description of **how** to handle the situation when software is not operating correctly.
 - ✓ Description of the state of software **when** scenario ends.

3.Prototypes:

- ❖ helps to clarify **unclear requirements**.
- ❖ helps users to **understand the information they need to provide** to software development team.

4.Quality function deployment (QFD) →Assignment (3)

Step 3: Requirement Analysis

It is the process of studying and refining requirements

Tasks performed in requirements analysis are:

- ❖ Understand the problem for which software is to be developed.
- ❖ Develop analysis model to analyze the requirements in the software.
- ❖ Detect and resolve conflicts that arise due to unclear and unstated requirements.
- ❖ Determine operational characteristics of software and how it interacts with its environment.

Step 4: Requirements Specification

Development of SRS document (software requirement specification document).

Characteristics of SRS

1. Correct:

SRS is correct when

- ✓ all user requirements are stated in the requirements document.
- ✓ The stated requirements should be according to the desired system.

2. Unambiguous:

- ✓ SRS is unambiguous when every stated requirement has only one interpretation i.e. each requirement is uniquely interpreted.

3. Complete:

- ✓ SRS is complete when the requirements clearly define what the software is required to do.

4. Modifiable:

- ✓ The requirements of the user can change, hence, requirements document should be created in such a manner where those changes can be modified easily.

5. Ranked for importance and stability:

- ✓ All requirements are not equally important.

6. Verifiable:

- ✓ SRS is verifiable when the specified requirements can be verified with a cost-effective process to check whether the final software meets those requirements or not.

7. Consistent:

- ✓ SRS is consistent when the individual requirements defined does not conflict with each other.
- ✓ e.g., a requirement states that an event ‘a’ is to occur before another event ‘b’. But then another set of requirements states that event ‘b’ should occur before event ‘a’.

8. Traceable:

- ✓ SRS is traceable when the source of each requirement is clear and it facilitates the reference of each requirement in future.

1.0	Introduction
1.1	Purposes
1.2	Scope
1.3	Definitions, Acronyms, and Abbreviations
1.4	References
1.5	Overview
2.0	The Overall Description
2.1	Product Perspective
2.1.1	System Interface
2.1.2	Interface
2.1.3	Hardware Interface
2.1.4	Software Interface
2.1.5	Communications Interface
2.1.6	Memory Constraints
2.1.7	Operations
2.1.8	Site Adaptation Requirements
2.2	Product Functions
2.3	User Characteristics
2.4	Constraints
2.5	Assumptions and Dependency
2.6	Apportioning of Requirements
3.0	Specific Requirements
3.1	External Interface
3.2	Functions
3.3	Performance Requirements
3.4	Logical Database of Requirement
3.5	Design Constraints
3.5.1	Standards Compliance
3.6	Software System Attributes
3.6.1	Reliability
3.6.2	Availability
3.6.3	Security
3.6.4	Maintainability
3.6.5	Portability
3.7	Organizing the Specific Requirements
3.7.1	System Mode
3.7.2	User Class
3.7.3	Objects
3.7.4	Feature
3.7.5	Stimulus
3.7.6	Response
3.7.7	Functional Hierarchy
3.8	Additional Comments
4.0	Change Management Process
5.0	Document Approvals
6.0	Supporting Information

Fig : SRS Document template

Step 5 : Requirements Validation

Why Validation ?

- ❖ Errors present in the SRS will adversely affect the cost if they are detected later in the development process or when the software is delivered to the user.

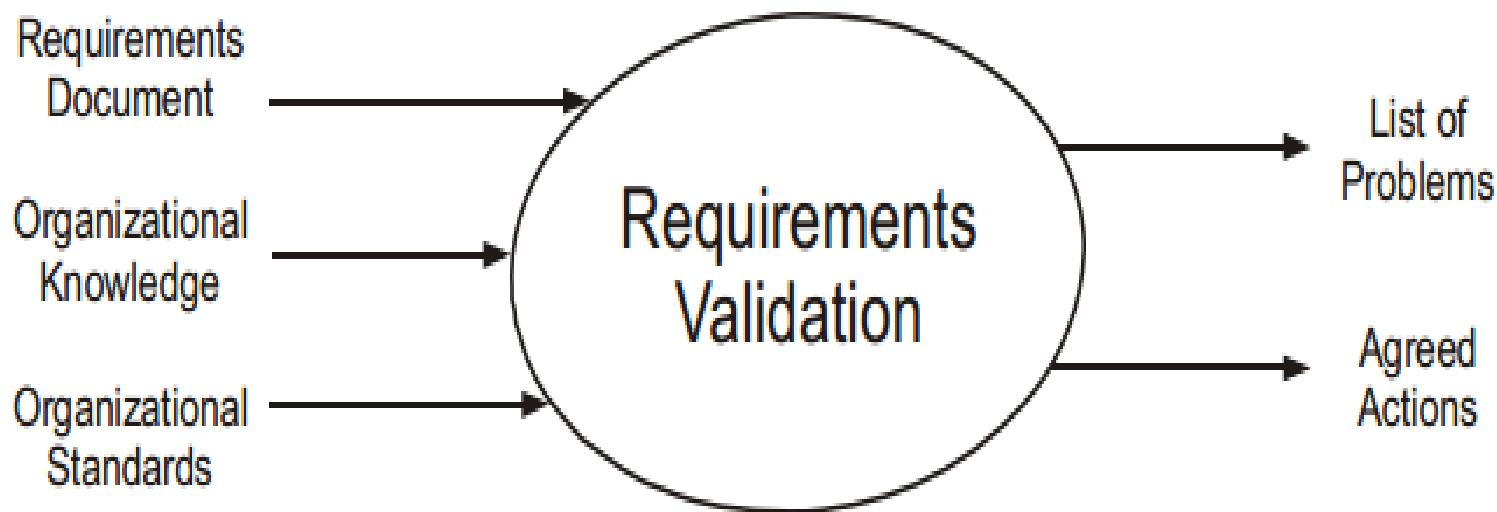


Fig: Requirement Validation

Step 6: Requirements Management

Why ??

- ❖ To understand and control changes to system requirements.

Advantages of requirements management

Better control of complex projects:

- ✓ Provides overview to development team with a clear understanding of what, when and why software is to be delivered.

Improves software quality:

- ✓ Ensures that the software performs according to requirements to enhance software quality.

Reduced project costs and delays:

- ✓ Minimizes errors early in the development cycle, as it is expensive to ‘fix’ errors at the later stages of the development cycle. As a result, the project costs also reduced.

Improved team communications:

- ✓ Facilitates early involvement of users to ensure that their needs are achieved.

Requirements Management Process

- ❖ Requirements management starts with **planning**,
- ❖ Then, each requirement is assigned a unique ‘identifier’ so that it can be crosschecked by other requirements. Once requirements are identified, requirements **tracing** is performed.
- ❖ The **objective** of requirement **tracing** is to ensure that all the requirements are well understood and are included in test plans and test cases.
- ❖ Traceability information is stored in a **traceability matrix**, which relates requirements to stakeholders or design module. **Traceability matrix** refers to a **table that correlates requirements**.

Req. ID	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		U	R					
1.2			U			R		U
1.3	R			R				
2.1			R		U			U
2.2								U
2.3		R		U				
3.1								R
3.2							R	

U → dependency
R → weaker Relationship

Requirements change management

- ❖ It is used when there is a request or proposal for a change to the requirements.



Fig: Requirement change management

Software Prototyping

Software prototyping

- ❖ A prototype is a version of a software product developed in the early stages of the product's life cycle for specific experimental purposes.
- ❖ A prototype enables you to fully understand how easy or difficult it will be to implement some of the features of the system.
- ❖ It also can give users a chance to comment on the usability and usefulness of the user interface design and it lets you assess the fit between the software tools selected, the functional specification, and the users' needs.

Software prototyping

The purpose of the prototype review is threefold:

1. To demonstrate that the prototype has been developed according to the specification and that the final specification is appropriate.
2. To collect information about errors or other problems in the system, such as user interface problems that need to be addressed in the intermediate prototype stage.
3. To give management and everyone connected with the project the first (or it could be second or third ...) glimpse of what the technology can provide.

System Modeling

System Modeling

- ❖ System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- ❖ It is about representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
- ❖ Models help the analyst to understand the functionality of the system; they are used to communicate with customers.

System Modeling

Models can explain the system from different perspectives:

- ❖ An external perspective, where you model the context or environment of the system.
- ❖ An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
- ❖ A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- ❖ A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

System Modeling

Five types of UML diagrams that are the most useful for system modeling:

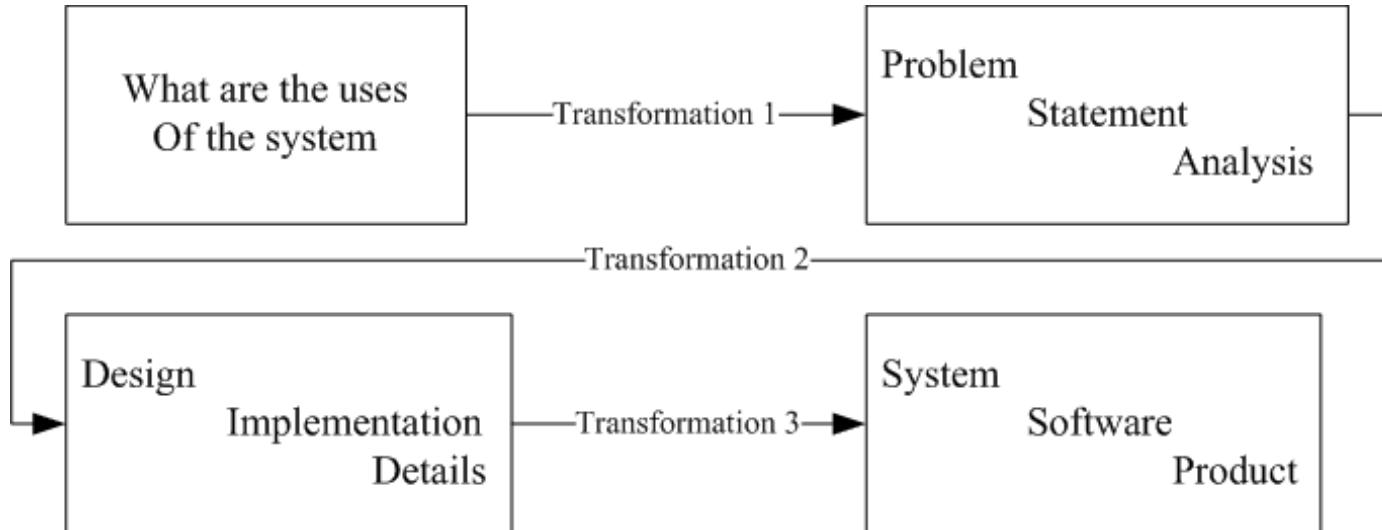
- ❖ **Activity diagrams**, which show the activities involved in a process or in data processing.
- ❖ **Use case diagrams**, which show the interactions between a system and its environment.
- ❖ **Sequence diagrams**, which show interactions between actors and the system and between system components.
- ❖ **Class diagrams**, which show the object classes in the system and the associations between these classes.
- ❖ **State diagrams**, which show how the system reacts to internal and external events.

Object Oriented Software Development

Object Oriented Software Development Cycle

- ❖ Object oriented systems development is a way to develop software by building self – contained (independent) modules or objects that can be easily replaced, modified and reused.
- ❖ In an object–oriented environment, software or application is a collection of discrete/separate objects that encapsulate their data as well as the functionality.
- ❖ It works by allocating tasks among the objects of the application.
- ❖ The software development approach for object oriented modeling goes through following stages:
 - © Analysis
 - © Design
 - © Implementation and Testing

Object Oriented Software Development Cycle



❖ Transformation 1 (analysis):

- Translates the users' needs into system requirements & responsibilities.

❖ Transformation 2 (design):

- Begins with a problem statement and ends with a detailed design that can be transformed into an operational system.

❖ Transformation 3 (implementation)

- Refines the detailed design into the system deployment that will satisfy the users' needs.
- It represents embedding s/w product within its operational environment.

**Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering**

**OBJECT ORIENTED SOFTWARE ENGINEERING
Chapter TWO**

Object oriented concepts and modeling

**by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.**

Course Outline

1.1. Object oriented concepts

1.2. Object oriented system development

1.3. Identifying the elements of an object model

Object oriented concepts

Class

- ❖ It is a user-defined data type, which has data members and member functions.
- ❖ Data members are the data variables and member functions are the functions or operations used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- ❖ For example, Consider the Class of '**Car**'. There may be many cars with different names and brand but all of them will share some common **properties** like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.
- ❖ In the example of class **Car**, the **data member (properties)** will be speed limit, mileage etc and **member functions (operations)** can be apply brakes, increase speed etc.

Object

- ❖ An object is an instance of a class.
- ❖ **Object** is a bundle of data and its behaviour (often known as methods).
- ❖ An Objects have two characteristics: **states** and **behaviors**.

→ *For e.g. an object 'House' has:*

State: Address, Color, Area

Behavior: Open door, close door

So if I had to write a class based on states and behaviours of House. I can do it like this: States can be represented as instance variables and behaviours as methods of the class.

Class and Object in C++

```
class House
{
    char Address[20];
    char Color;
    char Area[20];
public:
    void Open door(){}
    Void Close door(){}
};

int main()
{
    House h1; // h1 is a object (instance of class House)
}
```

→ Let's take another example of an object '**Car**':

State: Color, Brand, Weight, Model

Behavior: Break, Accelerate, Slow Down, Gear change.

As we have seen above, the states and behaviors of an object, can be represented by variables and methods in the class respectively.

Characteristics of Objects

- ❖ Abstraction
- ❖ Encapsulation
- ❖ Message passing

❖ **Encapsulation**

Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation.

❖ **Message passing**

A single object by itself may not be very useful. An application contains many objects. One object interacts with another object by invoking methods on that object. It is also referred to as Method Invocation.

❖ **Abstraction (detail with examples in the next slides)**

Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.

Abstraction

- ❖ One of the most fundamental concept of OOPs is Abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.
- ❖ For example, when we login to our **Facebook account**, we enter our user_id (email/PhNumber) & password and press login, what happens when we press login, how the input data sent to facebook server, how it gets verified is all abstracted away from the us.
- ❖ Another example of abstraction: A car in itself is a well-defined object, which is composed of several other smaller objects like a gearing system, steering mechanism, engine, which are again have their own subsystems. But for humans car is a one single object, which can be managed by the help of its subsystems, even if their inner details are unknown.

Encapsulation

- ❖ Encapsulation is the process of binding the data with the code that manipulates it.
- ❖ It keeps the data and the code safe from external interference
- ❖ Looking at the example of a power steering mechanism of a car. Power steering of a car is a complex system, which internally have lots of components tightly coupled together, they work synchronously to turn the car in the desired direction. It even controls the power delivered by the engine to the steering wheel. But to the external world there is only one interface is available and rest of the complexity is hidden. Moreover, the steering unit in itself is complete and independent. It does not affect the functioning of any other mechanism.

Inheritance

- ❖ Inheritance is the mechanism by which an object acquires the some/all properties of another object.
- ❖ It supports the concept of hierarchical classification.

Example:

Suppose we have a class **FourWheeler** and a sub class of it named **Car** (Car is a four wheeler vehicle so assume Car as a sub class of FourWheeler).

Here Car acquires the properties of a class FourWheeler. Also other classifications could be a jeep, tempo, van etc.

Inheritance

FourWheeler defines a class of vehicles that have four wheels, and specific range of engine power, load carrying capacity etc. Car (termed as a sub-class) acquires these properties from FourWheeler, and has some specific properties, which are different from other classifications of FourWheeler, such as luxury, comfort, shape, size, usage etc.

A car can have further classification such as an open car, small car, big car etc, which will acquire the properties from both Four Wheeler and Car, but will still have some specific properties.

This way the level of hierarchy can be extended to any level.

Polymorphism

- ❖ Polymorphism means to process objects differently based on their data type.
- ❖ In other words it means, one method with multiple implementation, for a certain class of action. And which implementation to be used is decided at runtime depending upon the situation (i.e., data type of the object)
- ❖ This can be implemented by designing a generic interface, which provides generic methods for a certain class of action and there can be multiple classes, which provides the implementation of these generic methods.

Polymorphism

Example:

Lets us look at same example of a car.

A car have a gear transmission system. It has four front gears and one backward gear. When the engine is accelerated then **depending upon which gear is engaged different amount power and movement is delivered to the car**. The action is same applying gear but based on the type of gear the action behaves differently or you can say that it shows many forms (polymorphism means many forms).

Object oriented System Development

Function and data methods

- ❖ The existing methods for system development can basically be divided into function/data methods and object-oriented methods.
- ❖ By function/ data methods treat functions and/or data as being more or less separate where as object-oriented methods view functions and data as highly integrated.
- ❖ Function/ data methods thus distinguish between functions and data, where functions, in principle, are active and have behavior, and data is a passive holder of information which is affected by functions.
- ❖ The system is typically broken down into functions, whereas data is sent between those functions. The functions are broken down further and eventually converted into source code.
- ❖ A system developed using a function/ data method often becomes difficult to maintain. A major problem with function/ data methods is that, in principle, all functions must know how the data is stored, that is, its data structure.

Function and data methods.....

- ❖ It is often the case that different types of data have slightly different data formats, which means that we often need condition clauses to verify the data type. The programs, therefore, often need to have either IF-THEN or CASE structures, which really do not have anything to do with the functionality, but relate only to the different data formats.
- ❖ The programs thus become difficult to read, as we are often not interested in data format, but only in the functionality. Furthermore, to change a data structure, we must modify all the functions relating to the structure.
- ❖ Systems developed using such methods often become quite unstable; a slight modification will generate major consequences.

Function and data methods.....

- ❖ Another problem with function/ data methods is that people do not naturally think in the terms they structure in.
- ❖ The requirement specification is normally formulated in normal human language. It often describes in 'What-terms' what the system will do, what functionality the system will support and what items should exist in the system.
- ❖ This is often reformulated into 'How-terms' for the functional breakdown when the focus changes. In this way we create a large semantic gap between the external and internal views of the system.
- ❖ Note that nothing in the methods explicitly tells you to change focus to the implementation. We shall see that, with an object-oriented method, the system will even be internally structured from the model taken from the requirement specification.

Object oriented Analysis

- ❖ The purpose of object-oriented analysis, as with all other analysis, is to obtain an understanding of the application: an understanding depending only on the system's functional requirements.
- ❖ The difference between object-oriented analysis and function/data analysis is, as previously mentioned, the means of expression. While function/ data analysis methods commence by considering the system's behavior and/or data separately, object oriented analysis combines them and regards them as integrated objects. Object-oriented analysis can be characterized as an iteration between analyzing the behavior and information of the system. Moreover, object-oriented analysis uses the object-oriented techniques introduced in the previous chapter.
- ❖ Object-oriented analysis contains, in some order, the following activities:
 - ✓ Finding the objects
 - ✓ Organizing the objects
 - ✓ Describing how the objects interact
 - ✓ Defining the operations of the objects
 - ✓ Defining the objects internally

Finding the objects

- ❖ The objects can be found as naturally occurring entities in the application domain. An object becomes typically a noun which exists in the domain and, because of this, it is often a good start to learn the terminology for the problem domain. By means of learning what is relevant in the application domain, the objects will be found. It is often the case that there is no problem in finding objects; the difficulty is usually in selecting those objects relevant to the system.
- ❖ The aim is to find the essential objects, which are to remain essential throughout the system's life cycle. As they are essential, they will probably always exist and, in this way, we hope to obtain a stable system. Stability also depends on the fact that modifications often begin from some of these items and therefore are local.
- ❖ For example, in an application for controlling a water tank, typical objects would include Contained Water, Regulator, Valve and Tank; for a banking application typical objects would include Customer, Account, Bank and Clerk.

Finding the objects

- ❖ The majority of object-oriented methods today have only one type of object. In this way, one obtains a simple and general model. Yet there are reasons for having several different object types: with only one object type, it can be quite difficult to see the difference between different objects. By having different object types, one can more quickly obtain an overview of the system. One can also obtain more support in improving the system's structure by having different rules for different objects. An example of this is that a passive object containing persistent information should not be dependent on objects which deal with the interface, as modifications in the interface are very common.
- ❖ Different object types can be organized according to different criteria. Some examples are to group them by characteristics, such as active/passive, physical/conceptual, temporary/permanent/persistent, part/whole, generic/specific, private/public, shared/non-shared.

Organizing the objects

- ❖ There are a number of criteria to use for the classification and organization of objects and classes of objects. One classification starts by considering how similar classes of objects are to each other.
- ❖ This is normally the basis of inheritance hierarchy; a class can inherit another class. Another classification can be made by considering which objects work with which other objects, or how an object is a part of another; for example a house can be built of doors and windows.
- ❖ A similar classification is to see which objects are in some way dependent on another and thus have modification as a basis of grouping into subsystems.

Object interaction

- ❖ In order to obtain a picture of how the object fits into the system, we can describe different scenarios or use cases in which the object takes part and communicates with other objects. In this way, we can fully describe the object's surroundings and what the other objects expect from our object. The object's interface can be decided from these scenarios. We then also consider how certain objects are part of other objects.

Operations on objects

- ❖ The object's operations come naturally when we consider an object's interface. The operations can also be identified directly from the application, when we consider what can be done with the items we model. They can be primitive (for example, create, add, delete) or more complex such as putting together some report of information from several objects. If one obtains very complex operations, new objects can be identified from them. Generally, it is better to avoid objects that are too complex.

Object implementation

- ❖ Finally, the object should be defined internally, which includes defining the information that each object must hold. Even the number of instances that can be created of each object is interesting; one may wish to have alternative ways of storing information. Some of the attributes can be inherited.

Object oriented construction

- ❖ Object-oriented construction means that the analysis model is designed and implemented in source code.
- ❖ This source code is executed in the target environment, which often means that the ideal model produced by the analysis model must be molded to fit into the implementation environment.

For details see book (I. Jacobson) pg. 79

Object Oriented Design

- ❖ Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis.
- ❖ In OOD, concepts in the analysis model, which are technology–independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.
- ❖ Grady Booch has defined object-oriented design as “A method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design”.

Object Oriented Design

- ❖ The implementation details generally include –
 - ✓ Restructuring the class data (if necessary),
 - ✓ Implementation of methods, i.e., internal data structures and algorithms,
 - ✓ Implementation of control, and
 - ✓ Implementation of associations.

Object Oriented Programming

- ❖ Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.
- ❖ Grady Booch has defined object-oriented programming as “A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships”.

Object Oriented Programming

- ❖ The important features of object-oriented programming are –
 - ✓ Bottom-up approach in program design
 - ✓ Programs organized around objects, grouped in classes
 - ✓ Focus on data with methods to operate upon object's data
 - ✓ Interaction between objects through functions
 - ✓ Reusability of design through creation of new classes by adding features to existing classes
- ❖ Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

details on *book (I Jacobson) pg. 84*

Identifying Elements of Object Model

Identifying Elements of Object Model

- ✓ Identifying Classes and Objects
- ✓ Specifying Attributes
- ✓ Defining Operations
- ✓ Finalizing the Object Definition

Identifying Classes and Objects..

- ❖ If we look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when we "look around" the problem space of a software application, the objects may be more difficult to comprehend.
- ❖ We can begin to identify objects by examining the problem statement or performing a "grammatical parse" on the processing narrative for the system to be built.
- ❖ Objects are determined by underlining each noun or noun clause and entering it in a simple table.

Identifying Classes and Objects..

An objects can be:

- ✓ **External entities** (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- ✓ **Things** (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- ✓ **Occurrences or events** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- ✓ **Roles** (e.g., manager, engineer, salesperson) played by people who interact with the system.
- ✓ **Organizational units** (e.g., division, group, team) that are relevant to an application.
- ✓ **Places** (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- ✓ **Structures** (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or in the extreme, related classes of objects.

Identifying Classes and Objects..

To illustrate how objects might be defined during the early stages of analysis, lets take an example of SafeHome security system. The processing narrative for SafeHome is as follow:

- ✓ SafeHome software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the SafeHome control panel.
- ✓ During installation, the SafeHome control panel is used to "program" and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.
- ✓ When a sensor event is sensed by the software, it rings an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting and the nature of the event that has been detected.
- ✓ The number will be redialed every 20 seconds until telephone connection is obtained. All interaction with SafeHome is managed by a user-interaction subsystem that reads input provided through the keypad and function keys, displays prompting messages on the LCD display, displays system status information on the LCD display. Keyboard interaction takes the following form . . .

Identifying Classes and Objects..

Extracting the nouns, we can propose a number of potential objects:

Potential Object/Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

Note:

The list would be continued until all nouns in the processing narrative have been considered. Note that we call each entry in the list a potential object. We must consider each further before a final decision is made

Identifying Classes and Objects..

Coad and Yourdon suggest six selection characteristics that should be used to consider each potential object for inclusion in the analysis model:

1. Retained information

The potential object will be useful during analysis only if information about it must be remembered.

2. Needed services

The potential object must have a set of identifiable operations that can change the value of its attributes in some way.

3. Multiple attributes

During requirement analysis, the focus should be on "major" information; an object with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another object during the analysis activity.

4. Common attributes

A set of attributes can be defined for the potential object and these attributes apply to all occurrences of the object.

Identifying Classes and Objects..

5. Common operations

A set of operations can be defined for the potential object and these operations apply to all occurrences of the object.

6. Essential requirements

External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as objects in the requirements model.

To be considered a legitimate object for inclusion in the requirements model, a potential object should satisfy all (or almost all) of these characteristics. With this in mind, we apply the selection characteristics to the list of potential SafeHome objects:

Potential Object/Class	Characteristic Number That Applies
homeowner	rejected: 2 fails
sensor	accepted
control panel	accepted
installation	rejected
system (alias security system)	accepted
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted
audible alarm	accepted
monitoring service	rejected: 2 fails

Note:

1. *The preceding list is not all-inclusive, additional objects would have to be added to complete the model.*
2. *Some of the rejected potential objects will become attributes for those objects that were accepted (e.g., number and type are attributes of sensor, and master password and telephone number may become attributes of system)*
3. *Different statements of the problem might cause different "accept or reject" decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the homeowner object would satisfy characteristics 2 and would have been accepted).*

Specifying Attributes

- ❖ Attributes describe an object that has been selected for inclusion in the analysis model. In essence, it is the attributes that define the object—that clarify what is meant by the object in the context of the problem space.
- ❖ For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the object **player** would be quite different than the attributes of the same object when it is used in the context of the professional baseball pension system. In the former, attributes such as **name, position, batting average, fielding percentage, years played, and games played** might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like **average salary, credit toward full vesting, pension plan options chosen, mailing address** etc.

Specifying Attributes

- ❖ To develop a meaningful set of attributes for an object, the analyst can again study the processing narrative (or statement of scope) for the problem and select those things that reasonably "belong" to the object.
- ❖ To illustrate, we consider the system object defined for SafeHome. Homeowner can configure the security system to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. Using the content description notation defined for the data dictionary , we can represent these composite data items in the following manner:

sensor information: sensor type, sensor number, alarm threshold

alarm response information = delay time , telephone number, alarm type

activation/deactivation information = master password, number of allowable tries , temporary password

identification information = system ID, verification phone number, system status

Defining Operations

- ❖ Operations define the behavior of an object and change the object's attributes in some way. More specifically, an operation changes one or more attribute values that are contained within the object.
- ❖ Although many different types of operations exist, they can generally be divided into three broad categories:
 - ✓ operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting)
 - ✓ operations that perform a computation, and
 - ✓ operations that monitor an object for the occurrence of a controlling event.

Finalizing the Object\ Class Definition

- ❖ The definition of operations is the last step in completing the specification of an object.
- ❖ For example I few consider object named **system** in **SafeHome**, this can be expanded to reflect known activities that occur during its life. For example, sensor event will send a message to system to display the event location and number; control panel will send system a reset message to update system status; the audible alarm will send a query message; the control panel will send a modify message to change one or more attributes without reconfiguring the entire system object; sensor event will also send a message to call the phone number(s) contained in the object. Other messages can be considered and operations derived. So the resulting object definition for object **system** is shown in figure (**next slide**)

Finalizing the Object\ Class Definition

System
System ID
Verification PH
System status
Sensor type
Sensor number
Alarm threshold
Alarm delay time
Telephone number(s)
Master password
Temporary password
Number of tries
Program()
Display()
Reset()
Query()
Modify()
Call()

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

OBJECT ORIENTED SOFTWARE ENGINEERING
Chapter Three

Structural behavioral and architectural modeling

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Conceptual model of UML

Conceptual model of UML

- © To understand the UML, we need to form a conceptual model of the language, and this requires learning three major elements:
 - ✓ The UML's basic building blocks
 - ✓ The rules that dictate how those building blocks may be put together and
 - ✓ Some common mechanisms that apply throughout the UML.

Basic Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:

- 1. *Things***
- 2. *Relationships***
- 3. *Diagrams***

Note:

Things are the abstractions that are first-class citizens in a model

Relationships tie these things together

Diagrams group interesting collections of things.

Things in the UML

There are four kinds of things in the UML:

1. *Structural Things*

- © These are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical.
- © Collectively, the structural things are called classifiers.
- © *Class, Interface, collaboration, use case, component, node* etc. are example of Structural things in UML.

2. *Behavioral Things*

- © These are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space.
- © In all, there are three primary kinds of behavioral things : *interaction, state machine and Activity*.

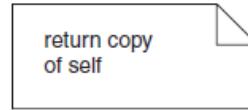
Things in the UML

3. Grouping Things

- © These are the organizational parts of UML models.
- © These are the boxes into which a model can be decomposed.
- © There is one primary kind of grouping thing called *packages*.

4. Annotational Things

- © These are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a *note*.
- © A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.
- © Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment, as in Figure below



Relationship in the UML

Relationships in the UML There are four kinds of relationships in the UML:

1. *Dependency*
2. *Association*
3. *Generalization*
4. *Realization*

Details on later slides

Diagrams in the UML

- © A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships).
- © We draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. The UML includes thirteen kinds of diagrams:
 1. Class diagram
 2. Object diagram
 3. Component diagram
 4. Composite structure diagram
 5. Use case diagram
 6. Sequence diagram
 7. Communication diagram
 8. State diagram
 9. Activity diagram
 10. Deployment diagram
 11. Package diagram
 12. Timing diagram
 13. Interaction overview diagram

UML

Use Case Diagram

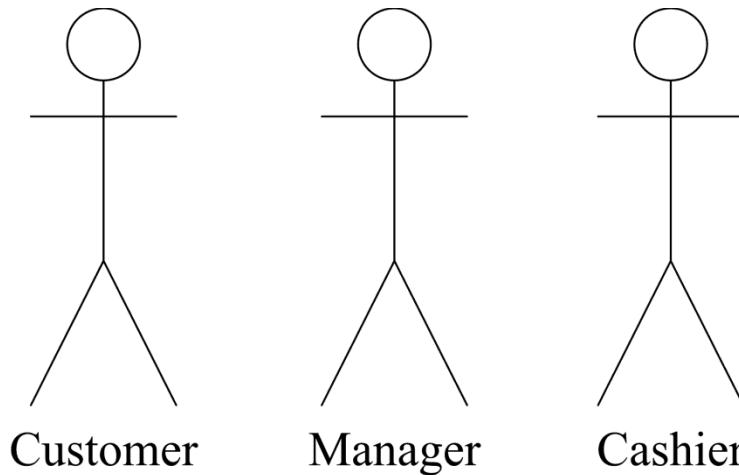
UML Use Case Diagram

- ④ Use case is used to model the system or subsystem of an application.
- ④ A single use case diagram captures a particular functionality of a system.
- ④ The purpose of use case diagrams can be as follows:
 - © Used to gather **requirements of a system**
 - © Used to get an **outside view** of a system
 - © Identify **external factors** influencing the system
 - © Show the **interaction among the requirements** also called use cases.
- ④ To draw a use case diagram we should have following items identified or components of use case diagrams are:
 - © Actor
 - © Use case
 - © System boundary
 - © Relationship

Use Case Diagram

Actor

- © An actor is someone or something that must interact with the system under development.
- © An UML notation of Actor is represented as:



- © Actors are not part of the system they represents anyone or anything that must interact with the system.
- © A single actor may perform more than one use cases (functionality)

Use Case Diagram

- ⓧ An actor may be:
 - © Input information to the system
 - © Receive information from the system
 - © Input to and out from the system.

- ⓧ How do we find the actors?

Ask the following questions:

- © Who uses the system?
- © Who install the system?
- © Who starts up the system?
- © What other system use this system?
- © Who gets the information to the system?

Note: *An actor is always external to the system.*

Use Case Diagram

Categories of Actor:

© **Principle**

who uses the main system functions

© **Secondary**

who takes care of administration and maintenance

© **External h/w**

The h/w devices which are part of application domain and must be used

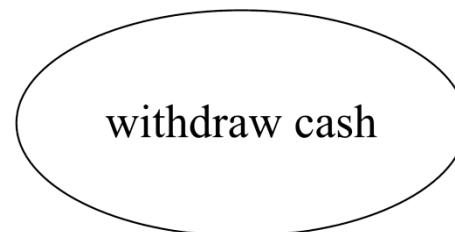
© **Other system**

The other system with which the system must interact.

Use Case Diagram

Use cases:

- © Use cases represents functionality of a system
- © which are the specific roles played by the actors within and around the system



How do we find the use cases?

- © What functions will the actor want from the system?
- © Does the system store information? If yes then which actors will create, read, update or delete that information?
- © Does the system need to notify an actor about changes in its internal state?

Use Case Diagram

Generic format for documenting an use case

Pre condition:	if any
Use case:	name of the use case
Actors:	list of actors, indicating who initiates the use case
Purpose:	intention of the use case.
Overview:	Description.
Type:	primary/secondary.
Post condition:	if any

Use Case Diagram

Example:

Description of opening a new account in the bank

Use case Open new account

Actors Customer, Cashier, Manager

Purpose Like to have new saving account.

Description A customer arrives in the bank to open the new account.

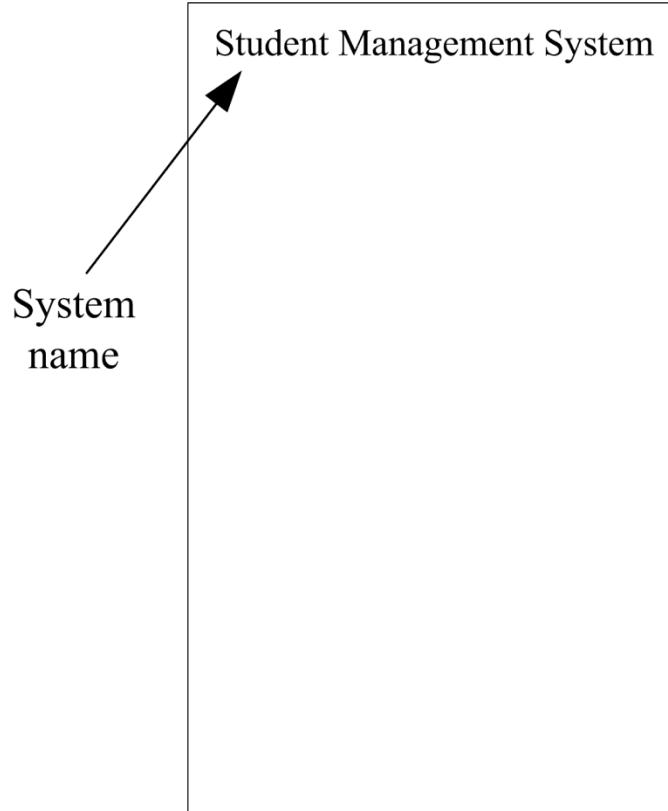
Customer requests for the new account form, fill the form and submits, along with the minimal deposit.

At the end of complete successful process customer receives the passbook.

Use Case Diagram

System boundary:

- © It helps to identify what is an external versus internal.
- © External environment is represented only by actors.
- © Represented as a rectangle.



UML Use Case Diagram

Relationship:

Relationship between use case and actor

- © communicates

Relationship between two use cases

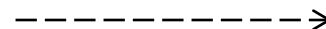
- © Include
- © extend

Notation used to show the include and extend relationship

«include»



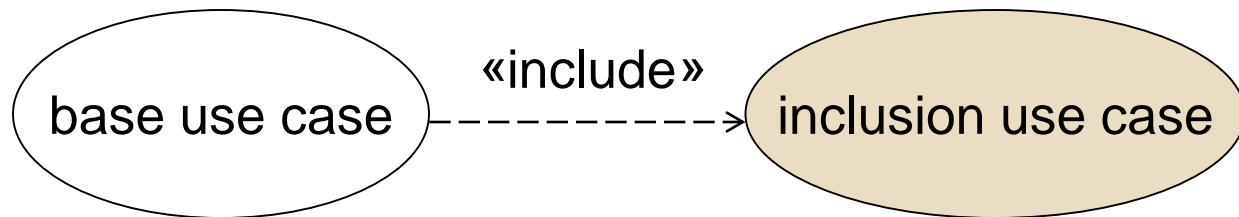
«extend»



Use Case Diagram

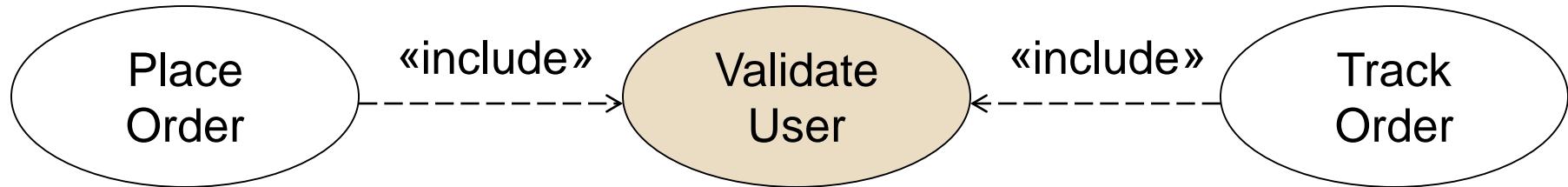
«include» Relationship:

- © A use cases may contain the functionality of another use case.
- © It is used to show how the system can use a pre existing components
- © An include relationship is a relationship in which one use case (the base use case) includes or uses the functionality of another use case (the inclusion use case).
- © Represented as a dashed line with an open arrow pointing from the base use case to the inclusion use case. The keyword «include» is attached to the connector.



Use Case Diagram

«include» relationship example:



- © The following figure illustrates an restaurant order management system that provides customers with the option of placing orders as well as tracking orders. This behavior is modeled with two base use cases called *PlaceOrder* & *TrackOrder* that has an inclusion use case called *ValidateUser*. The *ValidateUser* use case is a separate inclusion use case because it contains behaviors that several other use cases in the system use. That include relationship points from the *PlaceOrder* & *TrackOrder* use cases to the *ValidateUser* use case indicate that, the *PlaceOrder* & *TrackOrder* use cases always includes the behaviors of the *ValidateUser* use case.

UML Use Case Diagram

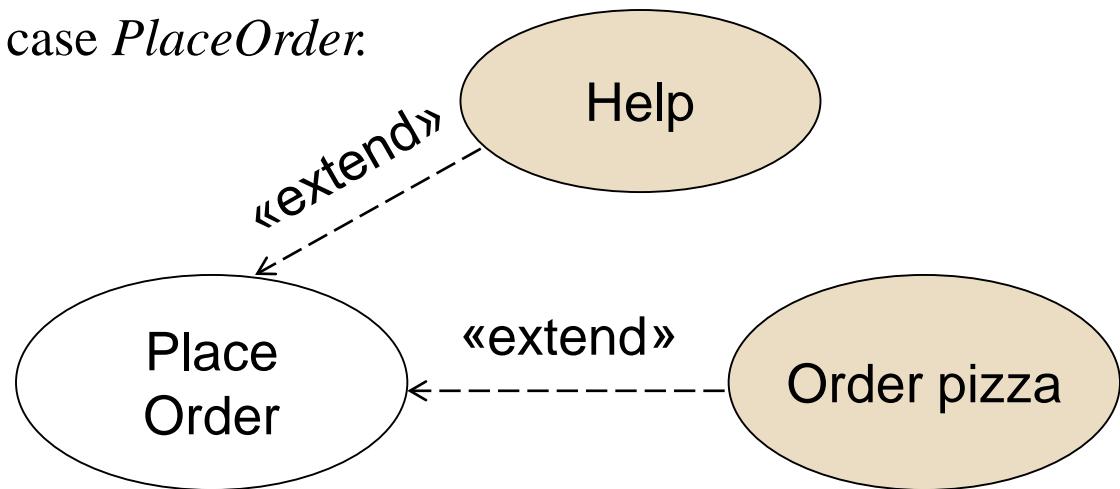
«extend» relationship

- Used to show optional behavior, which is required only under certain condition. This is typically used in exceptional circumstances.
- Represented as dotted line labeled «extend» with an arrow toward the base case.



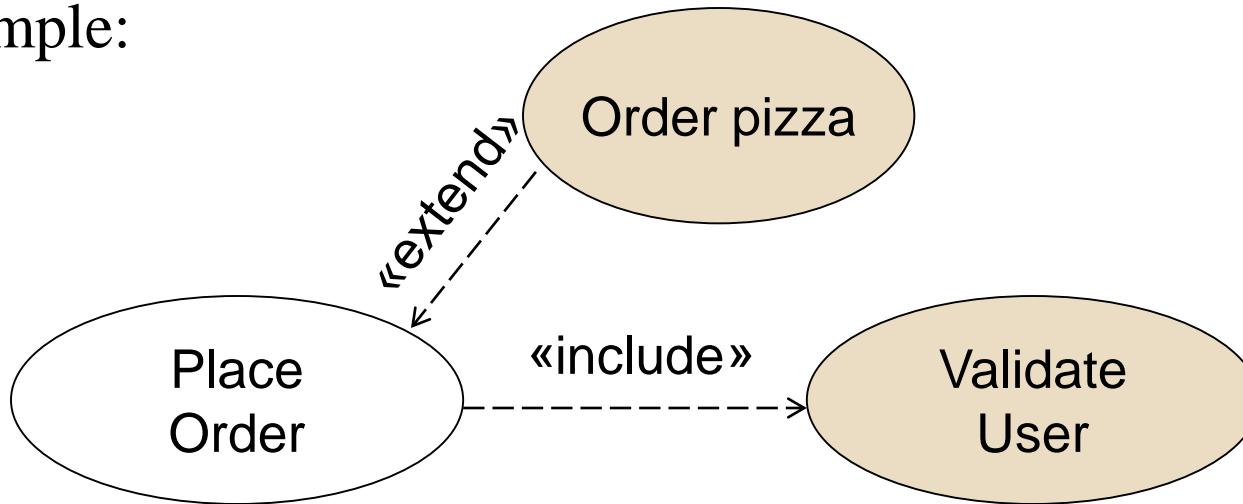
Example:

Here in following example the *OrderPizza* & *Help* use cases are optional to base use case *PlaceOrder*.



Includes vs. Extend

Example:



Key Points:

	include	extend
Is this use case optional?	<i>No</i>	<i>Yes</i>
Is the base use case complete without this use case?	<i>No</i>	<i>Yes</i>
Is the execution of this use case conditional?	<i>No</i>	<i>Yes</i>
Does this use case change the behavior of the base use case?	<i>No</i>	<i>Yes</i>

Description: Withdraw money from ATM.

Use case Scenario name: Withdraw money from ATM.

Participating actors:

- Customer
- ATM Machine
- Bank

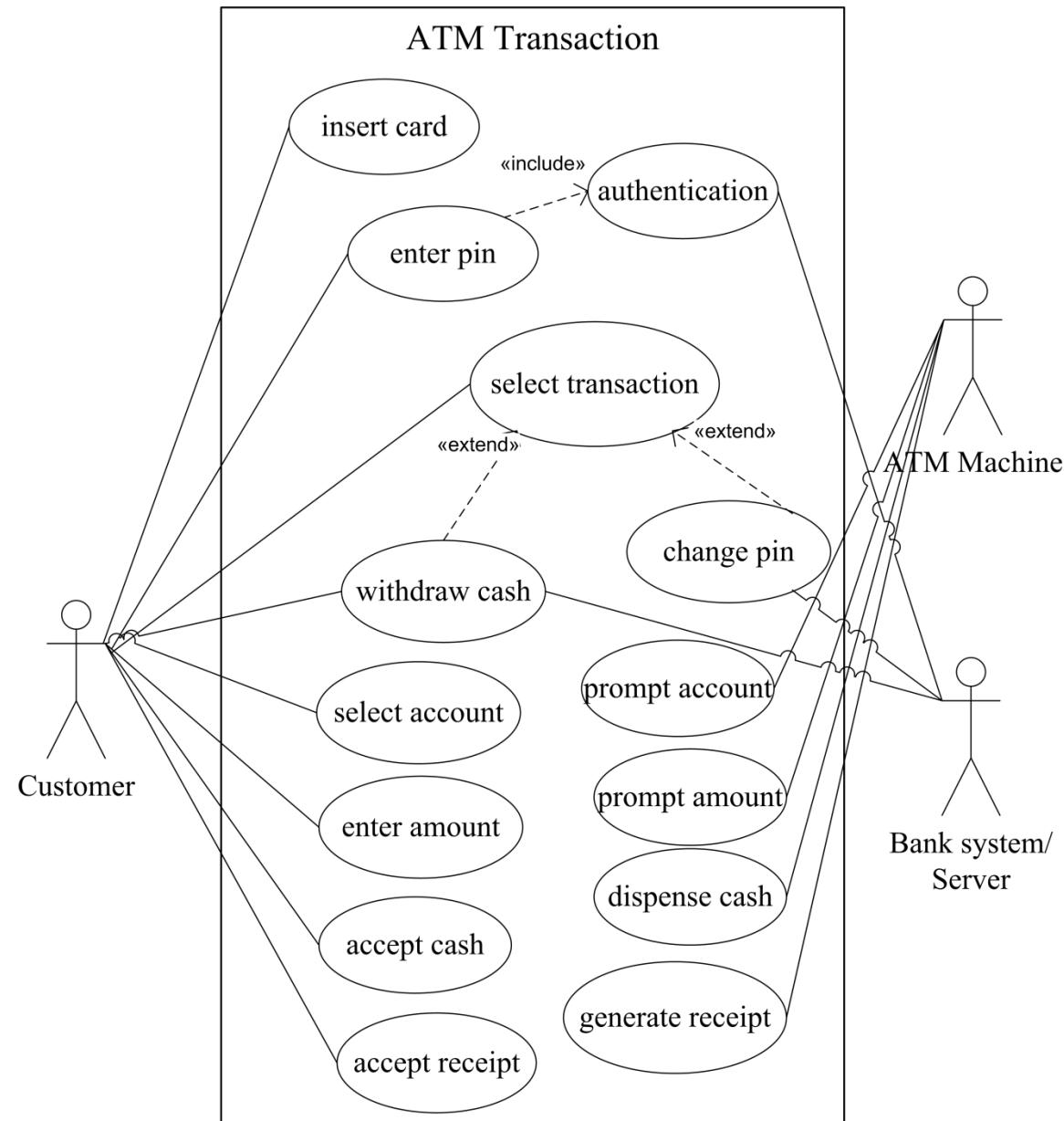
Preconditions:

- Network connection is active
- ATM has available cash

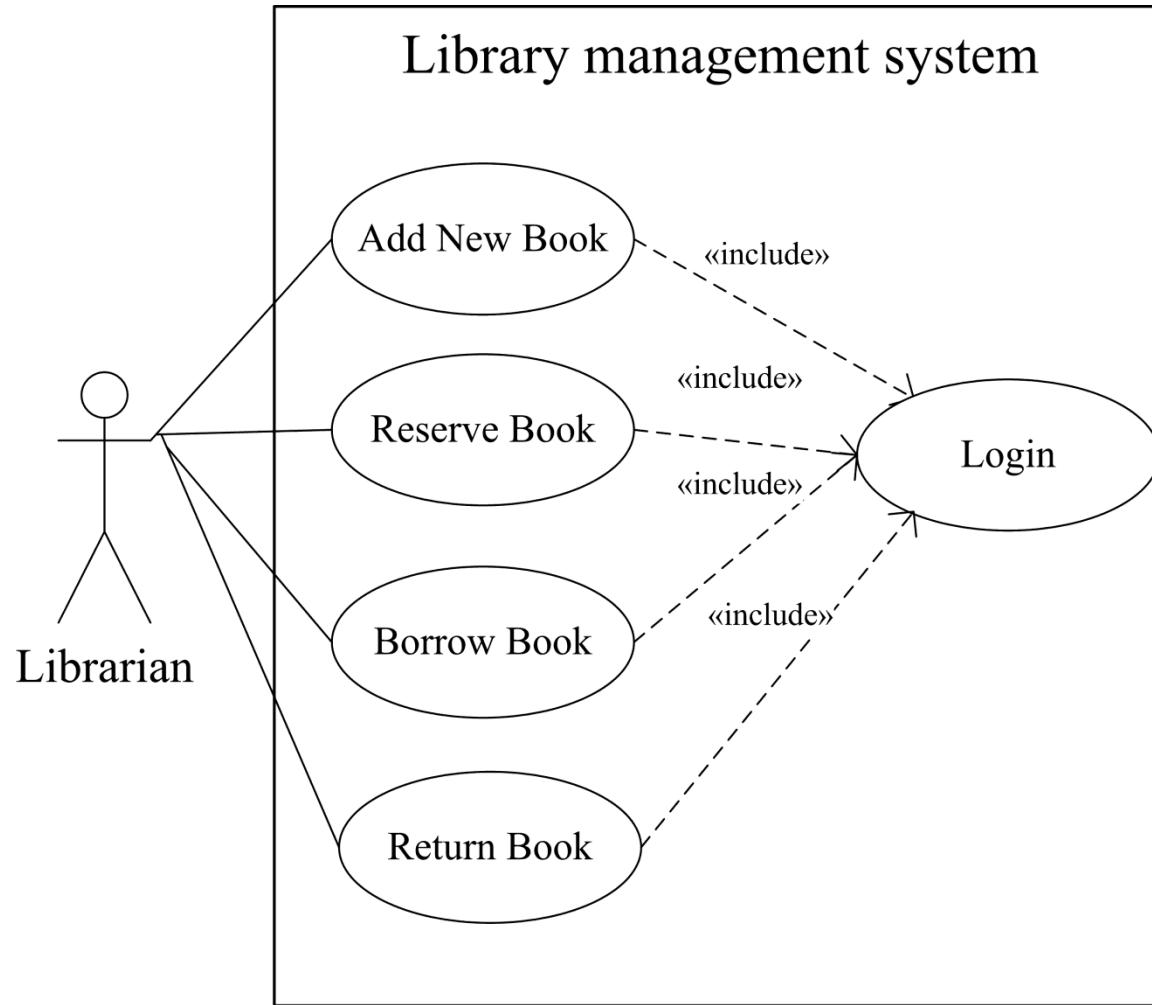
Flow of events:

- Bank customer inserts ATM card and enters PIN.
- Customer is validated.
- ATM displays actions available on ATM unit. Customer selects Withdraw Cash.
- ATM prompts account.
- Customer selects account.
- ATM prompts amount.
- Customer enters desired amount.
- Information sent to Bank, inquiring if sufficient funds/allowable withdrawal limit.
- Money is dispensed and receipt prints.

ATM Transaction Example



Library Management System



Example

A coffee vending machine dispenses coffee to customers. Customers order coffee by selecting a recipe from a set of recipes. Customers pay using coins. Change is given back if any to the customer. The service staff loads ingredients into machine. The service staff can also add a recipe by indicating name of coffee, units of coffee powder, milk, sugar, water and chocolate to be added as well as the cost of coffee.

Actors:

Customer, Service staff

Use Cases:

Dispense coffee

Order

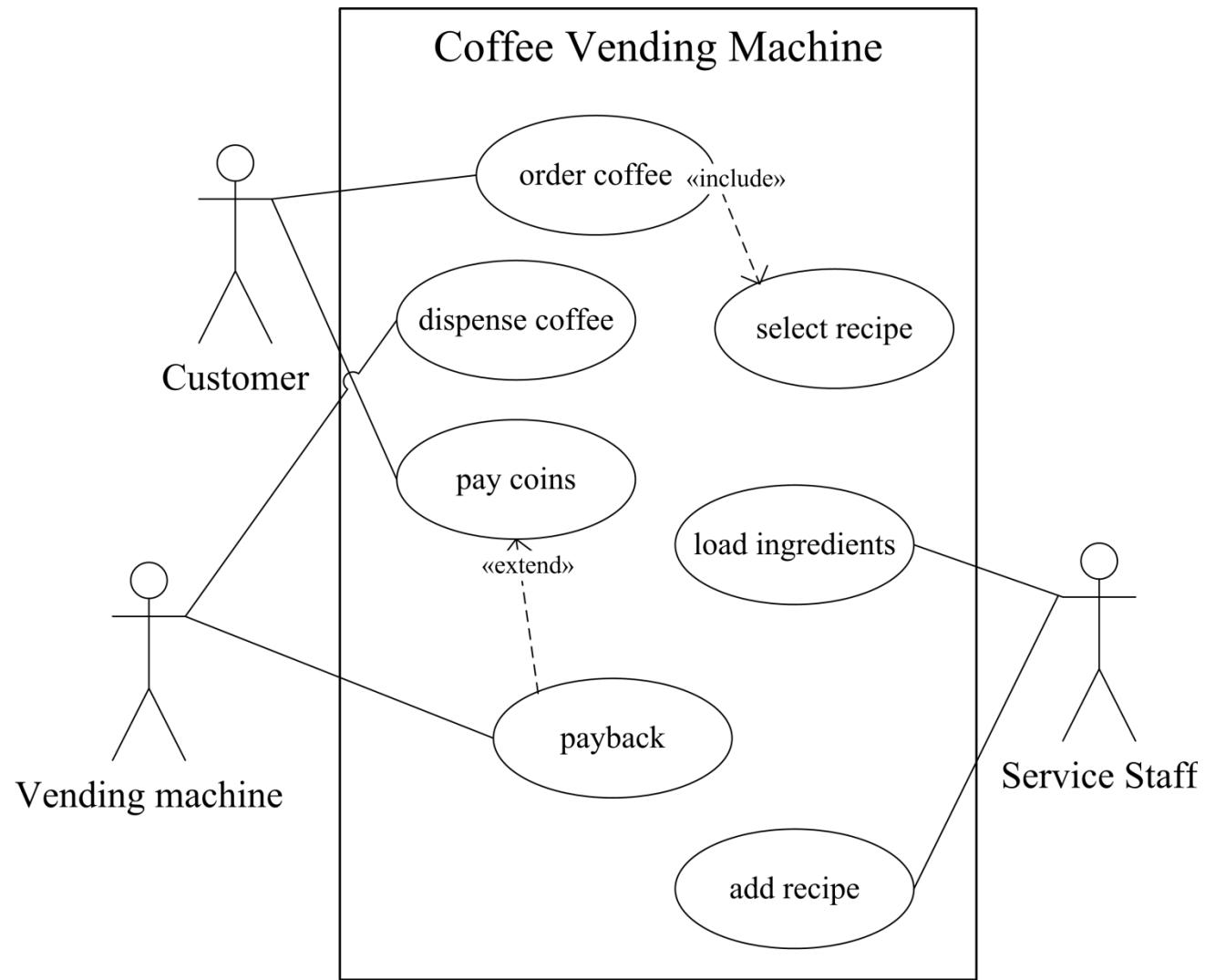
Pay coins

Payback

Load ingredients

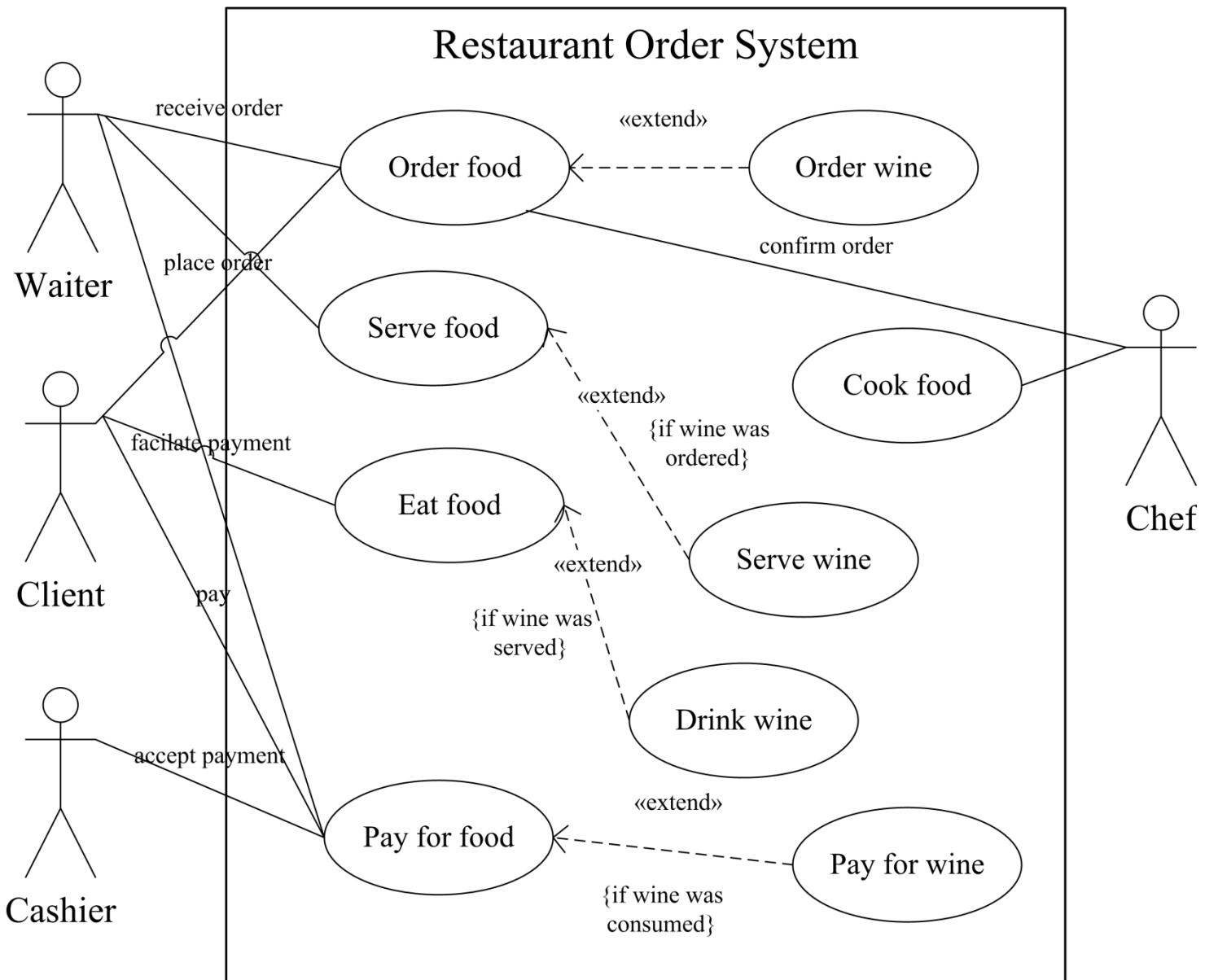
Add recipe

Use Case Diagram Example



Restaurant order System (Class Work)

Restaurant order System (Class Work)



UML

Interaction Diagrams

- ❖ *Sequence Diagram*
- ❖ *Collaboration (Commⁿ) Diagram*

UML

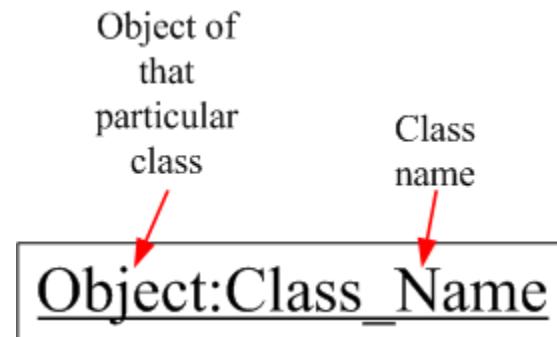
Sequence Diagram

Sequence Diagram

- ❖ Sequence diagram is interaction diagram that shows message exchanged or interaction between objects in the system.
- ❖ It mainly emphasizes on time ordering of messages between objects.
- ❖ It is used to illustrate the dynamic view of the system

Object or participants:

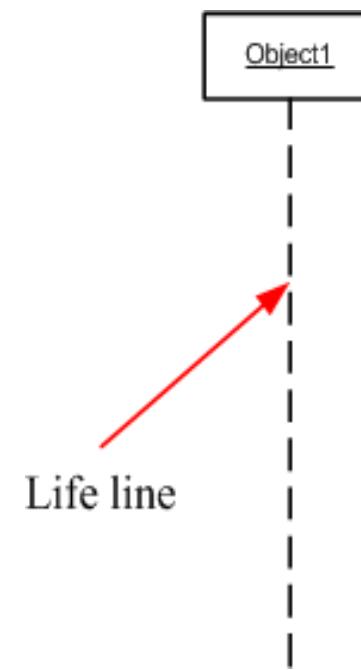
- ✓ The sequence diagram is made up of collection of **participants or objects**. Participants are system parts that interact each other during sequence diagram.
- ✓ The participants interact with each other by sending and receiving message.
- ✓ The object is represented by as:



Sequence Diagram

Lifeline:

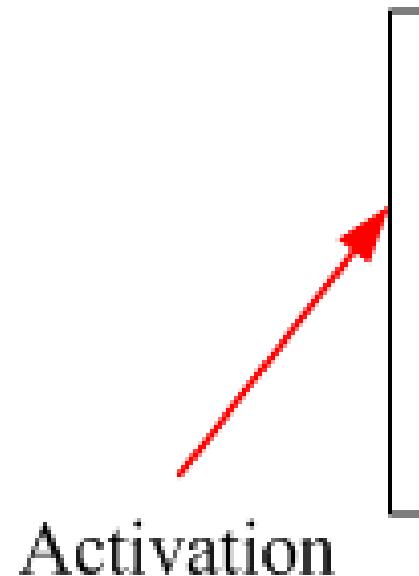
- ❖ Lifeline represents the existence of an object over a period of time.
- ❖ It is represented by vertical dashed line
- ❖ Most objects that appeared in ‘Interaction diagram’ will be in existence for the duration of an interaction. So, these objects are aligned at top of diagram with their lifeline from top to bottom of diagram.



Sequence Diagram

Activation bar:

- ❖ It is represented by tall thin rectangle.
- ❖ The top of rectangle is aligned with start of the action.
- ❖ The bottom is aligned with its completion and can be marked by a written message
- ❖ It is also called as focus of control. It shows the period of time during which an object is performing an action or operation.



Sequence Diagram

Messages:

- ❖ Messages can be flow in whatever direction required for interaction from left to right and right to left.
- ❖ The messages on sequence diagram are specifies using an arrow from participant that wants to pass the messages to the participant that receive the messages.
- ❖ The interaction in a sequence diagram between the objects can be shown by using messages.

Message types:

1) Asynchronous messages:-

- ✓ It is a message where the sender is not blocked and can continue executing.
- ✓ Representing by solid line with open arrowhead.

Asynchronous message



Sequence Diagram

2) Synchronous messages

- ❖ It is a message where the sender is blocked and waits until the receiver has finished processing of message.
- ❖ It is invoked the caller waits for the receiver to return from the message invocation.
- ❖ It is represented by solid line with filled arrowhead.

Synchronous message



Sequence Diagram

3) Reflexive messages/self message:-

- ❖ If the object sends the message to itself then it is called as ‘Reflexive message’.
- ❖ It is represented by solid line with loops the lifeline of object.

Self call



4) Return messages:-

- ❖ It can be used at the end of activation bar to show that control flow of activation returns to the participant that pass the original message.
- ❖ It is represented by dashed line from sender to receiver.

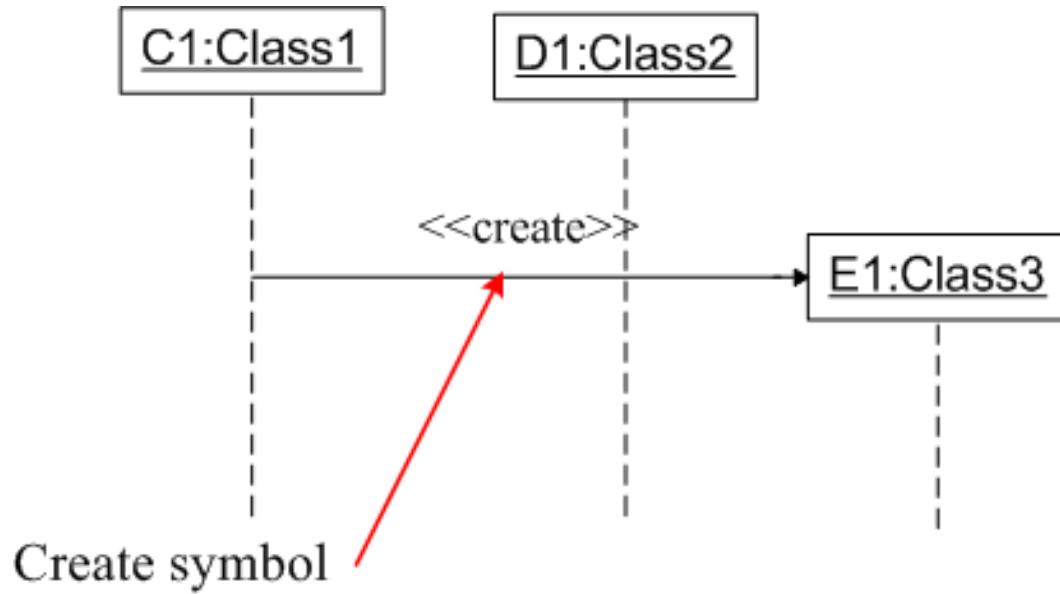
Return message



Sequence Diagram

5) Create messages:

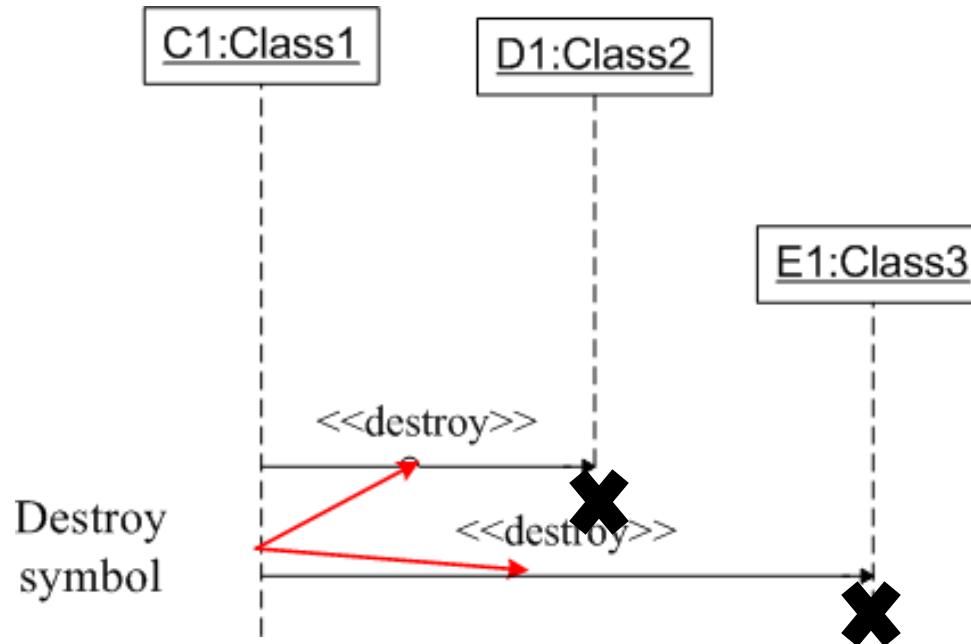
- ❖ It is used to create object during interaction.
- ❖ The object can be created by using <<create>> to indicate the timing of creation.
- ❖ Creating message can be shown as below:



Sequence Diagram

6) Destroy messages:

- ❖ It is used to destroy the objects during interaction.
- ❖ The objects can be terminated using <<destroy>> which points to an “x”.
- ❖ It indicates that object named message is terminated.



Note: Avoid modeling object destruction unless memory management is critical.

Sequence Diagram

Time:

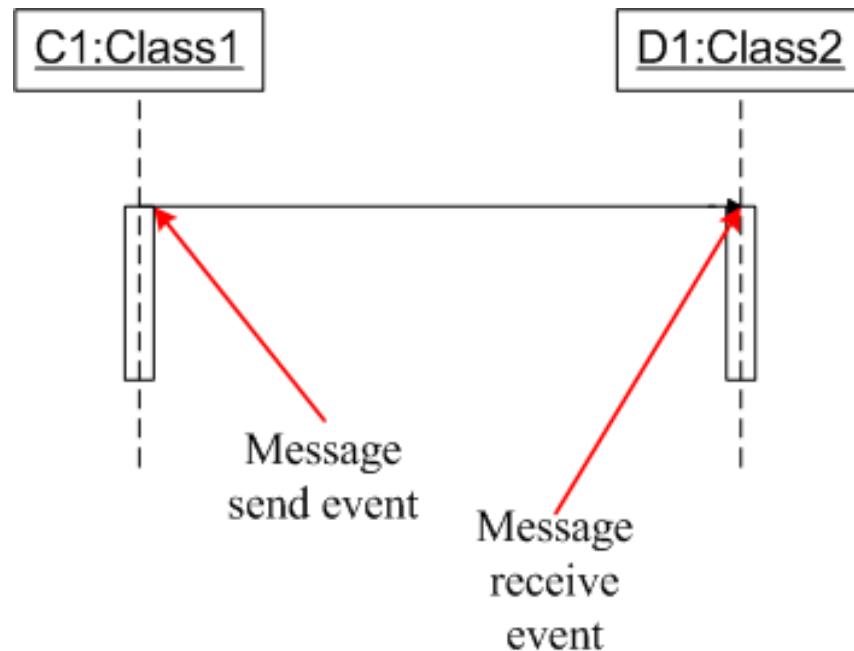
- ❖ Time is all about ordering but not duration.
- ❖ So time is an important factor.
- ❖ The time on sequence diagram starts at top of the page just below the object and then progresses down the page.
- ❖ The sequence diagram describes the order in which interaction takes place.



Sequence Diagram

Event:

- ❖ Event is created while sending and receiving message.
- ❖ When interaction takes place, Events are called as build in blocks for messages and signals.
- ❖ It can be referred as smallest part of an interaction and event can occur at any given point in a Time.



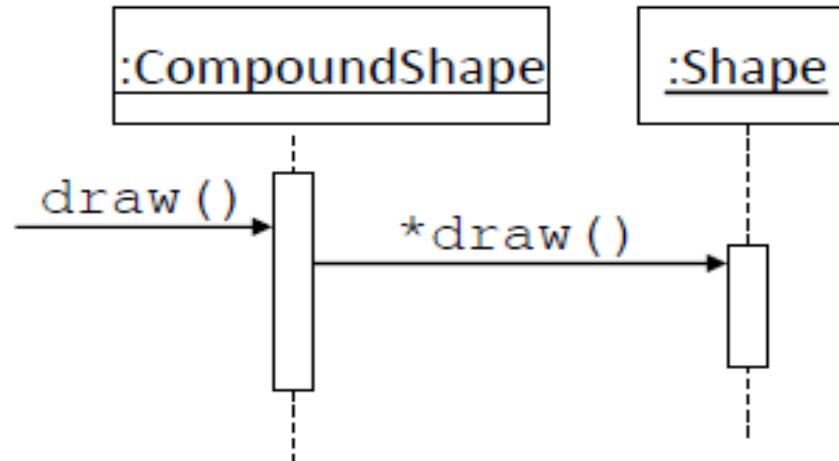
Sequence Diagram: Control Information

Condition

- ❖ **syntax:** [expression or condition] message-label
- ❖ The message is sent only if the condition is true
- ❖ example: [user valid= “true”] give access

Iteration

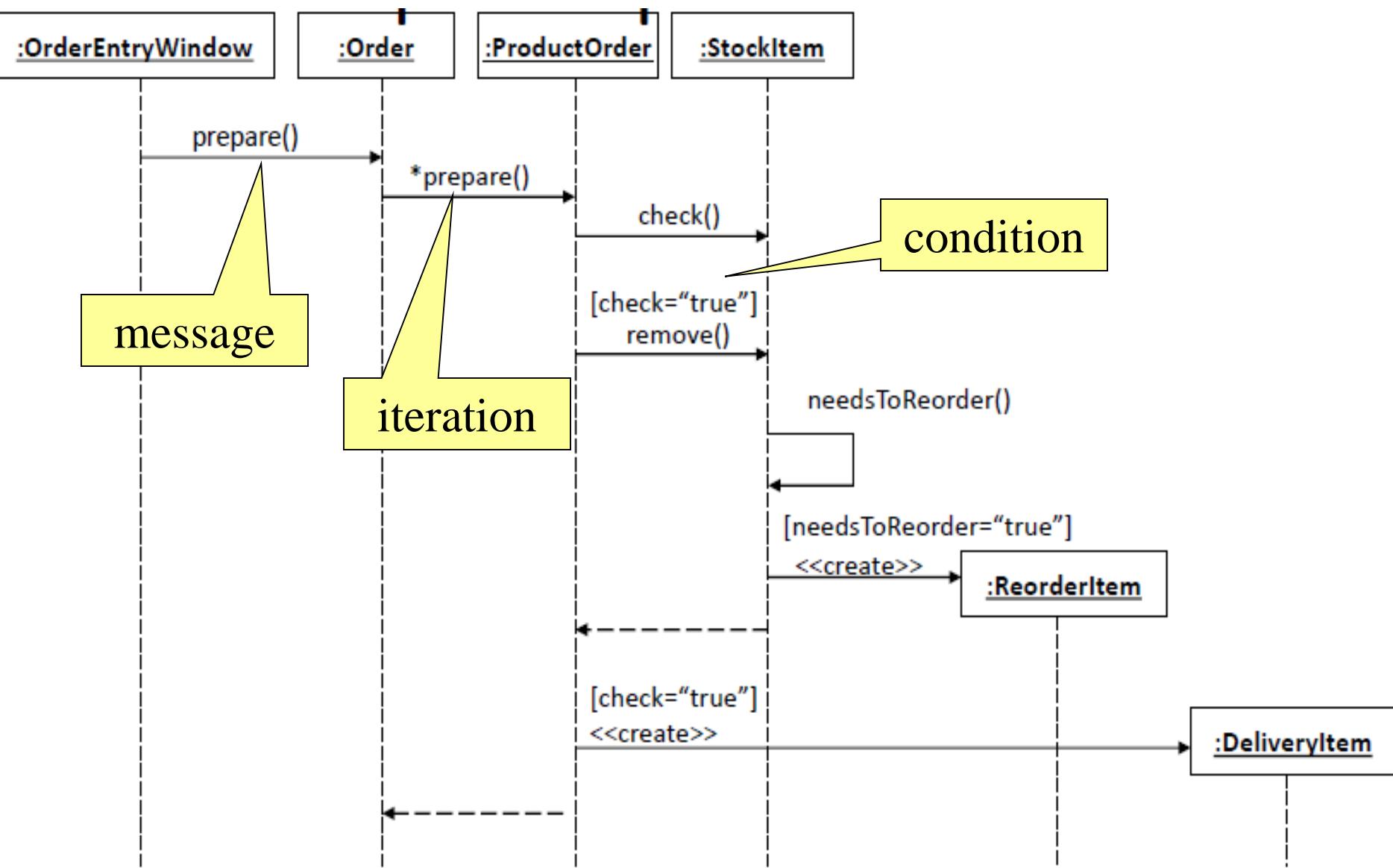
- ❖ **syntax:** * [expression] message-label or *message-label
 - Note:** * [expression] message-label is not standard syntax
- ❖ The message is sent many times to possibly multiple receiver objects.



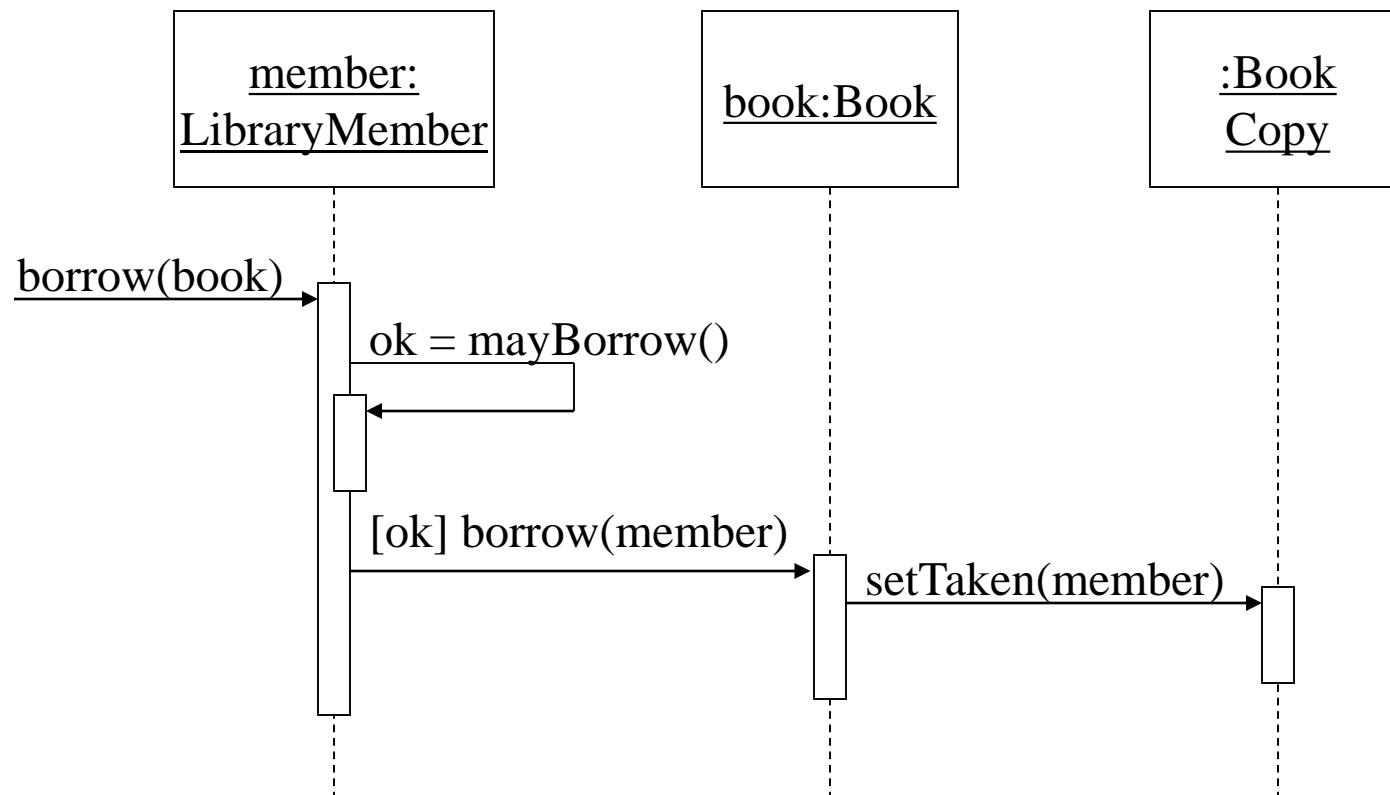
Sequence Diagram

- ❖ The control mechanisms of sequence diagrams suffice only for modeling simple alternatives.
- ❖ Consider drawing several diagrams for modeling complex scenarios.
- ❖ Don't use sequence diagrams for detailed modeling of algorithms (this is better done using *activity diagrams, pseudo-code or state-charts*).

Order processing → POS

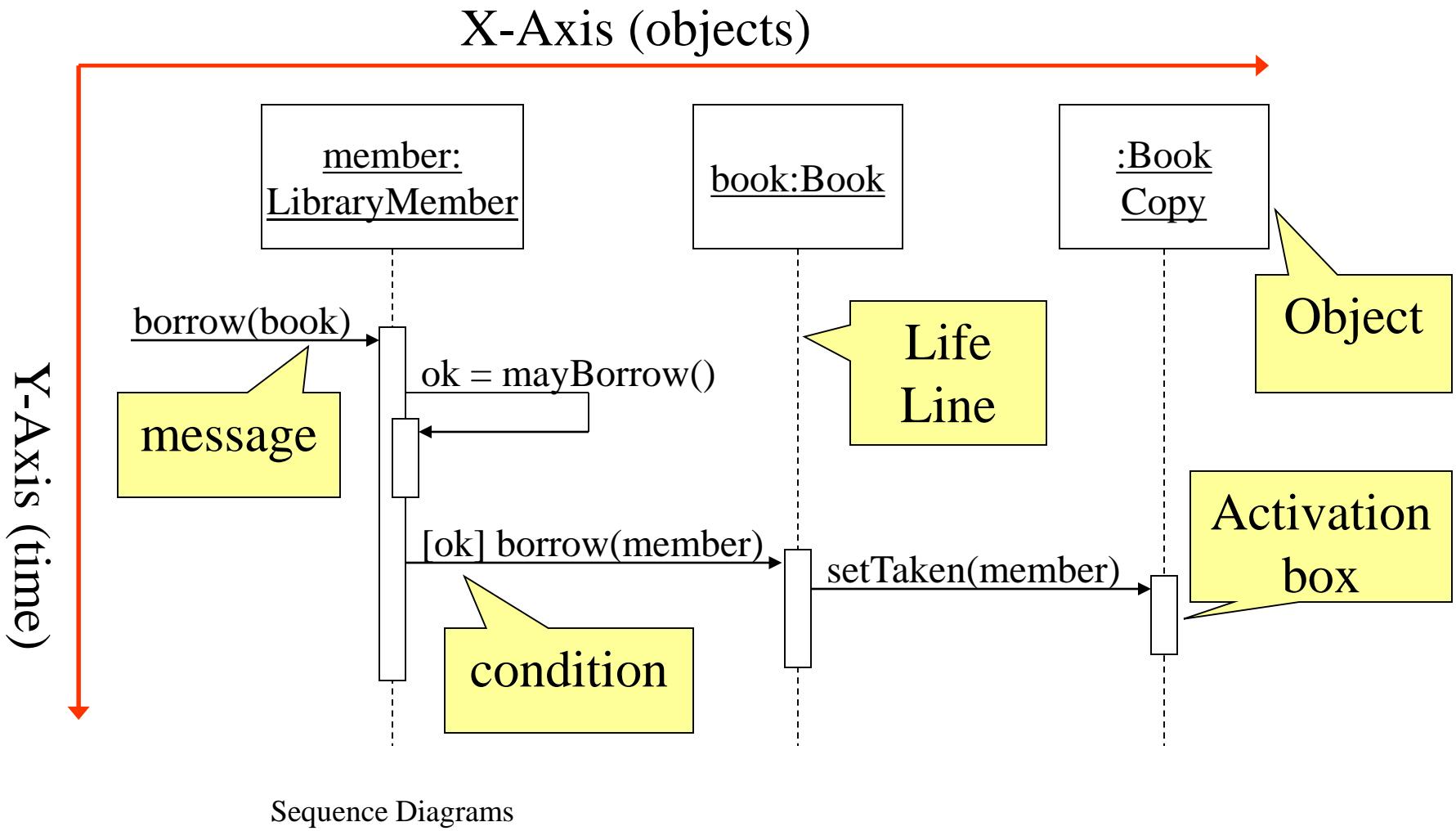


Borrowing book from library

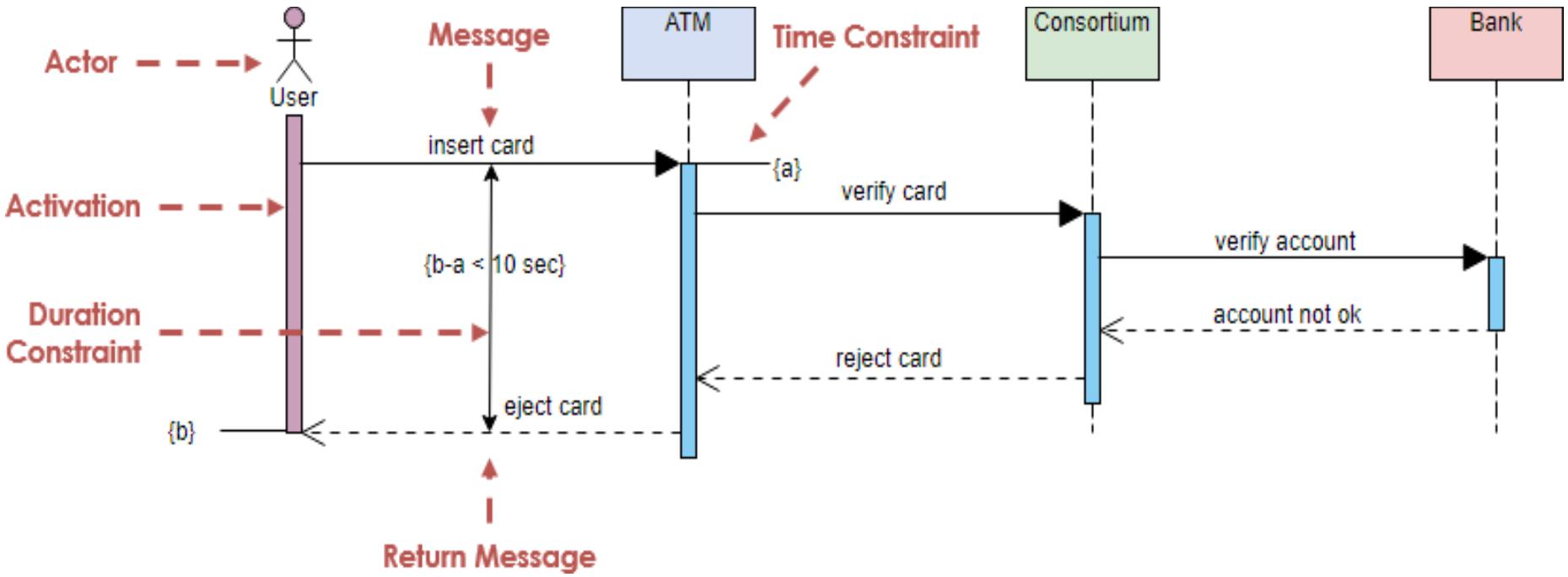


Sequence Diagrams

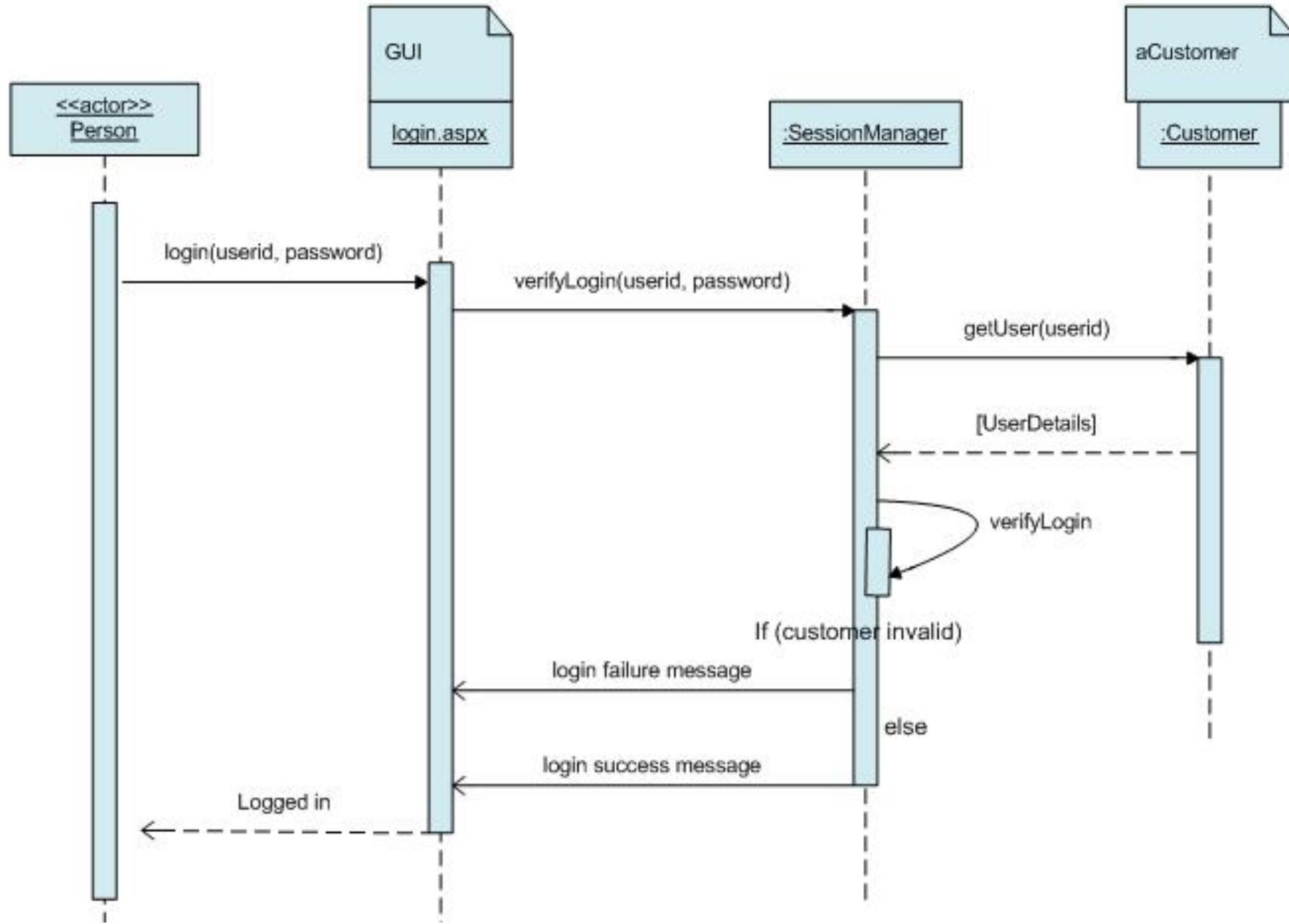
Borrowing book from library



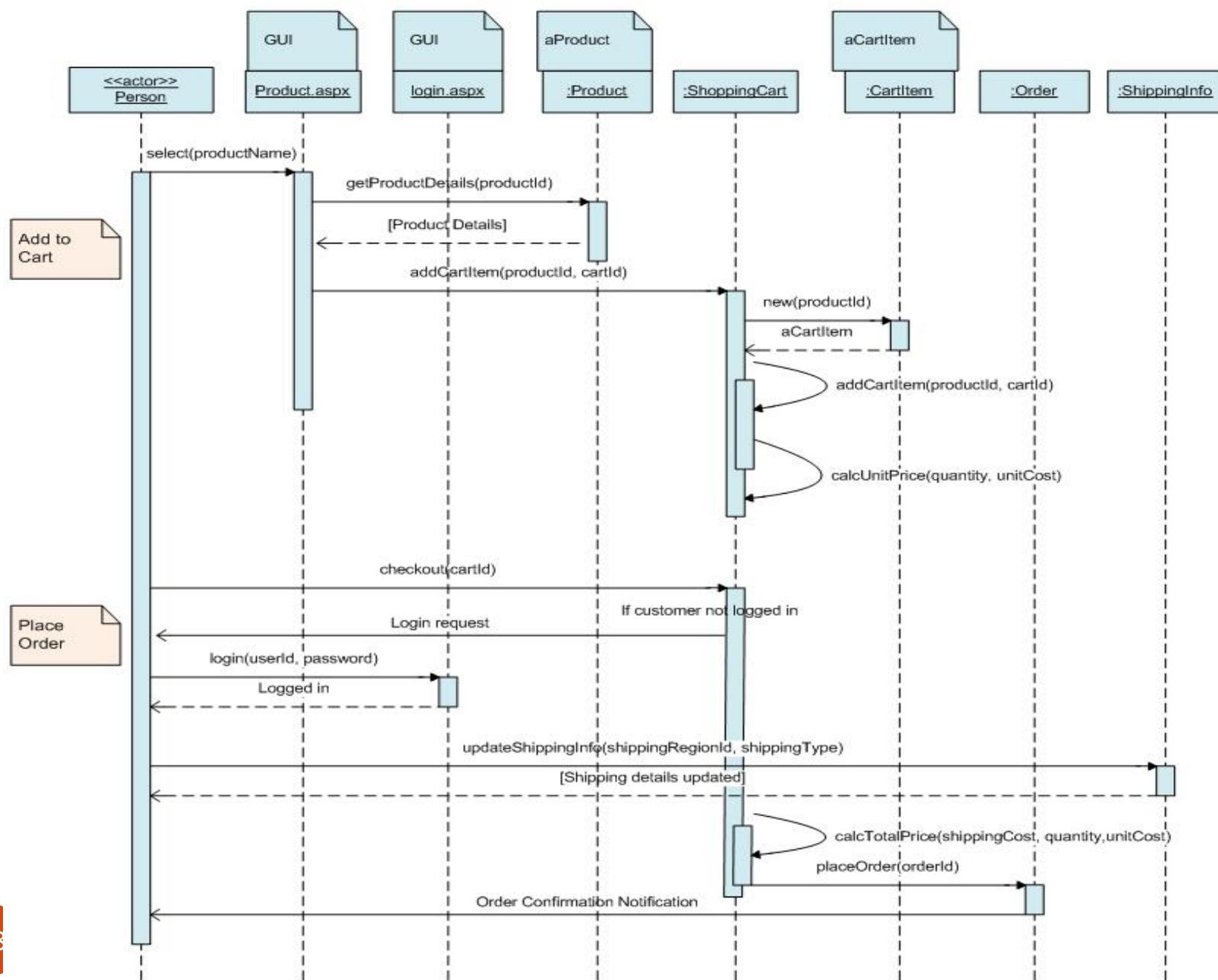
ATM (Invalid Pin)



Login



Online shopping: Explanation [Assignment]



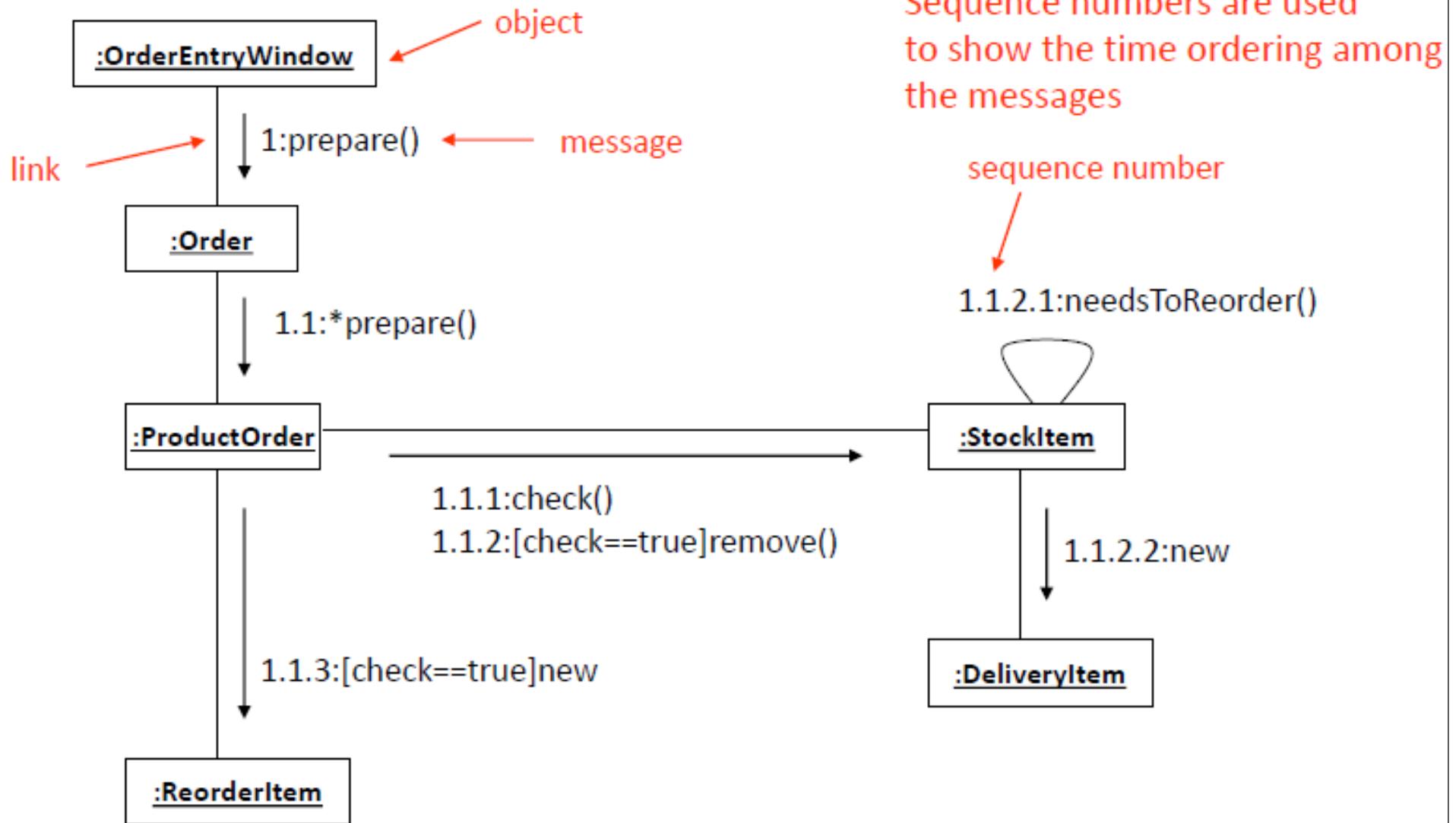
UML

Collaboration/ Commⁿ Diagram

Collaboration (Communication) Diagrams

- ❖ Collaboration diagrams (also called Communication diagrams) show a particular sequence of messages exchanged between a number of objects
 - ✓ This is what sequence diagrams do too!
- ❖ Sequence diagram highlight more the temporal aspect of the system i.e. it shows object interaction in timely manner(so no need of numbering the messages).
- ❖ In Collaboration diagram, the temporal aspect can be shown here too, by numbering the interactions with sequential labels. (so need to numbering the messages).
- ❖ So Sequence numbers are used to show the time ordering among the messages.

Collaboration (Communication) Diagrams



Sequence numbers are used to show the time ordering among the messages

UML

Activity Diagram

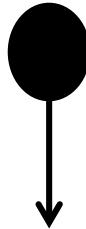
Activity Diagram

- ❖ An activity diagram visually represents a series of actions or flow of control in a system similar to a flowchart.
- ❖ Activities modeled can be sequential and concurrent.
- ❖ In both cases an activity diagram will have a beginning (an initial state) and an end (a final state) and in between them series of actions to be performed by the system.

Symbols in Activity Diagram

Initial State or Start Point

- ❖ A small filled circle followed by an arrow represents the initial action state or the start point for any activity diagram.



Symbols in Activity Diagram

Activity or Action State

- ❖ An activity represents execution of an action or performing some operation.
- ❖ It is represented using a rectangle with rounded corners.



Action Flow

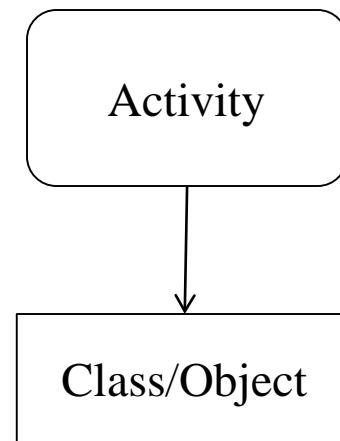
- ❖ Action flows, also called edges and paths, illustrate the transitions from one action state to another.
- ❖ They are usually drawn with an arrowed line.



Symbols in Activity Diagram

Object Flow

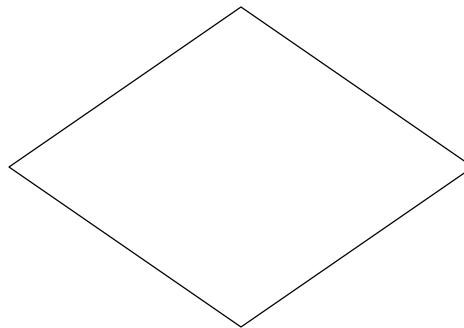
- ❖ Object flow refers to the creation and modification of objects by activities.
- ❖ An object flow arrow from an action to an object means that the action creates or influences the object.
- ❖ An object flow arrow from an object to an action indicates that the action state uses the object.



Symbols in Activity Diagram

Decisions and Branching

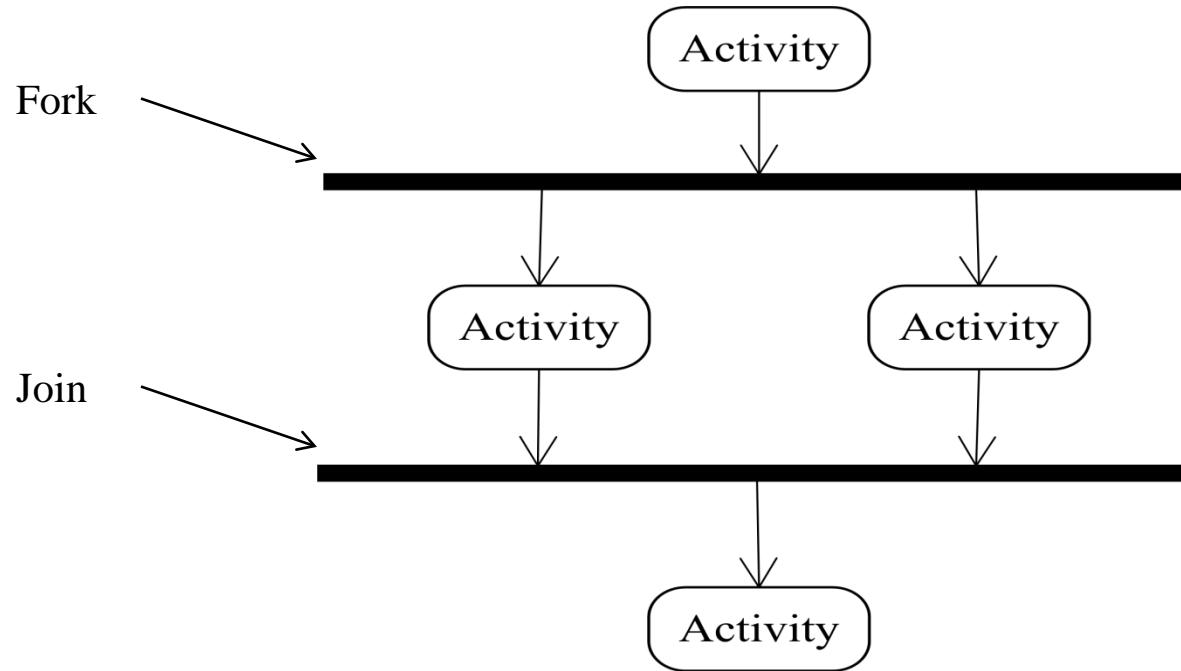
- ❖ A diamond represents a decision with alternate paths.
- ❖ When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities.
- ❖ The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."



Symbols in Activity Diagram

Synchronization

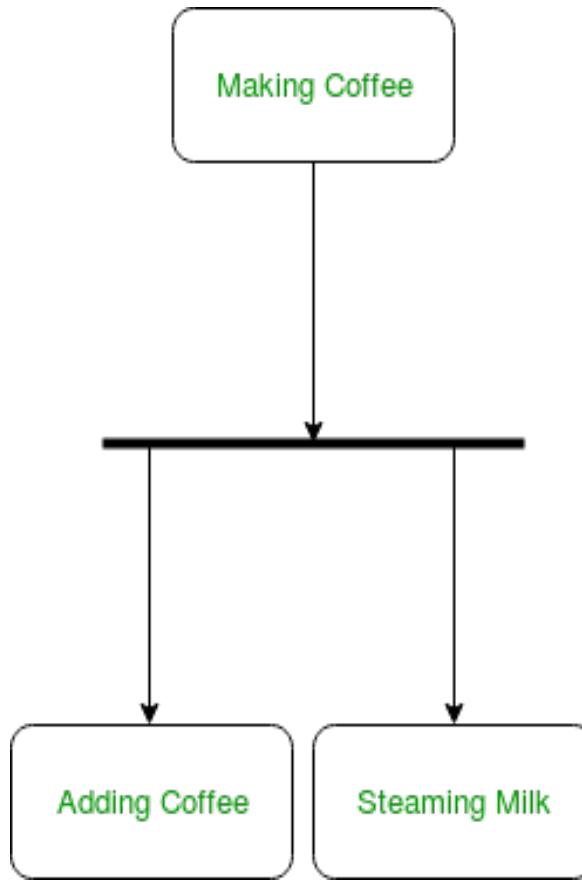
- ❖ A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- ❖ A join node joins multiple concurrent flows back into a single outgoing flow.
- ❖ A fork and join mode used together are often referred to as synchronization.



Symbols in Activity Diagram

Fork

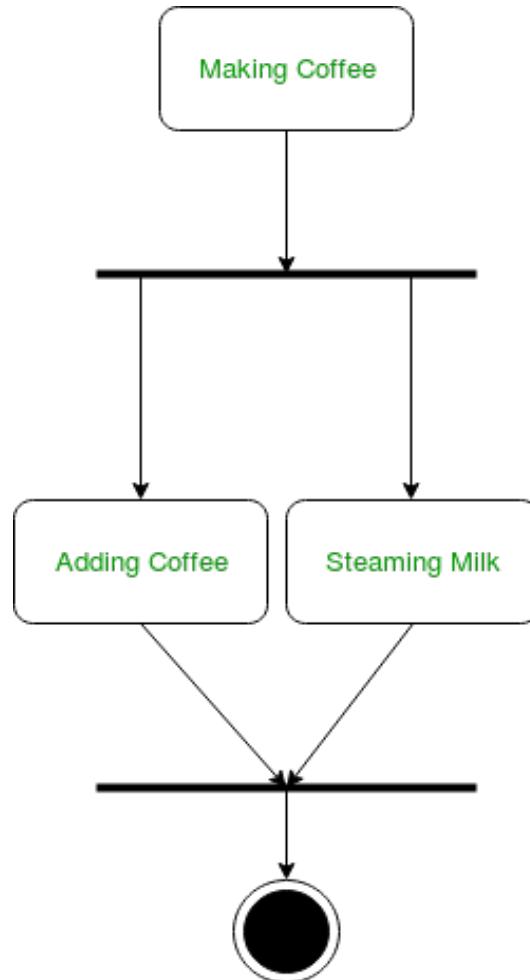
- ❖ A fork node is used to split a single incoming flow into multiple concurrent flows.
- ❖ It is represented as a straight, slightly thicker line in an activity diagram.



Symbols in Activity Diagram

Join

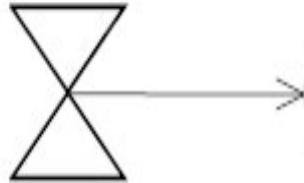
- ❖ A join node joins multiple concurrent flows back into a single outgoing flow.



Symbols in Activity Diagram

Time Event

- ❖ This refers to an event that stops the flow for some amount of time.
- ❖ It is represented by a hourglass.



Time Event

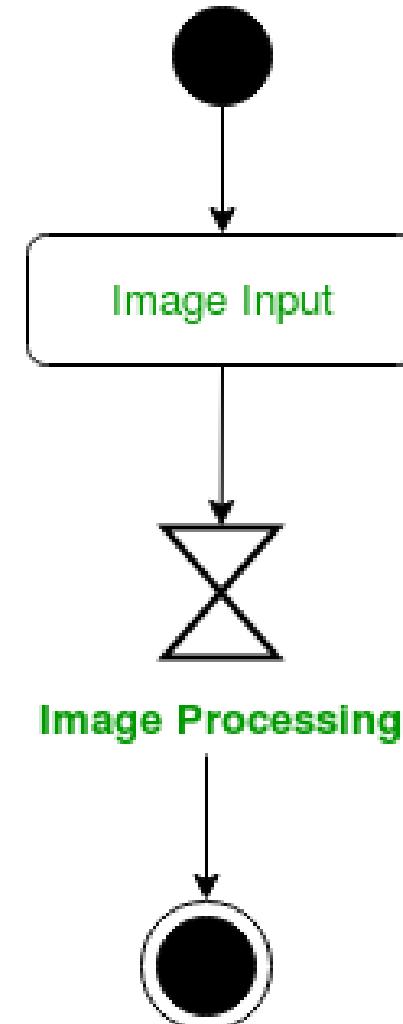


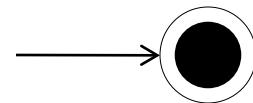
Image Input

Image Processing

Symbols in Activity Diagram

Final State or End Point

- ❖ An arrow pointing to a filled circle nested inside another circle represents the final action state.



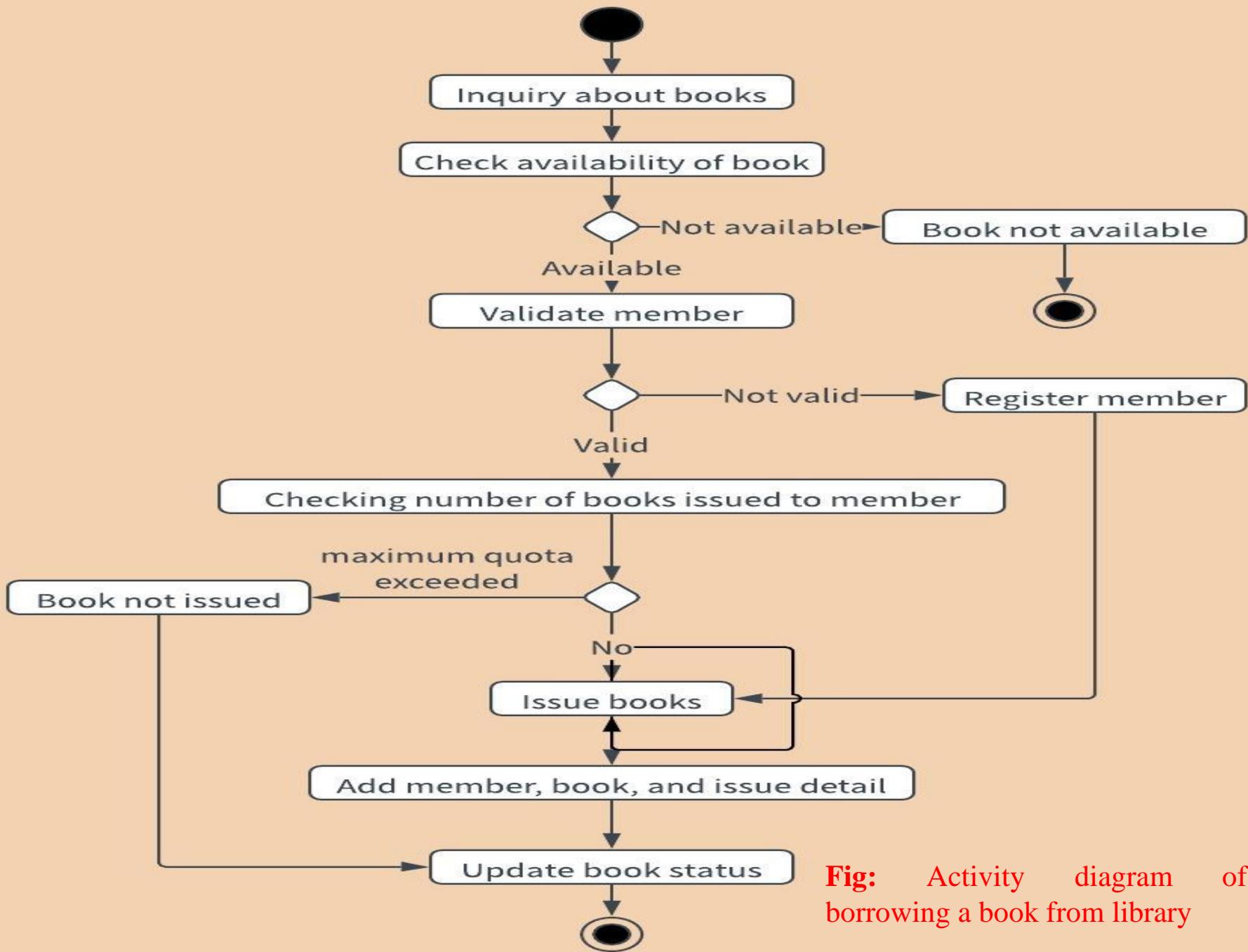


Fig: Activity diagram of borrowing a book from library

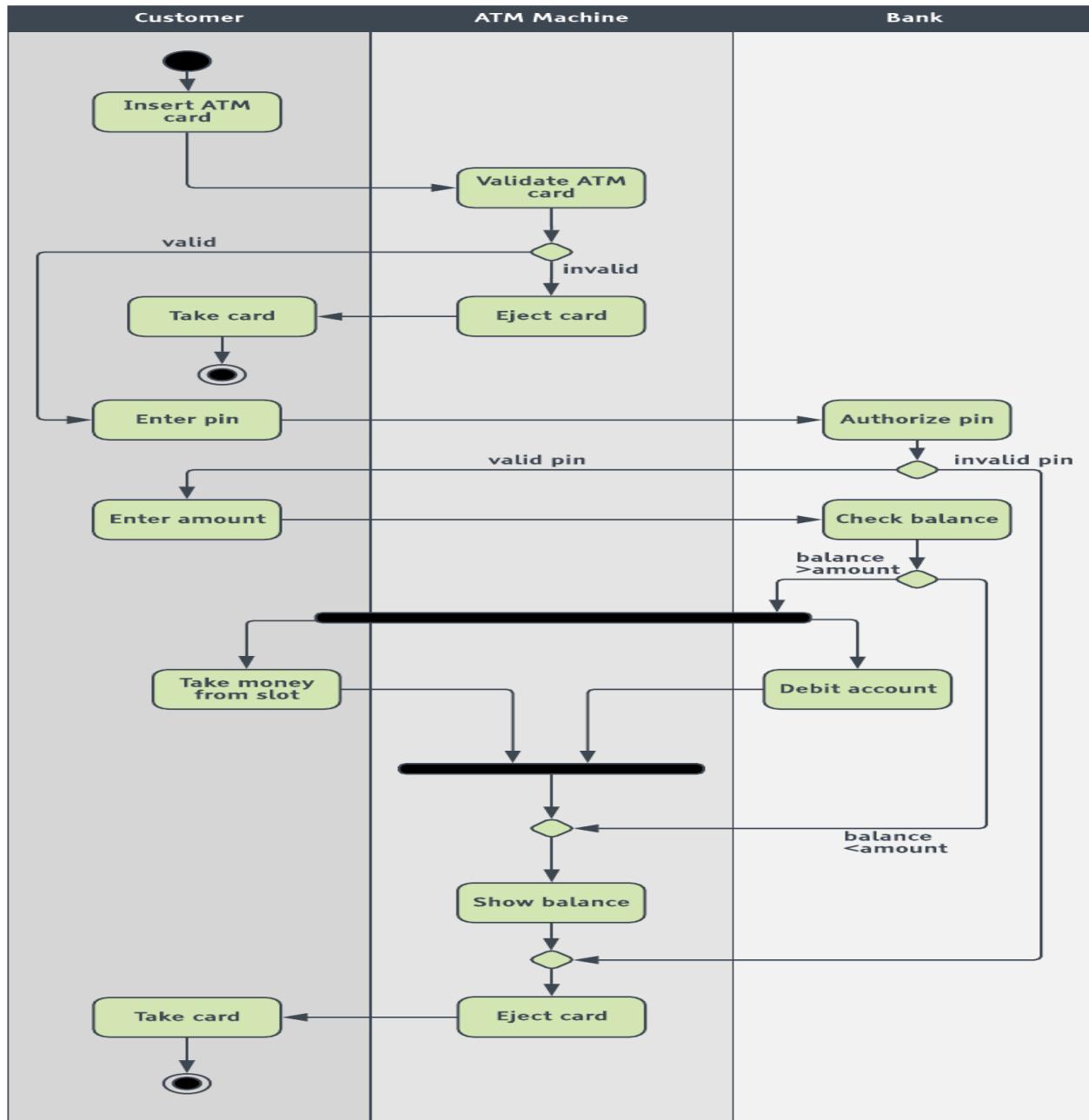
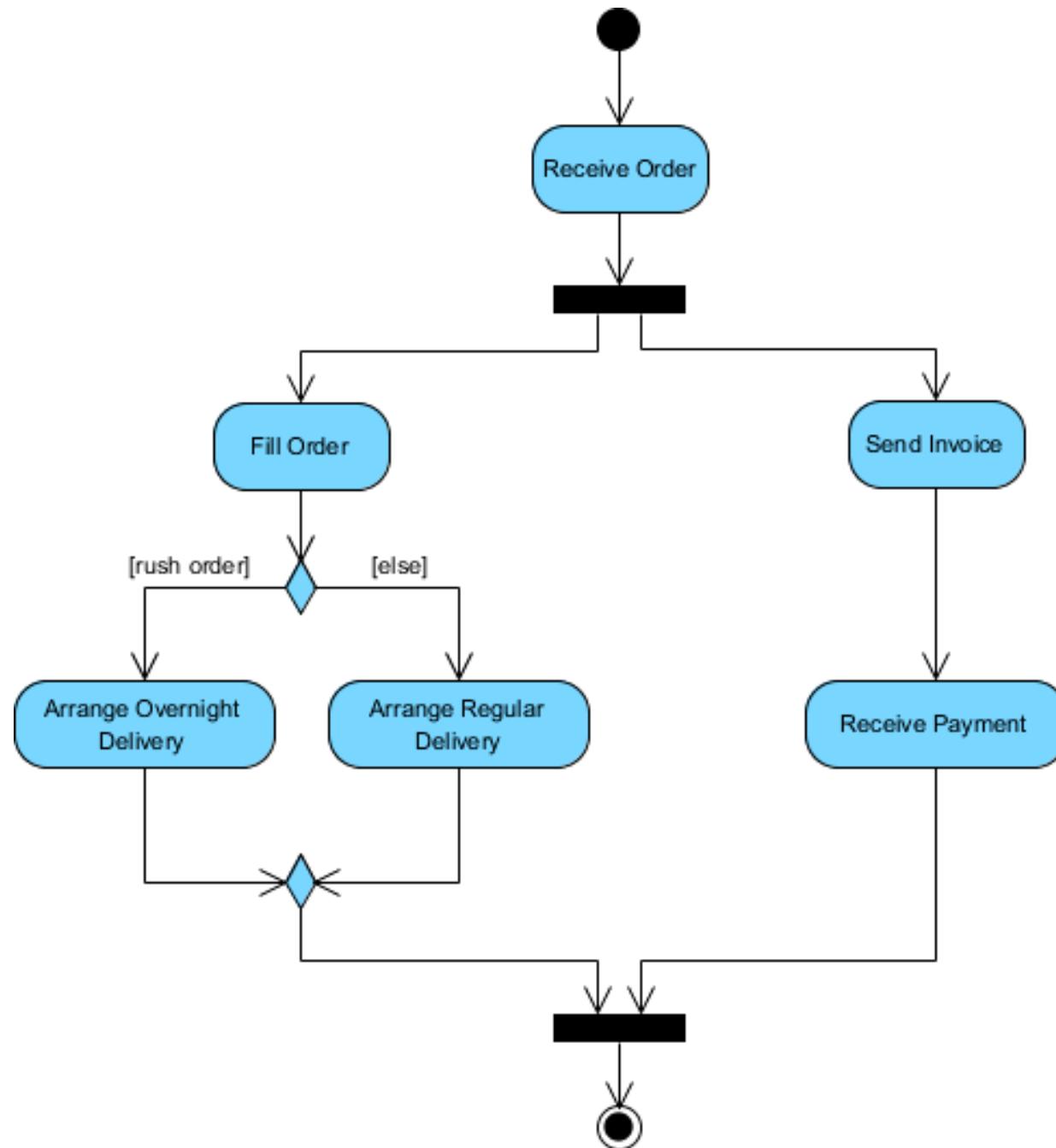


Fig: Swimlane diagram of ATM transaction

Activity Diagram

Process Order - Problem Description (Class Work)

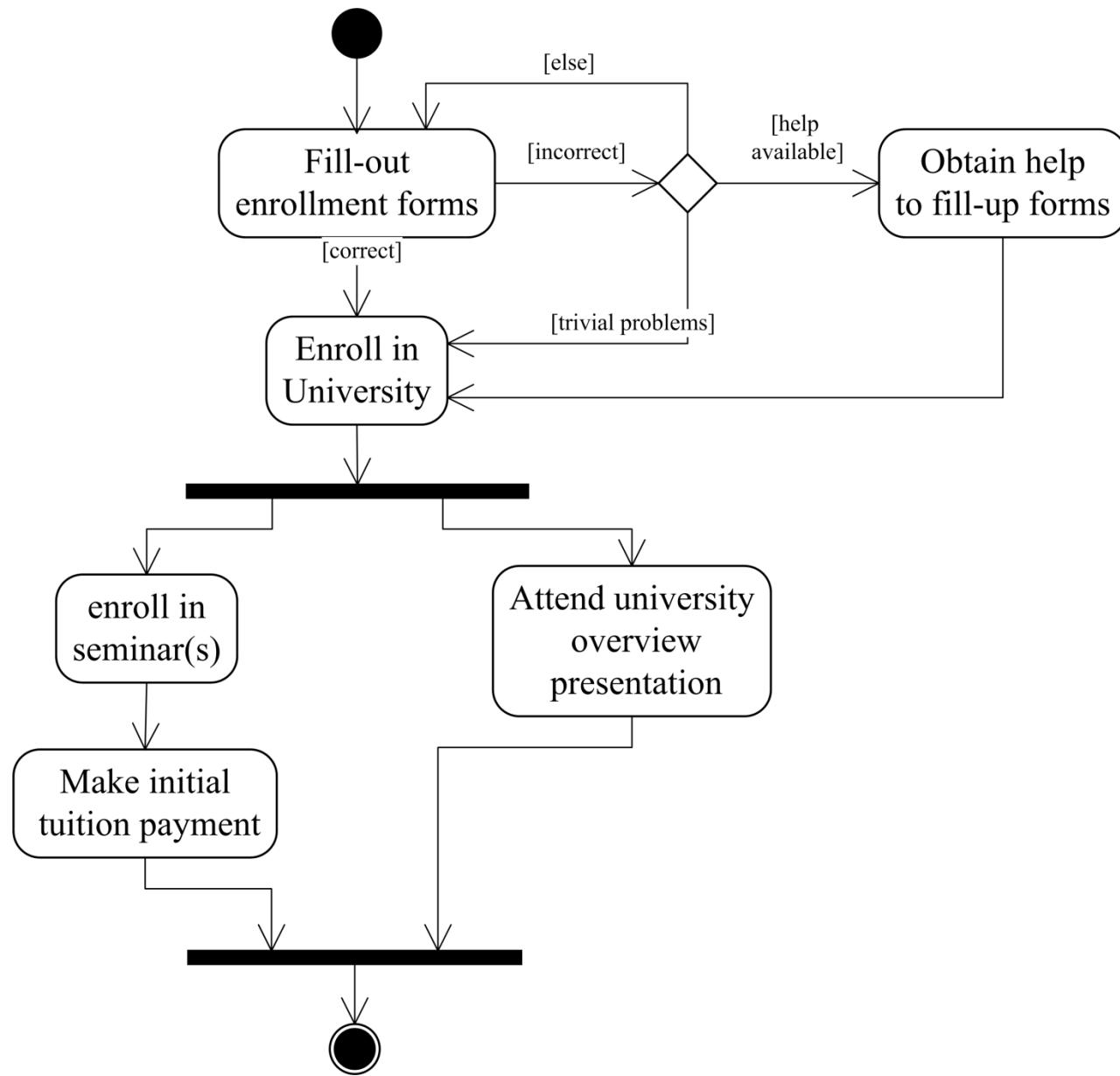
- ❖ Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.
- ❖ On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.
- ❖ Finally the parallel activities combine to close the order.



Activity Diagram

Activity Diagram Example - Student Enrollment (CW)

- ❖ An applicant wants to enroll in the university.
- ❖ The applicant hands a filled out copy of Enrollment Form.
- ❖ The registrar inspects the forms.
- ❖ The registrar determines that the forms have been filled out properly.
- ❖ The registrar informs student to attend in university overview presentation.
- ❖ The registrar helps the student to enroll in seminars
- ❖ The registrar asks the student to pay for the initial tuition.



UML

*State Machine/ State
Chart Diagram*

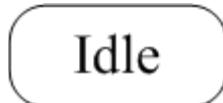
State Machine Diagram

- ❖ An object responds differently to the same event depending on what state it is in.
- ❖ A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.
- ❖ State diagrams are used to show possible states a single object can get into
 - ✓ i.e. shows states of an object .
- ❖ How object changes state in response to events
 - ✓ shows transitions between states

State Machine Diagram

States

- ❖ A state is denoted by a **round-cornered rectangle** with the name of the state written inside it.



Initial and Final States

- ❖ The **initial state** is denoted by a **filled black circle** and may be labeled with a name. The **final state** is denoted by a **circle with a dot inside** and may also be labeled with a name.



Initial state

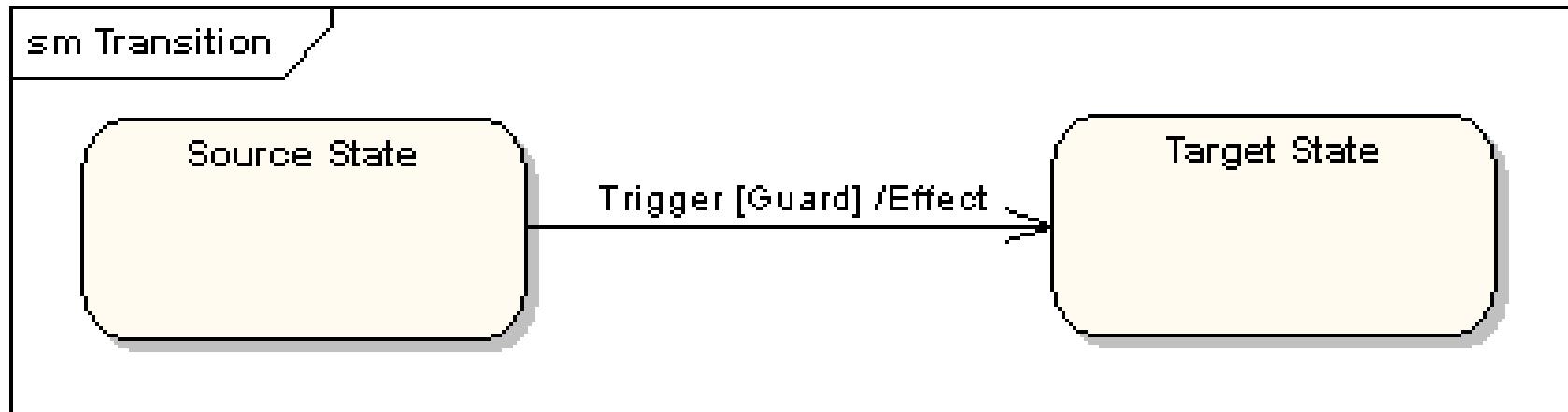


Final state

State Machine Diagram

Transitions

- ❖ Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.



- ❖ "Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State Machine Diagram

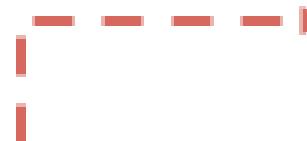
State Actions

- ❖ For each transition, an effect was associated with the transition.

- ✓ Entry
- ✓ Exit
- ✓ Do
- ✓ Defer

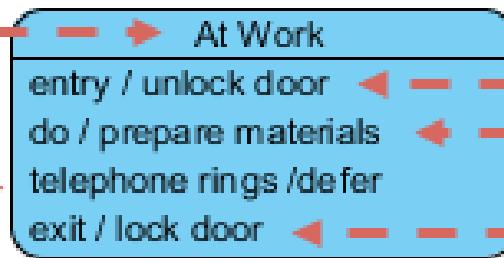
State Name

Name of state



Deferrable Trigger

Event that does not trigger any state transition,
but remain in the event pool ready for processing
when the object transitions to another state



Entry Activity

Action performed on entry to state

Do Activity

Action performed while in state

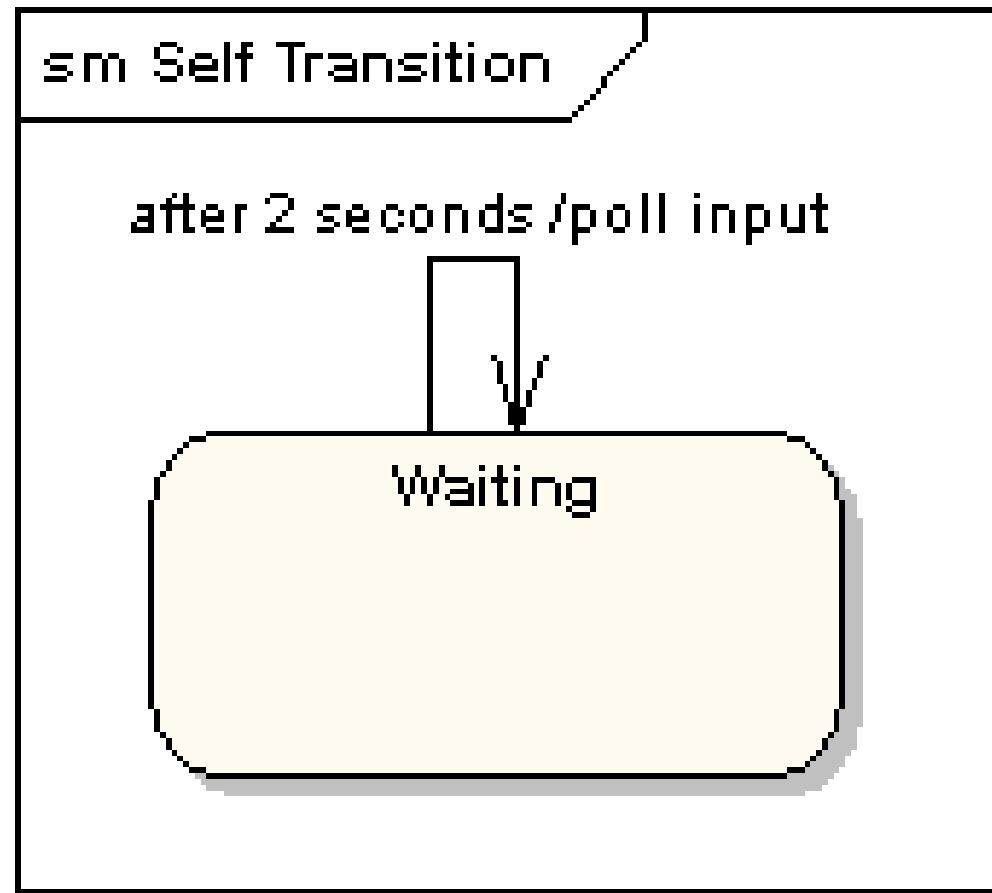
Exit Activity

Action performed on leaving state

State Machine Diagram

Self-Transitions

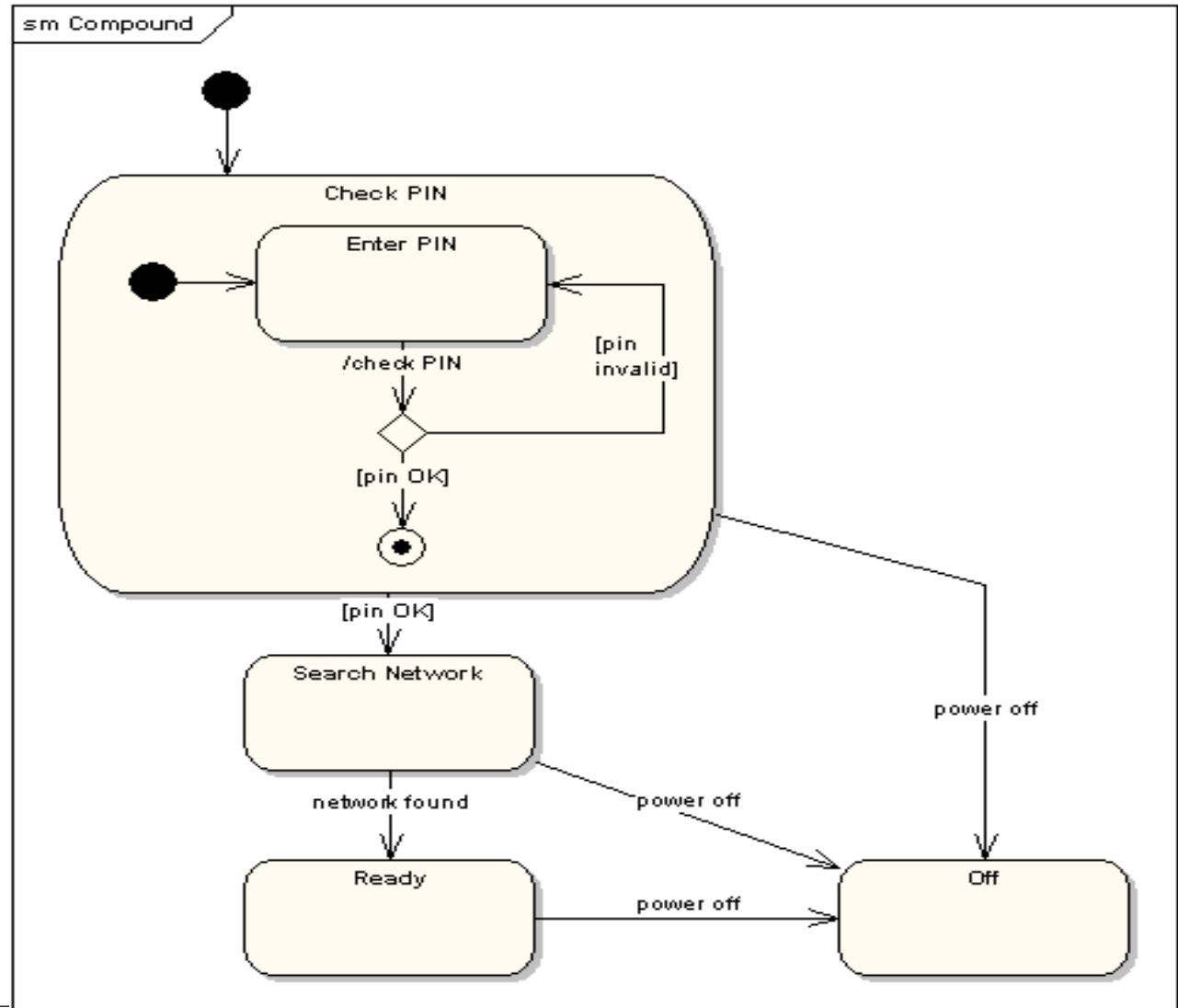
- ❖ A state can have a transition that returns to itself, as in the following diagram



State Machine Diagram

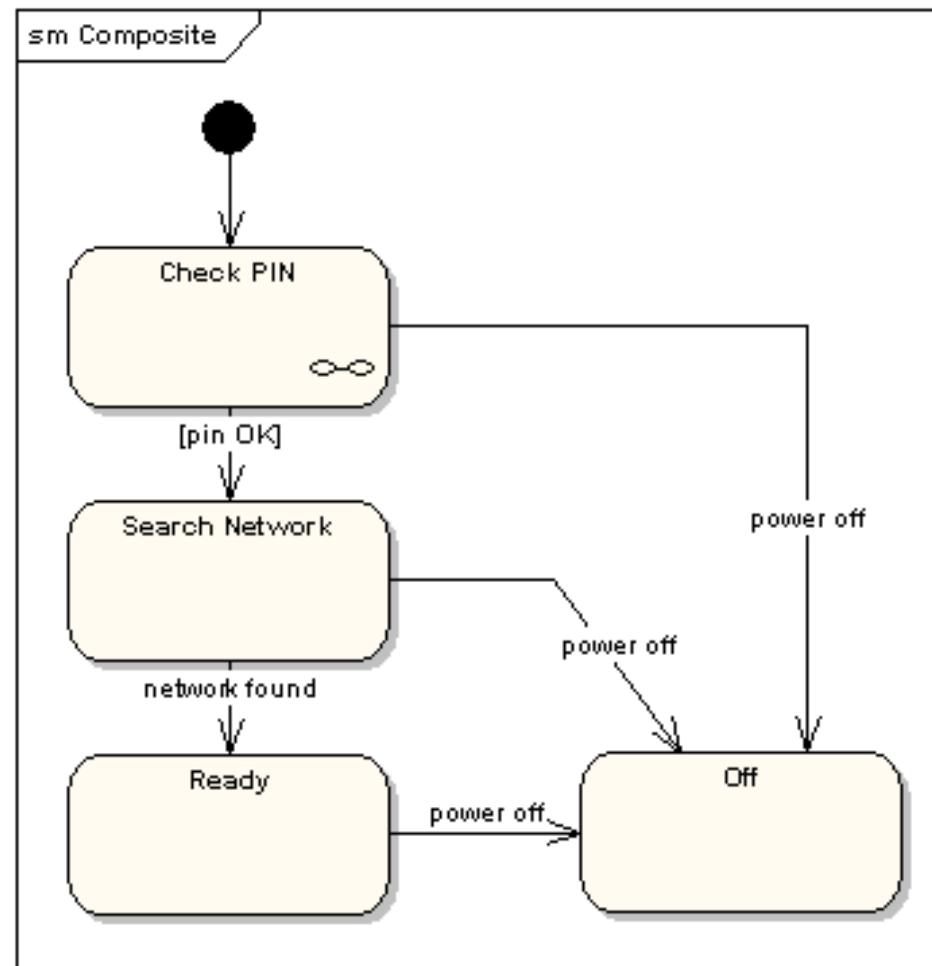
Compound States or Sub States

- A state machine diagram may include sub-machine diagrams, as in the example:



State Machine Diagram

- The alternative way to show the same information is as follows.

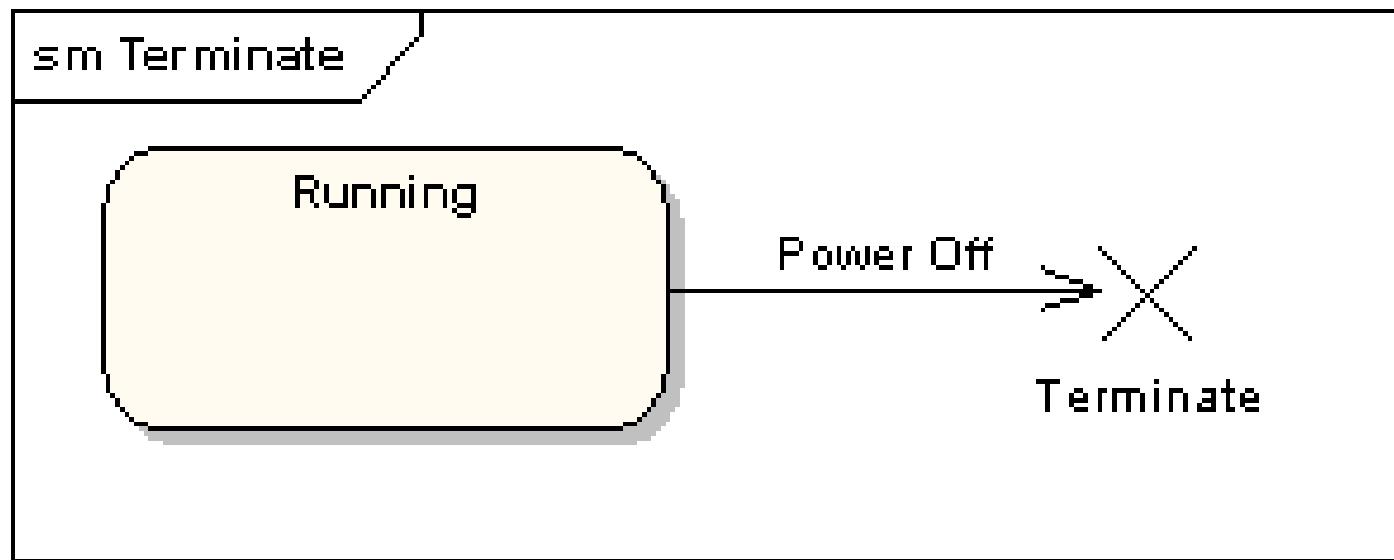


- The notation in above fig. indicates that the details of the **Check PIN** sub-machine are shown in a separate diagram.

State Machine Diagram

Terminate Pseudo-State

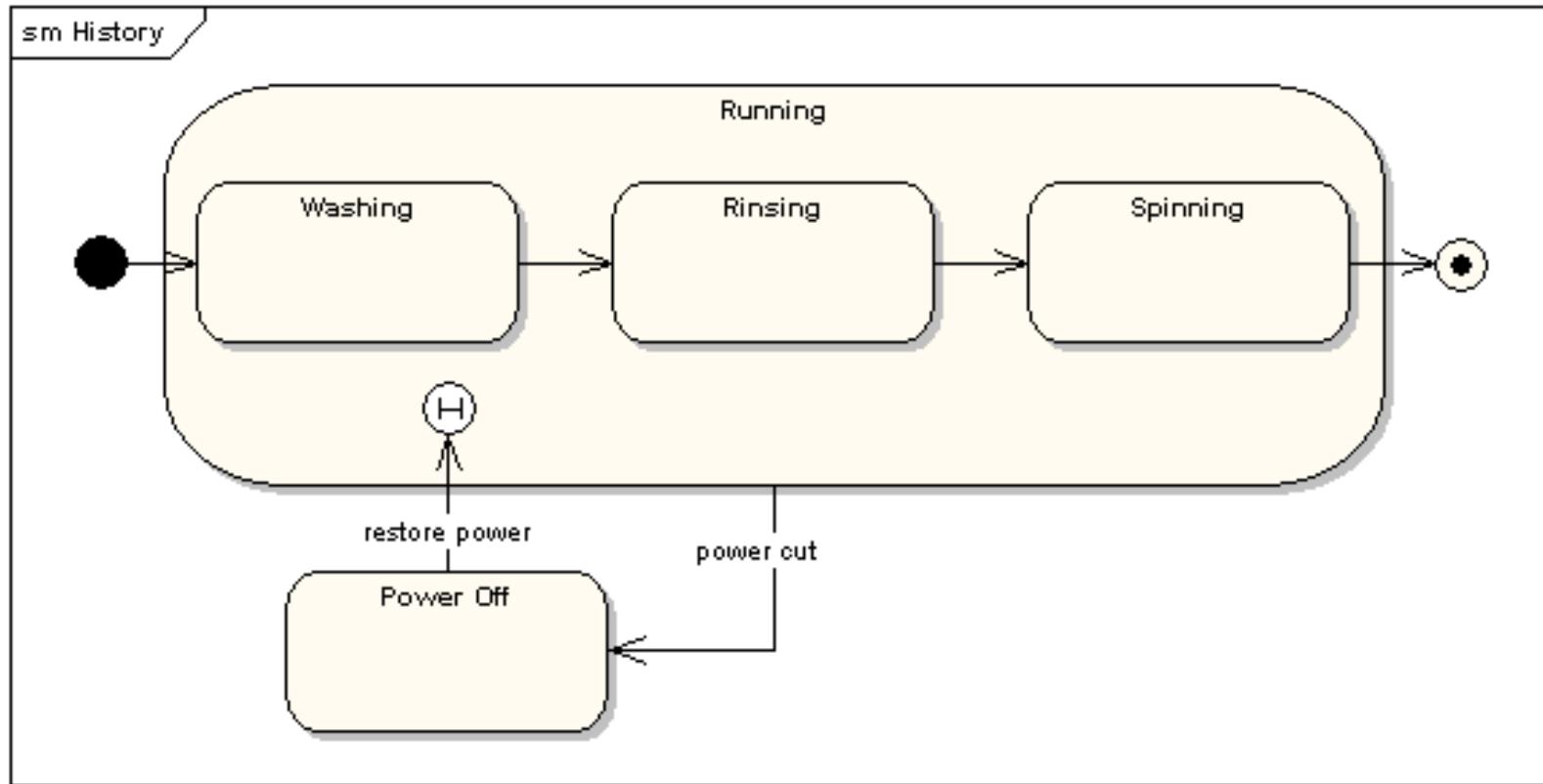
- ❖ Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended.
- ❖ A terminate pseudo-state is notated as a cross.



State Machine Diagram

History States

- ❖ A history state is used to remember the previous state of a state machine when it was interrupted.



- ❖ The above diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.

State Machine Diagram

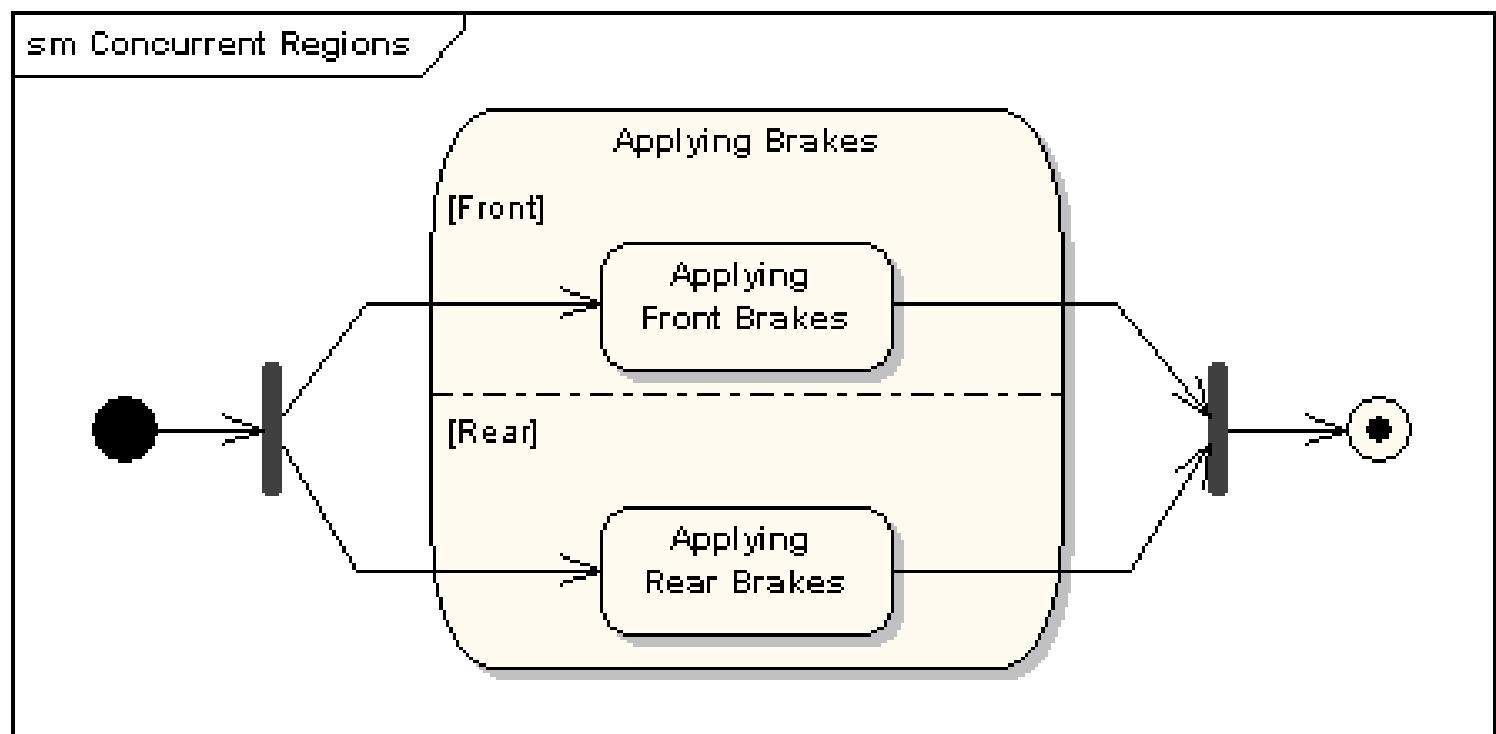
History States

- ❖ In this state machine shown in previous slide, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning".
- ❖ If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

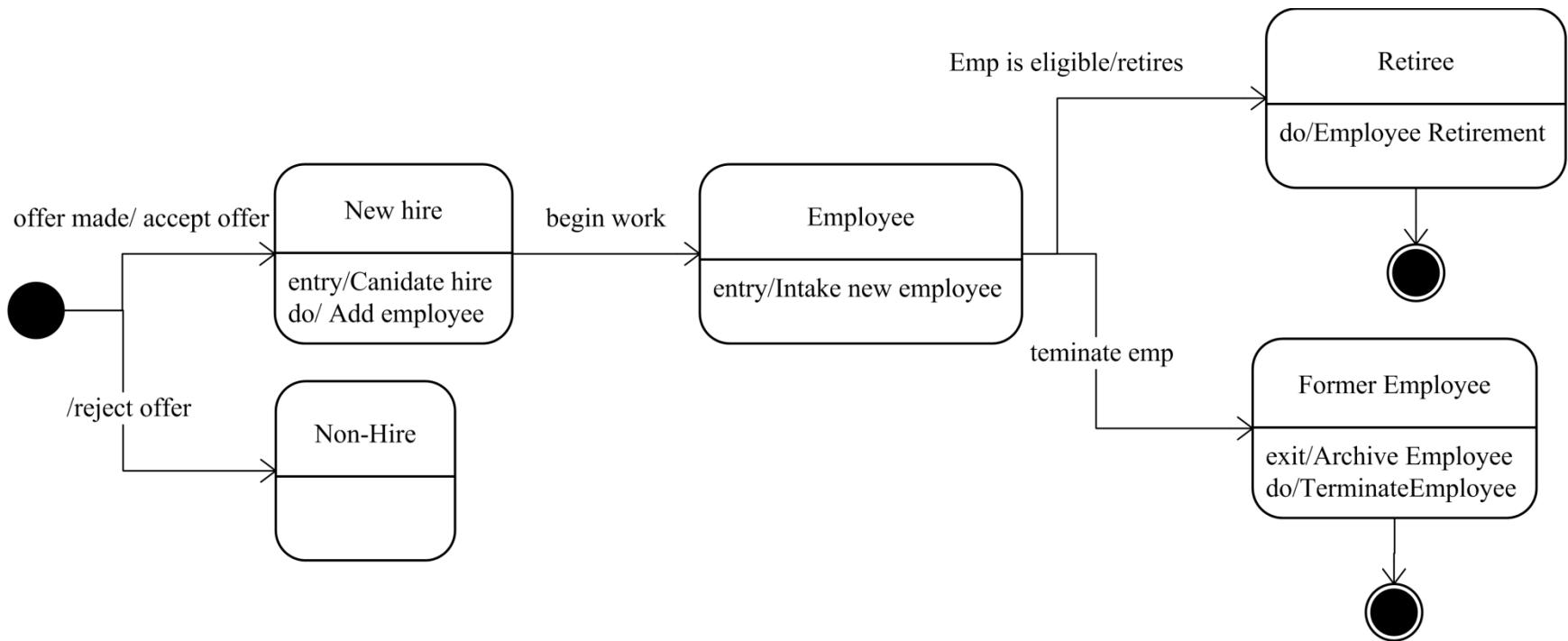
State Machine Diagram

Concurrent state machine

- ❖ A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

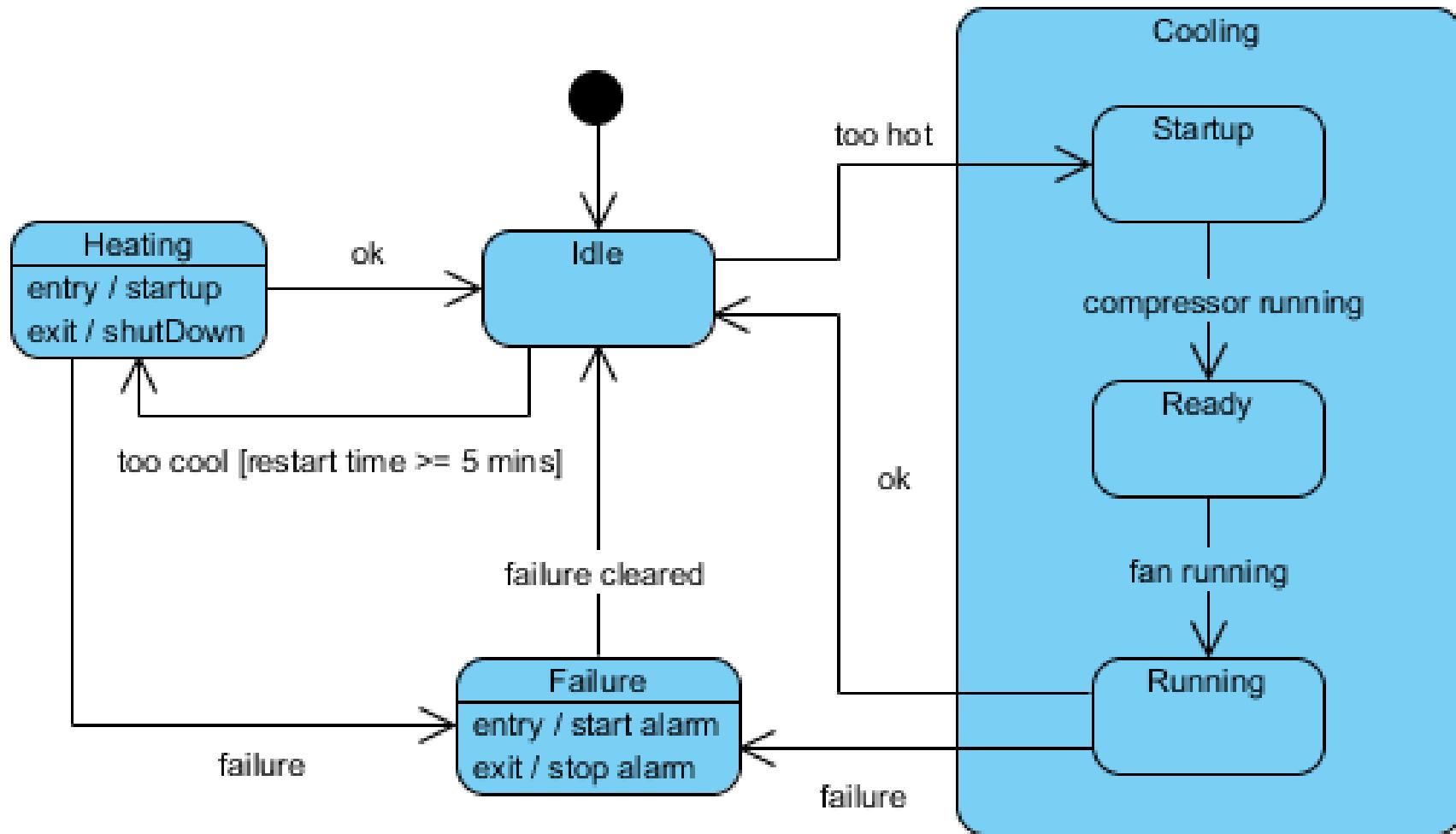


SMD of Recruitment process



State Machine Diagram

❖ Sub states or compounded states example



❖ Class work (description)

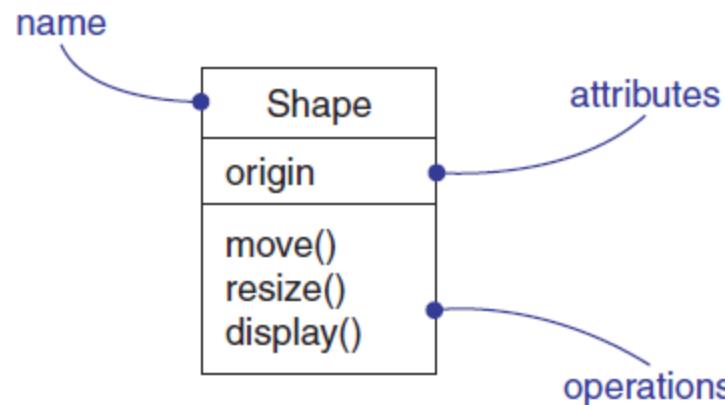
->Even roll -> **Too Hot**

->**Odd roll -> Too Cool**

Class Diagram

Class Diagram

- ❖ A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle.
- ❖ This notation permits you to visualize an abstraction apart from any specific programming language and in a way that lets you emphasize the most important parts of an abstraction: its name, attributes, and operations.



Class Diagram

Class Name:

- Every class must have a name that distinguishes it from other classes. A name is a textual string. That name alone is known as a simple name; a qualified name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name, as shown below.

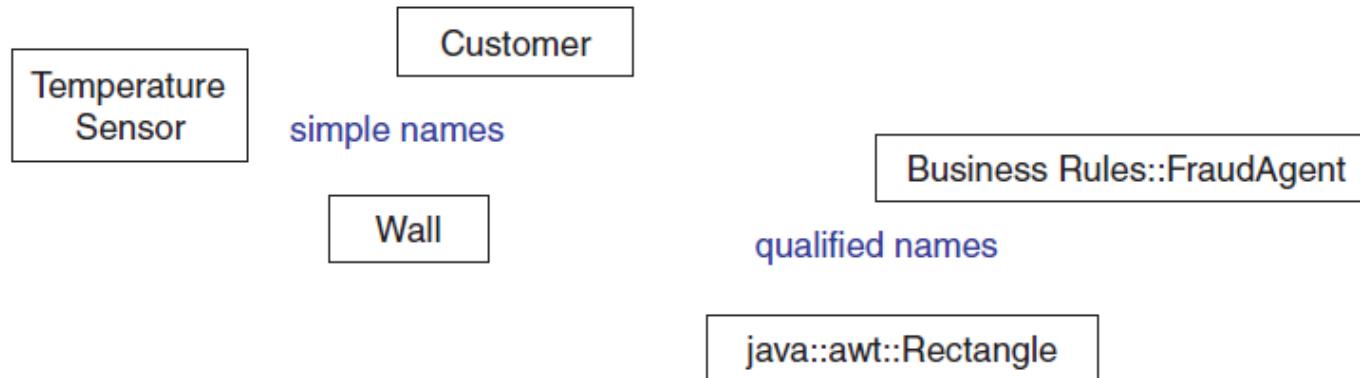


Fig: Simple and Quantified names

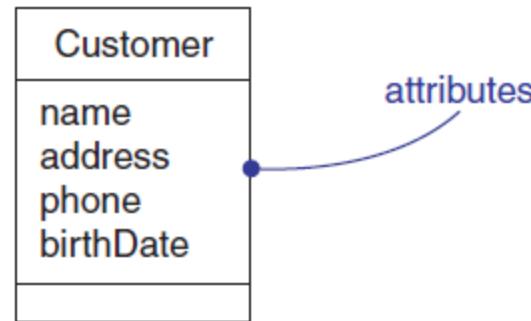
Notes:

A class name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the double colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, class names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling. **Typically, you capitalize the first letter of every word in a class name, as in Customer or TemperatureSensor.**

Class Diagram

Attributes:

- ❖ An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all.
- ❖ An attribute represents some property of the thing you are modeling that is shared by all objects of that class.
- ❖ For example, every wall has a height, width, and thickness; you might model our customers in such a way that each has a name, address, phone number, and date of birth.



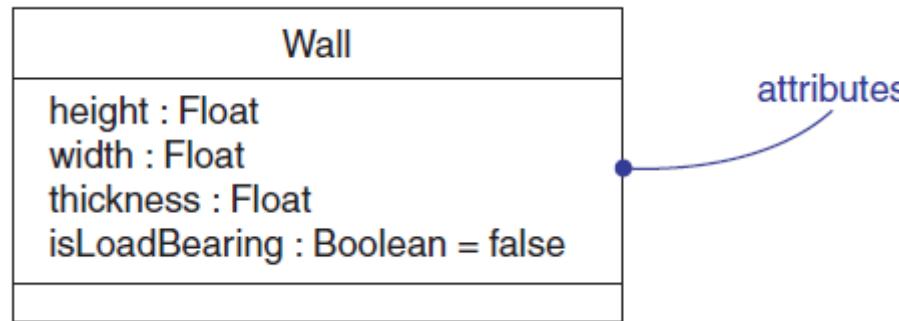
Note:

An attribute name may be text, just like a class name. In practice, an attribute name is a short noun or noun phrase that represents some property of its enclosing class.

Class Diagram

Attributes:

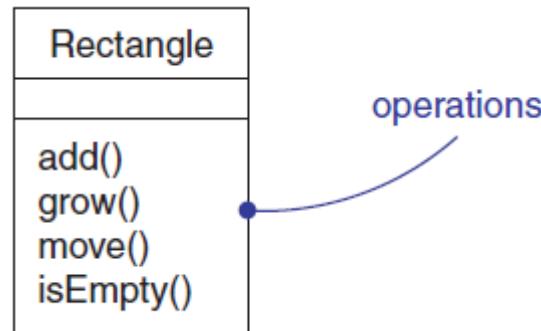
- we can further specify an attribute by stating its type and possibly a default initial value, as shown Figure below.



Class Diagram

Operations

- ❖ An operation is the implementation of a service that can be requested from any object of the class to affect behavior.
- ❖ In other words, an operation is an abstraction of something you can do to an object that is shared by all objects of that class.
- ❖ Operations may be drawn showing only their names, as in Figure below.



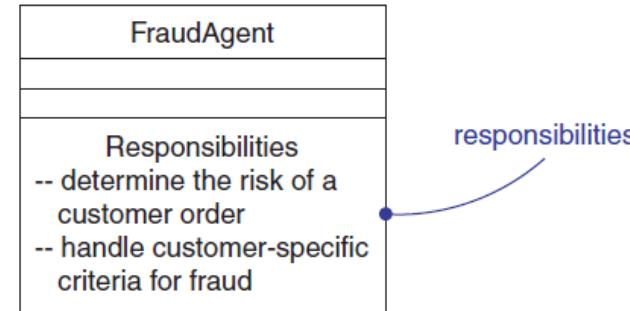
Note:

An operation name may be text, just like a class name. In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class.

Class Diagram

Responsibilities

- ❖ A responsibility is a contract or an obligation of a class.
- ❖ For example, A Wall class is responsible for knowing about height, width, and thickness; a FraudAgent class, as we might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.
- ❖ Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown below.



Note:

Responsibilities are just free-form text. In practice, a single responsibility is written as a phrase, a sentence, or (at most) a short paragraph.

Advance Class

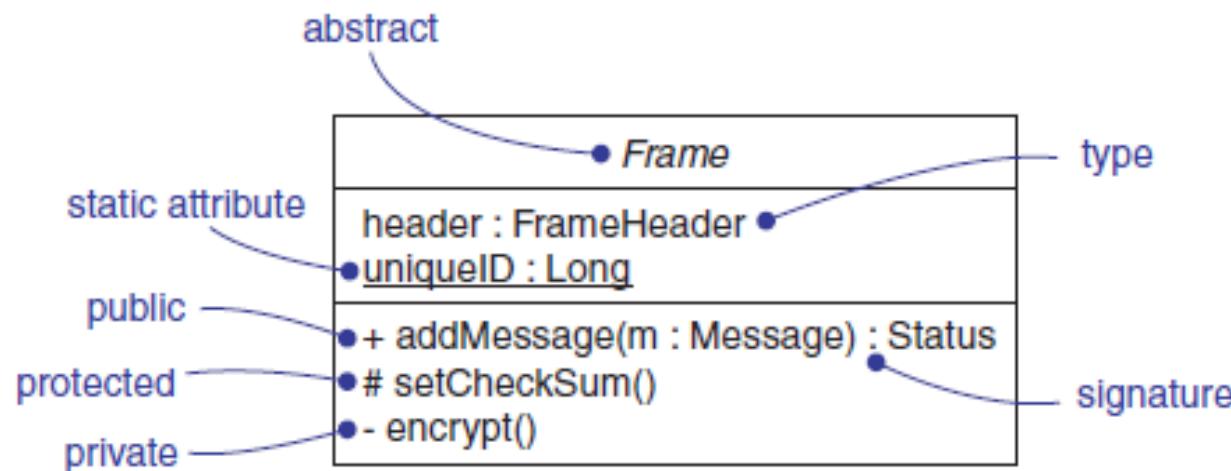
Advance class

Advance class

- ❖ Classifiers (and especially classes) have a number of advanced features beyond the simpler properties of attributes and operations described in the previous part: We can model multiplicity, visibility, signatures, polymorphism, and other characteristics.
- ❖ In the UML, we can model the semantics of a class so that you can state its meaning to whatever degree of formality you like.
- ❖ Early in a project, it's sufficient to say that we'll include a Customer class that carries out certain responsibilities.
- ❖ As we refine our architecture and move to construction, we'll have to decide on a structure for the class (its attributes) and a behavior (its operations) that are sufficient and necessary to carry out those responsibilities.
- ❖ Finally, as we evolve to the executable system, we'll need to model details, such as the visibility of individual attributes and operations, the concurrency semantics of the class as a whole and its individual operations, and the interfaces the class realizes.

Advance class

- ❖ The UML provides a representation for a number of advanced properties, as Figure below shows.
- ❖ This notation permits we to visualize, specify, construct, and document a class to any level of detail we wish.



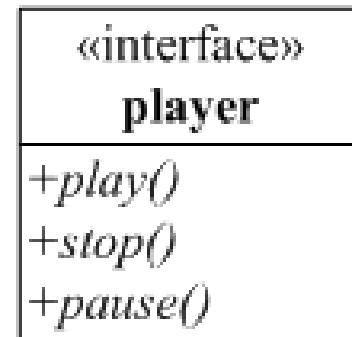
Advance class

Classifier

- ❖ A classifier is a mechanism which describes structural and behavioral features. The most important kind of classifier in the UML is the class.
 - ❖ A classifier represents structural aspects in terms of properties and behavioral aspects in terms of operations. Beyond these basic features, there are several advanced features like multiplicity, visibility, signatures, polymorphism and others.
 - ❖ *Not only the class, the UML provides a number of other kinds of classifiers to help we model.*

i. Interface

A collection of operations that are used to specify a service of a class or a component



Advance class

ii. Datatype

A type whose values are immutable, including primitive built-in types (such as numbers and strings) as well as enumeration types (such as Boolean).

iii. Association

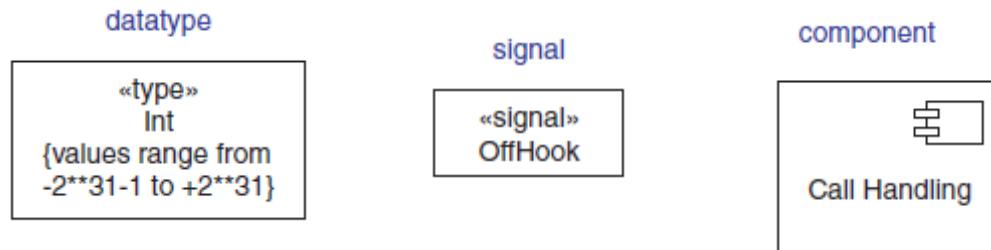
A description of a set of links, each of which relates two or more objects.

iv. Signal

The specification of an asynchronous message communicated between instances.

v. Component

A modular part of a system that hides its implementation behind a set of external interfaces.



Advance class

vi. Node

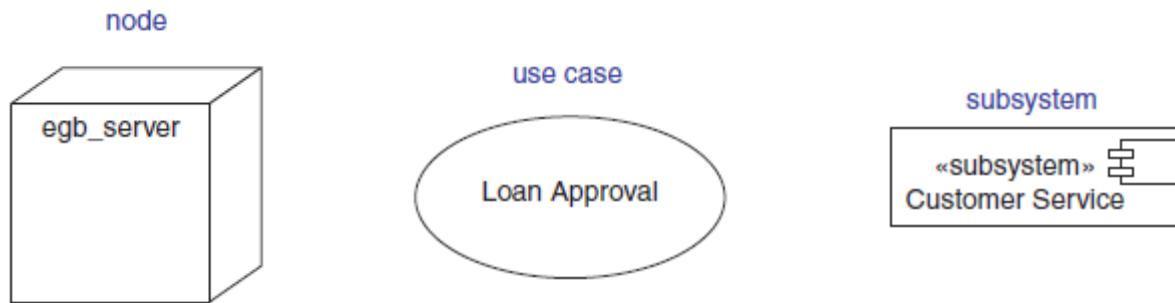
A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability.

vii. Use case

A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor

viii. Subsystem

A component that represents a major part of a system



Advance class

Visibility

The visibility of a feature specifies whether it can be used by other classifiers. In the UML, we can specify any of four levels of visibility.

1. Public

Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +.

2. Protected

Any descendant of the classifier can use the feature; specified by prepending the symbol #.

3. Private

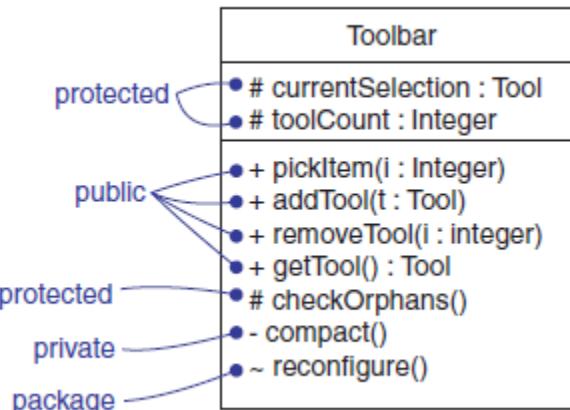
Only the classifier itself can use the feature; specified by prepending the symbol -.

3. Package

Only classifiers declared in the same package can use the feature; specified by prepending the symbol ~.

Note:

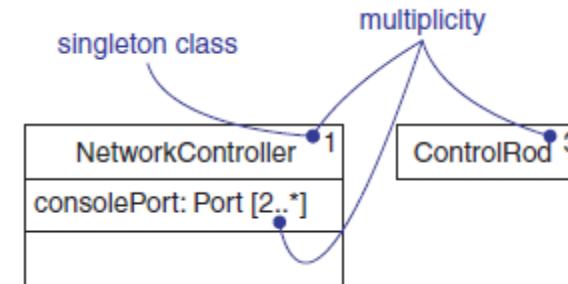
When we specify the visibility of a classifier's features, we generally want to hide all its implementation details and expose only those features that are necessary to carry out the responsibilities of the abstraction.



Advance class

Multiplicity

- ❖ The number of instances a class may have is called its multiplicity.
- ❖ In the UML, we can specify the multiplicity of a class by writing a multiplicity expression in the upper-right corner of the class icon
- ❖ For example, in Figure below, *NetworkController* is a singleton class (it can have *exactly one instance*). Similarly, there are exactly three instances of the class *ControlRod* in the system.
- ❖ Multiplicity applies to attributes, as well. we can specify the multiplicity of an attribute by writing a suitable expression in brackets just after the attribute name.
- ❖ For example, in the figure, there are two or more *consolePort* instances in the instance of *NetworkController*.



Advance class

Attributes:

- ❖ At the most abstract level we simply write each *attribute's name*. That's usually enough information for the average reader to understand the intent of our model.
- ❖ However, We can also specify the visibility, scope, and multiplicity of each attribute. There's still more. We can also specify the type, initial value, and changeability of each attribute.
- ❖ For example, the following are all legal attribute declarations:

■ origin	←Name only
■ + origin	←Visibility and name
■ origin : Point	←Name and type
■ name : String[0..1]	←Name, type, and multiplicity
■ origin : Point = (0,0)	←Name, type, and initial value
■ id: Integer {readonly}	←Name and property

Unless otherwise specified, attributes are always changeable. we can use the *readonly* property to indicate that the attribute's value may not be changed after the object is initialized.

Advance class

Operations:

- ❖ At the most abstract level, we will simply write each operation's name. That's usually enough information for the average reader to understand the intent of our model.
- ❖ however, we can also specify the visibility and scope of each operation. There's still more: we can also specify the parameters, return type, concurrency semantics, and other properties of each operation. Collectively, the name of an operation plus its parameters (including its return type, if any) is called the operation's signature.

For example, the following are all legal operation declarations:

- | | |
|-----------------------------|-----------------------|
| ■ display | ←Name only |
| ■ + display | ←Visibility and name |
| ■ set(n : Name, s : String) | ←Name and parameters |
| ■ getID() : Integer | ←Name and return type |
| ■ restart() {guarded} | ←Name and property |

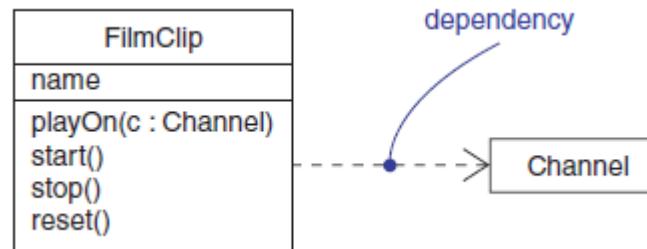
Class Relationships

Class Relationships

A relationship is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations.

Dependencies

- ❖ A dependency is a relationship that states that one thing (for example, class Window) uses the information and services of another thing (for example, class Event), but not necessarily the reverse.
- ❖ Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on.



Class Relationships

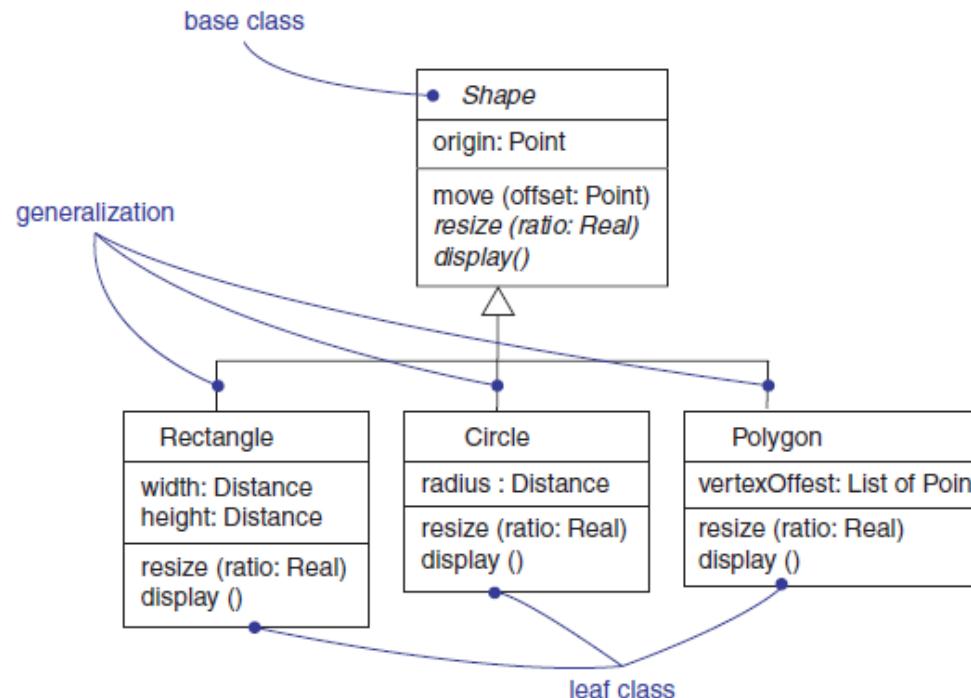
Dependencies

- ❖ Most often, we will use dependencies between classes to show that one class uses operations from another class or it uses variables or arguments typed by the other class..
- ❖ This is very much a using relationship—if the used class changes, the operation of the other class may be affected as well, because the used class may now present a different interface or behavior.
- ❖ In the UML we can also create dependencies among many other things, especially notes and packages.

Class Relationships

Generalizations

- ❖ A generalization is a relationship between a general kind of thing (called the superclass or parent) and a more specific kind of thing (called the subclass or child).
- ❖ Graphically, generalization is rendered as a solid directed line with a large unfilled triangular arrowhead, pointing to the parent, as shown below.
- ❖ Use generalizations when we want to show parent/child relationships.



Class Relationships

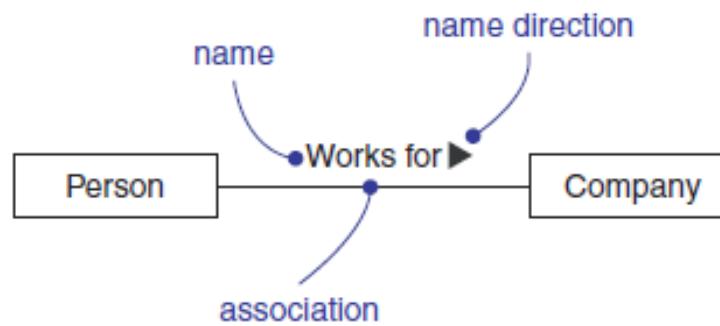
Associations

- ❖ An association is a structural relationship that specifies that objects of one thing are connected to objects of another.
- ❖ Given an association connecting two classes, we can relate objects of one class to objects of the other class.
- ❖ An association that connects exactly two classes is called a *binary* association. Although it's not as common, we can have associations that connect more than two classes; these are called *n-ary* associations.
- ❖ Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when we want to show structural relationships.
- ❖ Beyond this basic form, there are four adornments that apply to associations.
 - ✓ Name
 - ✓ Role
 - ✓ Multiplicity
 - ✓ Aggregation

Class Relationships → Associations

Name

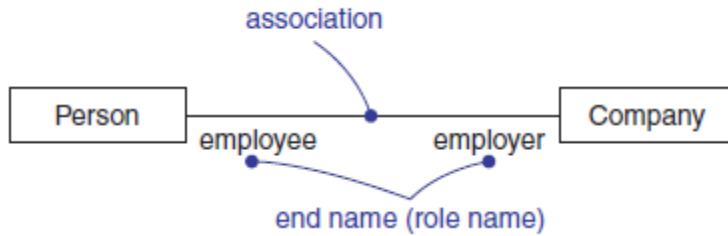
- ❖ An association can have a name, and we use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning
- ❖ Also we can give a direction to the name by providing a direction triangle that points in the direction we intend to read the name, as shown in Figure below.



Class Relationships → Associations

Role

- ❖ When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association.
- ❖ we can explicitly name the role a class plays in an association. The role played by an end of an association is called an end name.
- ❖ As in Figure below, *the class Person playing the role of employee is associated with the class Company playing the role of employer*.



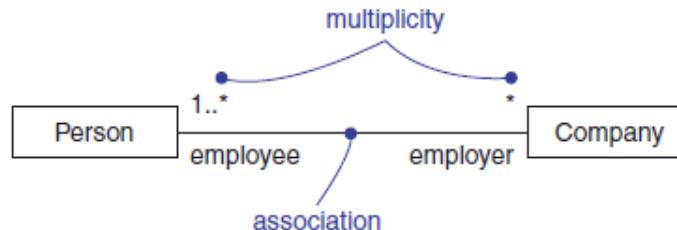
Note:

The same class can play the same or different roles in other associations.

Class Relationships → Associations

Multiplicity

- ❖ In many modeling situations, it's important for us to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role.
- ❖ It represents a range of integers specifying the possible size of the set of related objects. It is written as an expression with a minimum and maximum value, which may be the same; two dots are used to separate the minimum and maximum values.
- ❖ *The number of objects must be in the given range. we can show a multiplicity of as:*
 - ✓ exactly one (1)
 - ✓ zero or one (0..1)
 - ✓ many (0..*), or one or more (1..*).
 - ✓ integer range (such as 2..5)
 - ✓ exact number (for example, 3, which is equivalent to 3..3).

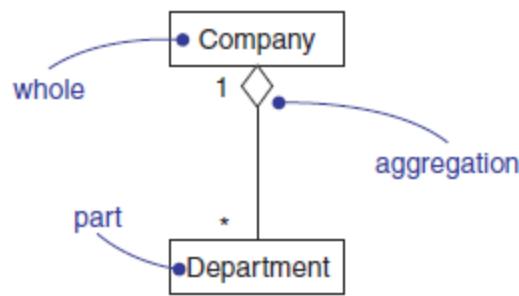


In above figure, each company object has as employee one or more person objects (multiplicity 1..*); each person object has as employer zero or more company objects (multiplicity *, which is equivalent to 0..*).

Class Relationships → Associations

Aggregation

- ❖ A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other.
- ❖ Sometimes we will want to model a “whole/part” relationship, in which one class represents a larger thing (the “whole”), which consists of smaller things (the “parts”). This kind of relationship is called aggregation, which represents a “has-a” relationship, meaning that an object of the whole has objects of the part.
- ❖ Aggregation is really just a special kind of association and is specified by adorning a plain association with an unfilled diamond at the whole end, as shown in Figure below.



Advance Relationships

→*Assignment*

→See book *G. Booch, J. Rumbaugh, I Jacobson/ pg. 133*

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

OBJECT ORIENTED SOFTWARE ENGINEERING
Chapter Four
Object Oriented Analysis

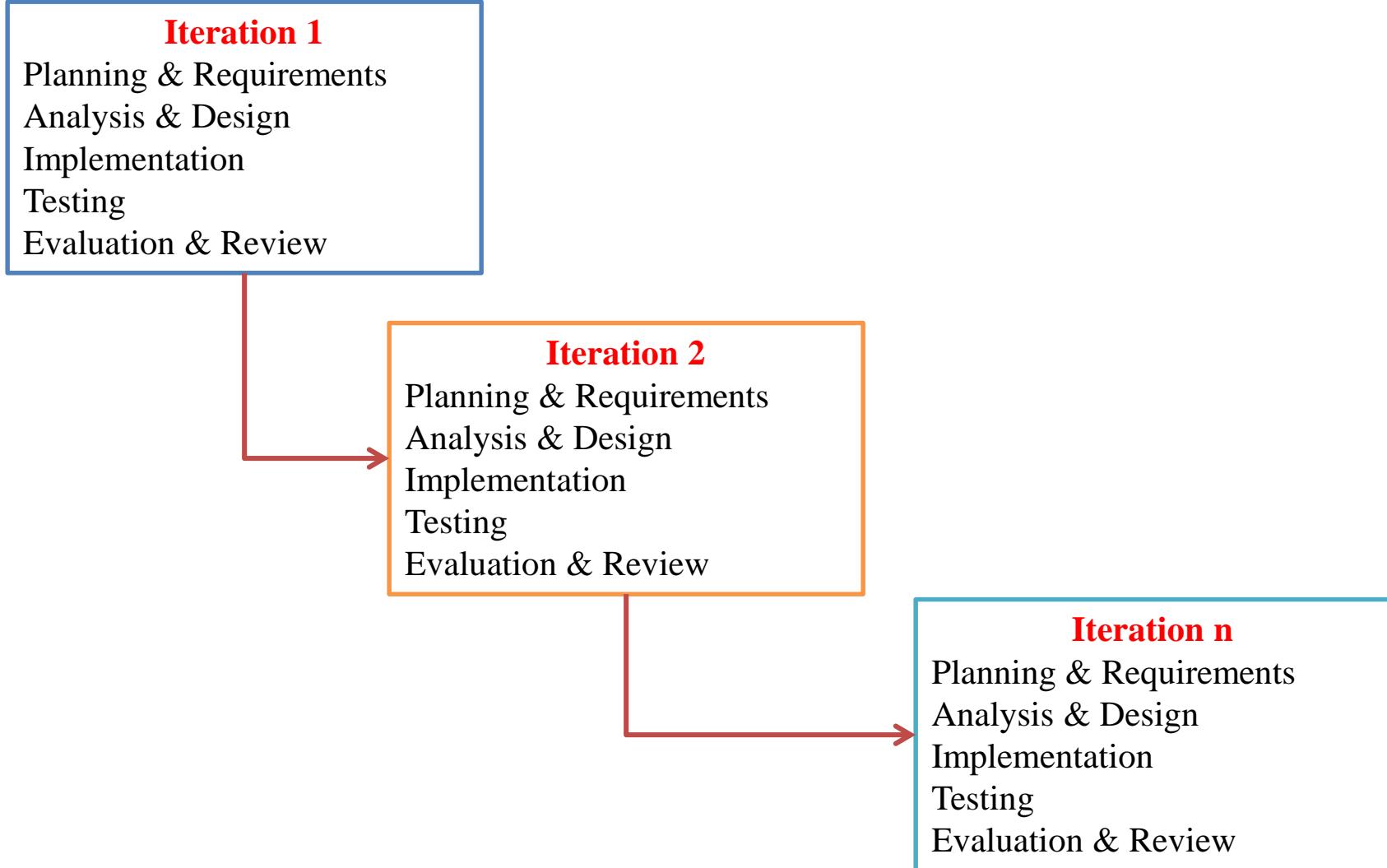
by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Iterative Development

Iterative Development

- ❖ The iterative process model is a cyclical process in which you make and test incremental adjustments. The iterative model is popular in technology, engineering, software development, design, qualitative research, project management (especially in Agile and Scrum), and more.
- ❖ At the most basic level, the process relies on a continual cycle of planning, analysis, implementation, testing, and evaluation. The iterative process starts with initial planning and overall requirements. Then the first, prioritized portion of the project becomes the initial cycle of development. That segment is refined by trial and error. Once finished, it forms the basis for the next chunk of the project. Each cycle improves on the overall product or project. The pace of this process is related to how effectively you work through the cycle. The iterative development process is a five steps process.

Iterative Development



Iterative Development

© **Planning and Requirements:**

In this stage, map out the initial requirements, gather the related documents, and create a plan and timeline for the first iterative cycle.

© **Analysis and Design:**

Finalize the business needs, database models, and technical requirements based on the plan. Create a working architecture, schematic, or algorithm that satisfies your requirements.

© **Implementation:**

Develop the functionality and design required to meet the specifications.

© **Testing:**

Identify and locate what's not working or performing to expectations. Stakeholders, users, and product testers weigh in with their experience.

© **Evaluation and Review:**

Compare this iteration with the requirements and expectations.

Iterative Development

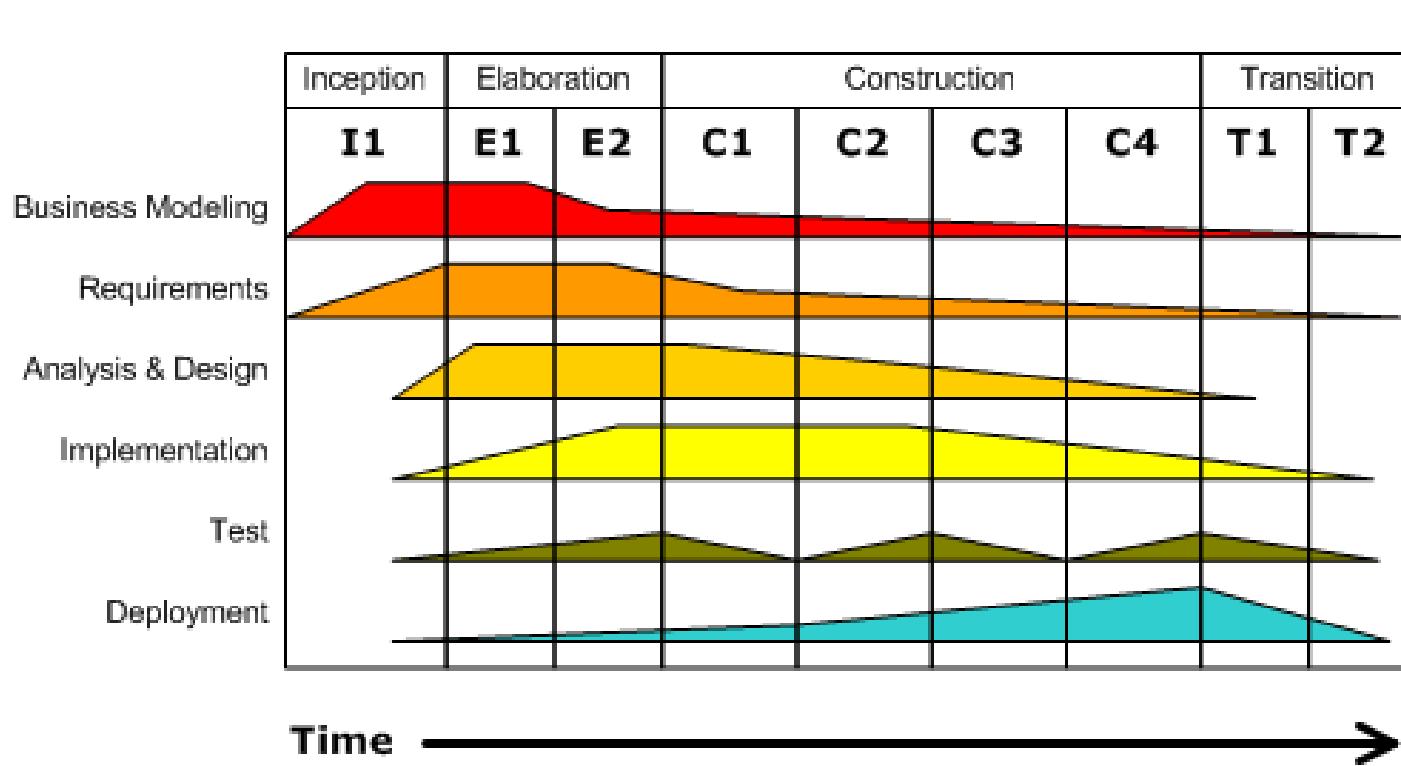
- © After we complete these steps, it's time to tackle the next cycle. In the iterative process, the product goes back to step one to build on what's working. Identify what you learned from the previous iteration.
- © This iterative development, sometimes called circular or evolutionary development, is based on refining the first version through subsequent cycles, especially as you gather and include requirements.
- © It allows you to remain flexible as you identify new needs or unexpected business issues.

Unified process and UP phases

Unified Process

Unified Process

- The most successful approach for object oriented software development is Unified process/ Rational Unified Process (UP/ RUP).
- It is an approach that combines iterative and risk driven development into a well documented process description.



UP Phases (*Abstract*)

① Inception:

- © The idea for the project is stated. The development team determines if the project is worth pursuing and what resources will be needed.

② Elaboration

- © The project's architecture and required resources are further evaluated. Developers consider possible applications of the software and costs associated with the development.

③ Construction

- © The project is developed and completed. The software is designed, written, and tested.

④ Transition

- © The software is released to the public. Final adjustments or updates are made based on feedback from end users.

UP Phases (*description*)

Inception

- © The requirements are gathered.
- © Feasibility study and scope of the project are determined.
- © Actors and their interactions are analyzed.

Elaboration

- © Project plan is developed.
- © Risk assessment is performed.
- © Non-functional requirements are elaborated.
- © Software architecture is described.
- © Use case model is completed.

UP Phases

① Construction

- © All the components are developed and integrated.
- © All features are tested.
- © In each iteration, refactoring (clarifying and simplifying the design of existing code, without changing its behavior) is done.
- © Stable product should be released.

② Transition

- © Software product is launched to user.
- © Deployment baseline should be complete.
- © Final product should be released.

UP Conclusion:

- © Input to a process is the needs or requirements of the system.
- © Process is the set of activities to reach goal.
- © Output is the software product.

UP Disciplines

There are nine disciplines in Unified process, they are:

Model

The goal of this discipline is to understand the business of the organization, the problem domain being addressed by the project, and to identify a viable solution to address the problem domain.

Requirements

The goal of this discipline is to elicit stakeholder feature/function requirements in order to define the scope of the project.

Analysis and Design

The goal of this discipline is to define the requirements into actionable and executable designs and models.

UP Disciplines

Implementation

The goal of this discipline is to transform your model(s) into executable code and to perform a basic level of testing, in particular unit testing.

Test

The goal of this discipline is to perform an objective evaluation to ensure quality. This includes finding defects, validating that the system works as designed, and verifying that the requirements are met.

Deployment

The goal of this discipline is to plan for the delivery of the system and to execute the plan to make the system available to end users.

UP Disciplines

Configuration Management

The goal of this discipline is to manage access to your project artifacts. This includes not only tracking artifact versions over time but also controlling and managing changes to them.

Project Management.

The goal of this discipline is to direct the activities that takes place on the project. This includes managing risks, directing people (assigning tasks, tracking progress, etc.), and coordinating with people and systems outside the scope of the project to be sure that it is delivered on time and within budget.

UP Disciplines

Environment

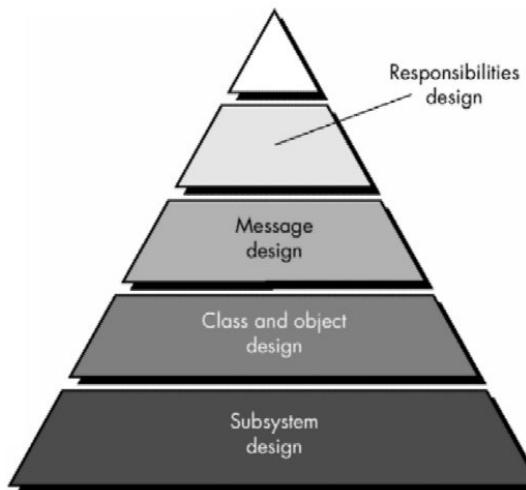
The goal of this discipline is to support the rest of the effort by ensuring that the proper process, guidance (standards and guidelines), and tools (hardware, software, etc.) are available for the team as needed.

Agile Unified Process

→Assignment

Object-Oriented Design

Four layers of the OO design pyramid:

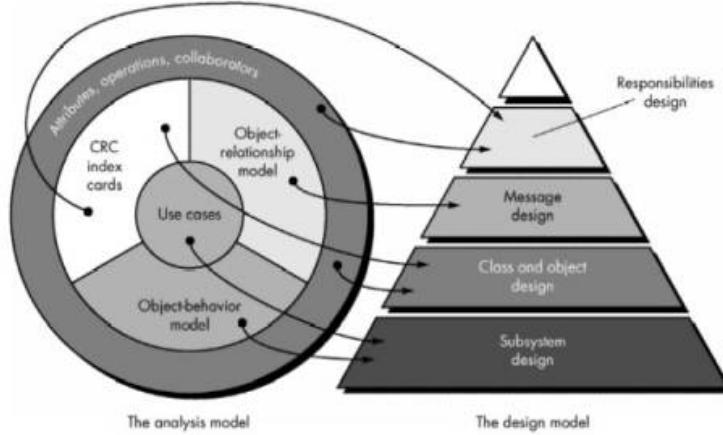


1. **Subsystem layer** contains a representation of each of the subsystems that:
 - a. enable the software to achieve its customer-defined requirements
 - b. implement the technical infrastructure that supports customer requirements.
 2. **Class and object layer** contains:
 - a. class hierarchies that enable system to be created using generalizations
 - b. representations of each object.
 3. **Message layer :**
 - a. contains the design details that enable each object to communicate with its collaborators
 - b. establishes the external and internal interfaces for the system.
 4. **Responsibilities layer** contains the data structure and algorithmic design for all attributes and operations for each object.
- **Hidden Underlying Layer -> Foundation layer:**
 - Focuses on the design of *domain objects* (called *design patterns*).
 - Domain objects provide support for human/computer interface activities, task management, and data management.
 - Domain objects can be used to flesh out the design of the application itself.

Conventional vs. OO Approaches

- Conventional approaches: apply a distinct notation and set of heuristics to map the analysis model into a design model.
- Each element of the conventional analysis model maps into one or more layers of the design model.
- OOD applies:
 - **data design** when attributes are represented
 - **interface design** when a messaging model is developed
 - **component-level** (procedural) design for the design of operations.

- Architecture of an OO design has more to do with the collaborations among objects than with the flow of control between components of the system.
- Relationship between the OO analysis model and design model:



- **Subsystem design** is derived by:
 - considering overall customer requirements (represented with use-cases)
 - events and states that are externally observable (the object-behavior model)
- **Class and object design :**
 - mapped from the description of attributes, operations, and collaborations contained in the CRC model.
- **Message design** driven by:
 - the object-relationship model
 - responsibilities design is derived using the attributes, operations, and collaborations described in the CRC model.

Design Issues

Five criteria for judging a design method's ability to achieve modularity:

- *Decomposability*—the facility with which a design method helps the designer to decompose a large problem into subproblems that are easier to solve.
- *Composability*—the degree to which a design method ensures that program components (modules), once designed and built, can be reused to create other systems.
- *Understandability*—the ease with which a program component can be understood without reference to other information or other modules.
- *Continuity*—the ability to make small changes in a program and have these changes manifest themselves with corresponding changes in just one or a very few modules.
- *Protection*—an architectural characteristic that will reduce the propagation of side effects if an error does occur in a given module.

Five basic design principles that can be derived for modular architectures:

- (1) **linguistic modular units**
 - programming language used should be capable of supporting the modularity defined directly
- (2) **few interfaces**
 - to achieve low coupling - the number of interfaces between modules should be minimized
- (3) **small interfaces (weak coupling)**
 - to achieve low coupling - the amount of information that moves across an interface should be minimized
- (4) **explicit interfaces**

- components should communicate in an obvious and direct way
- (5) information hiding**
- all information about a component is hidden from outside access

The OOD Landscape

Most important early OOD methods:

- **The Booch method.**
 - encompasses both "micro development process" and "macro development process."
 - **design context**
 - **macro development** encompasses:
 - architectural planning activity that:
 - clusters similar objects in separate architectural partitions
 - layers objects by level of abstraction
 - identifies relevant scenarios
 - creates a design prototype
 - validates the design prototype by applying it to usage scenarios.
 - **micro development:**
 - defines a set of "rules" that govern:
 - the use of operations and attributes and the domain-specific policies for memory management
 - error handling and other infrastructure functions
 - develops scenarios that describe the semantics of the rules and policies
 - creates a prototype for each policy
 - instruments and refines the prototype
 - reviews each policy so that it "broadcasts its architectural vision"
 - **The Rumbaugh method.**
 - encompasses a design activity that encourages design to be conducted at two different levels of abstraction:
 - **System design -**
 - focuses on the layout of components that are needed to construct a complete product or system
 - **analysis model** is partitioned into subsystems -> allocated to processors and tasks.
 - strategy for implementing data management is defined
 - global resources and the control mechanisms required to access them are identified
 - **Object design -**
 - emphasizes the detailed layout of an individual object.
 - operations are selected from the analysis model and algorithms are defined for each operation.
 - data structures appropriate to attributes and algorithms are represented.
 - classes and class attributes are designed so they optimize data access and improve computational efficiency.
 - a messaging model is created to implement the object relationships (associations).
 - **The Jacobson method.**
 - design model emphasizes traceability to the OOSE analysis model
 1. idealized analysis model is adapted to fit the real world environment.
 2. primary design objects, called *blocks* are created and categorized as interface blocks, entity blocks, and control blocks
 3. communication between blocks during execution is defined
 4. blocks are organized into subsystems.
 - **The Coad and Yourdon method.**
 - design approach addresses not only the application but also the infrastructure for the application
 - focuses on the representation of four major system components:
 1. problem domain component
 2. human interaction component
 3. task management component
 4. data management component.

To perform object-oriented design, a software engineer should perform the following generic steps:

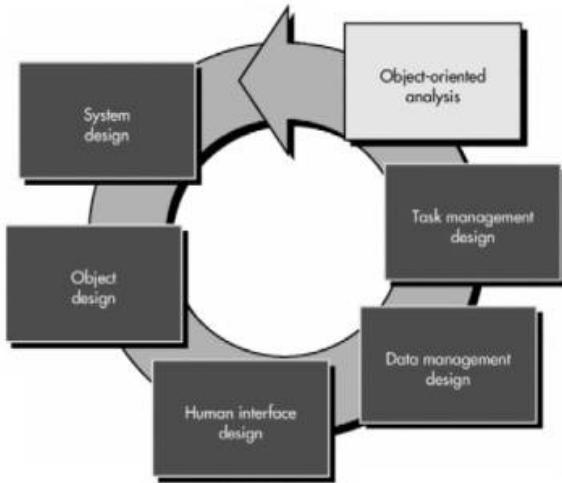
1. Describe each subsystem and allocate it to processors and tasks.
2. Choose a design strategy for implementing data management, interface support, and task management.
3. Design an appropriate control mechanism for the system.
4. Perform object design by creating a procedural representation for each operation and data structures for class attributes.
5. Perform message design using collaborations between objects and object relationships.
6. Create the messaging model.
7. Review the design model and iterate as required.

Note: design steps are iterative.

Unified Approach to OOD

Unified Modeling Language (UML)

- During analysis modeling the user model and structural model views are represented
 - provide insight into the usage scenarios for the system (providing guidance for behavioral modeling)
 - establish a foundation for the implementation and environment model views by identifying and describing the static structural elements of the system.
- UML is organized into two major design activities:
 - **system design:**
 - Primary objective of UML *system design* is to represent the software architecture
 - **conceptual architecture** is concerned with the structure of the static class model and the connections between components of the model
 - **module architecture** describes the way the system is divided into subsystems or modules and how they communicate by exporting and importing data
 - **code architecture** defines how the program code is organized into files and directories and grouped into libraries.
 - **execution architecture** focuses on the dynamic aspects of the system and the communication between components as tasks and operations execute.
 - **object design**
 - UML *object design* focuses on a description of objects and their interactions with one another.
 - detailed specification of attribute data structures and a procedural design of all operations are created.
 - visibility for all class attributes is defined and interfaces between objects are elaborated to define the details of a complete messaging model
- System and object design are extended to consider:
 - design of user interfaces
 - user model view drives the user interface design process
 - provides a scenario that is elaborated iteratively to become a set of interface classes
 - data management with the system to be built
 - establishes a set of classes and collaborations that allow the system (product) to manage persistent data (e.g., files and databases)
 - task management for the subsystems that have been specified.
 - establishes the infrastructure that organizes subsystems into tasks and then manages task concurrency.
- Process flow for OOD



The System Design Process

- System design develops the architectural detail required to build a system or product.
- System design encompasses the following activities:
 1. Partition the analysis model into subsystems.
 2. Identify concurrency that is dictated by the problem.
 3. Allocate subsystems to processors and tasks.
 4. Develop a design for the user interface.
 5. Choose a basic strategy for implementing data management.
 6. Identify global resources and the control mechanisms required to access them.
 7. Design an appropriate control mechanism for the system, including task management.
 8. Consider how boundary conditions should be handled.
 9. Review and consider trade-offs.

1. Partition the Analysis Model

- Partition the analysis model to define cohesive collections of classes, relationships, and behavior -> **subsystems**
- These design elements are packaged as a **subsystem**.
 - all of the elements of a subsystem share some property in common
 - all may be involved in accomplishing the same function
 - they may reside within the same product hardware, or they may manage the same class of resources
- **Subsystems** are characterized by their responsibilities -> identified by the services that they provide
 - a **service** is a collection of operations that perform a specific function (e.g., managing word-processor files, producing a three-dimensional rendering, translating an analog video signal into a compressed digital image)
- **Subsystems** should conform to the following design criteria:
 - Subsystem should have a well-defined interface through which all communication with the rest of the system occurs.
 - classes within a subsystem should collaborate only with other classes within the subsystem.
 - The number of subsystems should be kept low.
- A subsystem can be partitioned internally to help reduce complexity.
- When two subsystems communicate with one another, they can establish a **client/server link** or a **peer-to-peer link**
 - **client/server link** –
 1. each subsystem takes on one of the roles implied by client and server.

- 2. service flows from server to client in only one direction.
 - **peer-to-peer link** –
 - 3. services may flow in either direction.
- **layering** occurs when a system is partitioned into subsystems.
 - each layer of an OO system contains one or more subsystems and represents a different level of abstraction of the functionality required to accomplish system functions.
 - levels of abstraction are determined by the degree to which the processing associated with a subsystem is visible to an end-user.
 - e.g. four-layer architecture might include
 - (1) **presentation layer** (the subsystems associated with the user interface)
 - (2) **application layer** (the subsystems that perform the processing associated with the application)
 - (3) **data formatting layer** (the subsystems that prepare the data for processing)
 - (4) **database layer** (the subsystems associated with data management).
 - Each layer moves deeper into the system, representing increasingly more environment-specific processing.
- Design approach for layering:
 1. Establish layering criteria. That is, decide how subsystems will be grouped in a layered architecture.
 2. Determine the number of layers. Too many introduce unnecessary complexity; too few may harm functional independence.
 3. Name the layers and allocate subsystems (with their encapsulated classes) to a layer. Be certain that communication between subsystems (classes) on one layer and other subsystems (classes) at another layer follow the design philosophy for the architecture
 4. Design interfaces for each layer.
 5. Refine the subsystems to establish the class structure for each layer.
 6. Define the messaging model for communication between layers.
 7. Review the layer design to ensure that coupling between layers is minimized (a client/server protocol can help accomplish this).
 8. Iterate to refine the layered design.

2 Concurrency and Subsystem Allocation

- Dynamic aspect of the object-behavior model provides an indication of concurrency among classes (or subsystems)
 - If classes (or subsystems) are not active at the same time:
 - no need for concurrent processing
 - classes (or subsystems) can be implemented on the same processor hardware.
 - If classes (or subsystems) must act on events asynchronously and at the same time, they are viewed as concurrent.
- When subsystems are concurrent, two allocation options exist:
 1. allocate each subsystem to an independent processor
 2. allocate the subsystems to the same processor and provide concurrency support through operating system features.
- Concurrent tasks are defined by examining the state diagram for each object
 - If the flow of events and transitions indicates that only a single object is active at any one time a thread of control has been established.
 - The thread of control continues even when one object sends a message to another object, as long as the first object waits for a response.
 - If the first object continues processing after sending a message, the thread of control splits.
 - Tasks in an OO system are designed by isolating threads of control
 - e.g. while the *SafeHome* security system is monitoring its sensors, it can also be dialing the central monitoring station for verification of connection.
 - Since the objects involved in both of these behaviors are active at the same time, each represents a separate thread of control and each can be defined as a separate task.

- If the monitoring and dialing activities occur sequentially, a single task could be implemented.
- To determine which of the processor allocation options is appropriate, the designer must consider performance requirements, costs, and the overhead imposed by interprocessor communication.

3 The Task Management Component

- Strategy for the design of the objects that manage concurrent tasks:
 - The characteristics of the task are determined
 - by understanding how the task is initiated
 - Event-driven and clock-driven tasks are most common
 - Both activated by an interrupts
 - A coordinator task and associated objects are defined.
 - The coordinator and other tasks are integrated.
- Priority and criticality of the task must also be determined.
 - High-priority tasks must have immediate access to system resources.
 - High-criticality tasks must continue to operate even if resource availability is reduced or the system is operating in a degraded state.
- Once the characteristics of the task have been determined, object attributes and operations required to achieve coordination and communication with other tasks are defined.

The basic task template (for a task object) takes the form

Task name—the name of the object
Description—a narrative describing the purpose of the object
Priority—task priority (e.g., low, medium, high)
Services—a list of operations that are responsibilities of the object
Coordinates by—the manner in which object behavior is invoked
Communicates via—input and output data values relevant to the task

4 The User Interface Component

- The user interface represents a critically important subsystem for most modern applications.
- OO analysis model contains:
 - usage scenarios (called *use-cases*)
 - descriptions of the roles that users play (called *actors*) as they interact with the system.
 - These serve as input to the user interface design process.
- Once the actor and its usage scenario are defined, a command hierarchy is identified.
 1. The command hierarchy defines major system menu categories (the menu bar or tool palette) and all subfunctions that are available within the context of a major system menu category (the menu windows).
 2. The command hierarchy is refined iteratively until every use-case can be implemented by navigating the hierarchy of functions.
- Because a wide variety of user interface development environments already exist, the design of GUI elements is not necessary

5 The Data Management Component

- Data management encompasses two distinct areas of concern:
 - the management of data that are critical to the application itself
 - the creation of an infrastructure for storage and retrieval of objects.
- Data management is designed in a layered fashion
 - isolate the low-level requirements for manipulating data structures from the higher-level requirements for handling system attributes.
- A Database management system is often used as a common data store for all subsystems
 - The objects required to manipulate the database are:
 - (1) members of reusable classes that are identified using domain analysis
 - (2) supplied directly by the database vendor

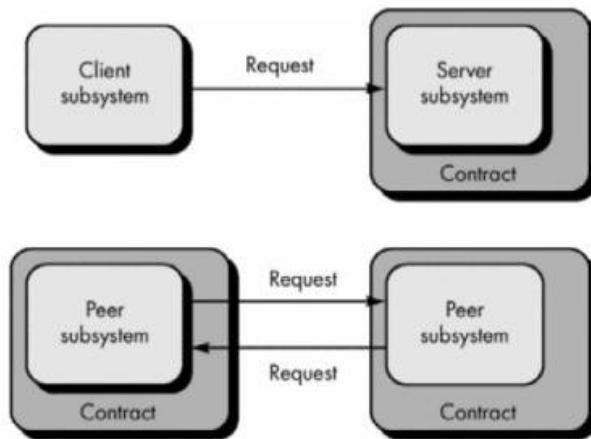
- The design of the data management component includes the design of the attributes and operations required to manage objects.

6 The Resource Management Component

- Different resources are available to an OO system or product
- Subsystems compete for these resources at the same time.
- Global system resources can be external entities (e.g., a disk drive, processor, or communication line) or abstractions (e.g., a database, an object).
- Regardless of the nature of the resource - design a control mechanism for it.

7 Intersubsystem Communication

- Once each subsystem has been specified define the collaborations that exist between the subsystems
- The model that we use for object-to-object collaboration can be extended to subsystems as a whole.
- A collaboration model:



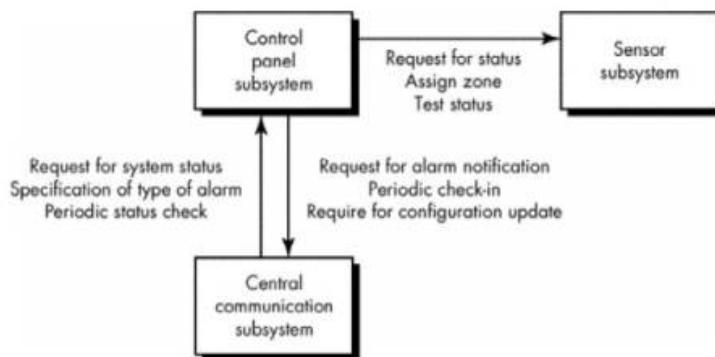
- Communication can occur by establishing a client/server link or a peer-to-peer link.
- Must specify the contract that exists between subsystems.
- A contract provides an indication of the ways in which one subsystem can interact with another.
- The following design steps can be applied to specify a contract for a subsystem
 - List each request that can be made by collaborators of the subsystem.**
 - Organize the requests by subsystem and define them within one or more appropriate contracts. Be sure to note contracts that are inherited from superclasses.
 - For each contract, note the operations (both inherited and private) that are required to implement the responsibilities implied by the contract.**
 - Associate the operations with specific classes that reside within a subsystem.
 - Considering one contract at a time, create a table of the form shown in below.**

Contract	Type	Collaborators	Class(es)	Operation(s)	Message Format

- For each contract, the following entries are made in the table:
 - Type*—the type of contract (i.e., client/server or peer-to-peer).
 - Collaborators*—the names of the subsystems that are parties to the contract.
 - Class*—the names of the classes (contained within a subsystem) that support services implied by the contract.
 - Operation*—the names of the operations (within the class) that implement the services.
 - Message format*—the message format required to implement the interaction between collaborators.
- Draft an appropriate message description for each interaction between the subsystems.

4. If the modes of interaction between subsystems are complex, a subsystem-collaboration diagram is created.

- Each subsystem is represented along with its interactions with other subsystems.
 - contracts that are invoked during an interaction are noted as shown.
 - details of the interaction are determined by looking up the contract in the subsystem collaboration table



The Object Design Process

- Object design is concerned with the detailed design of the objects and their interactions.
- It is completed within the overall architecture defined during system design and according to agreed design guidelines and protocols.
- Object design is particularly concerned with the specification of attribute types, how operations function, and how objects are linked to other objects.

Object Descriptions

- A design description of an object (an instance of a class or subclass) can take one of two forms:
 - a *protocol description* that establishes the interface of an object by defining each message that the object can receive and the related operation that the object performs when it receives the message

2. an *implementation description* that shows implementation details for each operation implied by a message that is passed to an object. Implementation details include information about the object's private part; that is, internal details about the data structures that describe the object's attributes and procedural details that describe operations.
- **Protocol description** is a set of messages and a corresponding comment for each message.
 - e.g. a portion of the protocol description for the object **motion sensor** to read the sensor
MESSAGE (*motion.sensor*) --> **read:** **RETURNS** *sensor.ID*, *sensor.status*;
 - **Implementation description** of an object provides the internal ("hidden") details that are required for implementation but are not necessary for invocation.
 - Composed of the following information:
 - (1) a specification of the object's name and reference to class
 - (2) a specification of private data structure with indication of data items and types
 - (3) a procedural description of each operation or pointers to such procedural descriptions.
 - Must contain sufficient information to provide for proper handling of all messages described in the protocol description.

Designing Algorithms and Data Structures

- Representations contained in the analysis model and the system design provide a specification for all operations and attributes.
- Algorithms and data structures are designed using an approach that differs little from the data design and component-level design approaches discussed for conventional software engineering.
- An algorithm is created to implement the specification for each operation.
- Data structures are designed concurrently with algorithms.
- Operations manipulate the attributes of a class
 - Three broad categories:
 1. operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting)
 2. operations that perform a computation
 3. operations that monitor an object for the occurrence of a controlling event.
 - e.g., *SafeHome* processing narrative contains the sentence fragments:
 "sensor is assigned a number and type"
 "a master password is programmed for arming and disarming the system."
- These two phrases indicate:
 - That an *assign* operation is relevant for the **sensor** object.
 - That a *program* operation will be applied to the **system** object.
 - That *arm* and *disarm* are operations that apply to **system** (also that **system status** may ultimately be defined (using data dictionary notation) as
system status = [armed | disarmed]
- The operation *program* is allocated during OOA, but during object design it will be refined into a number of more specific operations required to configure the system.
 - e.g., after discussions with product engineering, the analyst, and the marketing department, the designer might elaborate the original processing narrative and write the following for *program* (potential operations—verbs—are underlined):
Program enables the *SafeHome* user to configure the system once it has been installed. The user can (1) install phone numbers; (2) define delay times for alarms; (3) build a sensor table that contains each sensor ID, its type, and location; and (4) load a master password.
 - The single operation *program* and replaced it with the operations: *install*, *define*, *build*, and *load*.
 - Each of these new operations:
 - becomes part of the **system** object

- has knowledge of the internal data structures that implement the object's attributes
 - is invoked by sending the object messages of the form
MESSAGE (system) --> install: SENDS telephone.number;
 - an emergency phone number is given by an *install* message will be sent to system.
- Verbs connote actions or occurrences
 - For object design consider verbs and descriptive verb phrases and predicates (e.g., "is equal to") as potential operations.
 - Grammatical parse is applied recursively until each operation has been refined to its most-detailed level.
- After basic object model is created optimization occurs
 - Three major thrusts for OOD design optimization:
 - Review the object-relationship model to ensure that the implemented design leads to efficient utilization of resources and ease of implementation. Add redundancy where necessary.
 - Revise attribute data structures and corresponding operation algorithms to enhance efficient processing.
 - Create new attributes to save derived information, thereby avoiding recomputation.

Program Components and Interfaces

- An important aspect of software design quality is modularity;
- Object-oriented approach defines the object as a program component that is linked to other components (e.g., private data, operations).
- During design identify the interfaces between objects and the overall structure of the objects.
- A program component is a design abstraction
 - should be represented in the context of the programming language used for implementation
 - For OOD, the programming language should be capable of creating the following program component (modeled after Ada):

```

PACKAGE program-component-name IS
  TYPE specification of data objects
  :
  PROC specification of related operations . . .
PRIVATE
  data structure details for objects
PACKAGE BODY program-component-name IS
  PROC operation.1 (interface description) IS
  :
  END
  PROC operation.n (interface description) IS
  :
END
END program-component-name
  
```

- Program component is specified by indicating both data objects and operations
 - *specification part* of the component indicates all data objects (declared with the **TYPE** statement) and the operations (**PROC** for procedure) that act on them
 - *private part* (**PRIVATE**) of the component gives hidden details of data structure and processing.
- First program component to be identified should be the highest-level module from which all processing originates and all data structures evolve.
 - e.g. *SafeHome*, we can define the highest-level program component as
PROCEDURE SafeHome software
 - *SafeHome* software component can be coupled with a preliminary design for the following packages (objects):

```

PACKAGE system IS
    TYPE system data
    PROC install, define, build, load
    PROC display, reset, query, modify, call
PRIVATE
    PACKAGE BODY system IS
        PRIVATE
            system.id IS STRING LENGTH (8);
            verification.phone.number, telephone.number, ...
            IS STRING LENGTH (8);
            sensor.table DEFINED
                sensor.type IS STRING LENGTH (2),
                sensor.number, alarm.threshold IS NUMERIC;
        PROC install RECEIVES (telephone.number)
            {design detail for operation install}
        :
    END system
PACKAGE sensor IS
    TYPE sensor data
    PROC read, set, test
PRIVATE
    PACKAGE BODY sensor IS
        PRIVATE
            sensor.id IS STRING LENGTH (8);
            sensor.status IS STRING LENGTH (8);
            alarm.characteristics DEFINED
                threshold, signal.type, signal.level IS NUMERIC,
            hardware.interface DEFINED
                type, a/d.characteristics, timing.data IS NUMERIC,
    END sensor
:
END SafeHome software

```

- Data objects and corresponding operations are specified for each of the program components for *SafeHome* software
- Final step in design process: complete all information required to fully implement data structure and types contained in the **PRIVATE** portion of the package and all procedural detail contained in the **PACKAGE BODY**.
 - e.g. **sensor** package
 - data structures for **sensor** attributes have already been defined
 - first step is to define the interfaces for each of the operations attached to **sensor**:

```

PROC read (sensor.id, sensor.status: OUT);
PROC set (alarm.characteristics, hardware.interface: IN)
PROC test (sensor.id, sensor.status, alarm.characteristics: OUT);

```
 - next step -> stepwise refinement of each operation associated with the **sensor** package
 - e.g. develop a processing narrative (an informal strategy) for *read*:

When the sensor object receives a *read* message, the *read* process is invoked. The process determines the interface and signal type, polls the sensor interface, converts A/D characteristics into an internal signal level, and compares the internal signal level to a threshold value. If the threshold is exceeded, the sensor status is set to "event." Otherwise, the sensor status is set to "no event." If an error is sensed while polling the sensor, the sensor status is set to "error."
 - description of the *read* process can be developed:

```

PROC read (sensor.id, sensor.status: OUT);
    raw.signal IS BIT STRING
    IF (hardware.interface.type = "s" & alarm.characteristics.signal.type = "B"
    THEN
        GET (sensor, exception: sensor.status := error) raw.signal;
        CONVERT raw.signal TO internal.signal.level;
        IF internal.signal.level > threshold
            THEN sensor.status := "event";
            ELSE sensor.status := "no event";
        ENDIF
    ELSE {processing for other types of s interfaces would be specified}
    ENDIF
    RETURN sensor.id, sensor.status;
END read

```

- translate into the appropriate implementation language

Design Patterns

- Recurring patterns of classes and communicating objects exist in many object-oriented systems
- These patterns solve specific design problems and make object-oriented design more flexible, elegant, and ultimately reusable
- Help designers reuse successful designs by basing new designs on prior experience
- A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

Describing a Design Pattern

- All design patterns can be described by specifying the following information
 - the name of the pattern
 - the intent of the pattern
 - the "design forces" that motivate the pattern
 - the solution that mitigates these forces
 - the classes that are required to implement the solution
 - the responsibilities and collaboration among solution classes
 - guidance that leads to effective implementation
 - example source code or source code templates
 - cross-references to related design patterns
- Design pattern name conveys significant meaning once the applicability and intent are understood
- *Design forces:*
 - describe the data, functional, or behavioral requirements associated with part of the software for which the pattern is to be applied
 - define the constraints that may restrict the manner in which the design is to be derived
- Pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems
- Guidance associated with the use of a design pattern provides an indication of the ramifications of design decisions.

Using Patterns in Design

- Design patterns can be used by applying two different mechanisms:
 - Inheritance

An existing design pattern becomes a template for a new subclass.
 - Composition

Composition leads to aggregate objects

 - A complex object can be assembled by selecting a set of design patterns and composing the appropriate object (or subsystem).

- Each design pattern is treated as a black box, and communication among the patterns occurs only via well-defined interfaces.
- Object composition should be favored over inheritance when both options exist
 - Rather than creating large and unmanageable class hierarchies, composition favors small class hierarchies and objects that remain focused on one objective.

Object-Oriented Programming

- Software engineering viewpoint stresses OOA and OOD and considers OOP (coding) an important secondary activity that is an outgrowth of analysis and design
- As the complexity of systems increases, the design architecture of the end product has a significantly stronger influence on its success than the programming language that has been used

Object-Oriented Testing

- Construction of object-oriented software begins with the creation of analysis and design models
- These models begin as relatively informal representations of system requirements and evolve into detailed models of classes, class connections and relationships, system design and allocation, and object design (incorporating a model of object connectivity via messaging)
- Models can be tested at each stage to uncover errors prior to their propagation to the next iteration.

Testing OOA and OOD Models

Correctness of OOA and OOD Models

- Notation and syntax used to represent analysis and design models tied to the specific analysis and design method chosen for the project.
- Syntactic correctness is judged on proper use of the symbology
- Each model is reviewed to ensure that proper modeling conventions have been maintained.
- During analysis and design, semantic correctness must be judged based on the model's conformance to the real world problem domain.
- If the model accurately reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed), then it is semantically correct.
- To determine whether the model does reflect the real world, it should be presented to problem domain experts, who will examine the class definitions and hierarchy for omissions and ambiguity

Consistency of OOA and OOD Models

- Consistency of OOA and OOD models judged by "considering the relationships among entities in the model"
- To assess consistency, each class and its connections to other classes should be examined.
- One Approach: Use CRC and object-relationship model

To evaluate the class model use the following steps:

1. **Revisit the CRC model and the object-relationship model.** Cross check to ensure that all collaborations implied by the OOA model are properly represented.
2. **Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.**
3. **Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.**
4. **Using the inverted connections determine whether other classes might be required and whether responsibilities are properly grouped among the classes.**

- 5. Determine whether widely requested responsibilities might be combined into a single responsibility.
- 6. Steps 1 through 5 are applied iteratively to each class and through each evolution of the OOA model.
- Reviews of the system design and the object design should also be conducted once the OOD model is created.
 - The system design depicts:
 - the overall product architecture
 - the subsystems that compose the product
 - the manner in which subsystems are allocated to processors
 - the allocation of classes to subsystems
 - the design of the user interface.
 - The object model presents:
 - the details of each class
 - the messaging activities that are necessary to implement collaborations between classes
- The system design is reviewed by examining the object-behavior model developed during OOA and mapping required system behavior against the subsystems designed to accomplish this behavior.
- Concurrency and task allocation are reviewed within the context of system behavior.
 - The behavioral states of the system are evaluated to determine which exist concurrently.
- Use-case scenarios are used to exercise the user interface design.
- The object model is tested against the object-relationship network to ensure that all design objects contain the necessary attributes and operations to implement the collaborations defined for each CRC index card.
- The detailed specification of operation details (i.e., the algorithms that implement the operations) are reviewed using conventional inspection techniques.

Object-Oriented Testing Strategies

- Classical strategy for testing computer software: begin with "testing in the small" and work outward toward "testing in the large."

Unit Testing in the OO Context

- Encapsulation drives the definition of classes and objects
 - Each class and each instance of a class (object) packages attributes (data) and the operations (also known as *methods* or *services*) that manipulate these data.
- The smallest testable unit is the encapsulated class or object
- Class testing for OO software is the equivalent of unit testing for conventional software
- Unlike unit testing of conventional software class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class

Integration Testing in the OO Context

Two different strategies for integration testing of OO systems:

1. *thread-based testing* - integrates the set of classes required to respond to one input or event for the system
 - Each *thread* is integrated and tested individually.
 - Regression testing is applied to ensure that no side effects occur.
2. *use-based testing* - tests those classes (*independent classes*) that use very few of server classes.
 - After testing independent classes the next layer of classes, *dependent classes*, that use the independent classes are tested.
 - Sequence of testing layers of dependent classes continues until the entire system is constructed

Cluster testing is one step in the integration testing of OO software

- A cluster of collaborating classes
- is exercised by designing test cases that attempt to uncover errors in the collaborations.

Validation Testing in an OO Context

- Validation of OO software focuses on user-visible actions and user-recognizable output from the system
- The tester should draw upon the use-cases that are part of the analysis model
- Use-case provides a scenario that has a high likelihood of uncovered errors in user interaction requirements.
- Conventional black-box testing methods can be used to drive validations tests
- Test cases may be derived from the object-behavior model and from event flow diagram created as part of OOA

Test Case Design for OO Software

- Overall approach to OO test case design:
 1. Each test case should be uniquely identified and explicitly associated with the class to be tested.
 2. The purpose of the test should be stated.
 3. A list of testing steps should be developed for each test and should contain [BER93]:
 - a. A list of specified states for the object that is to be tested.
 - b. A list of messages and operations that will be exercised as a consequence of the test.
 - c. A list of exceptions that may occur as the object is tested.
 - d. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test).
 - e. Supplementary information that will aid in understanding or implementing the test.
- Object-oriented testing focuses on designing appropriate sequences of operations to exercise the states of a class.

The Test Case Design Implications of OO Concepts

- OO class is the target for test case design
- Encapsulation makes testing operations outside of the class unproductive
 - Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.
- Inheritance leads to challenges
 - Each new context of usage requires retesting, even though reuse has been achieved.
 - Multiple inheritance complicates testing by increasing the number of contexts for which testing is required
 - If subclasses instantiated from a superclass are used within the same problem domain, the set of test cases derived for the superclass can be used when testing the subclass.
 - If the subclass is used in an entirely different context, the superclass test cases will have little applicability and a new set of tests must be designed.

Applicability of Conventional Test Case Design Methods

- White-box testing methods can be applied to the operations defined for a class
 - Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested.
- Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods
 - Use-cases can provide useful input in the design of black-box and state-based tests.

Fault-Based Testing

- The object of *fault-based testing* is to design tests that have a high likelihood of uncovering plausible faults
- Test cases are designed to exercise the design or code
- Errors occur at the boundaries of a problem.
 - e.g., testing a SQRT operation that returns errors for negative numbers
 - try the boundaries: a negative number close to zero and zero itself.
 - "Zero itself" checks whether the programmer made a mistake like
`if (x > 0) calculate_the_square_root();`
 instead of the correct
`if (x >= 0) calculate_the_square_root();`
 - e.g., a Boolean expression:
`if (a && !b || c)`
 - Multicondition testing and related techniques probe for certain plausible faults in this expression, such as
 - **&&** should be **||**
 ! was left out where it was needed
 There should be parentheses around **!b || c**
 - For each plausible fault design test cases that will force the incorrect expression to fail.
 - **(a=0, b=0, c=0)** will make the expression as given evaluate *false*.
 - If the **&&** should have been **||**, the code has done the wrong thing and might branch to the wrong path.
- Integration testing looks for plausible faults in operation calls or message connections
- Three types of faults are encountered:
 - unexpected result
 - wrong operation/message used
 - incorrect invocation
- To determine plausible faults as functions (operations) are invoked, the behavior of the operation must be examined.
- Integration testing applies to attributes too.
- Object "behaviors" are defined by the values that its attributes are assigned.
- Testing should exercise the attributes to determine whether proper values occur for distinct types of object behavior.

The Impact of OO Programming on Testing

- Object-oriented programming's impact on testing
 - Some types of faults become less plausible (not worth testing for).
 - Some types of faults become more plausible (worth testing now).
 - Some new types of faults appear.
- When an operation is invoked, it may be hard to tell exactly what code gets exercised.
 - the operation may belong to one of many classes.
- Hard to determine the exact type or class of a parameter
 - e.g. function call:
`x = func (y);`
 - Must consider the behaviors of **base::func()**, of **derived::func()**, etc.
 - Each time **func** is invoked, the tester must consider the union of all distinct behaviors.

Scenario-Based Test Design

- Fault-based testing misses two main types of errors:
 1. incorrect specifications

2. interactions among subsystems.
- *Scenario-based testing* concentrates on what the user does, not what the product does.
 - It captures the tasks (via use-cases) that the user has to perform, then applies them and their variants as tests.
 - Scenarios uncover interaction errors
 - Scenario-based testing exercises multiple subsystems in a single test.
 - e.g. the design of scenario-based tests for a text editor.
- Use-Case: Fix the Final Draft**
- Background:** It's not unusual to print the "final" draft, read it, and discover some annoying errors that weren't obvious from the on-screen image. This use-case describes the sequence of events that occurs when this happens.
1. Print the entire document.
 2. Move around in the document, changing certain pages.
 3. As each page is changed, it's printed.
 4. Sometimes a series of pages is printed.
- Scenario describes two things: a test and specific user needs.
 - The user needs:
 - (1) a method for printing single pages
 - (2) a method for printing a range of pages.
 - There is a need to test editing after printing (as well as the reverse).
 - Tester hopes to discover that the *printing* function causes errors in the *editing* function.

Use-Case: Print a New Copy

- Background:** Someone asks the user for a fresh copy of the document. It must be printed.
1. Open the document.
 2. Print it.
 3. Close the document.
- Document was created in an earlier task. Does that task affect this one?
 - Modern editors - documents remember how they were last printed.
 - After the *Fix the Final Draft* scenario, just selecting "Print" in the menu and clicking the "Print" button in the dialog box will cause the last corrected page to print again.
 - According to the editor, the correct scenario should look like this:

Use-Case: Print a New Copy

1. Open the document.
 2. Select "Print" in the menu.
 3. Check if you're printing a page range; if so, click to print the entire document.
 4. Click on the Print button.
 5. Close the document.
- This scenario indicates a potential specification error
 - The editor does not do what the user reasonably expects it to do.
 - Customers will often overlook the check noted in step 3.
 - They will then be annoyed when they trot off to the printer and find one page when they wanted 100. Annoyed customers signal specification bugs.

Testing Surface Structure and Deep Structure

- *Surface structure* refers to the externally observable structure of an OO program.
 - Capturing tasks involves understanding, watching, and talking with users.
- *Deep structure* refers to the internal technical details of an OO program.
 - Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the system and object design of OO software.
 - Analysis and design models are used as the basis for deep structure testing.
 - Design representations of class hierarchy provide insight into inheritance structure.

Testing Methods Applicable at the Class Level

- Testing in the small focuses on a single class and the methods that are encapsulated by the class
- Random testing and partitioning are methods that can be used to exercise a class during OO testing

Random Testing for OO Classes

- e.g. consider a banking application in which an **account** class has the following operations: *open*, *setup*, *deposit*, *withdraw*, *balance*, *summarize*, *creditLimit*, and *close*.
 - Each operation may be applied for **account**
 - Certain constraints (e.g., the account must be opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem.
 - Even with these constraints, there are many permutations of the operations.
- Minimum behavioral life history of an instance of **account** includes the following operations:
open•setup•deposit•withdraw•close
 - Represents the minimum test sequence for account.
 - Other behaviors may occur within this sequence:
open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]ⁿ•withdraw•close
- A variety of different operation sequences can be generated randomly.
- For example:
Test case r₁: **open•setup•deposit•deposit•balance•summarize•withdraw•close**
Test case r₂: **open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close**
- These and other random order tests are conducted to exercise different class instance life histories.

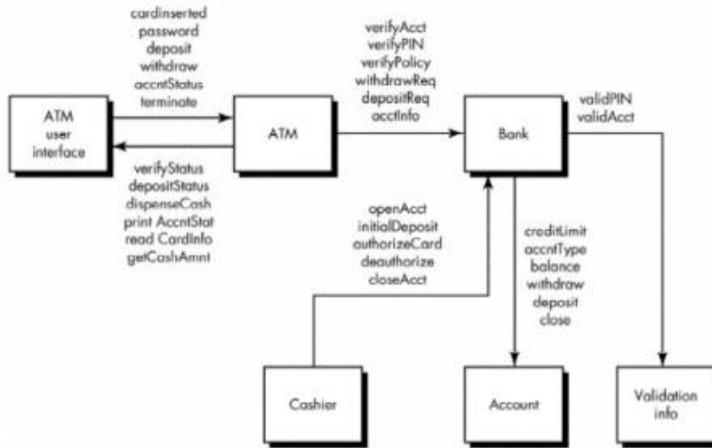
Partition Testing at the Class Level

- *Partition testing* reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning.
- *Partition testing* breaks down the input domain of a program into classes of data from which test cases can be derived.
- Input and output are categorized and test cases are designed to exercise each category.
- *State-based partitioning* categorizes class operations based on their ability to change the state of the class.
 - e.g. consider the **account** class:
 - state operations include *deposit* and *withdraw*
 - nonstate operations include *balance*, *summarize*, and *creditLimit*
 - Tests are designed in a way that exercises operations that change state and those that do not change state separately.
 - **Test case p₁:** **open•setup•deposit•deposit•withdraw•withdraw•close**
 - **Test case p₂:** **open•setup•deposit•summarize•creditLimit•withdraw•close**
 - Test case *p₁* changes state
 - Test case *p₂* exercises operations that do not change state
- *Attribute-based partitioning* categorizes class operations based on the attributes that they use.
 - For the **account** class:
 - attributes **balance** and **creditLimit** can be used to define partitions.
 - Operations are divided into three partitions:
 1. operations that use **creditLimit**
 2. operations that modify **creditLimit**
 3. operations that do not use or modify **creditLimit**.
 - Test sequences are then designed for each partition.
- *Category-based partitioning* categorizes class operations based on the generic function that each performs.
- For example, operations in the **account** class can be categorized in:

- initialization operations (*open*, *setup*)
- computational operations (*deposit*, *withdraw*)
- queries (*balance*, *summarize*, *creditLimit*)
- termination operations (*close*)

Interclass Test Case Design

- Test case design becomes more complicated as integration of the OO system begins.
- Testing of collaborations between classes must begin here.
- e.g. "interclass test case generation" - banking example includes classes and collaborations.



Class collaboration diagram for banking application

- Arrow direction in above indicates the direction of messages
- Labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.
- Class collaboration testing can be accomplished by applying:
 - random and partitioning methods
 - scenario-based testing
 - behavioral testing

Multiple Class Testing

- Sequence of steps to generate multiple class random test cases:
 1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
 2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
 3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
 4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.
- e.g. consider a sequence of operations for the **bank** class relative to an **ATM** class:

verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]]|depositReq|acctInfoREQⁿ

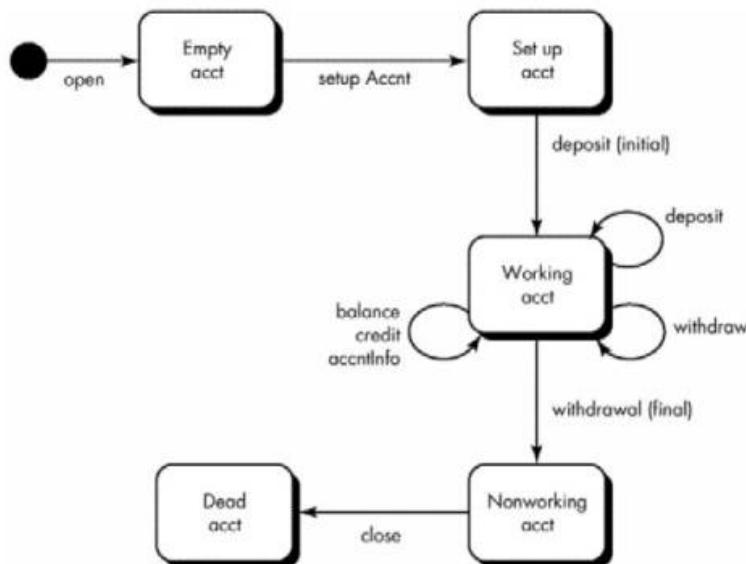
 - A random test case for the **bank** class might be
test case r₃ = verifyAcct•verifyPIN•depositReq
 - To test collaboration the messages associated with each of the operations noted in test case *r₃* are considered.
 - **Bank** must collaborate with **ValidationInfo** to execute the *verifyAcct* and *verifyPIN*.
 - **Bank** must collaborate with **account** to execute *depositReq*.

- Hence, a new test case that exercises these collaborations is

test case r_4 = **$\text{verifyAcctBank}[\text{validAcctValidationInfo}] \cdot \text{verifyPINBank}[\text{validPinValidationInfo}] \cdot \text{depositReq} \cdot [\text{depositAccount}]$**

- Multiple class partition testing is similar to partition testing of individual classes
 - A single class is partitioned
 - The test sequence is expanded to include those operations that are invoked via messages to collaborating classes.
- Alternative approach partitions tests based on the interfaces to a particular class.
 - The **bank** class receives messages from the **ATM** and **cashier** classes.
 - The methods within **bank** can therefore be tested by partitioning them into those that serve **ATM** and those that serve **cashier**.
 - State-based partitioning can be used to refine the partitions further.

Tests Derived from Behavior Models



- State transition diagrams represent the dynamic behavior of a class.
- STD for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it).
- Above STD for the **account** class
 - Initial transitions move through the *empty acct* and *setup acct* states.
 - Majority of all behavior for instances of the class occurs while in the *working acct* state.
 - A final withdrawal and close cause the account class to make transitions to the *nonworking acct* and *dead acct* states
- Tests should achieve all state coverage.

test case s_1 : **$\text{open} \cdot \text{setupAcct} \cdot \text{deposit} \text{ (initial)} \cdot \text{withdraw} \text{ (final)} \cdot \text{close}$**
- This sequence is the minimum test sequence. Adding additional test sequences to the minimum sequence

test case s_2 : **$\text{open} \cdot \text{setupAcct} \cdot \text{deposit} \text{ (initial)} \cdot \text{deposit} \cdot \text{balance} \cdot \text{credit} \cdot \text{withdraw} \text{ (final)} \cdot \text{close}$**

test case s_3 : **$\text{open} \cdot \text{setupAcct} \cdot \text{deposit} \text{ (initial)} \cdot \text{deposit} \cdot \text{withdraw} \cdot \text{acntInfo} \cdot \text{withdraw} \text{ (final)} \cdot \text{close}$**
- When class behavior results in a collaboration with one or more classes, multiple STDs are used to track the behavioral flow of the system.

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

OBJECT ORIENTED SOFTWARE ENGINEERING
Chapter Six
Object Oriented Testing

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Overview

Overview

What is it?

- ❖ The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span.
- ❖ Each elements of an Object oriented system (subsystems and classes) perform functions that help to achieve system requirements.
- ❖ So, It is necessary to test an OO system at a variety of different levels in an effort to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers.

Who does it?

- ❖ Object-oriented testing is performed by software engineers and testing specialists.

Overview

Why is it important?

- ❖ You have to execute the program before it gets to the customer with the specific intent of removing all errors, so that the customer will not experience the frustration associated with a poor-quality product.
- ❖ In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

What are the steps?

- ❖ Testing begins with the review different OO analysis and design models.
- ❖ Once code has been generated, OO testing begins “in the small” with class testing. A series of tests are designed that exercise class operations and examine whether errors exist as one class collaborates with other classes.
- ❖ As classes are integrated to form a subsystem, testing like thread-based, use-based, and cluster testing, along with fault-based approaches, are applied to fully exercise collaborating classes.
- ❖ Finally, use-cases (developed as part of the OO analysis model) are used to uncover errors at the software validation level.

Overview

What is the work product?

- ❖ A set of test cases to exercise classes, their collaborations, and behaviors is designed and documented
- ❖ Expected results defined; and actual results recorded.

oo

*Testing
strategies*

OO testing strategies

Unit Testing

- ❖ When object-oriented software is considered, the concept of the unit changes.
- ❖ Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as methods or services) that manipulate these data.
- ❖ Rather than testing an individual module(as in classical approach), the smallest testable unit is the encapsulated class or object.
- ❖ Class testing for OO software is the equivalent of unit testing for conventional software.

Integration Testing

There are two different strategies for integration testing of OO systems:

- i. Thread-based testing
- ii. Use-based testing

OO testing strategies

Thread-based testing

- ✓ It integrates the set of classes required to respond to one input or event for the system.
- ✓ Each thread is integrated and tested individually.

Use-based testing

- ✓ It begins the construction of the system by testing independent classes.
- ✓ After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested.
- ✓ This sequence of testing layers of dependent classes continues until the entire system is constructed.

Validation/system testing

- ❖ At the validation or system level, the details of class connections disappear.
- ❖ Like conventional validation, the validation of OO software focuses on user-visible actions and user-recognizable output from the system.

Test case design

For

OO software

Test case design for OO software

An overall approach to OO test case design has been defined as:

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.
2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain
 - ✓ A list of specified states for the object that is to be tested.
 - ✓ A list of messages and operations that will be exercised as a consequence of the test.
 - ✓ A list of exceptions that may occur as the object is tested.
 - ✓ A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test).
 - ✓ Supplementary information that will aid in understanding or implementing the test.

Test case design for OO software

Fault-Based Testing

- ❖ The objective of fault-based testing within an OO system is to design tests that have a high likelihood of uncovering plausible faults.
- ❖ The tester looks for plausible faults and to determine whether these faults exist, test cases are designed to exercise the design or code.
- ❖ Consider a simple example

Software engineers often make errors at the boundaries of a problem. For example, when testing a SQRT operation that returns errors for negative numbers, we know to try the boundaries: *a negative number close to zero and zero itself*. "Zero itself" checks whether the programmer made a mistake like

```
if(x > 0)  
calculate_the_square_root();  
instead of the correct statement  
if(x >= 0)  
calculate_the_square_root();
```

Test case design for OO software

Scenario-Based Test Design

- ❖ Fault-based testing misses two main types of errors:
 1. incorrect specifications and
 2. interactions among subsystems.
- ❖ When errors associated with incorrect specification occur, the product doesn't do what the customer wants. It might do the wrong thing or it might omit important functionality.
- ❖ But in either circumstance, quality (conformance to requirements) suffers.
- ❖ Errors associated with subsystem interaction occur when the behavior of one subsystem creates circumstances (e.g., events, data flow) that cause another subsystem to fail.
- ❖ Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.
- ❖ test cases are more complex than fault-based tests.

Test case design for OO software

Scenario-Based Test Design (example),

Consider the design of scenario-based tests for a text editor.

Scenario 1

Use-Case:

Fix the Final Draft

Background:

The normal scenario is to print the "final" draft, read it, and discover some annoying errors that weren't obvious from the on-screen image. This use-case describes the sequence of events that occurs when this happens.

1. Print the entire document.
2. Move around in the document, changing certain pages (if any).
3. As each page is changed, it's printed.
4. Sometimes a series of pages is printed (if required).

According to above case, the user needs are:

- (1) *a method for printing single page*
- (2) *a method for printing a range of pages.*

Test case design for OO software

Scenario-Based Test Design (example),

- ❖ Consider the design of scenario-based tests for a text editor.

Scenario 2

Use-Case:

Print a New Copy

Background:

Someone asks the user for a fresh copy of the document. It must be printed.

1. *Open the document.*
2. *Print it.*
3. *Close the document.*

- ❖ In many modern editors, documents remember how they were last printed. By default, they print the same way next time.
- ❖ After the **Fix the Final Draft** scenario, just selecting "Print" in the menu and clicking the "Print" button in the dialog box will cause the last corrected page to print again

Test case design for OO software

Scenario-Based Test Design (example),

- ❖ For above case, the correct scenario should be like this:

Use-Case:

Print a New Copy

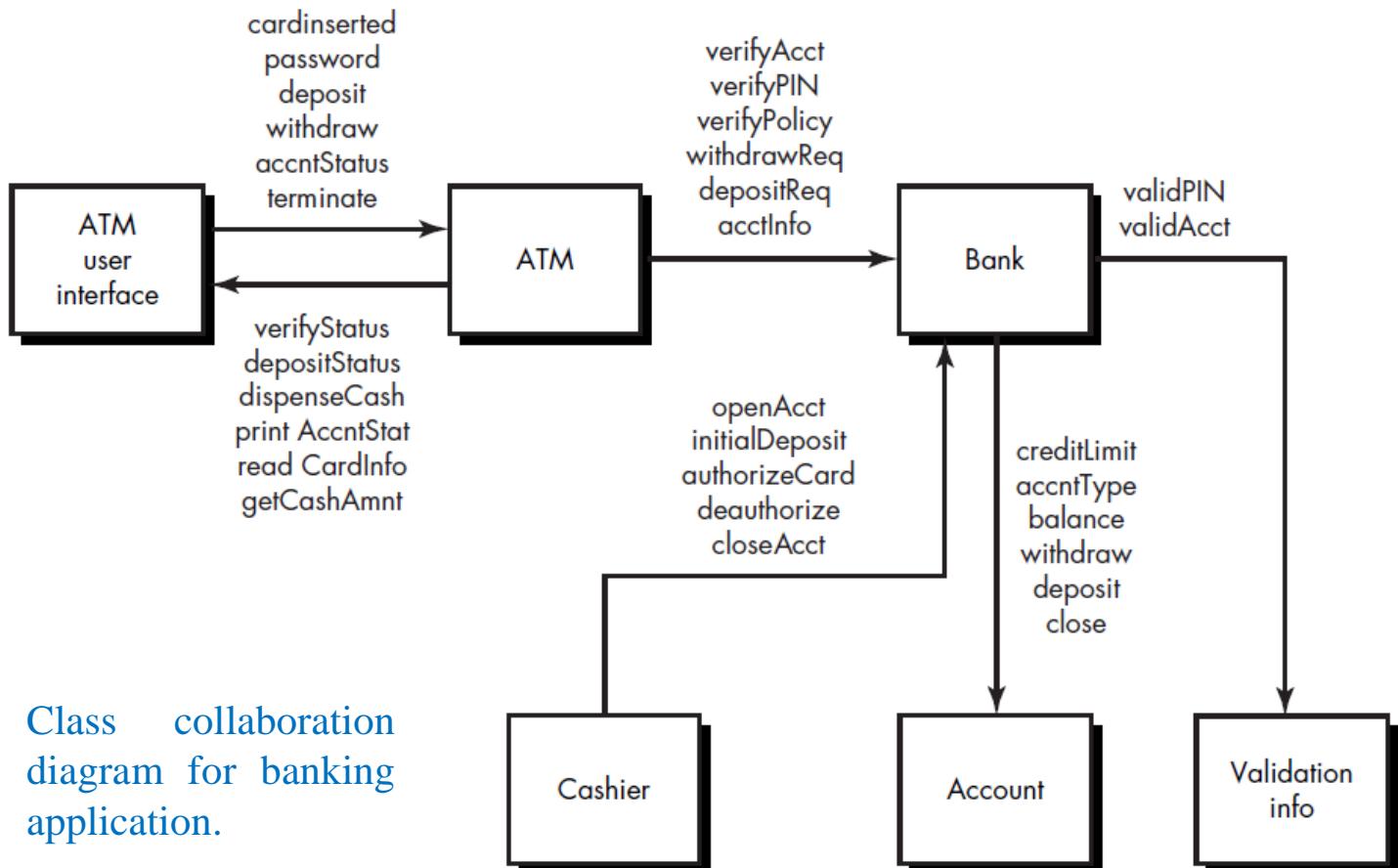
1. *Open the document.*
2. *Select "Print" in the menu.*
3. *"Check" if you're printing a page range; if so, click to print the range of page.*
4. *Click on the Print button.*
5. *Close the document.*

- ❖ Still, this scenario indicates a potential specification error. For example, if Customers overlook the check noted in step 3 then printer will print one page (last corrected page) instead of range of pages (suppose 100 pages).
- ❖ So the customer will be annoyed when he/she finds one page when he/she wanted was 100 pages. This indicates Annoyed customers signal specification bugs.

Interclass test case design

Inter class test case design

- ❖ Interclass testing is the testing of a set of classes composing a system or subsystem, usually performed during integration.
- ❖ Test case design becomes more complicated as integration of the OO system begins.



Inter class test case design

- ❖ To illustrate “interclass test case design”, we take example of banking system as shown in figure (in previous slide), which include the classes and collaborations between those classes.
- ❖ The direction of the arrows in the figure indicates the direction of messages and the labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.

Approaches for inter class testing

- i. **Multiple Class Testing**
- ii. **Tests Derived from Behavior Models**

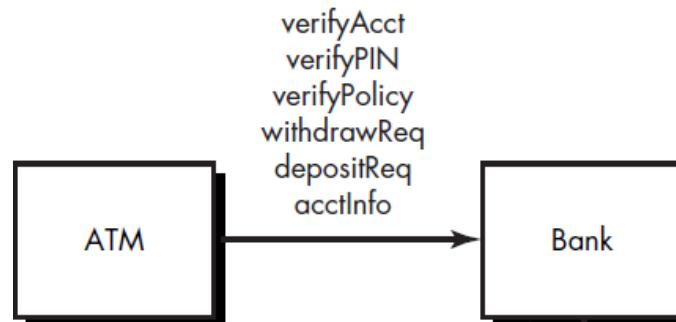
Inter class test case design

Multiple Class Testing

Steps to generate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

Consider a sequence of operations for the **bank** class relative to an **ATM** class as shown in figure.



Inter class test case design

Multiple Class Testing

- ❖ General sequence of operations for the **bank** class relative to an ATM class:

verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]ⁿ

- ❖ So, A random test case for the **bank** class might be

test case r_3 : verifyAcct•verifyPIN•depositReq

- ❖ In order to consider the collaborators involved in this *test case (r_3)*, the messages associated with each of the operations noted in test case r_3 is considered. In *test case r_3* ,

- **Bank** must collaborate with *ValidationInfo* to execute the verifyAcct and verifyPIN.
- **Bank** must collaborate with **account** to execute depositReq.

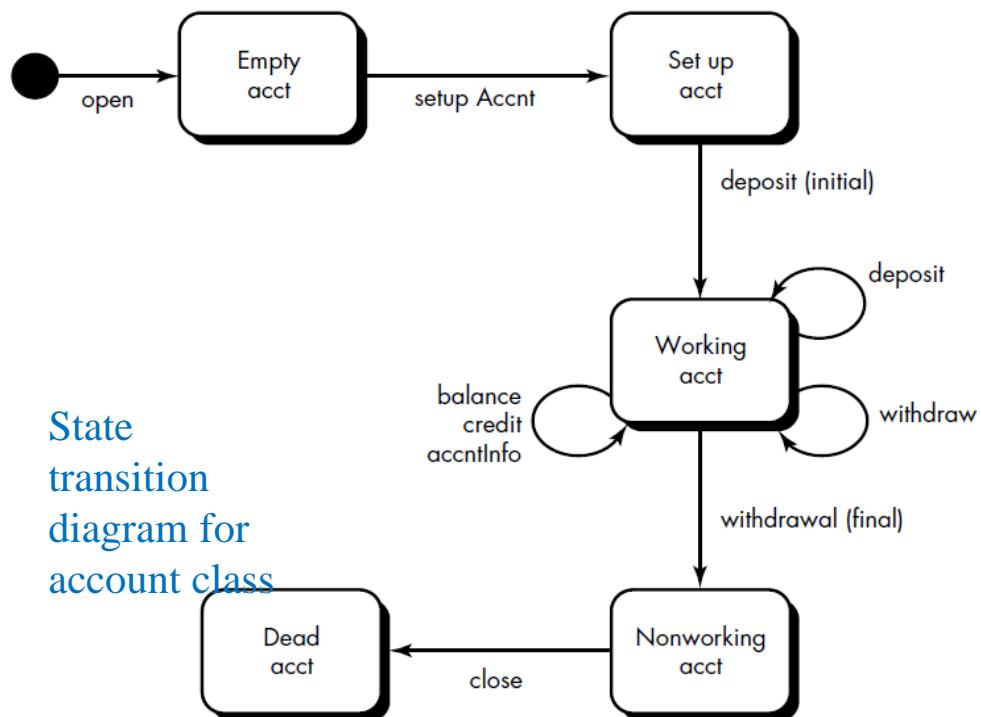
- ❖ Hence, a new test case (suppose r_4) that exercises these collaborations is

r_4 : verifyAcct_{Bank}[validAcct_{ValidationInfo}]•verifyPIN_{Bank}• [validPin_{ValidationInfo}]•depositReq•[deposit_{account}]

Inter class test case design

Tests Derived from Behavior Models

- ❖ As we discussed the use of the state transition diagram is a model that represents the dynamic behavior of a class.
- ❖ The STD for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it).
- ❖ Figure below illustrates an STD for the **account** class.



Inter class test case design

Tests Derived from Behavior Models

- ❖ Referring to the figure, initial transitions move through the `empty acct` and `setup acct` states. the majority of all behavior for instances of the class occurs while in the working acct state.
- ❖ A final withdrawal and close cause the account class to make transitions to the nonworking acct and dead acct states, respectively.
- ❖ So the tests to be designed should achieve all states i.e., the operation sequences should cause the account class to make transition through all allowable states. So the minimum test sequence should be:

test case s1: `open•setupAcnt•deposit (initial)•withdraw (final)•close`

- ❖ Additional test sequences can be:

test case s2: `open•setupAcnt•deposit(initial)•deposit•balance•credit•withdraw (final)•close`

test case s3: `open•setupAcnt•deposit(initial)•deposit•withdraw•acctInfo•withdraw(final)•close`

- ❖ Still more test cases could be derived to ensure that all behaviors for the class have been adequately exercised. In situations in which the class behavior results in a collaboration with one or more classes, multiple STDs are used to track the behavioral flow of the system.

Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

OBJECT ORIENTED SOFTWARE ENGINEERING
Chapter Seven

Managing Object Oriented Software engineering

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Overview

- ✓ *Project selection and preparation*
- ✓ *Project development, organization and management*
- ✓ *Software project planning, scheduling and techniques*
- ✓ *COCOMO model*
- ✓ *Risk Management process*
- ✓ *Software quality assurance*
- ✓ *software metrics*

Project selection and preparation

1. Selecting the first project

- ❖ When selecting the first project there are certain things to consider. Generally, it should have the optimal conditions possible, so that the method can be evaluated without any disturbances such as unnecessary shortage of staff or problems in defining the system responsibility.
- ❖ We summarize this in the following recommendations:
 - (1) Select a real project that is important, but not with a tight time schedule or any other hard constraints.
 - (2) Select a problem domain that is well known and well defined.
 - (3) Select people experienced in system development who have a positive view of changes. The management should .have confidence in them.
 - (4) Select a project manager with a high degree of interest in the task.
 - (5) The staff should work full time within the project and not be distracted by other projects.
 - (6) Base your work on a detailed plan developed in advance. Perform evaluation at all stages with criteria established in advance.

2. Education and Training

- ❖ All personnel involved in the new order of work need education and training. Given a strict method and process definition, more emphasis can be put on formal education and training.
- ❖ Therefore more material can be taught, and staff will need fewer projects on which to learn how to work in an orderly way. In a less formalized method, staff need several learning projects before getting familiar and highly productive with the method.
- ❖ The scope and amount of education needed varies according to each person's role in the project. Everyone should have a basic education, so that there is a common basis of concepts and way of working.
- ❖ More specialized education and training is needed for individual roles within the project according to their roles.
- ❖ We have summarized our experiences of the need for education on the table (next page).

2. Education and Training

issue \ role	Project manager	Analyst	Constructor	Tester	QA	Upper management
Concepts	x	x	x	+	x	+
Project management	x					
Overview	x	+	+	+	x	+
Analysis	+	x	x	+	+	
Construction	+	+	x	+	+	
Testing	+		+	x	+	

Table: The need for education when introducing a new development process:

x =necessary + = preferably

Concepts

- ✓ Basic concepts of object orientation and the fundamental concepts of the method.
- ✓ Duration: 1-2 days

Project management

- ✓ Specific characteristics for the new method. Appropriate metrics for the management within the new method.
- ✓ Duration: 1 day

2. Education and Training

Overview

- ✓ An overview of the method using examples to introduce the entire method in the system life cycle and the underlying ideas. This entire book may form a basis for this course.
- ✓ Attendees should be familiar with object orientation.
- ✓ Duration: 3- 4 days.

Analysis

- ✓ A thorough and detailed study of the entire analysis process.
- ✓ Emphasis should be on applying the method and process to a larger example, possibly on the current system.
- ✓ Duration: 3- 6 days.

Construction

- ✓ A thorough and detailed study of the entire design and implementation process. Emphasis should be on applying the method and process to a larger example, preferably the system to be built.
- ✓ Attending should be familiar with the programming language used.
- ✓ Duration: 3-6

Testing

- ✓ A thorough and detailed study of the testing activities and principles.
- ✓ Duration: 2-3 days.

3. Risk Analysis

- ❖ Introducing new technologies always brings potential risks. The number of new technologies introduced simultaneously increases risk exponentially.
- ❖ We will here discuss a simple technique for detecting and managing risks in OOSE. The method is divided into three steps.
 - (1) Risk identification
 - (2) Risk valuation
 - (3) Managing the risks

In **risk identification**, we define the potential and foreseeable risks of the project that, if they occur, may seriously injure the project. The starting point should be the goal of the project. What risks may occur that will make us fail to reach the goal? These risk areas could involve OOSE-specific risks, but also other risks. There are many risks that must be identified for each project. Few examples of potential risk areas involved in OOSE projects are:

(1) Paradigm shift

- Are we mature enough to adopt a new development strategy?
- Will the team be mature enough not to start coding too early?
- Is the project manager familiar with the new paradigm?
- Are the team members familiar with the new paradigm?
- Are we familiar with the programming language to be used?

3.1. Risk Identification

(2) Process

- Is our process well defined and well documented, and can we work through it?
- Is the process mature?
- Does the documentation produced fit our purpose?
- Do we have sufficient training capabilities?

(3) Tools

- Are we familiar with the tools to be used, and with the learning threshold?
- Are the tools to be used mature and stable?
- Will we have the tools when they are needed?
- Are the tools compatible? Are they integrated?

(4) The system

- Are we familiar with the application domain?
- Are the requirements clear, consistent and stable?
- Can or must we integrate existing systems? Is it possible?

(5) Organization

- Do we have a tight schedule?
- Will time-to-market be critical? Will the pressure from the market change?
- Does the organization have realistic expectations for the project?

3.2. Risk Valuation

- ❖ The next step is risk valuation. Now the potential risks should be evaluated to assess the consequences (disturbance or damage) if they occur.
- ❖ We then initially judge the probability that the risk will occur. A scale from 1 (very improbable) to 5 (very probable) could be used.
- ❖ Then we assess the consequences of each risk. A scale from 1 (negligible) to 5 (catastrophic) could be used.
- ❖ After that, we multiply the probability and the consequence factor for each risk (see Table). So, we will now have a relative measure of the size and potential threats of the risks.

Risk	Probability to occur (P)	Consequence (C)	P*C
Development tools not mature	3	4	12
Not familiar with application domain	2	4	8
Project team not used to OOSE	5	5	25
Weak support from upper management	1	4	4
Delayed delivery of stable DBMS	4	5	20
...

3.3. Managing the Risk

- ❖ The third step is managing the risks. From the first two steps, we establish a prioritized table of the most serious threats to the project.
- ❖ For each of these serious threats we propose active actions to prevent them from occurring or to reduce their consequences. This could be done by decreasing the probability of the risk occurring or decreasing the consequences, or both.
- ❖ In the table (previous table), we see that the major threats are that the project team are not used to OOSE and we are not sure about the delivery of the DBMSs.
- ❖ The measures to avoid these threats could be to put in place an education program for the team members and not to use the DBMS proposed.

Software Quality Assurance

Software Quality Assurance?

- ❖ Software Quality Assurance is a planned and systematic way of creating an environment to assure that the software product being developed meets the quality requirements.
- ❖ This process is controlled and determined at managerial level.

SQA Activities & Tasks

1. Prepare a SQA plan for a product

- ✓ The plan is developed during project planning and reviewed by all interested parties.
- ✓ SQA activities, performed by the s/w engineering team & SQA team are governed by the plan.

2. Participate in the development of the project's s/w process descriptions

- ✓ The s/w engineering team select a process for work to be performed.
- ✓ The SQA groups reviews the process descriptions for the agreements with the organizations policy, standards etc.

SQA Activities & Tasks

3. Review s/w engineering activities to verify compliance (agreement) with the defined s/w process

✓ The SQA group identifies, documents and tracks deviations from the process and verify that and corrections have been made.

4. Audits designed s/w work product to verify compliance with those defined as part of the s/w process

✓ The SQA groups reviews selected work products, identifies documents, tracks deviations and verify that correctness have been made and periodically reports the results of its work to the project manager.

SQA Activities & Tasks

5. Ensure that deviation in s/w work and work product are documented and handled according to the documented procedure

✓ Deviation occurred in s/w product development process has been documented according to documentation procedure.

6. Periodically reports any non-compliance(agreement) and reports to the senior management

✓ Non agreement items are tracked and reports to the senior management.

Software Metrics

Software metrics

Software metric

- ✓ Any type of **measurement** which relates to a software system, process or related documentation.
 - Lines of code in a program, the Fog index-**readability test**, number of person-days required to develop a component.
- ✓ Allow the software and the software process to be quantified.
- ✓ May be used to predict product attributes or to control the software process.
- ✓ Product metrics can be **used** for **general predictions** or to **identify anomalous components**.

Software product metrics

Software Metrics	Description
Fan-in/ Fan-out	Fan-in is a measure of the number of functions or methods that call some other function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of Code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error proneness in components.
Length of identifiers	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth conditional Nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document is to understand.

Object oriented metrics

OO Metrics	Description
Depth inheritance tree	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from super-classes. The deeper the inheritance tree, the more complex the design . Many different object classes may have to be understood to understand the object classes at the leaves of the tree.
Method fan-in/fan-out	This is directly related to fan-in and fan-out as described above and means essentially the same thing. However, it may be appropriate to make a distinction between calls from other methods within the object and calls from external methods .
Weighted methods per class	This is the number of methods that are included in a class , weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1 and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class . Complex objects are more likely to be more difficult to understand. They may not be logically cohesive so cannot be reused effectively as super-classes in an inheritance tree.
Number of overriding operations	This is the number of operations in a super-class that are over-ridden in a subclass . A high value for this metric indicates that the super-class used may not be an appropriate parent for the sub-class.

cocomo model

Constructive Cost Model(COCOMO)

- ❖ COCOMO is one of the most widely used software estimation models in the world.
- ❖ In this model, size is measured in terms of thousand of delivered lines of code (KDLOC).
- ❖ In order to estimate effort accurately, COCOMO model divides projects into three categories:

1. Organic projects:

- ✓ These projects are small in size (not more than 50 KDLOC.)
- ✓ Example of organic project are, business system, inventory management system, payroll management system, and library management system.

2. Semi-detached projects:

- ✓ The size of semi-detached project is **not more than 300 KDLOC**.
- ✓ Examples of semi-detached projects include **operating system, compiler design, and database design**.

3. Embedded projects:

- ✓ These projects are **complex in nature** (size is more than 300 **KDLOC**).
- ✓ Example of embedded projects are software system used in **avionics and military hardware**.

Constructive cost model is based on the hierarchy of three models

- ✓ basic model
 - ✓ intermediate model and
 - ✓ advance model.

1. Basic Model:

- In basic model, only the size of project is considered while calculating effort.
 - To calculate effort, use the following equation (known as effort equation):

where E is the effort in person-months and size is measured in terms of KDLOC.

The values of constants ‘A’ and ‘B’ depend on the type of the software project. In this model, values of constants (‘A’ and ‘B’) for three different types of projects are listed in Table.

Project Type	A	B
Organic project	3.2	1.05
Semi-detached project	3.0	1.12
Embedded project	2.8	1.20

Example:

if the project is an organic project having a size of 30 KDLOC, then effort is calculated using equation,

$$E = 3.2 \times (30)^{1.05}$$

$$E = 114 \text{ Person-Month}$$

2. Intermediate Model:

- ❖ In intermediate model, parameters like software reliability and software complexity are also considered along with the size, while estimating effort.
- ❖ To estimate total effort in this model, a number of steps are followed, which are listed below:
 - ✓ Calculate an initial estimate of development effort by considering the size in terms of KLOC.
 - ✓ Identify a set of 15 parameters, which are derived from attributes of the current project. All these parameters are rated against a numeric value, called multiplying factor.
 - ✓ Effort adjustment factor (EAF) is derived by multiplying all the multiplying factors with each other.

The COCOMO II Effort Equation:

$$\text{Effort(Person-Month)} = 2.94(\text{Initial calibration}) * \text{EAF} * (\text{KDLOC})^{\text{E}}$$

Where,

EAF: Effort Adjustment Factor derived from the 15 Cost Drivers or multiplying factors (make assumption if value are not given in exam)

E: Exponent derived from the five Scale Drivers (make Assumption if value not given)

Example:

A project with all **Nominal Cost Drivers** and **Scale Drivers** would have an “EAF” of 1.00 and exponent “E” of 1.0997. Assuming that the project is projected to consist of 8,000 source lines of code.

Then by Using COCOMO II estimation

$$\text{Effort} = 2.94 * (1.0) * (8)^{1.0997} = 28.9 \text{ Person-Months}$$

In the same example if effort multipliers are given then

If your project is rated Very High for Complexity (effort multiplier of 1.34), and Low for Language & Tools Experience (effort multiplier of 1.09), and all of the other cost drivers are rated to be Nominal (effort multiplier of 1.00) then,

$$\text{Effort Adjustment Factor (EAF)} = 1.34 * 1.09 * 1 = 1.46$$
$$\text{Effort} = 2.94 * (1.46) * (8)^{1.0997} = 42.3 \text{ Person-Months}$$

COCOMO II Schedule Equation:

The COCOMO II schedule equation predicts the number of months required to complete your software project. It is predicted as:

$$\text{Duration} = 3.67 \times (\text{Initial calibration}) * (\text{Effort})^{\text{SE}}$$

Where,

Effort: Effort from the COCOMO II effort equation.

SE: Schedule equation exponent derived from the five Scale Drivers

Example:

Continuing previous example and assuming the schedule equation exponent of 0.3179 that is calculated from the five scale drivers.

$$\text{Duration} = 3.67 * (42.3)^{0.3179} = 12.1 \text{ months}$$

$$\text{Average staffing} = \text{Effort} / \text{Duration}$$

$$= (42.3 \text{ Person-Months}) / (12.1 \text{ Months}) = 3.5 \text{ people}$$

Example:

If your project is rated Very High for Complexity (effort multiplier of 1.34), and Nominal for Language & Tools Experience (effort multiplier of 1.00), and all of the other cost drivers are rated to be Nominal (effort multiplier of 1.09) assuming that exponent “E” of 1.0997 and project is projected to consist of 10500 source lines of code and the **schedule equation exponent of 0.419** that is calculated from the five scale drivers. Find the duration and average staffing.