

CS2002

INTRODUCTION

TO

SOFTWARE

ENGINEERING



FUNDAMENTALS OF SOFTWARE ENGINEERING

CS2002

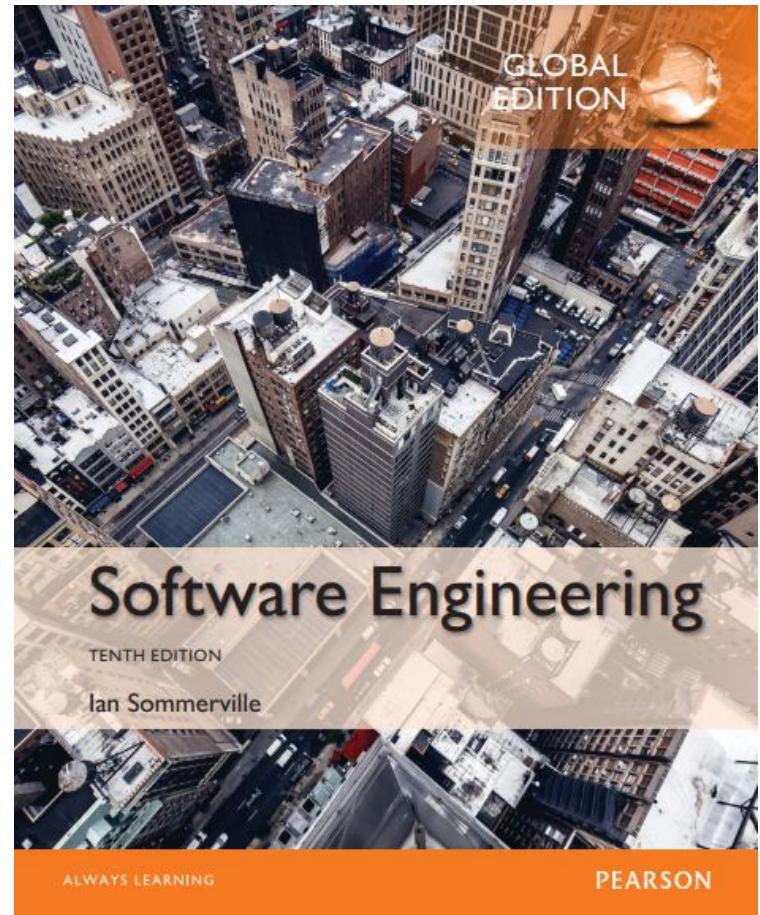


Course Information



Recommended Reading

- ❑ **Main Reference:** Ian Sommerville, Software Engineering, 10th Edition, Pearson, 2017.
- ❑ R Pressman, Software Engineering - A Practitioners Approach, 7th Edition, McGraw Hill.



Course Structure

3 Credit course

- Lectures

Course Assessment:

- Exam paper: 3 hours (70%)
- Assignments: Continuous Assignments + 1 Take home (30%)





Introduction to Software Engineering

Overall Objectives

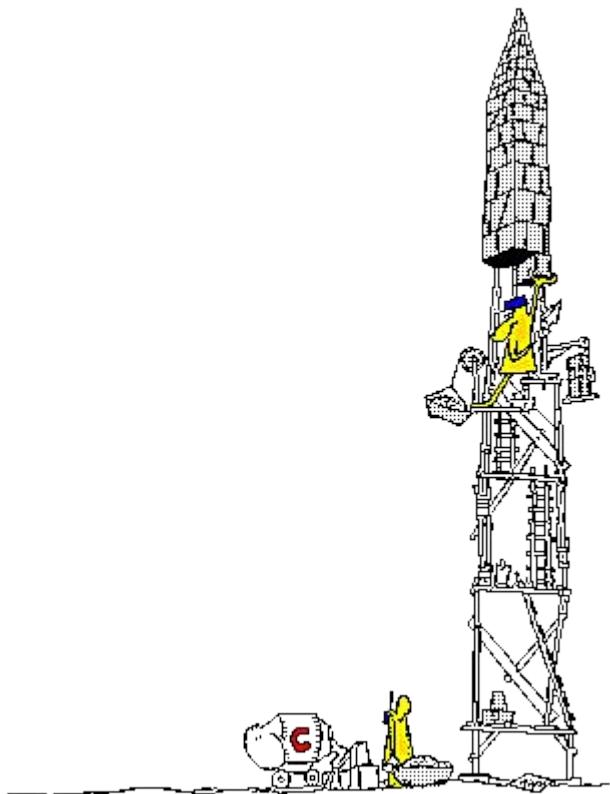
**After successfully completing this course,
students should be able to:**

- Understand what is software engineering, associated principles and concepts.
- Evaluate different software processes / techniques in systematic development of quality software products.
- Apply software engineering principles and techniques appropriately to design moderately complex software system.



Course Content

- Chapter 1 Introduction
- Chapter 2 Software processes
- Chapter 3 Agile software development
- Chapter 4 Requirements engineering
- Chapter 5 System modeling
- Chapter 6 Architectural design
- Chapter 7 Design and implementation
- Chapter 8 Software testing
- Chapter 9 Software evolution



Chapter 1 – Introduction to SE

Learning outcomes

Define what software means and its features

Define what software engineering is and why it is important?

Identify suitable software engineering techniques for development of different types of software

Recognize the ethical and professional issues that are important for software engineers

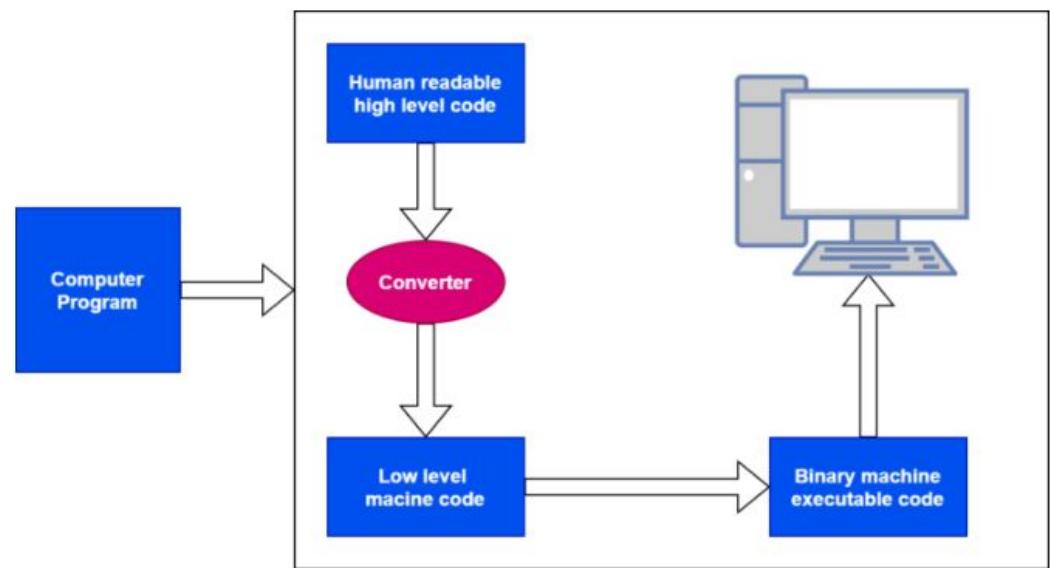
What is Software?

Many people think that software is simply another word for **computer programs**.

Computer Program

Set of instructions

Enables a computer to perform a specific task



What is Software?

Many people think that software is simply another word for computer programs.

**Software ≠
Computer Program**

What is Software?

Software is a combination of

- Computer programs
- System Documentation (documents produced during development)
- User Documentation (delivered with programs to customer at the time of release).

Computer Program

+

System and User Documentations

What does mean by a good
Software?

Feature of Software

- Software is intangible
 - Hard to understand development effort
- Software is easy to reproduce
 - Cost is in its *development*
 - in other engineering products, manufacturing is the costly stage
- The industry is labor-intensive
 - Hard to automate
- Untrained people can hack something together
 - Quality problems are hard to notice
- Software is easy to modify
 - People make changes without fully understanding it

Feature of Software

- Software does not 'wear out'
 - It deteriorates by having its design changed: erroneously, or in ways that were not anticipated, thus making it complex
- Complexity and Changes
 - Useful software systems are complex
 - Evolve with end user's need and the target environment
- Has poor design and is getting worse
- Demand for software is high and rising

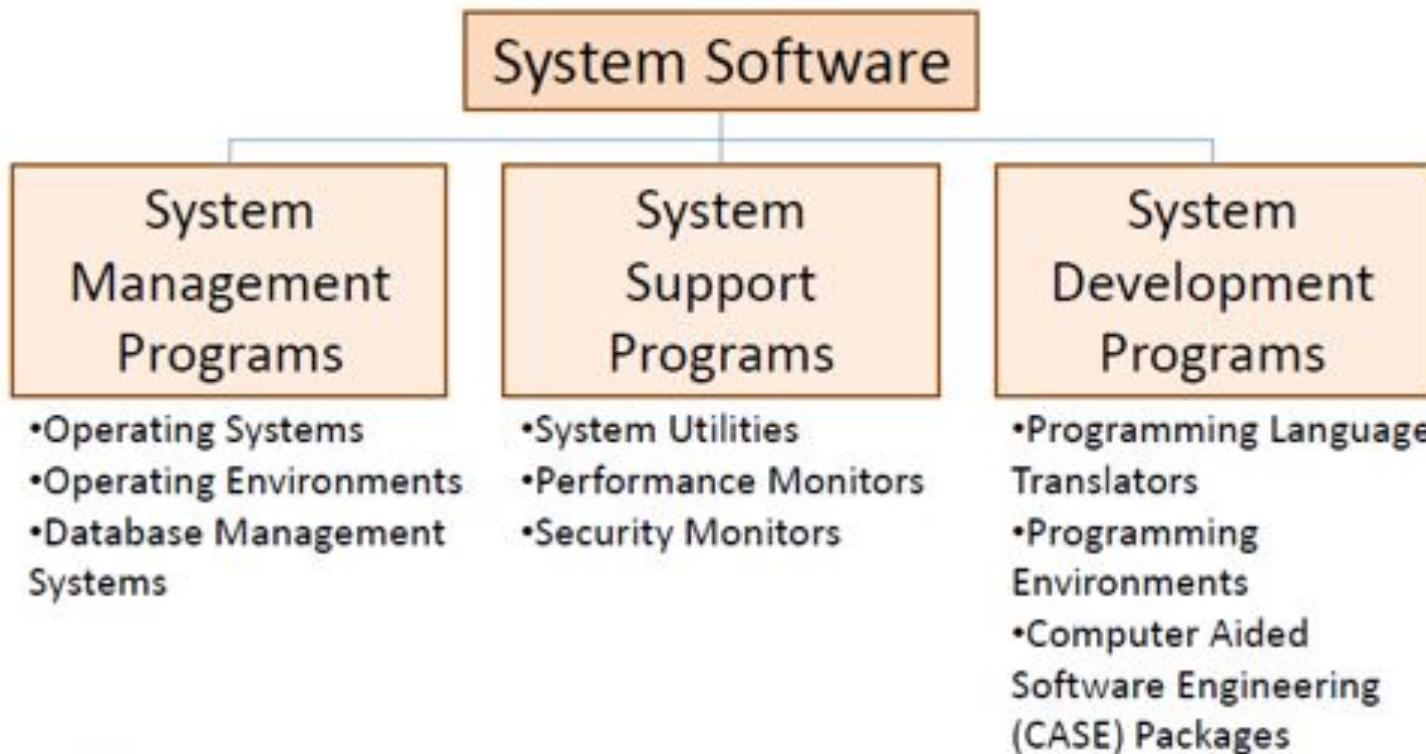
Software Types

System Software – Controls the operations of a computer and the other types of software that runs on the computer

e.g. Operating systems, device drivers, diagnostic tools, servers, etc...



System Software



Software Types

Application Software

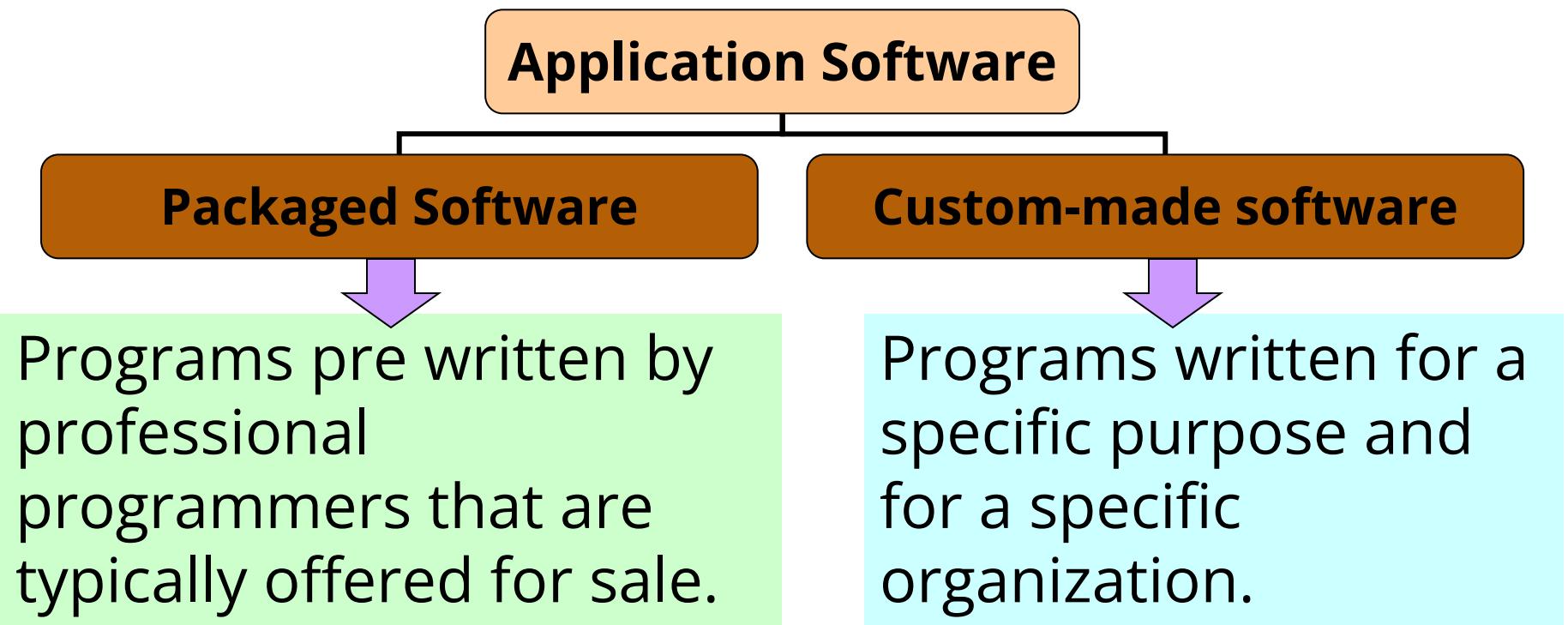
- “end-user” software
- Most commonly used software packages
- The type of program that you use once the operating system has been loaded

e.g:

- Word-processing programs: for producing letters, memos etc
- Spreadsheets: for doing accounts and working with numbers
- Databases: for organizing large amounts of information
- Graphics programs: for producing pictures, advertisements, manuals etc



Application software



What is a Software?

Software that can be sold to a customer.

May be developed for a particular customer or may be developed for a general market.

Software products may be

- Generic - developed to be sold to a range of different customers e.g. PC software such as Excel or Word.
- Customised (bespoke) - developed for a single customer according to their specification.

New software can be created by developing new programs, configuring generic software systems or reusing existing software.

Packaged or Generic Software

Stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them.

Examples:

- Office Packages (Microsoft Office, Open Office etc)
- Drawing Packages (AutoCAD, Adobe Photoshop ...)
- Databases (MySql, MSSQL)

Customized products

The systems which are commissioned by a particular customer.

A software contractor develops the software especially for that customer (i.e. developed for a single customer according to their specification).

Examples:

- Control systems for electronic devices
- Air traffic Control systems
- Healthcare systems
- Payroll systems

Exercise

Identify customized products from the following list?

- Adobe Photoshop
- Microsoft SQL Server
- Rent A Car system
- VLC Player
- Airline Reservation System
- Pharmacy Management System
- Python
- Library Management System

Common Software Types

- **System Software:**
 - System software is a collection of programs is written to service the other programs.
Eg: Operating system component, drivers, telecommunication process.



Common Software Types

- **Business software:**

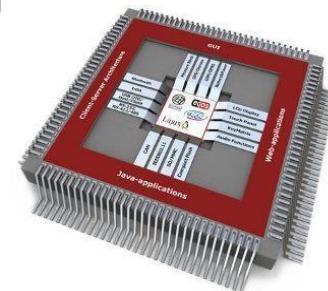
management information system software that access one or more large database containing business information



- **Embedded software:**

Embedded software resides in read-only memory and is used to control product and system for customer and industrial markets

Washing Machine
Microwave



Common Software Types

- **Web-based software:**

The network becomes a massive computer providing an almost unlimited software resources that can be accessed by anyone with a modem.



- **Artificial intelligence software:**

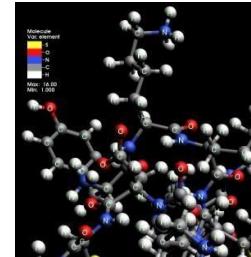
AI software makes use of non-numerical algorithms to solve the complex problems that are not amenable to computing or straightforward analysis.



Common Software Types

- **Engineering and scientific software:**

They have been characterized by number crunching algorithms



- **Personal computer software:**

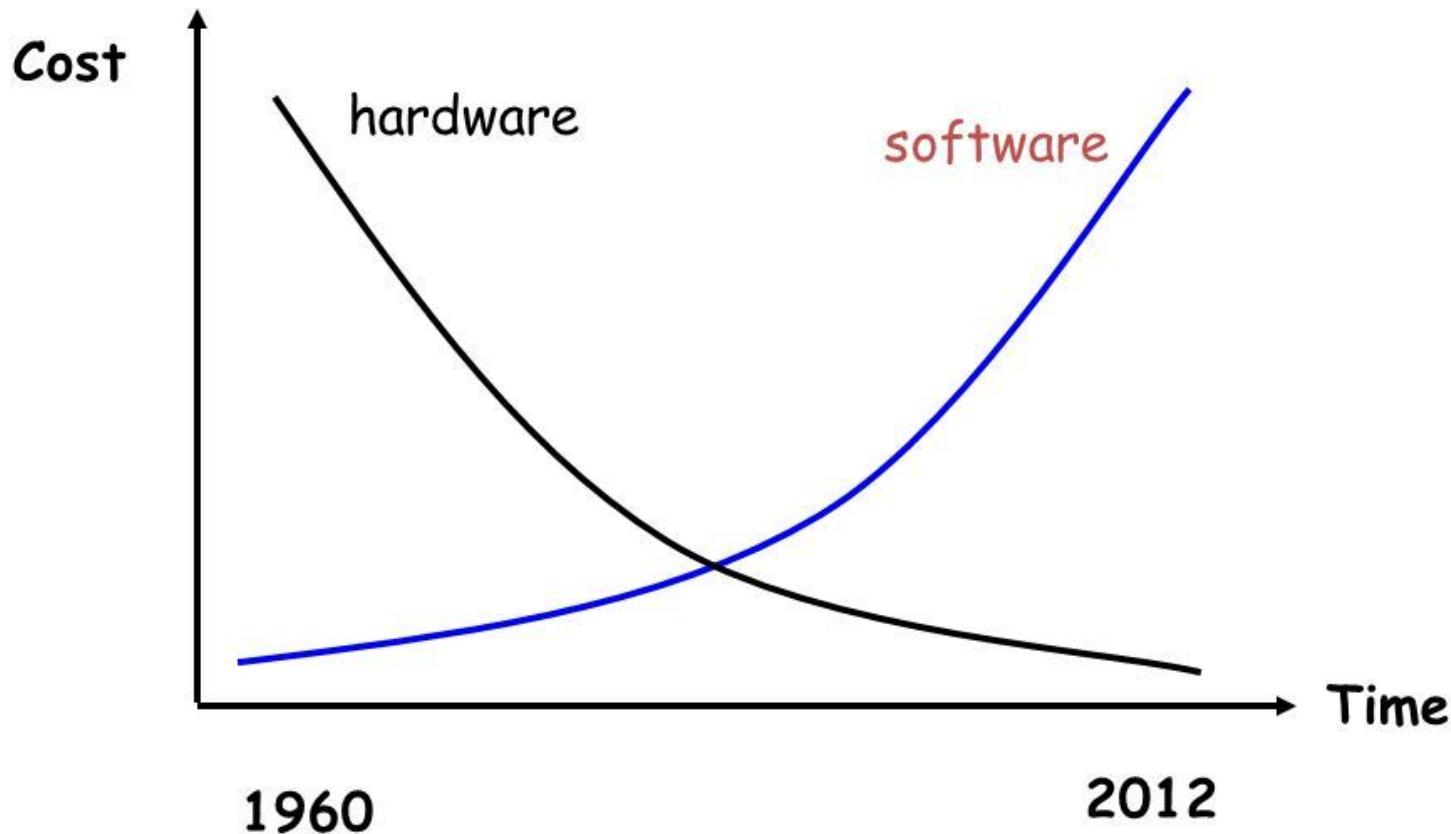
Personal computer software market has burgeoned over the past two decades.

Word processing, spreadsheets, computer graphic, multimedia and db management



Cost of software and
hardware

Cost of Hardware & Software



Software Cost

- Software costs often controlled by the computer system costs.
- The costs of software are often greater than the hardware cost.
- Software costs more to maintain than it does to develop.
- For systems with a long life, maintenance costs may be several times of development costs.
- Software engineering is concerned with cost effective software development

Software Cost Estimation

- How much **effort** is required to complete each activity?
- How much calendar **time** is needed to complete each activity?
- What is the **total cost** of each activity?

Software Development Cost

The total cost of a software development project is the sum of following costs

- Hardware and software costs including maintenance.
- Travel and training costs.
- Effort costs of paying software developers.

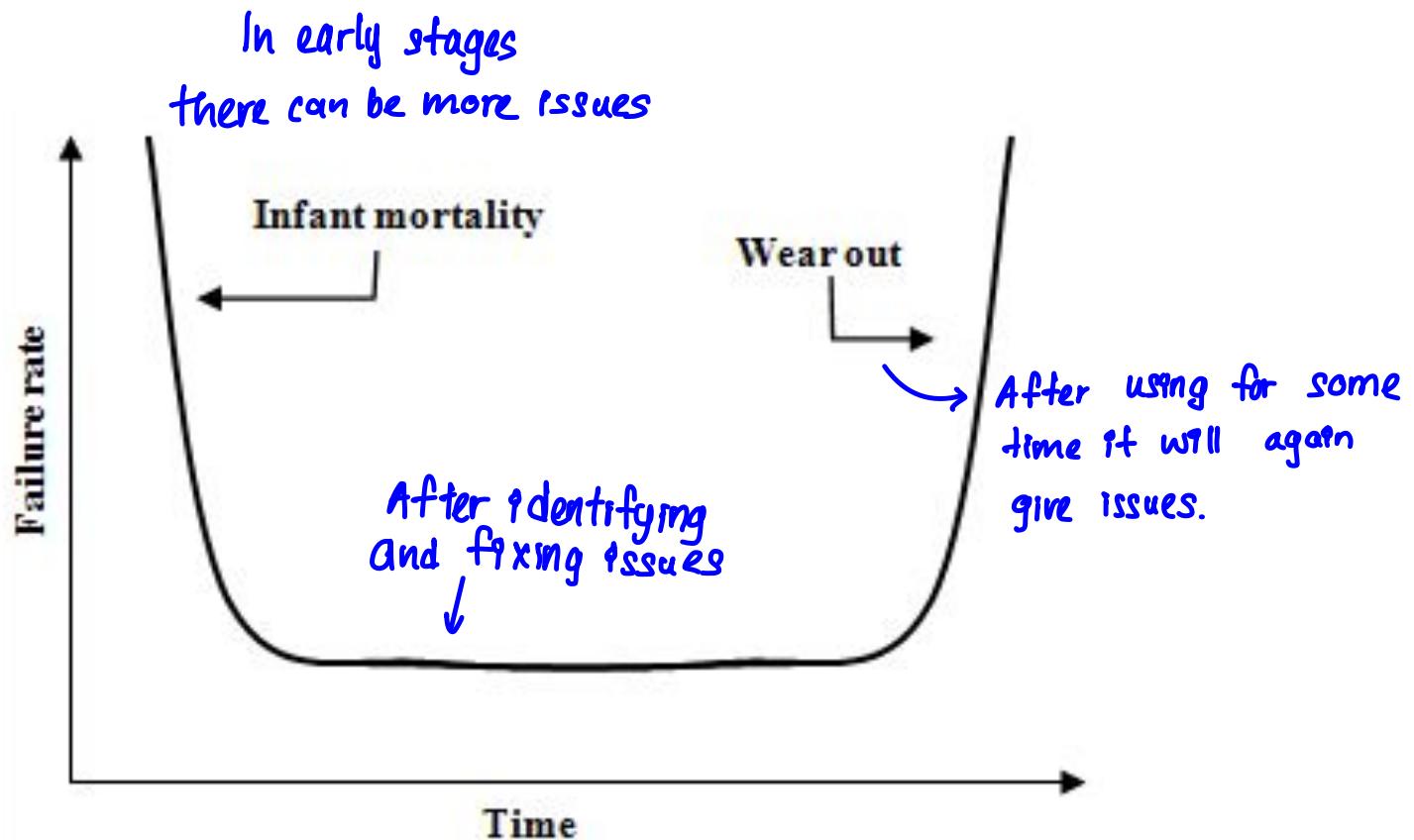
Distribution of Software Cost over Software Life Cycle

1. Requirements capture
2. Requirement specification 14%
3. Design
4. Implementation
5. Testing 6%
6. Maintenance 60%

40%

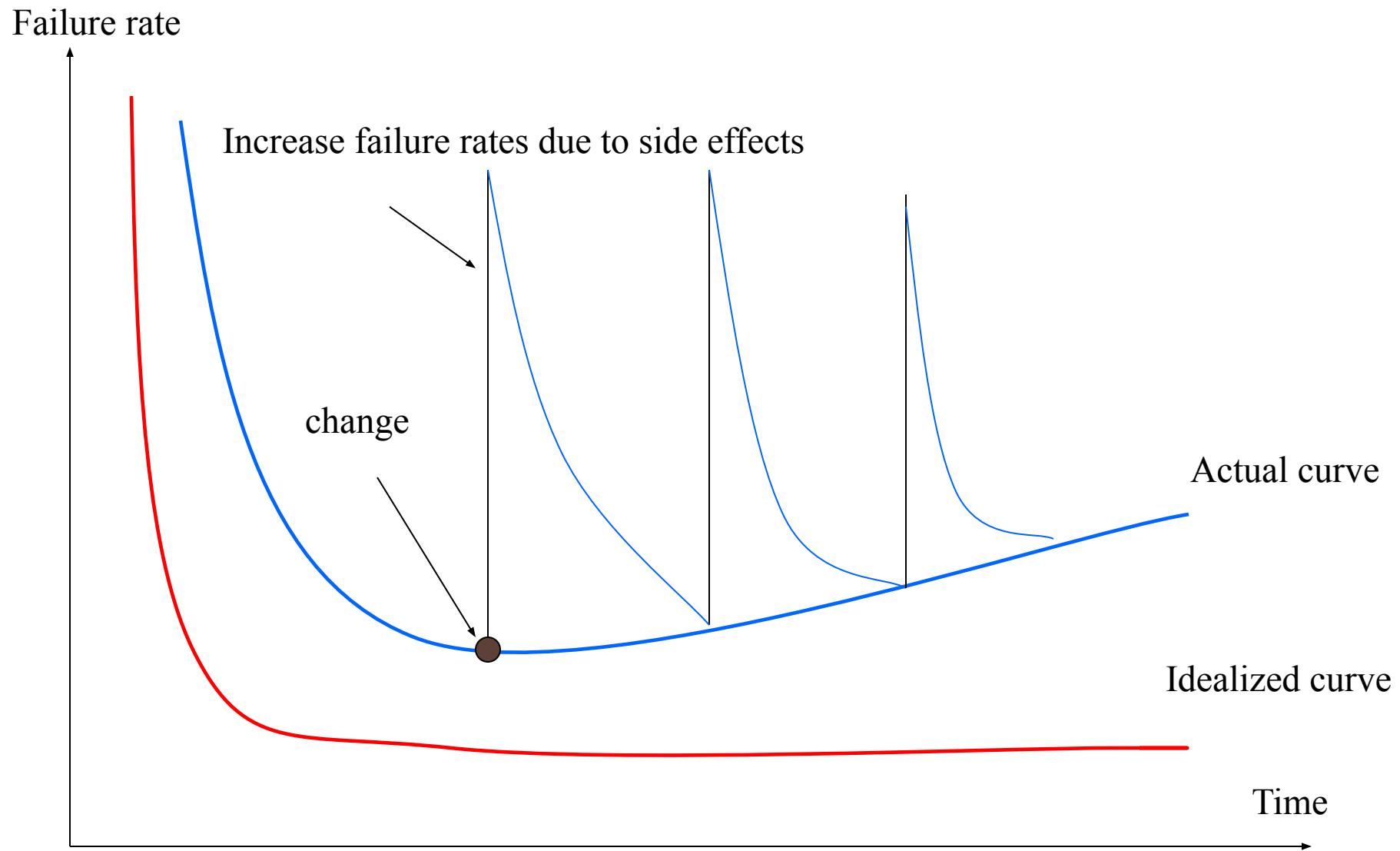
Failure curve of
hardware

Failure curve of hardware



Failure curve of software

Failure curves for software-Pressman



Project Success Criteria

- Deliver the software to the customer at the agreed time.
- Keep overall costs within budget.
- Deliver software that meets the customer's expectations.
- Maintain a happy and well-functioning development team.

Project Failure Causes

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Use of immature technology
- Inability to handle the project's complexity
- Careless development practices
- Poor project management



Software Development Failures

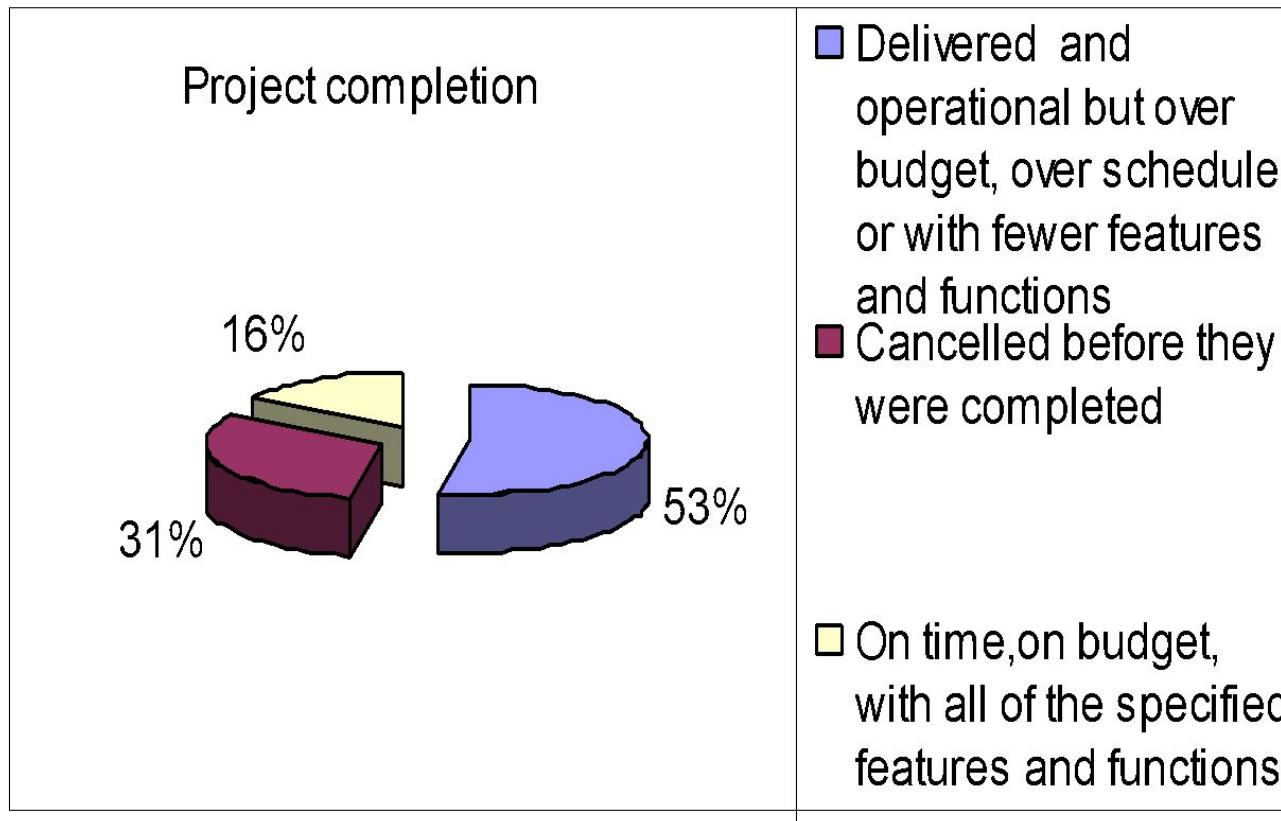
IBM Survey, 2000

- 55% of systems cost more than expected
- 68% overran the schedules
- 88% had to be substantially redesigned

Bureau of Labour Statistics (2001)

- for every 6 new systems put into operation, 2 cancelled
- probability of cancellation is about 50% for large systems
- average project overshoots schedule by 50%

Software Development Failures



Software Disasters

Patriot Missile System Timing Issue Leads To 28 Dead

By far the most tragic computer software blunder on our list occurred on February 25, 1991, during the Gulf War. While the Patriot Missile System was largely successful throughout the conflict, it failed to track and intercept a Scud missile that would strike an American barracks. The software had a delay and was not tracking the missile launch in real time, thus giving Iraq's missile the opportunity to break through and explode before anything could be done to stop it, according to the US Government Accountability Office. In all, 28 were killed with an additional 100+ injured.



Software Disasters...cont.

Apple Maps Goes Nowhere Fast

With the 2012 Apple iOS 6 update, the company decided to kick the superior Google Maps platform to the curb in favor of its own system. Unfortunately, it did a poor job of mapping out locations resulting in one of the most epic fails of the mobile computing movement. TPM IdeaLab noted the software was "missing entries for entire towns, incorrectly placed locations, incorrect locations given for simple queries, satellite imagery obscured by clouds and more" in a September 2012 report. David Pogue of the New York Times added that it was the most embarrassing, "least usable piece of software Apple has ever unleashed."



Software Disasters cont..

Airplane disasters



The crash of an Airbus plane, owned by China Airlines on the Nagoya Airport on April 16, 1994 due to the software problems.

Medical disasters



A medical disaster due to software-related accidents in safety-critical systems that used a computerized radiation therapy machine called the Therac-25. (contributed to the death of several cancer patients.)

.....FIND SOME MORE EXAMPLES

Problems of Software Development

User expectations:

- User expectations increase as the technology becomes more and more sophisticated.

The mythical man-month factor:

- Adding personnel to a project may not increase productivity.
- Adding personnel to a late project will just make it later.

Problems of Software Development

Communications:

- Communications among the various constituencies is a difficult problem. Sometimes different constituencies speak completely different languages.
- For example, developers may not have the domain knowledge of clients and / or users.
- The larger the project, the more difficult the communications problems become.



Problems of Software Development

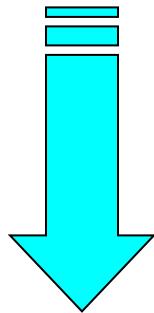
Project characteristics:

- size / complexity
- novelty of the application
- response-time characteristics
- security requirements
- user interface requirements
- reliability / criticality requirements

How to overcome these problems?

SOLUTION: SOFTWARE ENGINEERING

How to overcome such Problems?



Using a systematic process to
produce high quality software
products

What is Software Engineering?

Composed of two words 'software' and 'engineering'.

Engineering forces us to focus on systematic, scientific and well defined processes to produce a good quality product.



What is Software Engineering?

Software engineering is the establishment and use of sound engineering principles in order to obtain economical software that is reliable and works efficiently on real machines.

Software Engineering is a profession dedicated to designing, implementing, and modifying,

- High quality
- More affordable
- Maintainable software

Software Engineering - Evolution

1. Software development began as a single person activity in 1940s and 1950s.
2. Software engineering was considered a new scientific discipline in 1960s and 1970s.
3. In 1980s and 1990s engineering ideas dominated software development

Importance of Software Engineering

The economies of ALL developed nations are dependent on software.

More and more systems are software controlled

- Transportation
- Medical (Channeling, Pharmacy)
- Telecommunications
- Military
- Industrial (Car Manufacturing)
- Entertainment (Video and Audio Players)

Software is found in products and situations where very high reliability is expected

- E.g. Monitoring and controlling Nuclear power plants

Contain millions of lines of code

Comparably more complex

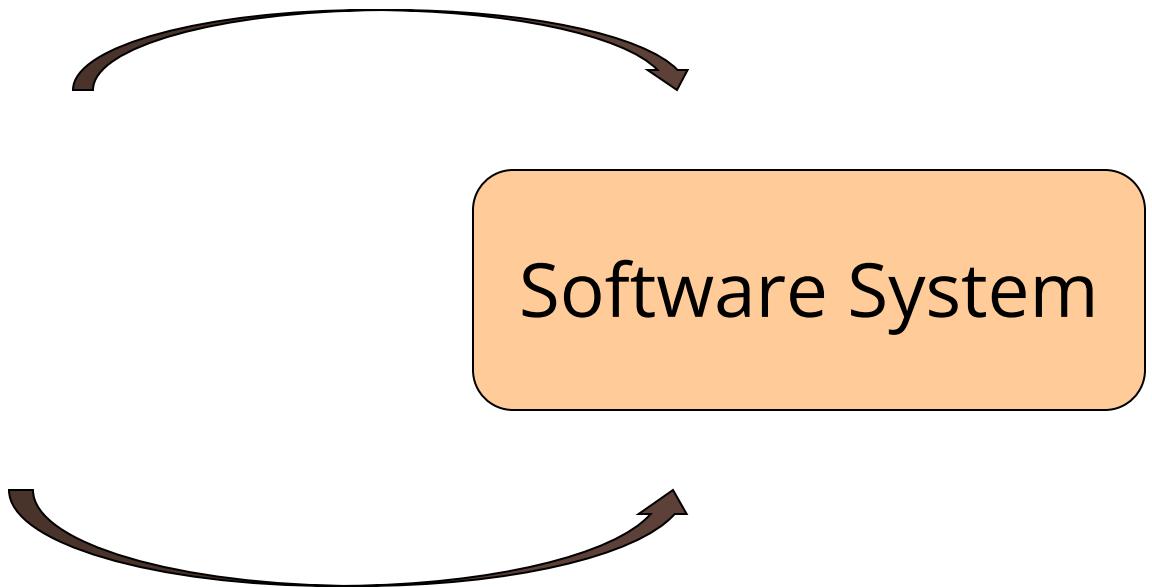
Importance of Software Engineering

Concerned with the

- Conception
- Development
- Verification

Deals with

- Identifying
- Defining
- Realizing
- Verifying



Importance of Software Engineering

Software engineering is concerned with theories, methods and tools for professional software development,

Which helps to

- reduce complexity
- minimize software cost
- decrease project time
- handle big projects
- develop reliable software
- develop more effective software

Software Engineering: Definitions

'Software engineering is concerned with the theories, methods and tools for developing, managing and evolving software products'

- I Sommerville

'The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them'

- B.W.Boehm

Software Engineering: Definitions

'The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines'

- F.L. Bauer

'The application of systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software'

- IEEE Standard 610.12

Software Engineering: Definitions

An **engineering discipline** that is concerned with **all aspects of software production** from the early stages of system specification through to maintaining the system after it has gone into use.

Software engineering

*In the given definition, there are **TWO** key phrases:*

- 1. Engineering discipline** - Engineers make things work. They apply concepts, theories, methods and tools where these are appropriate.
- 2. All Aspects of Software Production** - Software Engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management, quality management

Software Engineering vs Computer Science?

Software Engineering vs Computer Science?

Computer science is concerned with theory and fundamentals

Software engineering is concerned with the practicalities of developing and delivering useful software.

Computer science theories are still insufficient to act as a complete underpinning for software engineering

Software Engineering vs System Engineering?

Software Engineering vs System Engineering?

System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering.

Software engineering is part of this process concerned with developing the software infrastructure, control, applications and databases in the system.

Software Quality

- Quality means that a product should meet its specification

The different qualities can conflict

- Increasing efficiency can reduce maintainability or reusability
- Increasing usability can reduce efficiency

Setting objectives for quality is a key engineering activity

- You then design to meet the objectives
- Avoids 'over-engineering' which wastes money

Stakeholders in Software Engineering

1. Users

- Those who use the software

2. Customers/Clients

- Those who pay for the software

3. Software developers

- Those who develop and maintain software

4. Development Managers

- Those who run organization that is developing software

Software Quality Attributes

Customer:

solves problems at
an acceptable cost in
terms of money paid and
resources used

User:

easy to learn;
efficient to use;
helps get work done

Developer:

easy to design;
easy to maintain;
easy to reuse its parts



Development manager:

sells more and
pleases customers
while costing less
to develop and maintain

Types of Quality Attributes

(Developer's Perspective)

Internal Quality Attributes

Implementation Related

Maintainability, Flexibility, Portability, Re-usability, Readability, Testability, and Understandability.

(User Perspective)

External Quality Attributes

Usage Related.

Correctness, Usability, Efficiency, Reliability, Integrity, Adaptability, Accuracy, and Robustness

Think of examples
to explain these facts

External Quality Attributes

Availability

Installability

Integrity

Interoperability

↳ How easy to exchange from
one system to another system.

Performance

Reliability

Recoverability

Robustness

↳ How it will react in
unexpected situations.

Safety

Usability

Internal Quality Attributes

How it utilize the resources.

Efficiency

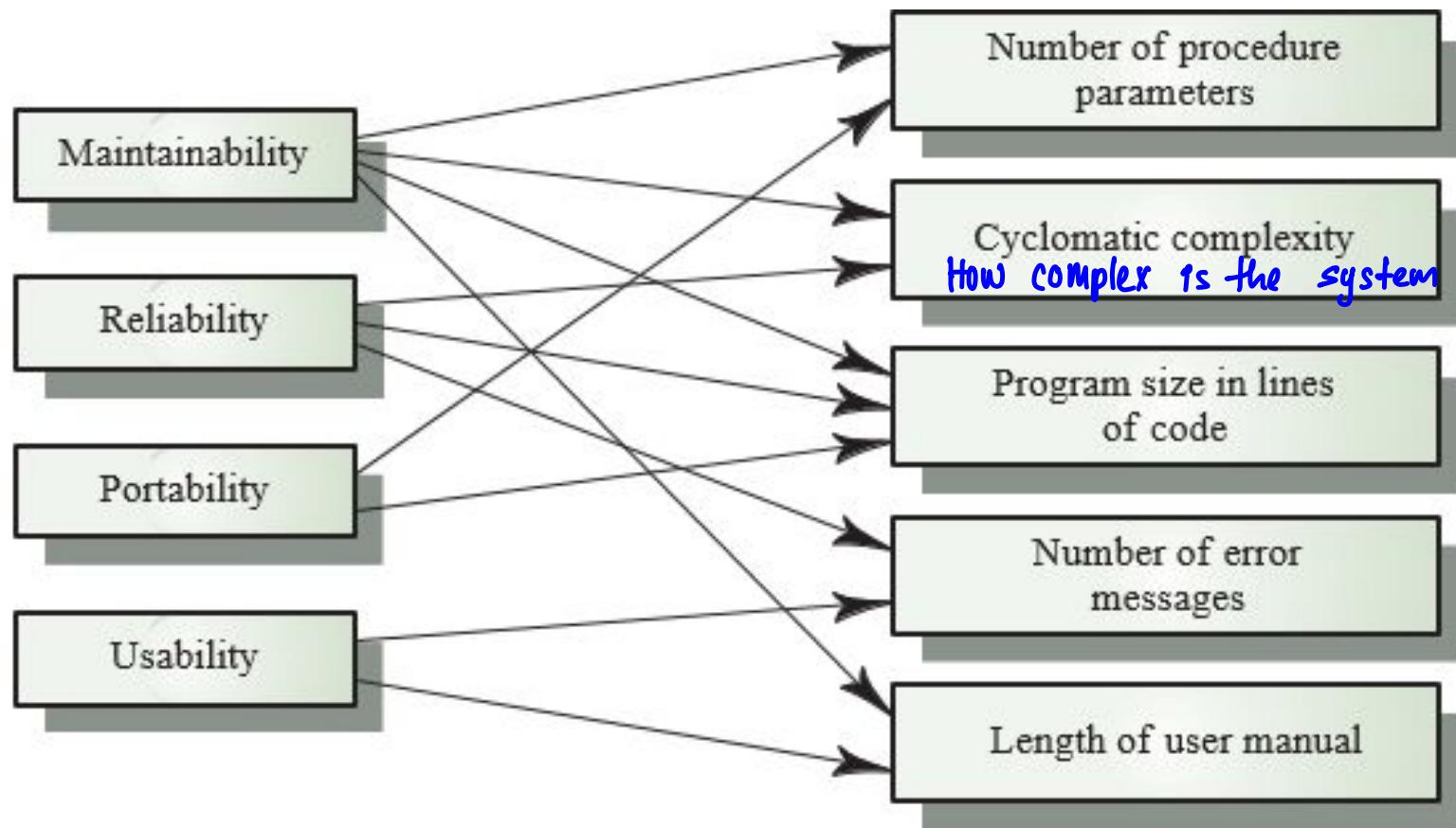
Flexibility

Maintainability

Portability

How easily adoptable for future changes

Reusability
Scalability
Testability



Software Quality Attributes – Engineering Aspect

- **Availability**
 - Availability is the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
- **Security**
 - Security is the ability of the software to remain protected from unauthorized access. This includes both change access and view access.
- **Maintainability**
 - Maintainability is the ability of a software to adapt to changes, improve over time, correct any bugs and be proactively fixed through preventive maintenance.
- **Testability**
 - Testability is the ability of a software to be tested thoroughly before putting into production.

Software Quality Attributes – Engineering Aspect

- **Flexibility**
 - Flexibility is the ability of a software to adapt when external changes occur.
- **Traceability**
 - Traceability is the ability of the Software to offer insight into the inner processing when required. A higher level of traceability is required at time of debugging a problem or at times of new interoperability testing.
- **Modifiability**
 - Modifiability is a measure of how easy it may be to change an application to cater for new functional and non-functional requirements.

Software Quality Attributes – Technology Aspect

- **Reliability**
 - Reliability is the measure of how a product behaves in varying circumstances. It is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified time interval.
- **Robustness**
 - Robustness is defined as the ability of a software product to cope with unusual situation.
- **Efficiency**
 - Efficiency is the ability of the software to do the required processing on least amount of hardware.

Software Quality Attributes – Technology Aspect

- **Compatibility**
 - Compatibility is the ability of the software to work with other systems.
- **Modularity**
 - Modularity is the measure of the extent to which software is composed of separate, interchangeable components, each of which accomplishes one function and contains everything necessary to accomplish this. Modularity increases cohesion and reduces coupling and makes it easier to extend the functionality and maintain the code.
- **Reusability**
 - Reusability is the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.

Software Quality Attributes

–Operational Aspect

- **Manageability**
 - Manageability is the ability of a software to be managed from different perspectives.
- **Usability**
 - Usability is the ability of a software to offer its interfaces in a user friendly and elegant way.
- **Accessibility**
 - Accessibility is the ability of a software to be accessible from a multitude of devices and for a number of different types of users. (e.g. users with disabilities)
- **Auditability**
 - Auditability is the ability of a software to keep track of what happened, who did it, from where, how and when.

Software Quality Attributes

–Operational Aspect

- **Portability**
 - Portability is the ability of a program (or system) to execute properly on multiple hardware platforms.
- **Performance**
 - Performance is an indication of the responsiveness of a system to execute any action within a given time interval. It can be measured in terms of latency or throughput. Latency is the time taken to respond to any event. Throughput is the number of events that take place within a given amount of time.

Over Engineering

- This happens when you try to address future needs/requirements
- Architecting and designing the application with more components than it really should have according to the requirements list
- Good design is design that leads to simplicity in implementation and maintenance, and makes it easy to understand the code. Over engineered design is design that leads to difficulty in implementation, makes maintenance a nightmare, and turns otherwise simple code into complex
- **The best way to avoid over-engineering is just don't design too far into the future**

Software Engineering Projects

- Software engineering work is normally organized into projects.

Categories of Software Projects

- ***Evolutionary Projects*** - Those that involve modifying an existing system
- ***Green Field Projects*** - Those that involve starting to develop a system from scratch
- ***Framework-based Projects*** - Those that involve building most of a new system from existing components

↳ Using libraries?

Activities Common to Software Projects

- Requirements and specification
 - Includes
 - Domain analysis
 - Defining the problem
 - Requirements gathering
 - Obtaining input from as many sources as possible
 - Requirements analysis
 - Organizing the information
 - Requirements specification
 - Writing detailed instructions about how the software should behave

Activities Common to Software Projects

- Design
 - Deciding how the requirements should be implemented, using the available technology
 - Includes:
 - Systems engineering: Deciding what should be implemented in hardware and what in software
 - Software architecture: Dividing the system into subsystems and deciding how the subsystems will interact
 - Detailed design of the internals of a subsystem
 - User interface design
 - Design of databases

Activities Common to Software Projects

- Modeling
 - Process of creating representations of the domain or the software
 - Use case modeling
 - Structural modeling
 - Dynamic and behavioral modeling
- Implementation/Programming
 - Translation of high-level design into programming language
 - Some tools can generate codes based on the design to some extent

Activities Common to Software Projects

- Deployment
 - Involves distributing and installing the software and any other components of the system (databases, special hardware, etc.)
- Software Project Management
 - An integral part of software engineering process
 - PM Knowledge Framework based on PMI Guidelines

Software Engineering Diversity

There are many different types of software systems.

There is no universal set of software techniques that are suitable for all systems and all companies.

The software engineering methods and tools used depend on the *type* of application being developed, the *requirements* of the *customer* and the *background* of the development team.

Software Engineering Ethics

Software engineering is carried out within a social and legal framework that limits the freedom of people working in that area.

As a software engineer, your job involves wider responsibilities than simply the application of technical skills

You should not use your skills and abilities to behave in a dishonest way

Definition: – Ethics is a set of moral principles that govern the behavior of a group or individual

Issues of Professional Responsibility

Confidentiality

Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

Competence

Engineers should not misrepresent their level of competence. They should not knowingly accept work which is out with their competence.

Issues of Professional Responsibility

Intellectual property rights

Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.

Computer misuse

Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

SOFTWARE PROCESS

Software Process

- A set of ordered tasks involving activities, constraints and resources that produce a software system.
- It guides our actions by allowing us to examine, understand, control and improve the activities that comprise the process



Software Process Models

Wasn't included in
our pdf document

- A software process model is an **abstract representation** of a process.
- It presents a description of a process from some particular perspective.
- By modeling the process:
 - forms a common understanding of the activities, resources and constraints involved in software development.
 - helps the team find inconsistencies, redundancies and commissions in the process
 - process becomes more effective
 - the model reflects the goals of development and shows explicitly how the product characteristics are to be achieved .
 - model helps people to understand these

SOFTWARE PROCESS ACTIVITIES



Software process descriptions

When we describe and discuss processes, we usually talk about the **activities** in these processes such as *specifying a data model, designing a user interface, etc.* and the *ordering of these activities*.

Software process descriptions

Process descriptions may also include:

- **Products**, which are the outcomes of a process activity;
- **Roles**, which reflect the responsibilities of the people involved in the process;
 ↳ Stakeholders (users, clients, Developers, Managers)
- **Pre- and post-conditions**, which are statements that are true before and after a process activity has been enacted or a product produced.

Login process ← Pre condition

Processes done after

the desired task ← Post Condition
is completed.

(Printing the booking /
Removing the reserved
seating from availables)

Eg:- Reserving Movie tickets

Login to the system - Pre

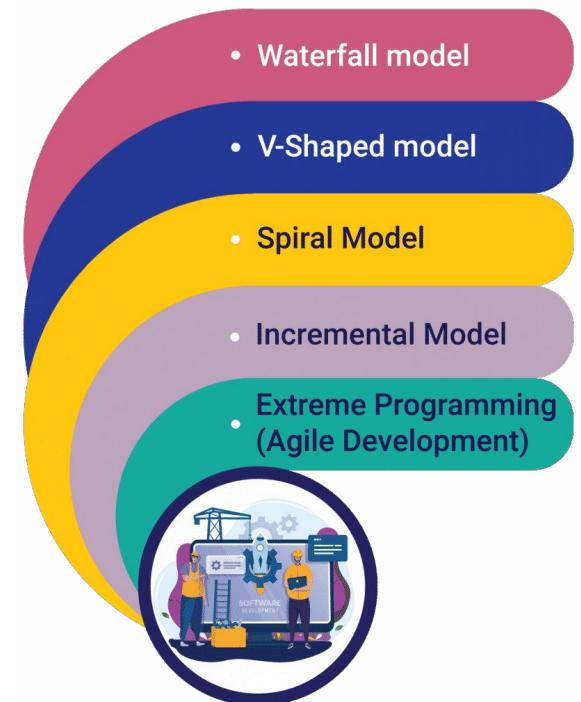
Printing the reserved ticket - Post

Plan-driven and agile processes

- **Plan-driven processes** - all of the process activities are planned in advance and progress is measured against this plan.
- **Agile processes** - planning is incremental and it is easier to change the process to reflect changing customer requirements.

Software process models

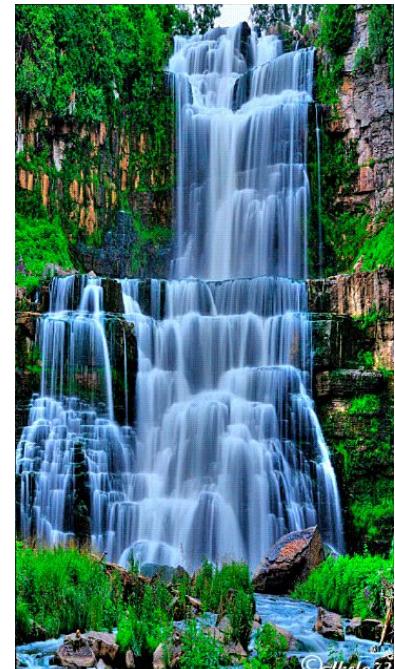
- Waterfall model
- Incremental development
- Prototyping
- The spiral model
- Rapid Application Development
- Unified Process



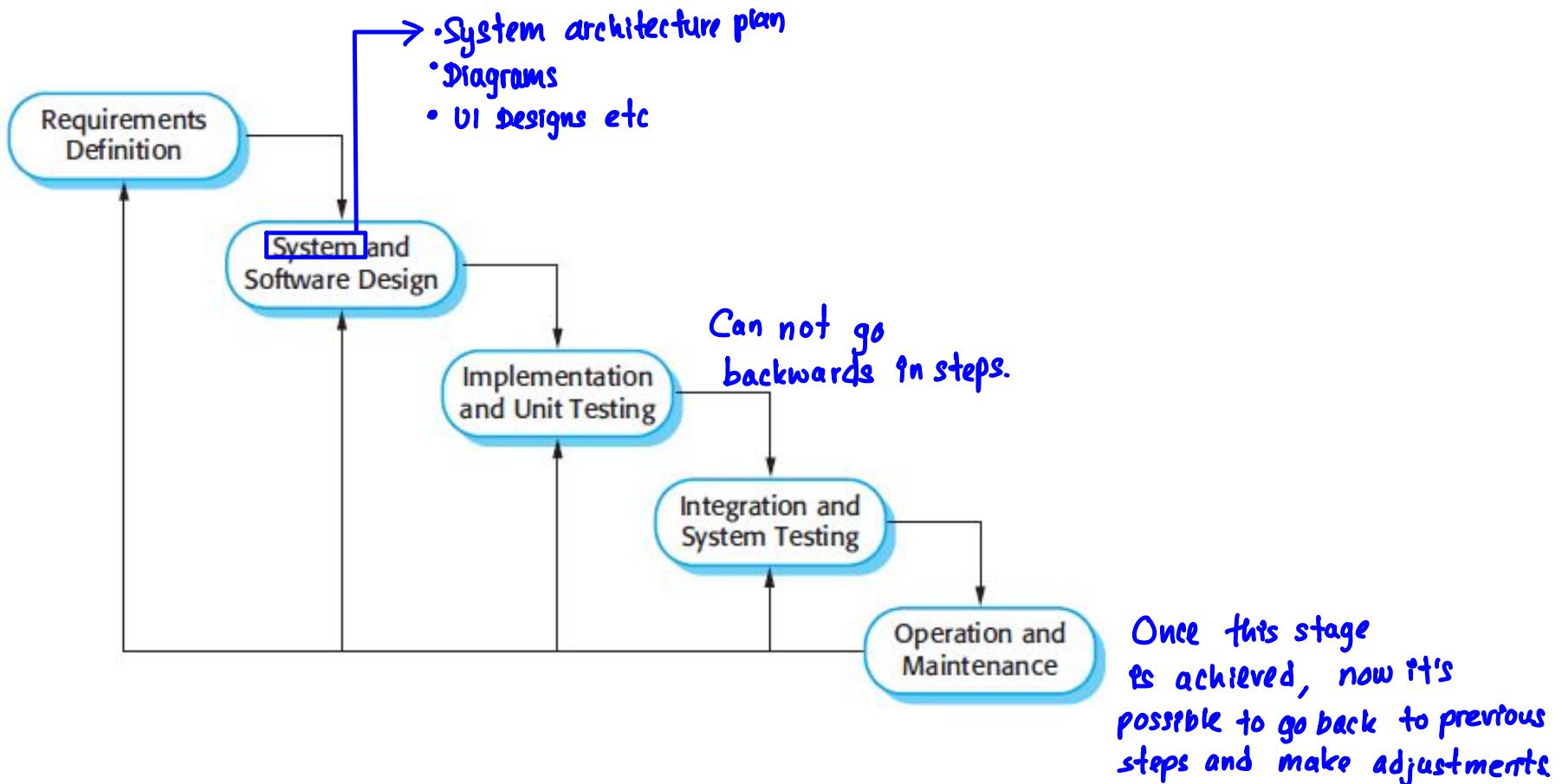
Waterfall model

Waterfall model

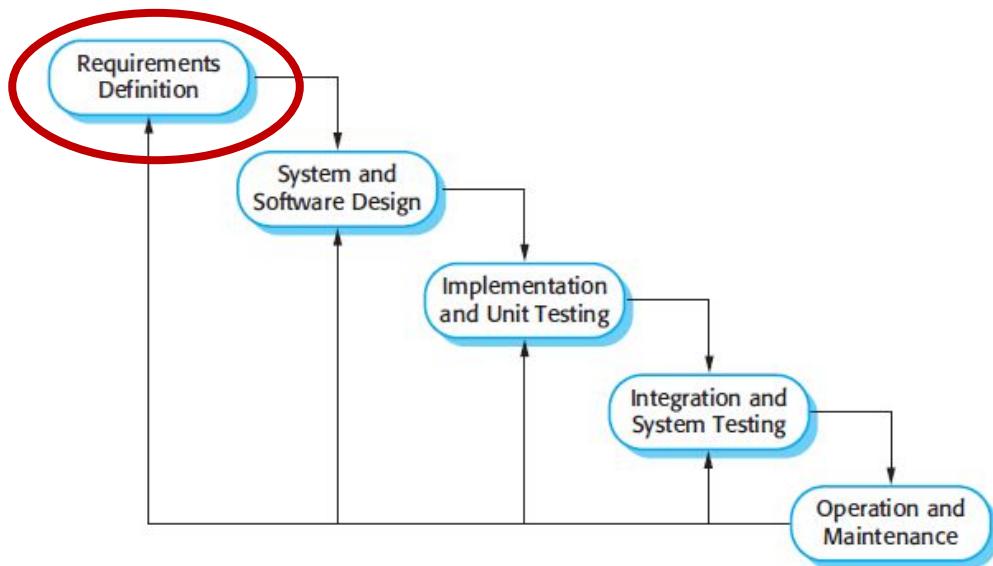
- The first published model of the software development process
- Is an example of a plan-driven process.
- In principle at least, you plan and schedule all of the process activities before starting software development



Waterfall model



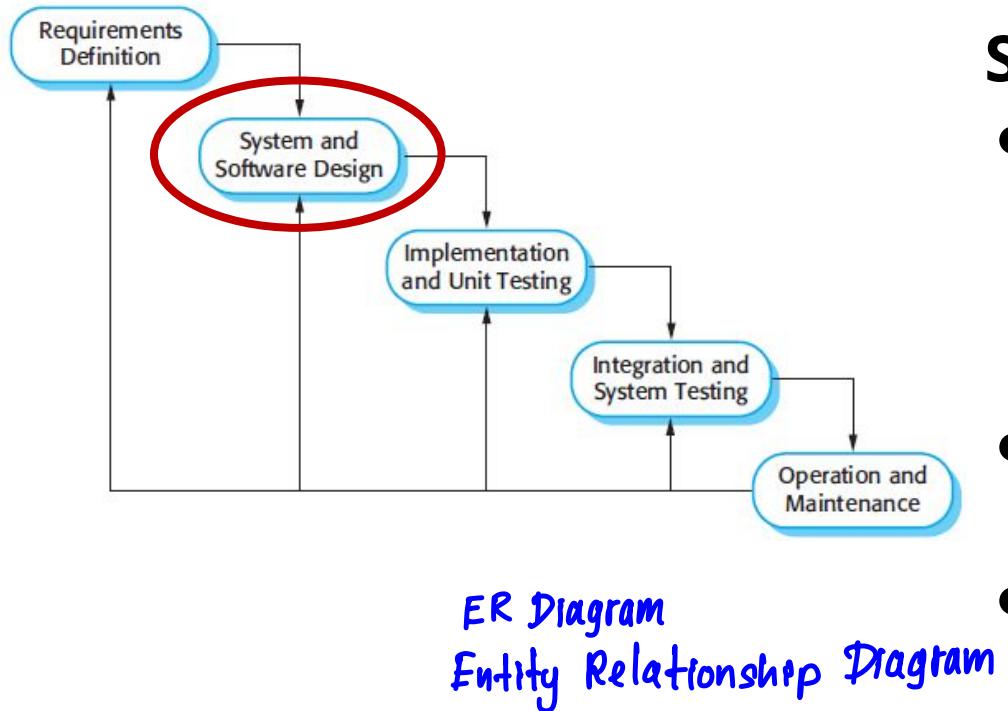
Waterfall model



Requirements analysis and definition

- The system's services, constraints, and goals are established by consultation with system users.
- They are then defined in detail and serve as a system specification.

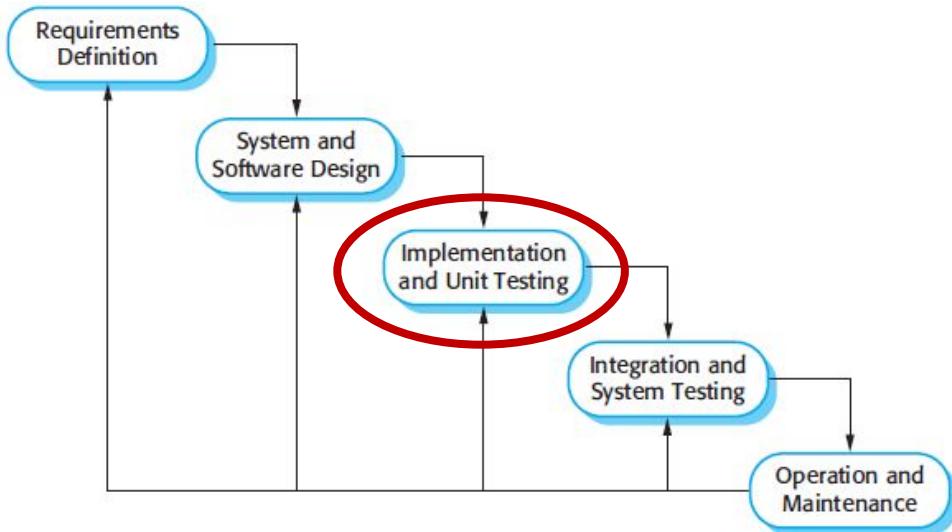
Waterfall model



System and software design

- The systems design process allocates the requirements to either hardware or software systems.
- It establishes an overall system architecture.
- Software design involves identifying and describing the fundamental software system abstractions and their relationships.

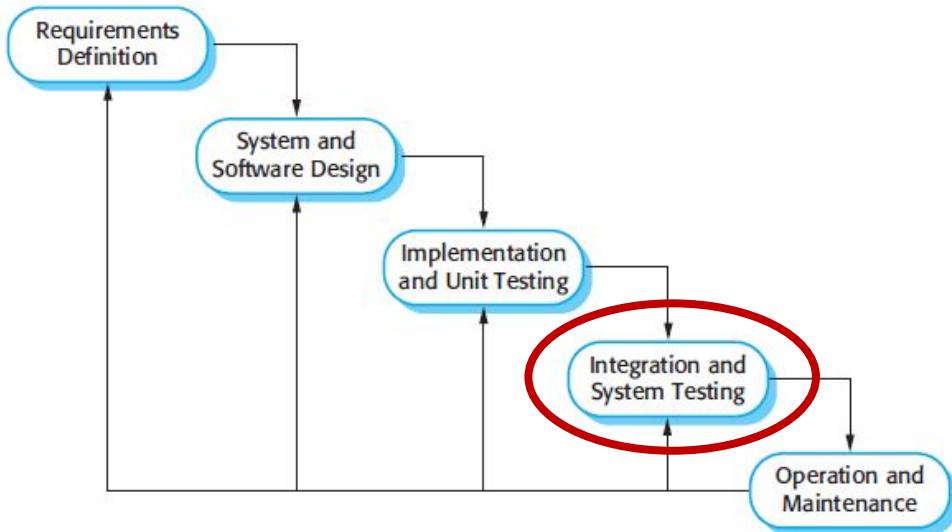
Waterfall model



Implementation and unit testing

- During this stage, the software design is realized as a set of programs or program units.
- Unit testing involves verifying that

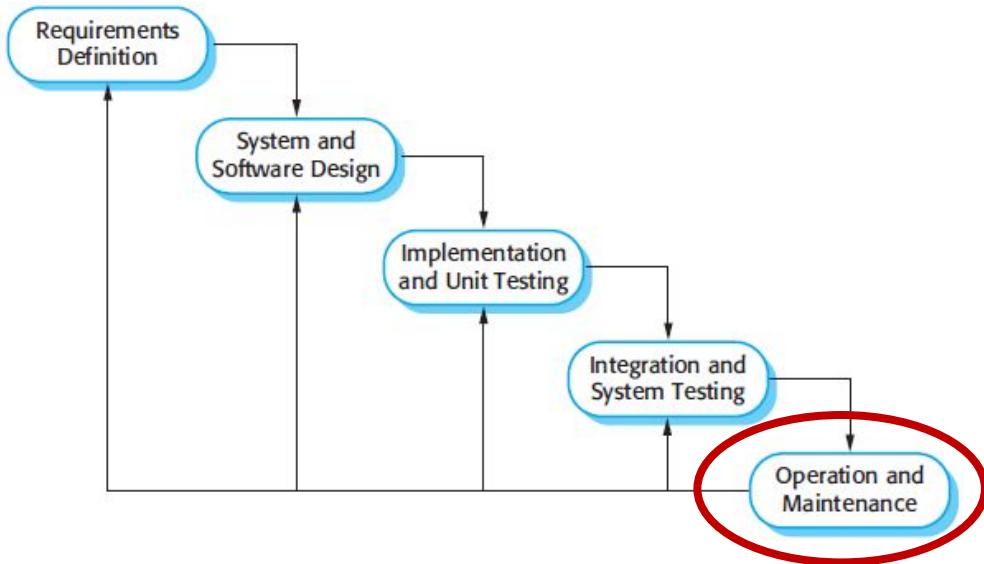
Waterfall model



Integration and system testing

- The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met.
- After testing, the software system is delivered to the customer.

Waterfall model



Operation and maintenance

- Normally, this is the longest life-cycle phase.
- The system is installed and put into practical use.
- Maintenance involves
 - correcting errors that were not discovered in earlier stages of the life cycle.
 - improving the implementation of system units.
 - enhancing the system's services as new requirements are discovered.

Waterfall model can be used when

- Requirements are not changing frequently
- Application/project is not complicated and big
- Project is short
- Requirements are clear
- Environment/requirements are stable
- Technology and tools used are not changing dynamically
- Resources are available and trained

Outcomes

Stage	Outputs
Requirements specifications	SRS document Draft user manual Maintenance plan
System design and software design	System design document Hardware design document Software design document Interface design document Unit test plan System test plan
Implementation and unit testing	Program code Unit test report
Integration and system testing	System test report Final user manual Working system
Operation and maintenance	\$ (if the software is without any defects)

Waterfall Model

- The main drawback of the waterfall model is the **difficulty of accommodating change** after the process is underway. One phase has to be complete before moving onto the next phase.
- This model is only appropriate when the **requirements are well-understood and changes will be fairly limited** during the design process.
- Few business systems have stable requirements.
- The waterfall model is mostly used for **large systems** engineering projects where a system is developed at several sites.

Incremental Development

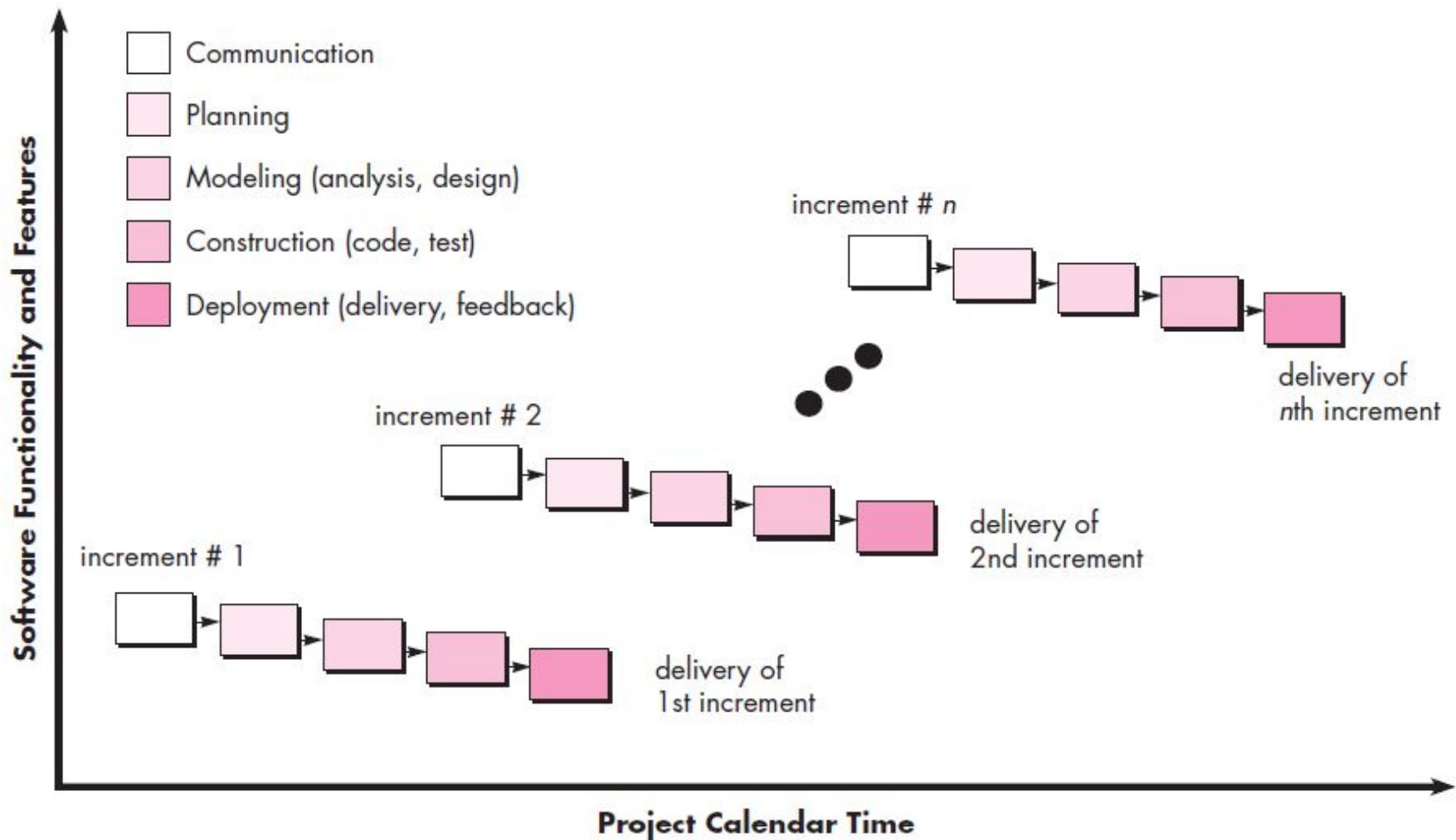
Incremental



Iterative



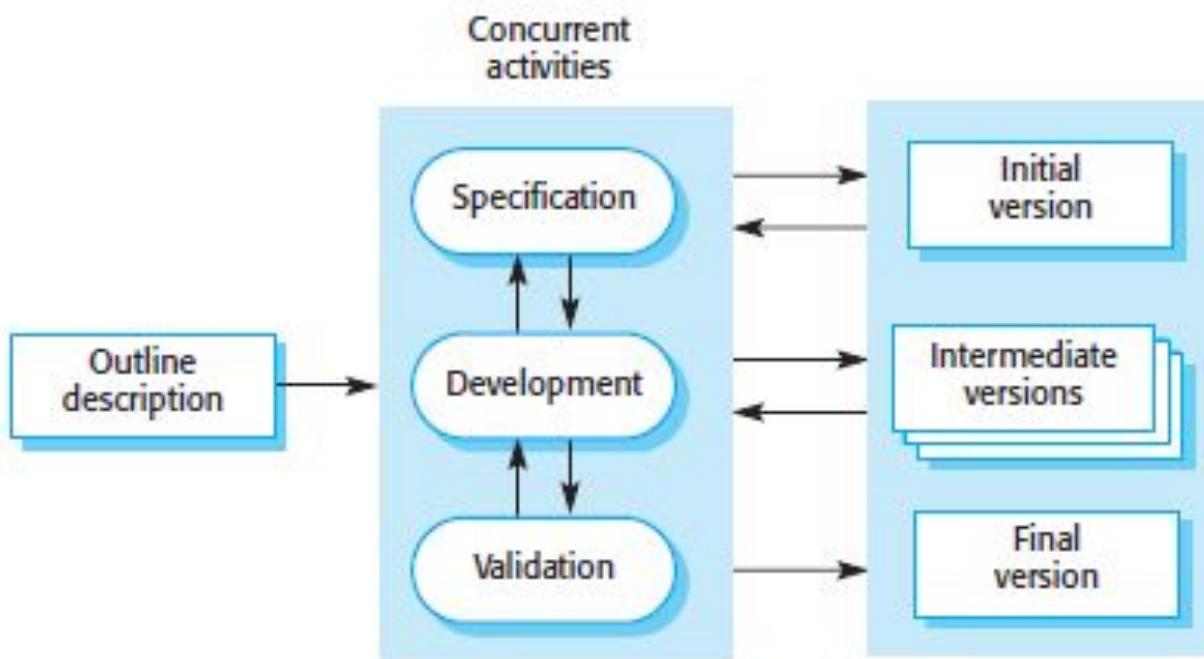
Incremental Development



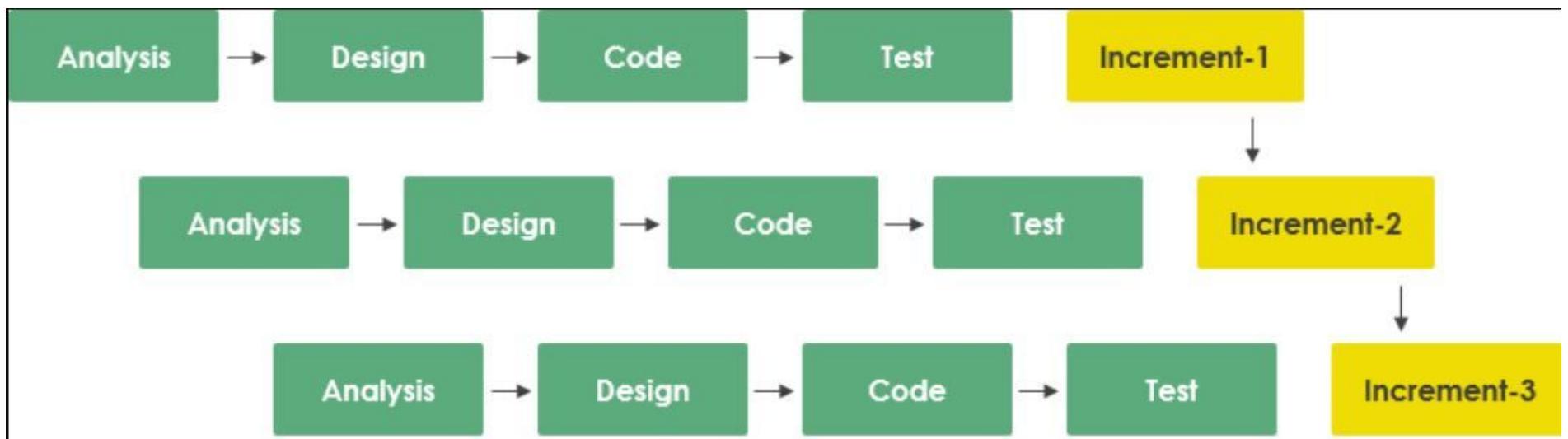
Incremental Development

- Incremental development is based on the idea of *developing an initial implementation, getting feedback from users and evolving the software through several versions* until the required system has been developed.
- System development is **broken** down into many **mini development projects**.
- Partial systems are successively built to produce a final total system.
- Once the development of an increment is started, the requirements are frozen, though requirements for later increments can continue to evolve.
- Highest priority requirement is tackled first

Incremental Development



Incremental Development



Benefits

- The software will be generated quickly during the software life cycle
- It is flexible and less expensive to change requirements and scope
- Through out the development stages, changes can be done
- This model is less costly compared to others
- A customer can respond to each building [User Involvement is very high]
- Errors are easy to be identified
- Lower risk of overall project failure

Problems

- The process is not visible.
 - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost effective to produce documents that reflect every version of the system.
- System structure tends to degrade as new increments are added .
 - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

Rapid Application Development [RAD]

RAD Model

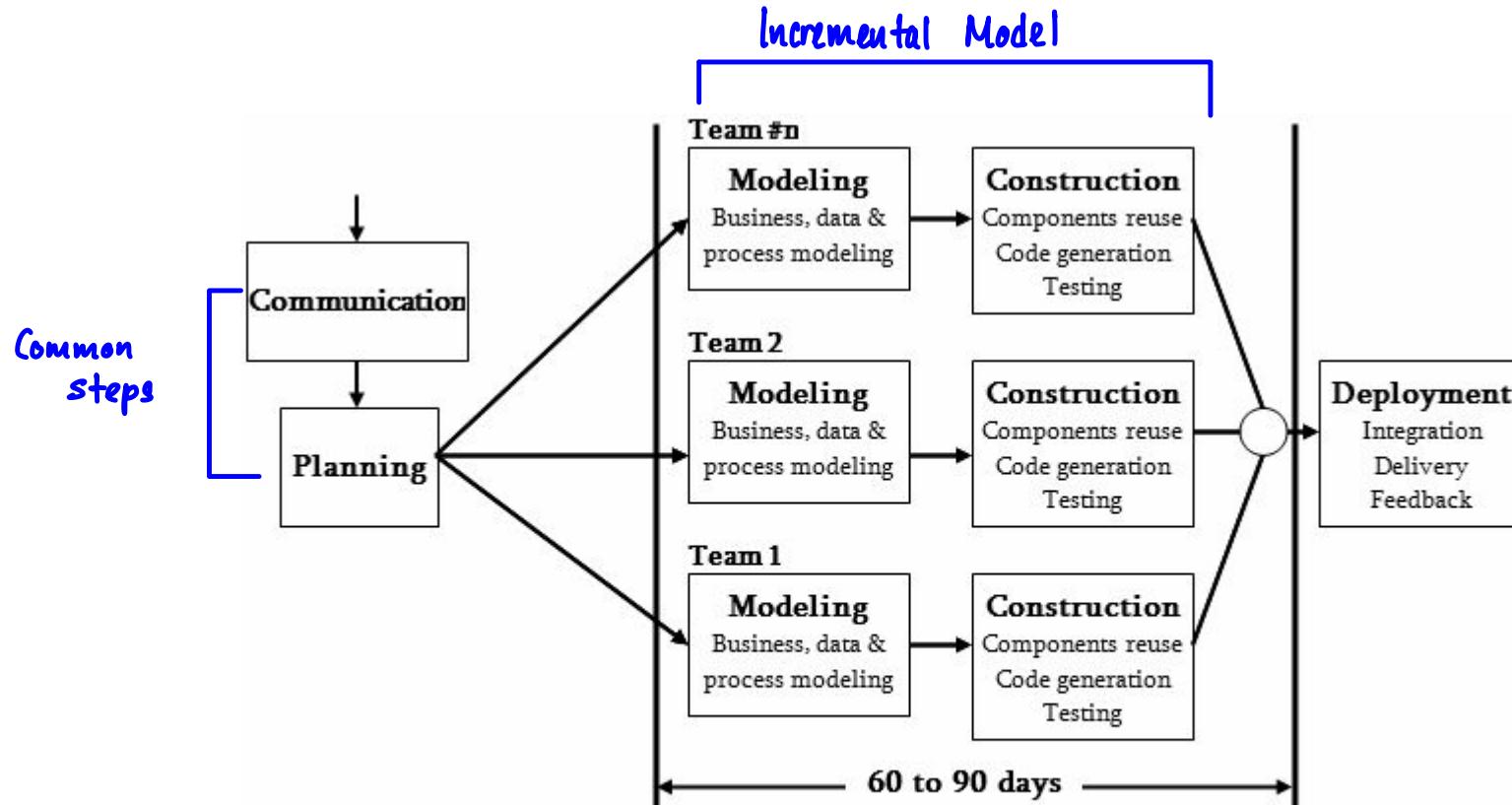


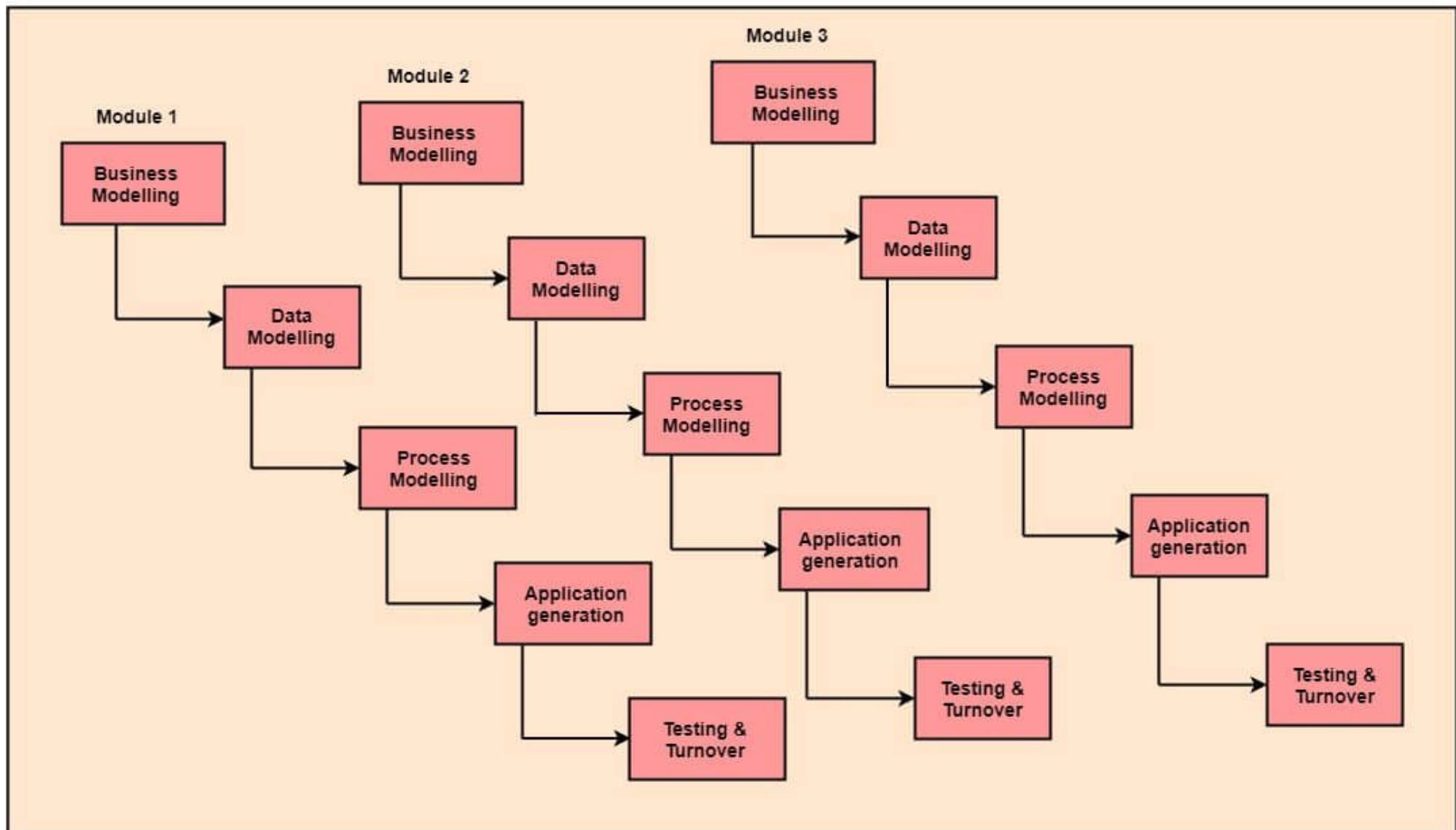
Figure : Flowchart of RAD model

Rapid Application Development

Rapid Application Development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle.

If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a 'fully functional system' within very short time periods (eg. 60 to 90 days)

Fig: RAD Model



Processes in the RAD model

Business modeling

The information flow in a business system considering its functionality.

Data Modeling

The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business

Process Modeling

The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement business function.

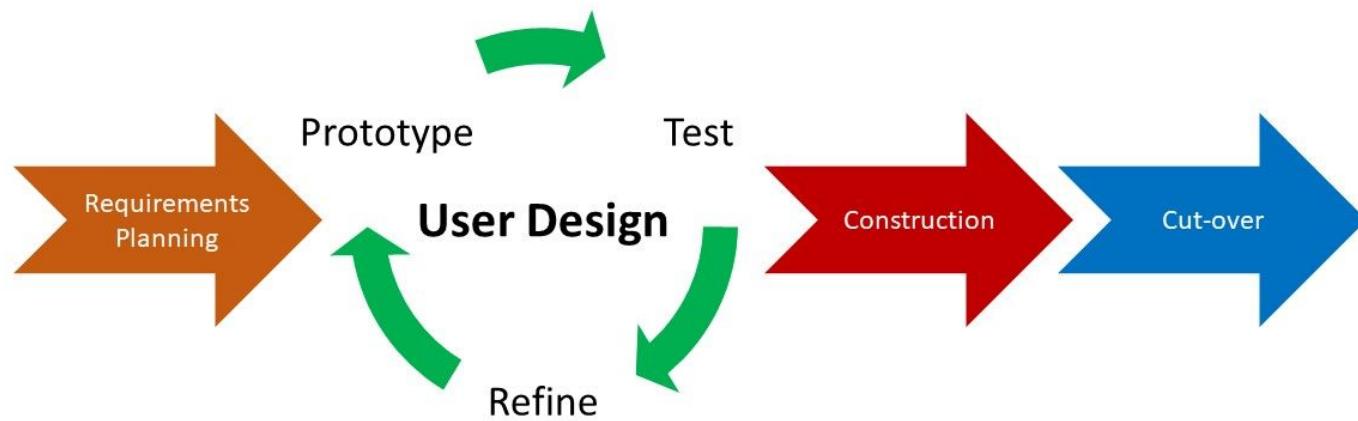
Application generation

RAD assumes the use of 4GL or visual tools to generate the system using reusable components.

Testing and turnover

New components must be tested and all interfaces must be fully exercised

Rapid Application Development



- Iterative development
- Construction of prototypes
- Use of Computer-aided software engineering (CASE) tools.

Advantages

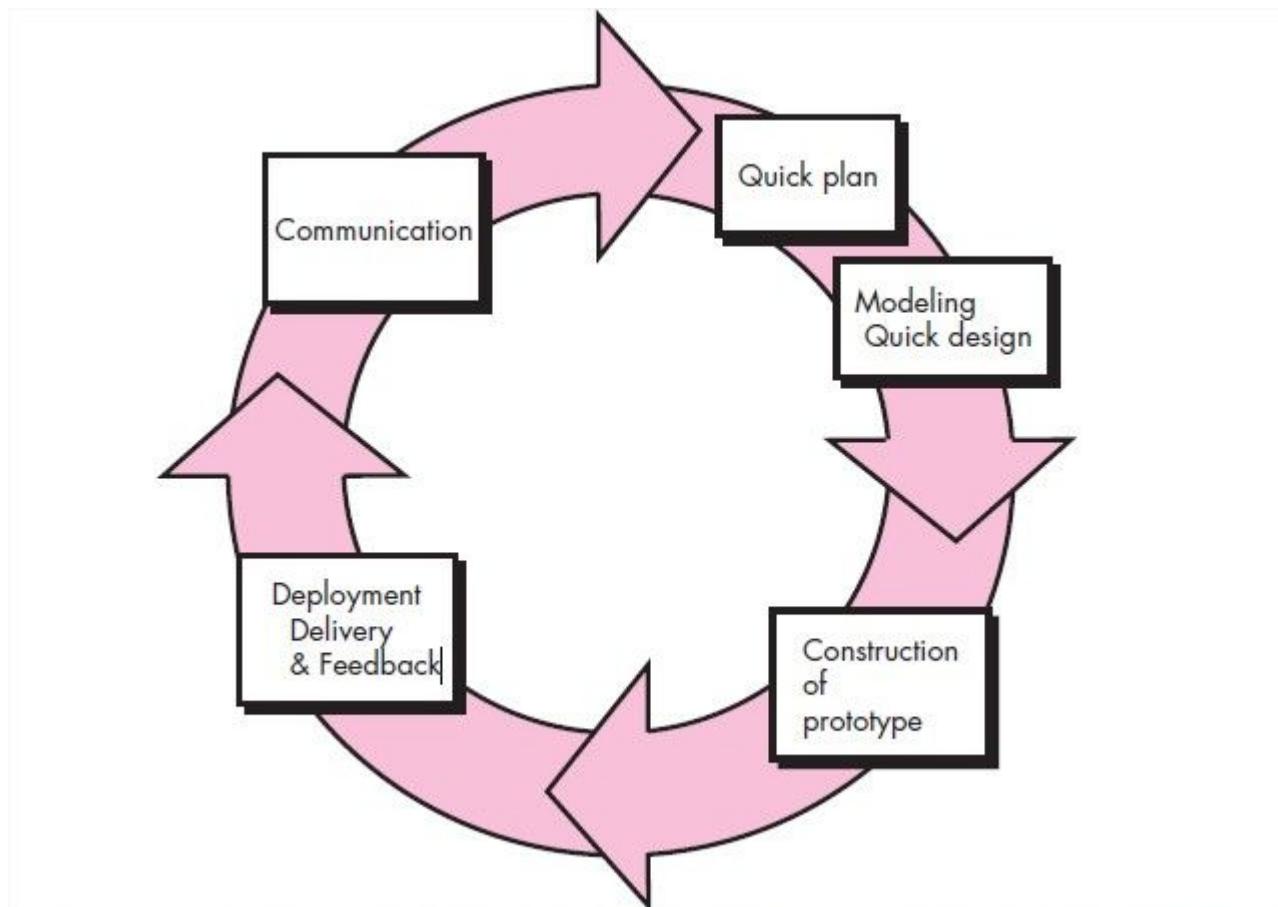
- Can be quickly developed
- Increased user involvement
- Cost reduces
- Due to code generators and code reuse, there is a reduction of manual coding
- Due to prototyping in nature, there is a possibility of lesser defects

Problems

- Depends on strong team and individual performances for identifying business requirements.
- Only system that can be modularized can be built using RAD
- Requires highly skilled developers/designers.
- High dependency on modeling skills
- Inapplicable to cheaper projects as cost of modeling and automated code generation is very high
- Not good for the technical risk high project

Evolutionary Model

Evolutionary Models



Evolutionary Models



Software Prototyping

Software Prototyping

- A prototype is a working model that is functionally equivalent to a component of the product
- The prototyping model is a systems development method in which a prototype is built, tested and then reworked as necessary until an acceptable outcome is achieved from which the complete system or product can be developed.
- This model works best in scenarios where not all of the project requirements are known in detail ahead of time.
- It is an iterative, trial and error process that takes place between the developers and the users.



Software Prototyping

A prototype can be used in:

- The requirements engineering process to help with requirements elicitation and validation;
- The design process to explore options and develop a UI design;
- In the testing process to run back-to-back tests.

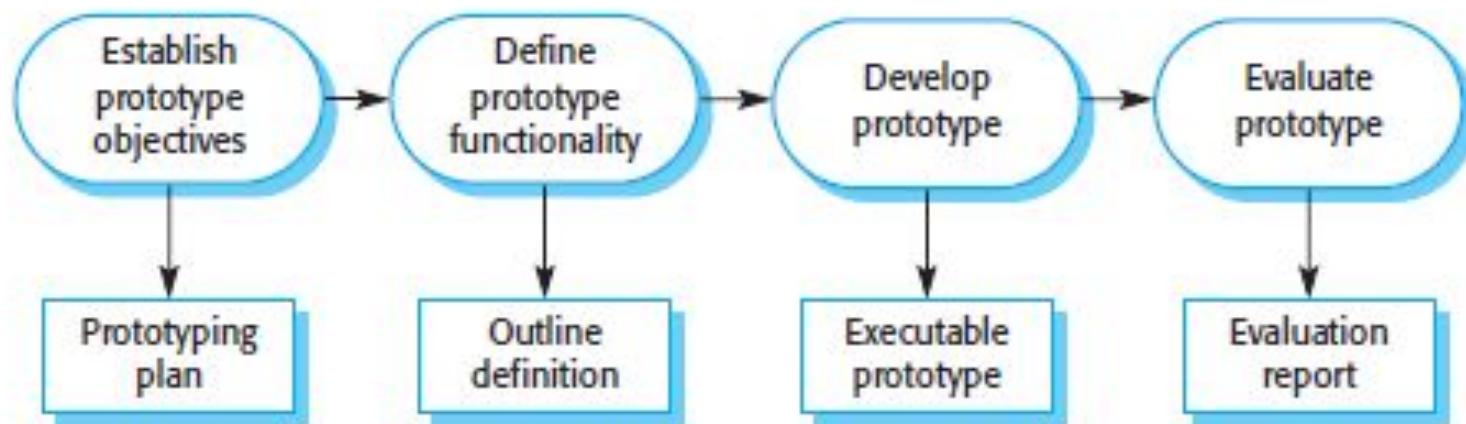
Benefits of prototyping

- Improved system usability.
- A closer match to users' real needs.
- Improved design quality.
- Improved maintainability.
- Reduced development effort.

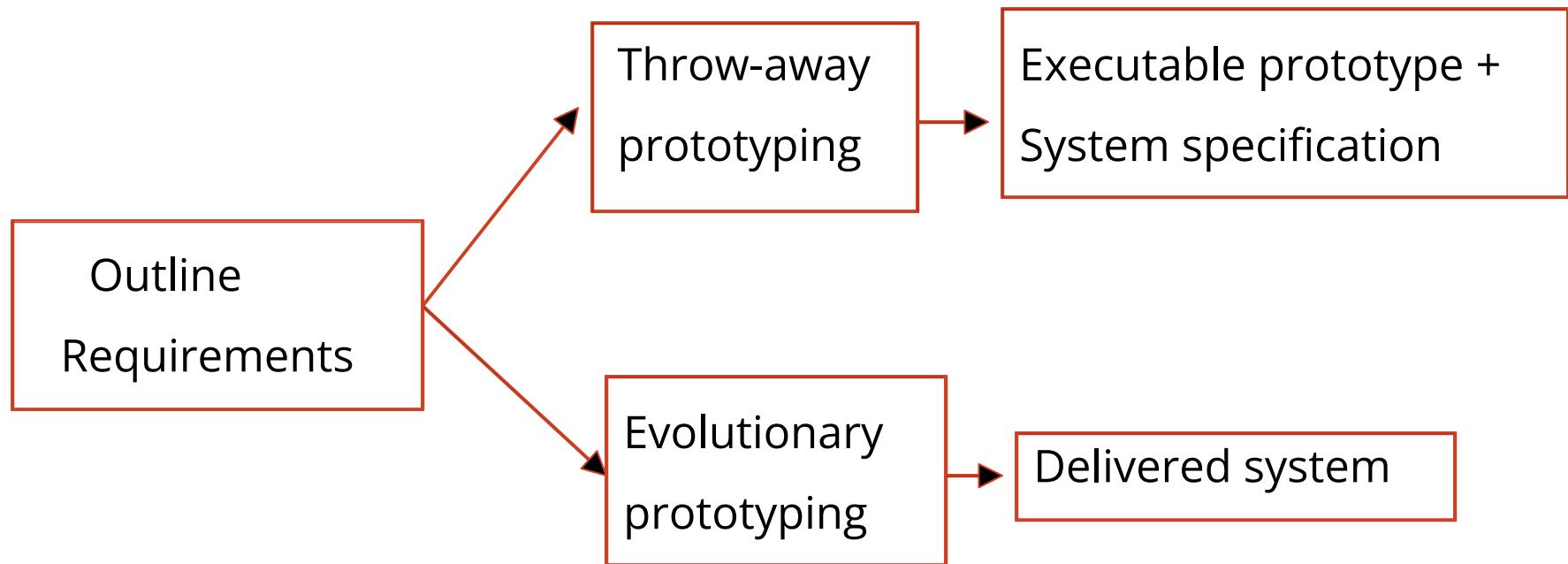
Problems with prototyping

- This model is costly.
- It has poor documentation because of continuously changing customer requirements.
- Effort in building a prototype may be wasted
- Customers may not be satisfied or interested in the product after seeing the initial prototype.

The process of prototype development



Throw-away and Evolutionary Prototyping



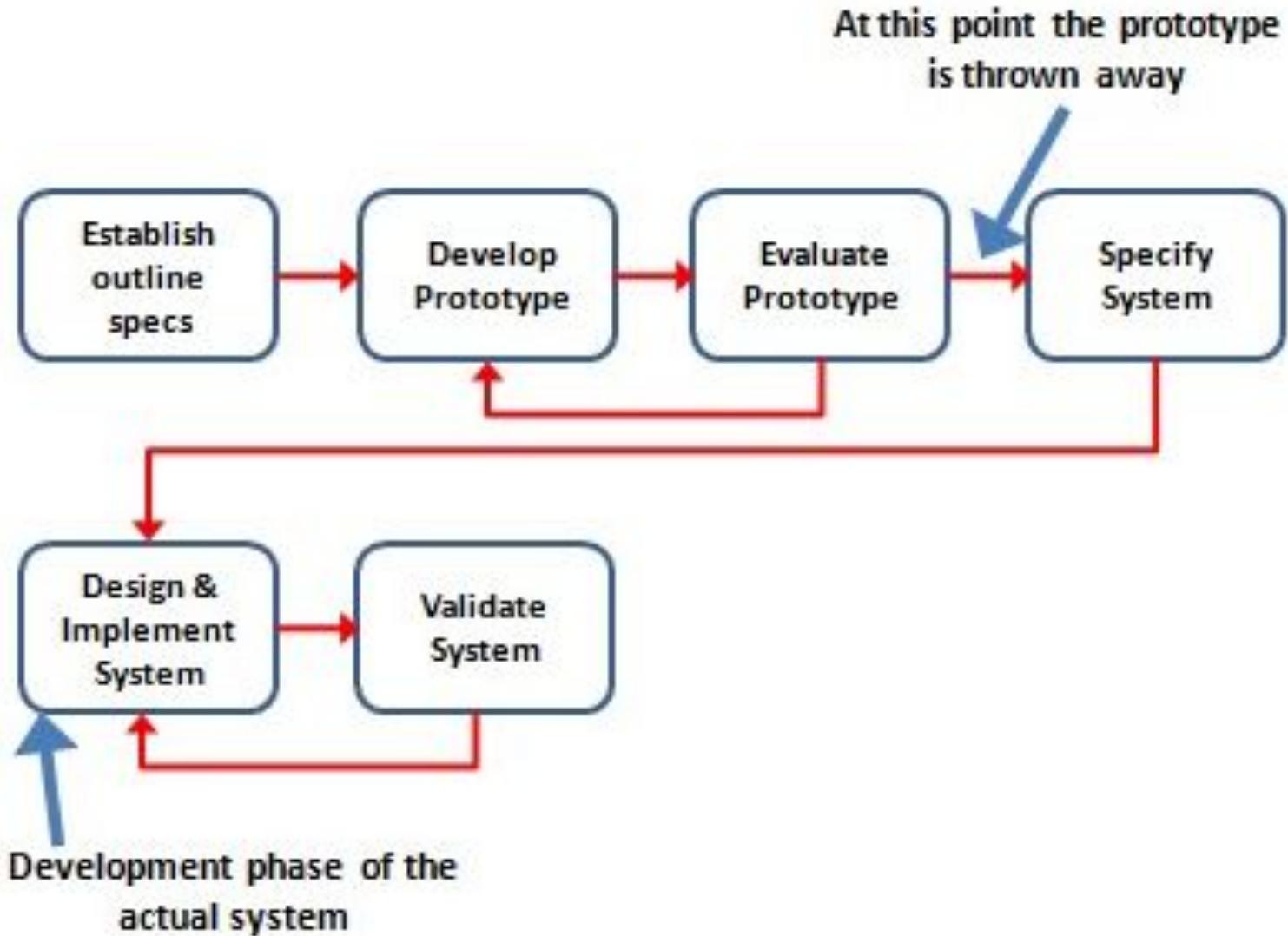
Throw-away Prototyping

The objective is to understand the system requirements clearly.

Starts with poorly understood requirements. Once the requirements are cleared, the system will be developed from the beginning.

This model is suitable if the requirements are vague but stable.

Throw-away Prototyping

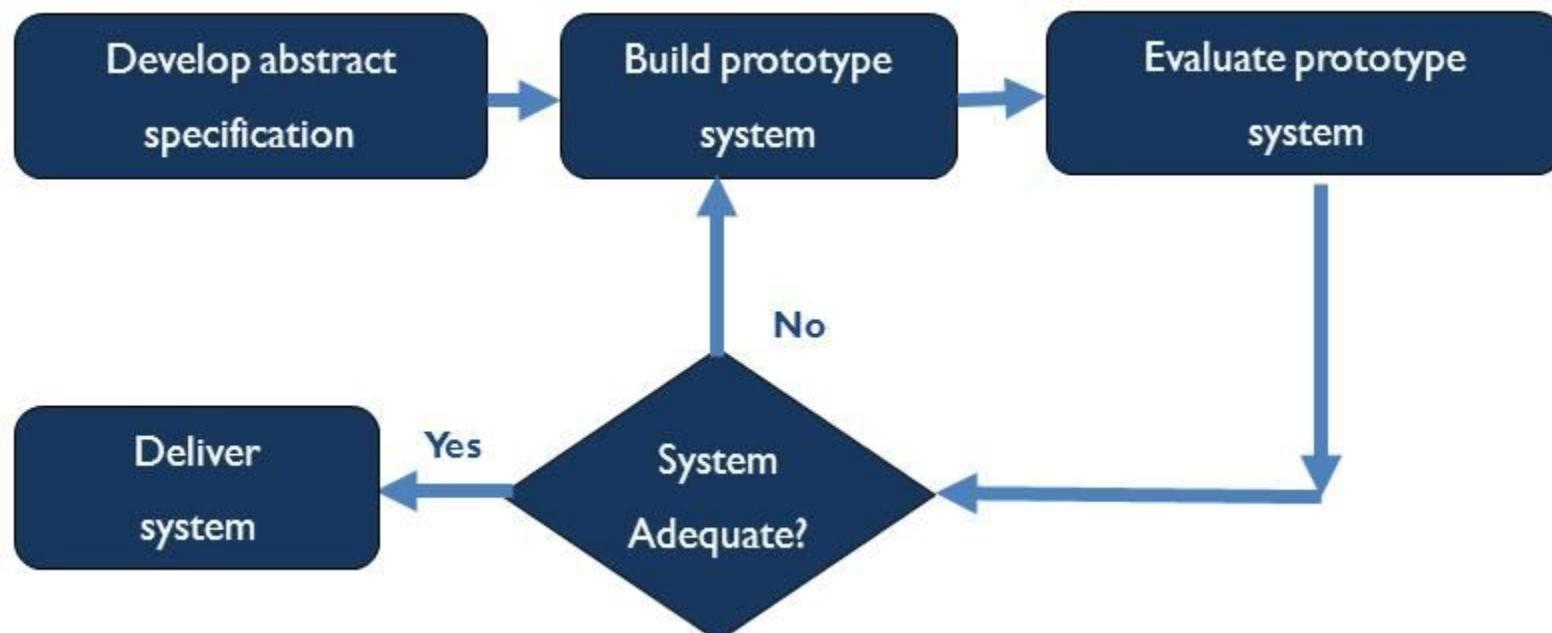


Problems with Throw-away Prototyping

- Important features may have been left out of the prototype to simplify rapid implementation. In fact, it may not be possible to prototype some of the most important parts of the system such as safety-critical functions.
- An implementation has no legal standing as a contract between customer and contractor.
- Non-functional requirements such as those concerning reliability, robustness and safety cannot be adequately tested in a prototype implementation.

Evolutionary Prototyping

Develop an initial implementation, exposing this to user comment and refining through many versions until an adequate system has been developed



Find Disadvantages

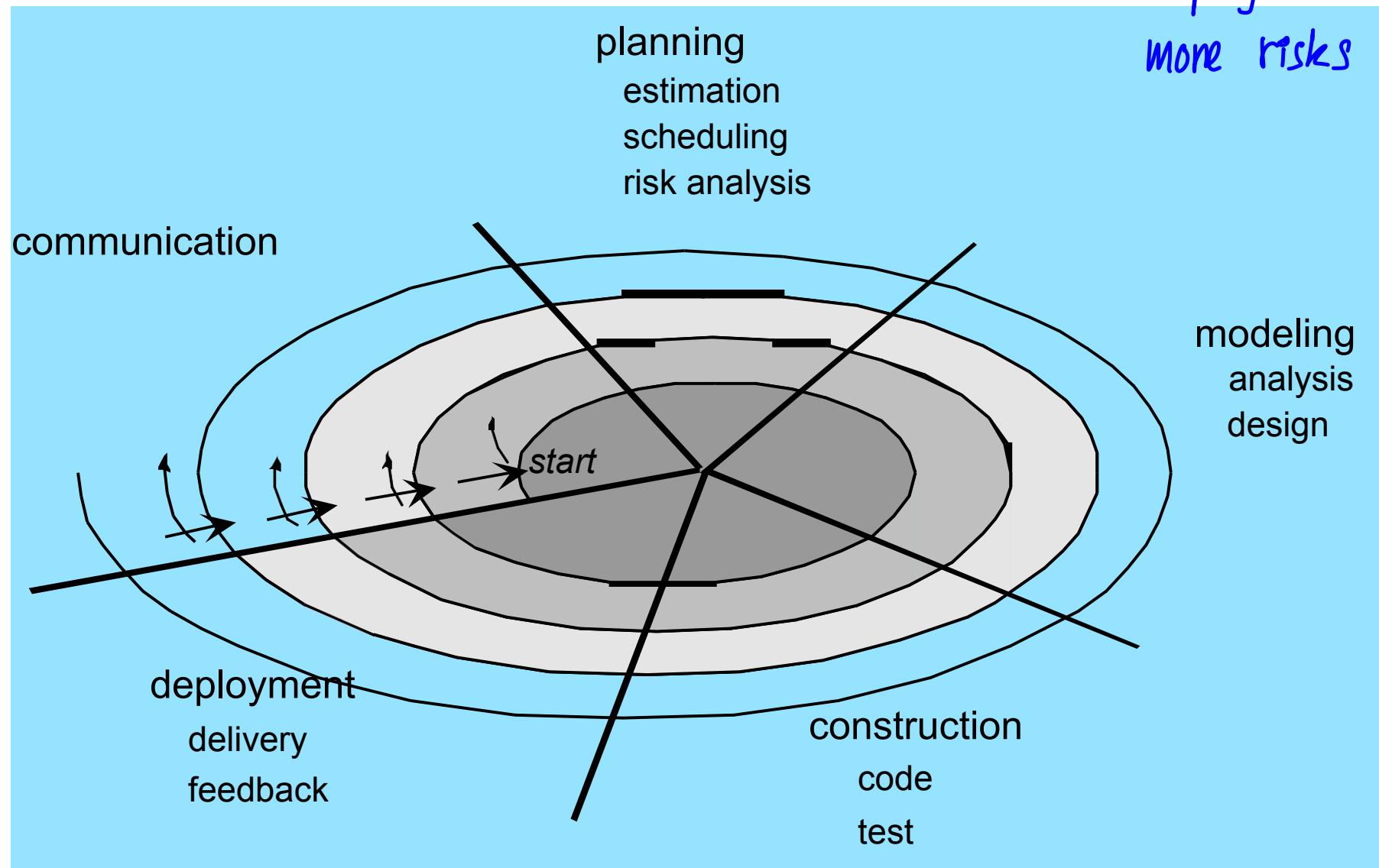
Advantages of Evolutionary Prototyping

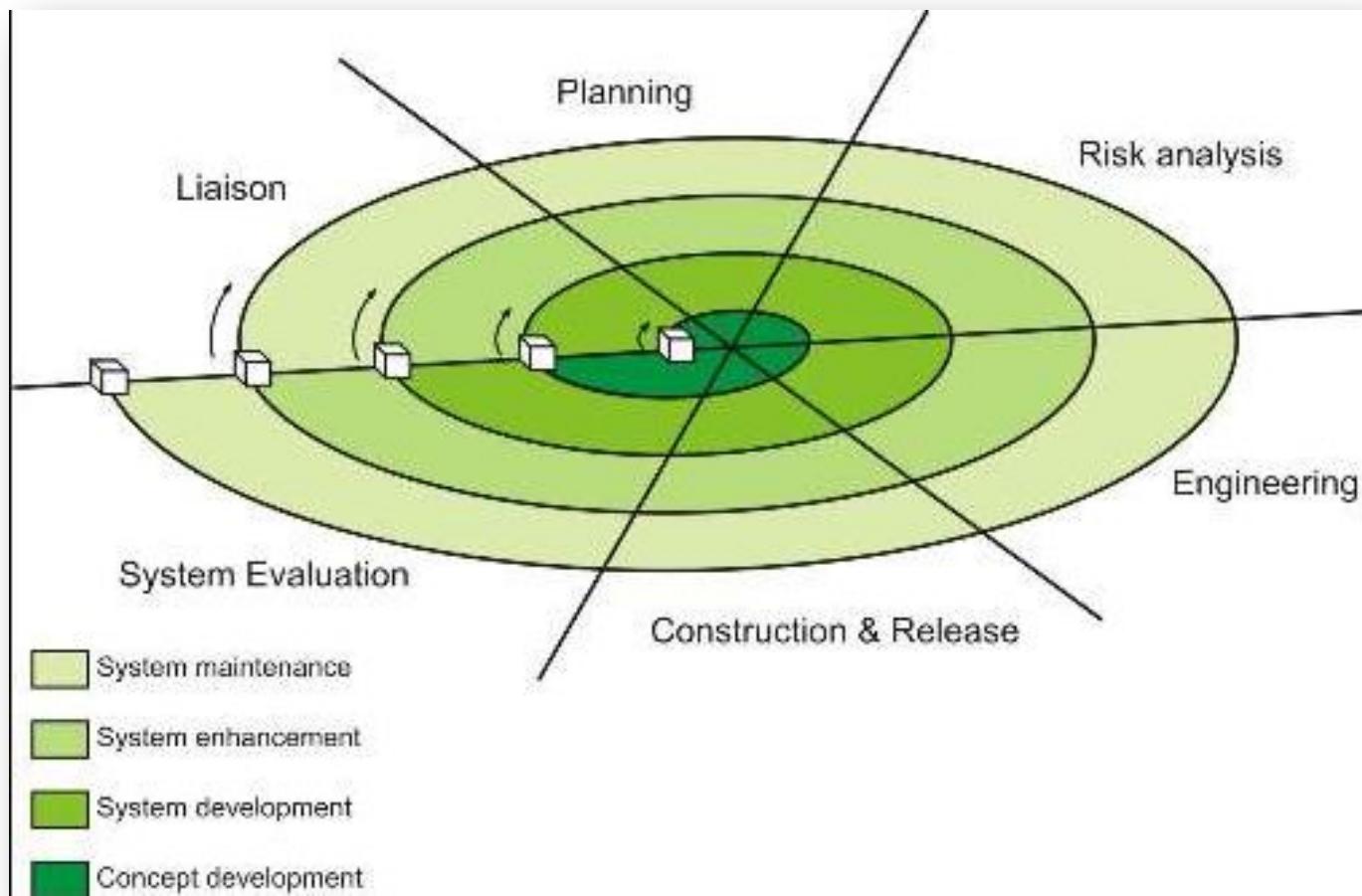
- Effort of prototype is not wasted
- Faster than the Waterfall model
- High level of user involvement from the start
- Technical or other problems discovered early – risk reduced
- mainly suitable for projects with vague and unstable requirements.

Spiral Model

Evolutionary Models: The Spiral

If the project has
more risks





Spiral development

Process is represented as a spiral rather than as a sequence of activities with backtracking.

Each loop in the spiral represents a phase in the process.

No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.

Risks are explicitly assessed and resolved throughout the process.

Other Process Models

Component based development—the process to apply when reuse is a development objective

Formal methods—emphasizes the mathematical specification of requirements

AOSD—provides a process and methodological approach for defining, specifying, designing, and constructing aspects

Unified Process—a “use-case driven, architecture-centric, iterative and incremental” software process closely aligned with the Unified Modeling Lan²g⁵ uage (UML)

Unified Process Model

Unified Process Model

What is this? “*Use-case Driven, Architecture-Centric, Iterative and Incremental Software Development Process Model*”

- Use-case Driven?
 - Capture functional requirements
 - Define the content on iterations
 - Recognize Customer View and Customer Communication
- Architecture Centric?
 - Heart of the overall system
 - Many architectural views/models
 - Executable architecture (foundation for remaining development)
 - most important deliverable, elaboration phase
- Process: Iterative and Incremental

UP – Risk focused

- The Unified Process requires the project team to focus on addressing the most *critical risks early in the project life cycle.*
- The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that *the greatest risks are addressed first.*

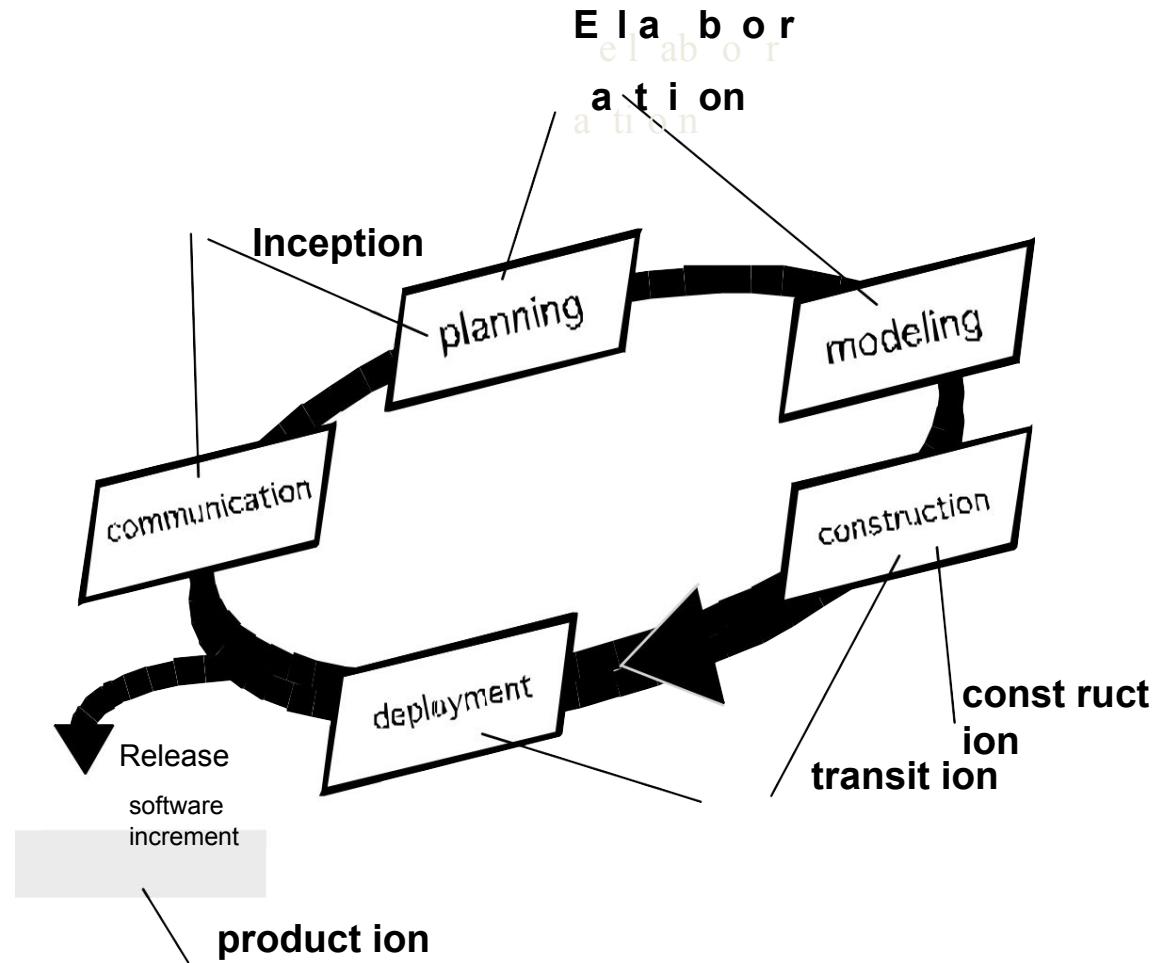
What is Special about it?

- UP identifies the best features and characteristics of conventional software process models and integrate them with the best principles/practices of agile software development

Unified Process

- Why we call Unified Process hybrid process model?
(it is iterative and incremental model, two dimensional view)
- How many phases in UP?
 - 4 phases
 - Inception, elaboration, construction and transitions
- How many abstract disciplines/workflows?
 - 6 disciplines
 - Business Modeling, Requirements, Analysis and design, implementation, testing and deployment

The Unified Process (UP)



Inception Phase

- Encompasses customer communication and planning activities
 - Identify business requirements
 - Initial Use Cases (functions, features, actors, sequence of actions, basis for project plan)
 - Initial (Rough) Architecture of the System
 - Outline of Major sub-systems, functions, features
 - Refine to model different views of the system
 - Plan for number of incremental and iterations
 - Identify resources, risks, schedule, basis for phases



Elaboration Phase

- Refine initial use-cases to establish 5 models
 - Use-case Model
 - Analysis Model
 - Design Model
 - Implementation Model
 - Deployment Model
 - Milestones
- Create “Executable Architectural Baseline”
 - Justify viability of system irrespective of all functions and features

Elaboration Phase

- Deliverable
 - Plan for the construction phase (accurate and credible) address risk factors
- Review the plan
 - Scope
 - Risks
 - Delivery Dates

If necessary, modify the plan as necessary

Construction Phase

- Make each use case operation system (Increment)
 - All increments started in elaboration phase to be completed
 - Develop test cases for Acceptance test
 - Unit Testing for each Increment as and when completed
 - Integration activities (component assembly and Integration Testing)

Transition Phase

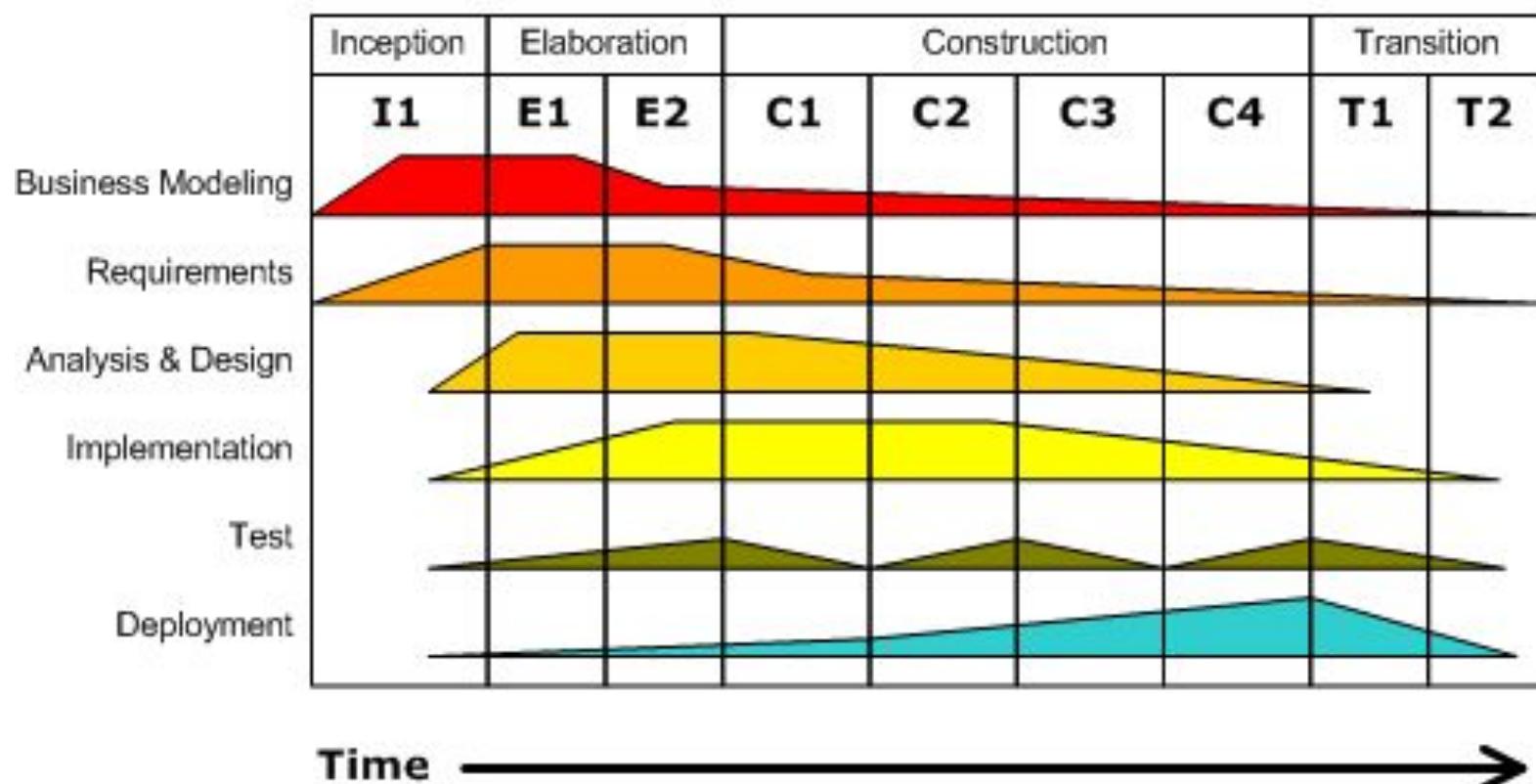
- Beta Testing (s/w is given to users)
- Gather users feedback (errors and changes required)
- Develop supporting materials
 - User manuals, trouble-shooting guidelines, installation procedures etc.
- At the end, release should be a usable product

Workflows (process areas)

- What are they?
- Distributed over all UP phases
- A workflow includes several tasks
- A result of tasks are “Work Products”

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.



UP Work Products

Inception phase

Vision document
Initial use-case model
Initial project glossary
Initial business case
Initial risk assessment.
Project plan,
phases and iterations.
Business model,
if necessary.
One or more prototypes

Elaboration phase

Use-case model
Supplementary requirements
including non-functional
Analysis model
Software architecture
Description.
Executable architectural
prototype.
Preliminary design model
Revised risk list
Project plan including
iteration plan
adapted workflows
milestones
technical work products
Preliminary user manual

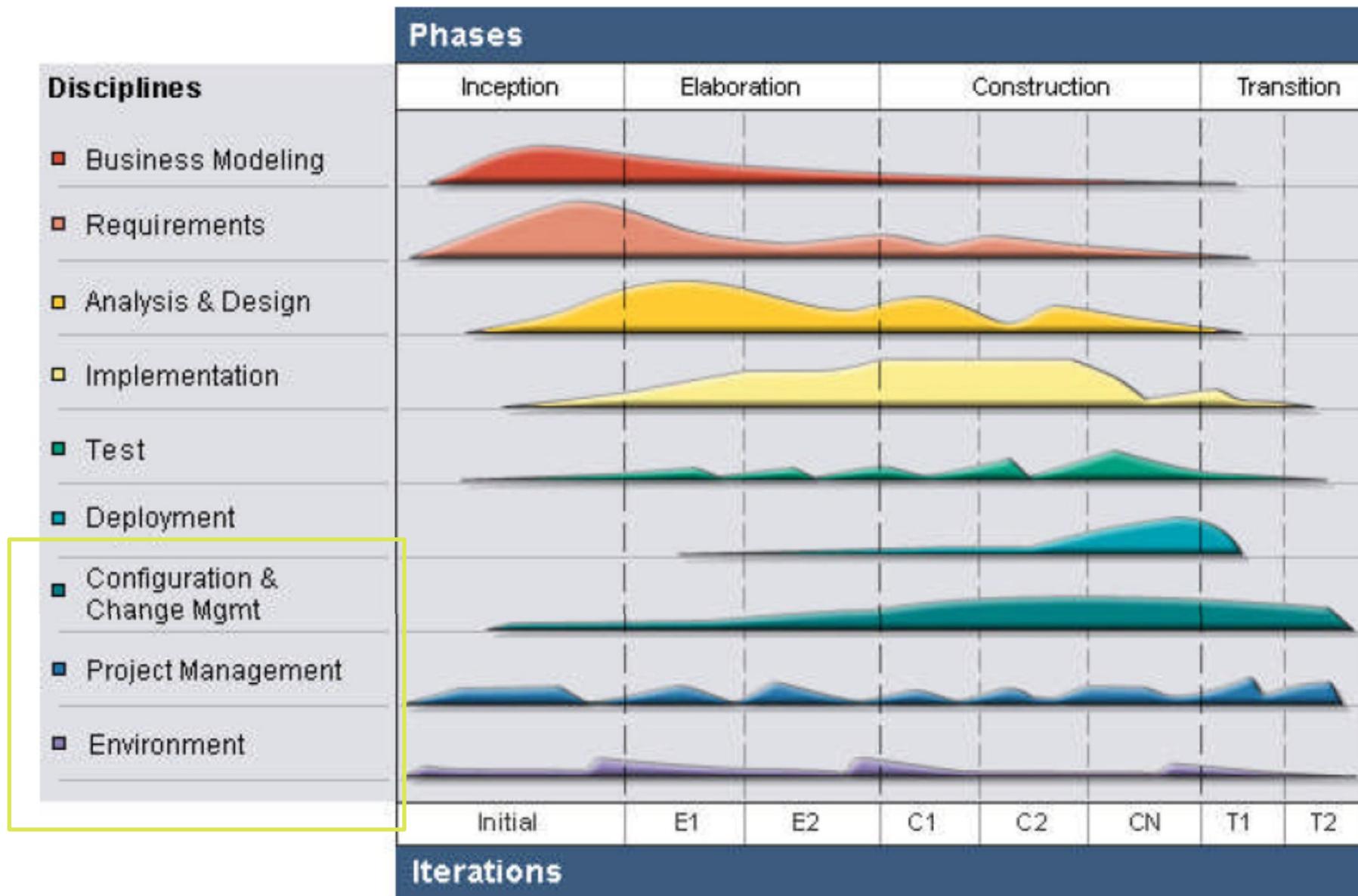
Construction phase

Design model
Software components
Integrated software
increment
Test plan and procedure
Test cases
Support documentation
user manuals
installation manuals
description of current
increment

Transition phase

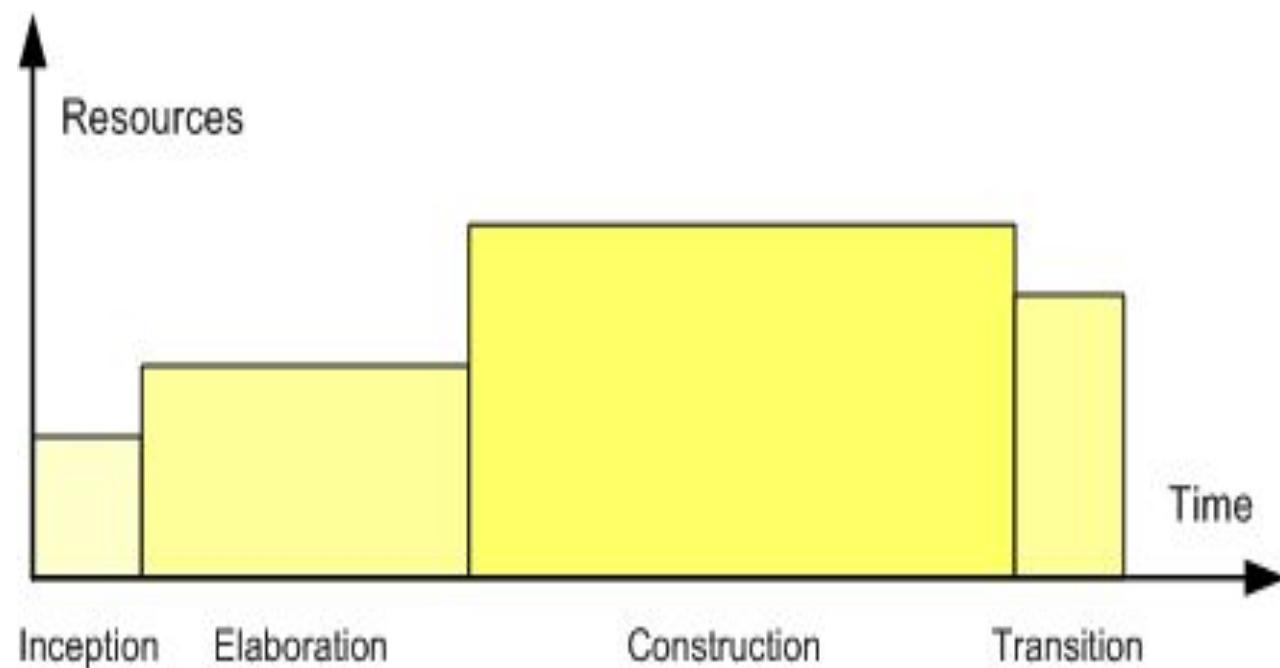
Delivered software increment
Beta test reports
General user feedback

RUP (Rational Unified Process)



UP – Project Life Cycle

- Resources required in the cycle



Different refinements of UP

- Agile Unified Process (AUP), a lightweight variation developed by Scott W. Ambler
- Basic Unified Process (BUP), a lightweight variation developed by IBM and a precursor to Open UP
- Enterprise Unified Process (EUP), an extension of the Rational Unified Process
- Open Unified Process (OpenUP), the Eclipse Process Framework software development process
- Rational Unified Process (RUP), the IBM / Rational Software development process

Open UP

- Refinement of UP
- Popular process model
- Freely available on Eclipses
- Several reason for failures in software project

How to handle it?

Open UP

- Strong Process Development Framework
 - iterative, incremental, risk focused, use-case driven, architecture centric
 - Complete and Extensible Architecture across different application domains and organizational cultures
- Good practices
 - Individual focus
 - Team focus
- Collaboration and Cooperation

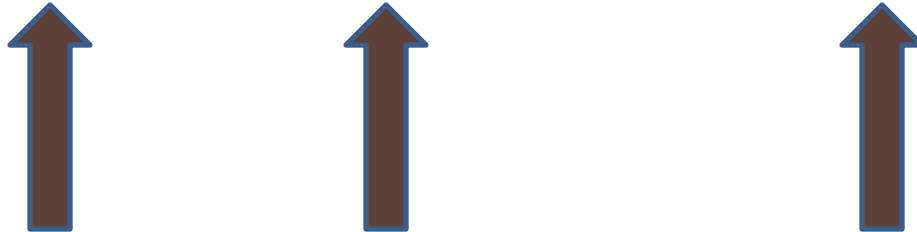
Unified + Agile + Community

Open UP

"An Agile Inspired process with its roots in the UP"

Support Collaborative Software Development

An iterative software development process that is minimal , complete , and extensible



Open UP

OpenUP incorporates a three-tiered governance model to plan, execute, and monitor progress.

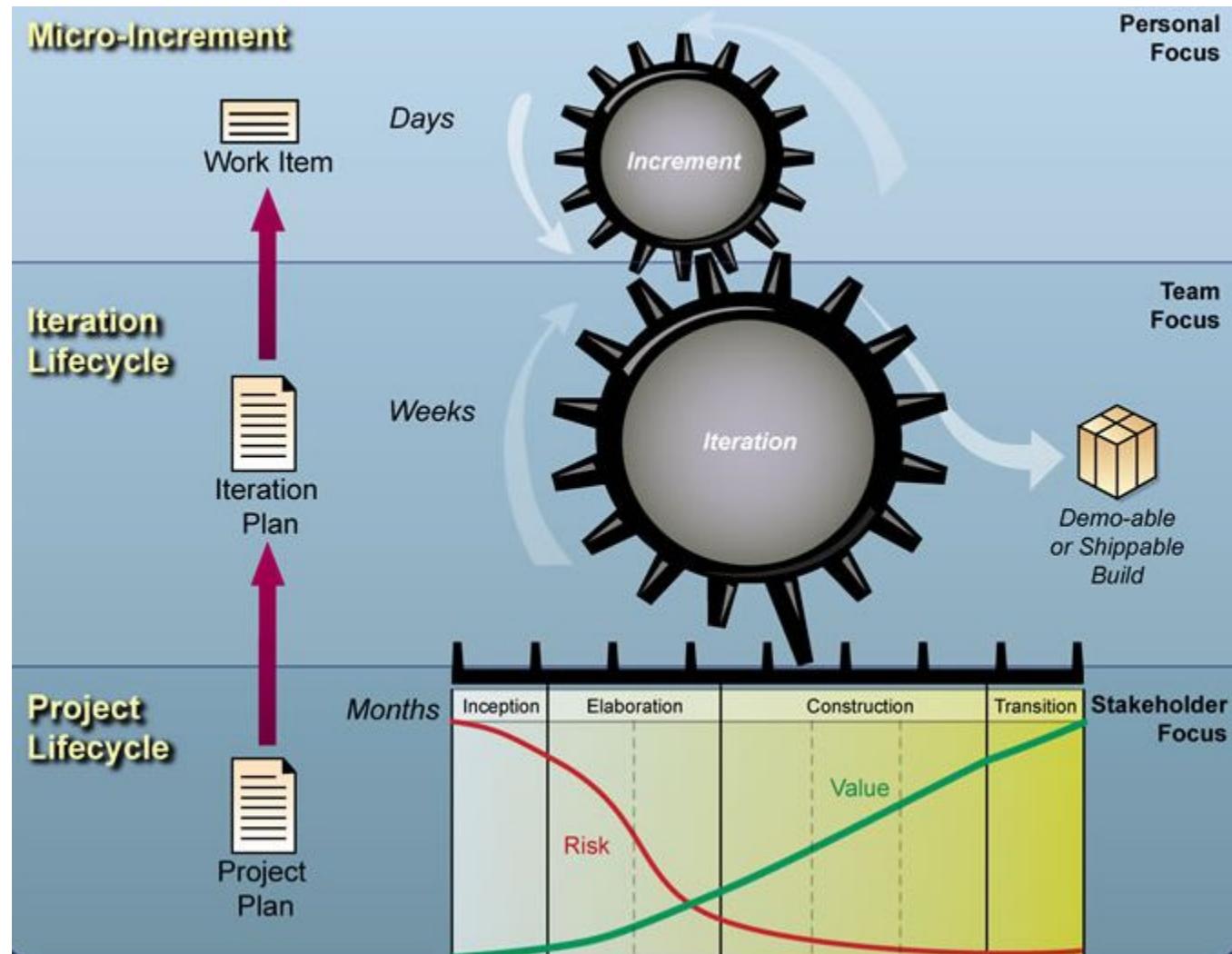
It concerns on,

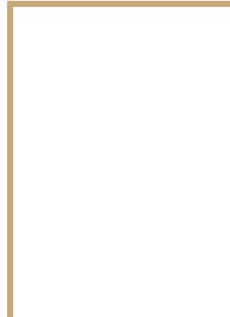
Personal focus – individuals works on daily(micro increments)

Team focus – combine micro increments, takes weeks

Stakeholders focus – project life cycle (UP)

Open UP





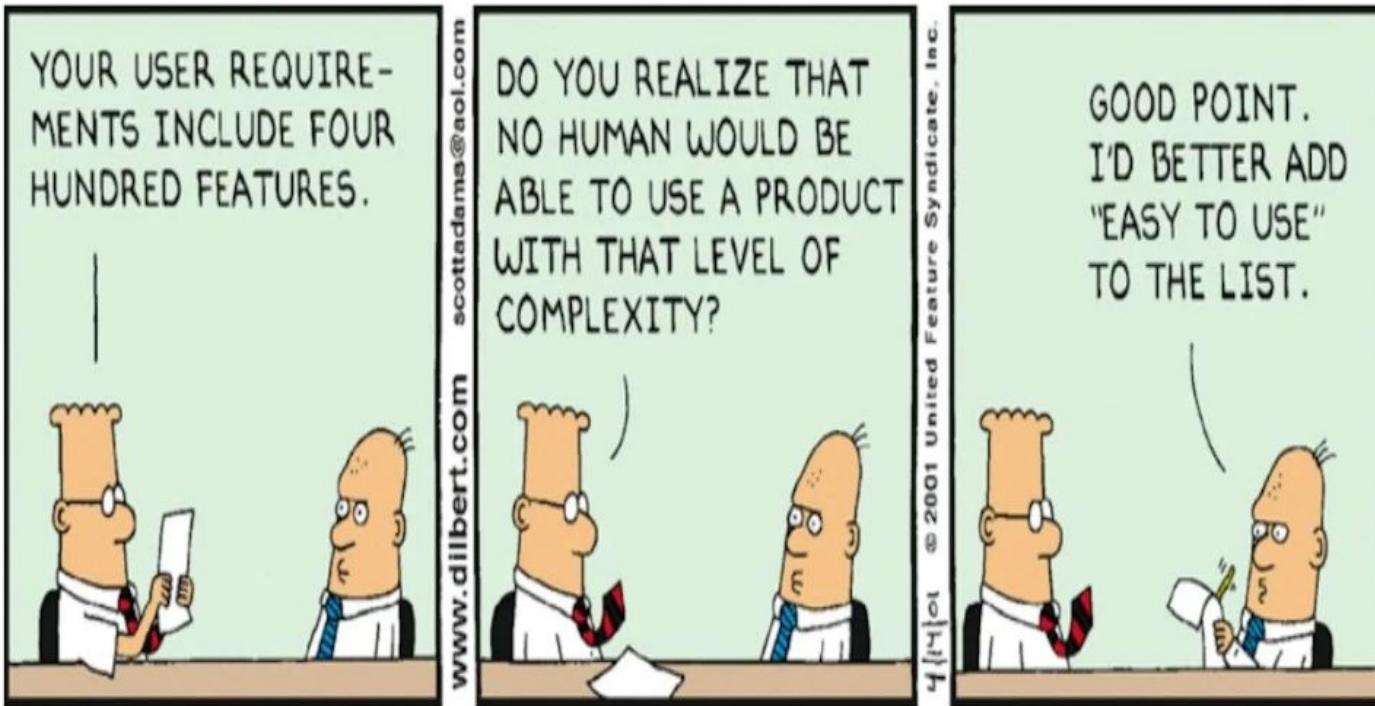
Requirements Engineering



Topics covered

- ✓ What is a Requirement?
- ✓ Functional and Non-functional Requirements
- ✓ Requirements Engineering Processes
- ✓ Requirements Elicitation and Analysis
- ✓ Requirements Specification
- ✓ Requirements Validation
- ✓ Requirements Change

What is a requirement?



What is a requirement?

- A **need** to be satisfied
- A requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system.
- At the other extreme, it is a detailed, formal definition of a system function

Requirement Engineering

- The requirements for a system are the **descriptions** of the **services** that a system should provide and the **constraints** on its operation.
- These requirements reflect the **needs of customers** for a system that serves a certain purpose such as controlling a device, placing an order, or finding information.
- The process of **finding out, analyzing, documenting and checking** these services and **constraints** is called **Requirements engineering (RE)**.

Representation of Requirement

- ❖ User requirements
 - Statements in natural language plus diagrams of the **services the system provides and its operational constraints**. Written for customers.
- ❖ System requirements
 - A structured document setting out **detailed descriptions of the system's functions, services and operational constraints**. Defines what should be implemented so may be part of a contract between client and contractor.
- ❖ User requirement can be broken into several system requirements

User requirements definition

1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

- 1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
- 1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
- 1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
- 1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc.) separate reports shall be created for each dose unit.
- 1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

User Requirements

- ◆ Written for customers.
- ◆ Statements in natural language.
- ◆ Describe the services the system provides and its operational constraints.
- ◆ May include diagrams or tables.
- ◆ Should describe functional and non-functional requirements.
- ◆ Should be understandable by system users who don't have detailed technical knowledge.

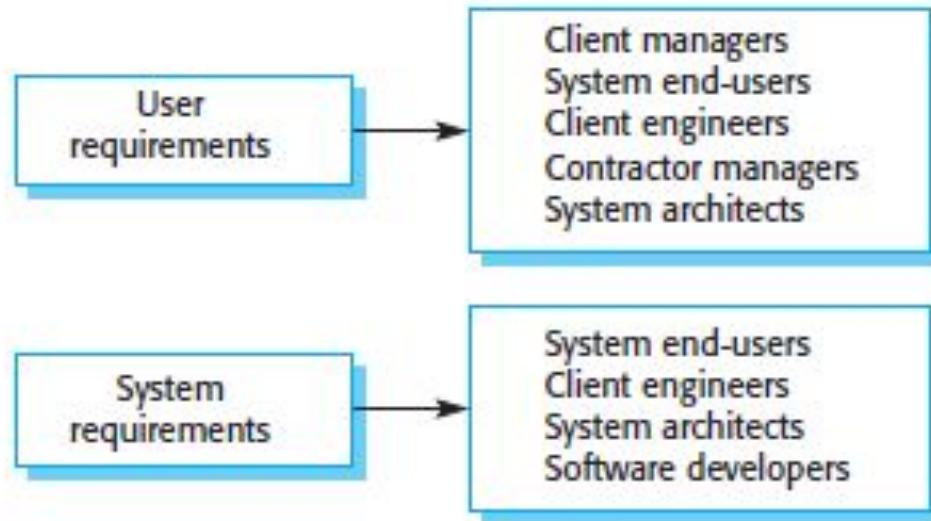
We provide a **definition** for a user requirement.

System Requirements

- ◆ Statements that set out detailed descriptions of the system's functions, services and operational constraints.
- ◆ Defines what should be implemented so may be part of a contract between client and contractor.
- ◆ Intended to be a basis for designing the system.
- ◆ Can be illustrated using system models.

We provide a **specification** for a system requirement.

Readers of different types of requirements specification



Types of requirement

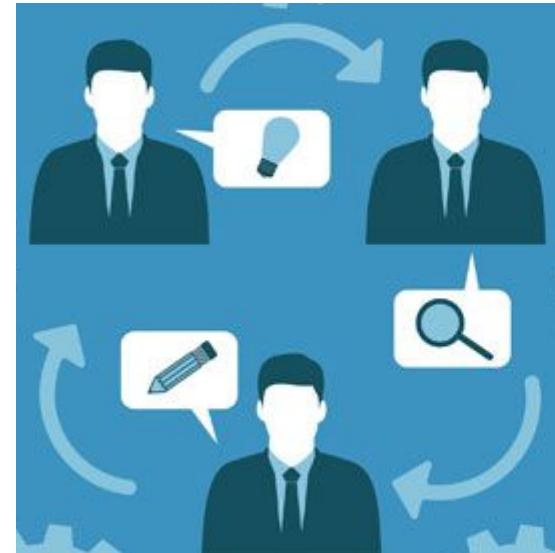
- ❖ **Functional Requirements** – typically focus on the “**what**” of the system and identify the functions and data related requirements.
- ❖ **Non-functional Requirements** – focus on the “**how well**” aspects of the system and identify attributes like
 - Performance
 - Maintainability
 - Interoperability
 - Reliability
 - Portability
 - Constraintson which the system needs to operate.

Types of requirement

- ✧ Domain requirements
 - Restrictions on the system from the domain of operation

Project stakeholders

- ❖ A **project stakeholder** is any **individual** or an **organization** that is **actively involved in a project**, or whose interest might be affected (positively or negatively) as a result of project execution or completion.



Types of Stakeholders

- The project manager
- The project team
- The project sponsor
- The performing organizations
- The partners
- The client
- The rest: anyone who might be affected by the project outputs



Question 1

❖ Who are the stakeholders in the Mentcare system, a the mental health care patient information system ?

-  Patients
-  Doctors
-  Nurses
-  Medical receptionists
-  IT staff
-  Managers

Question 2

Airline Reservation System

- Write down all the Stakeholders in this system
- Identify main subsystems of the proposed system
- Write down functional requirements and explain
- Write down non-functional requirements and explain

Functional and non-functional requirements

Functional and non-functional requirements

- ❖ Functional requirements (**What the system should do?**)
 - Statements of **services** the system **should provide**, how the system should **react** to particular **inputs** and how the system should behave in particular situations.
 - May state what the system should not do.
- ❖ Non-functional requirements (**How the system performs a certain function?**)
 - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
 - Often apply to the system as a whole rather than individual features or services.

Functional requirements

- ❖ Describe **functionality** or system services.
- ❖ Depend on the **type of software**, expected **users** and the type of system **where the software is used**.
- ❖ Functional user requirements may be high-level statements of what the system should do.
- ❖ Functional system requirements should describe the system services in detail.
- ❖ Functional requirements should be **written in natural language** so that system users and managers can understand them.

Functional requirements : Example

- ⑤ Registered users shall be able to login with valid username and password
- ⑤ On successful login, user shall be redirected to a landing page in the system
- ⑤ On failure, for not registered username prompt "Username not registered" message and for invalid credentials prompt "Invalid credentials"
- ⑤ New users shall be able to register with the system by clicking on the "Sign Up" link
- ⑤ Users shall be able to recover password by clicking on "Forgot Password" link

Requirements imprecision

- ❖ Problems arise when functional requirements are not precisely stated.
 - Ambiguous requirements may be interpreted in different ways by developers and users.
- ❖ Consider the term '**search**' in requirement 1
 - User intention – search for a patient name across all appointments in all clinics;
 - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

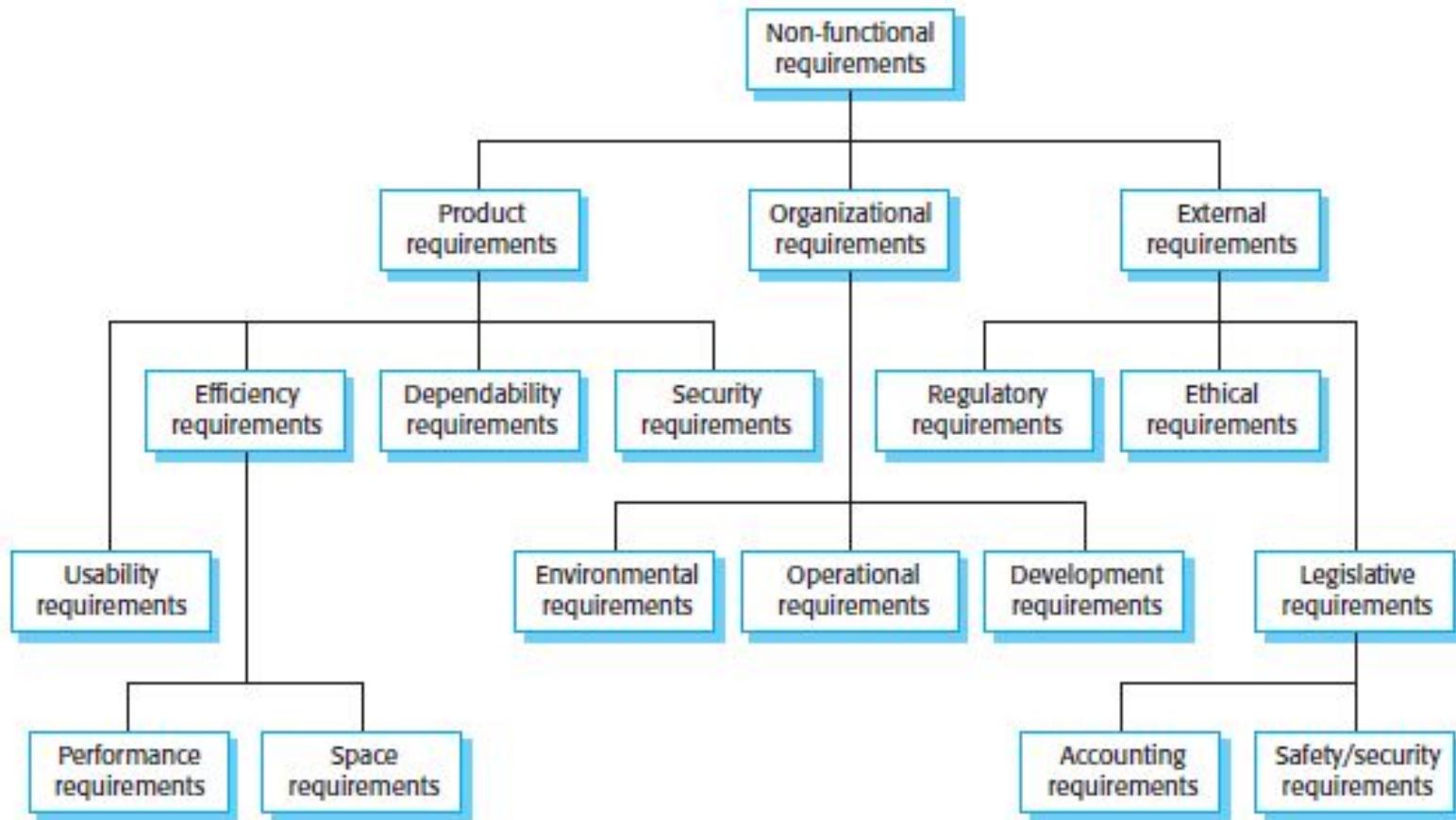
Non-functional requirements

- Non-functional requirements are **requirements that are not directly concerned with the specific services delivered by the system to its users.**
- Non-functional requirements specify the system's **quality characteristics** or '**quality attributes**'.
- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- May be more critical than functional requirements

Non-functional requirements implementation

- ❖ Can identify the missing requirements
- ❖ Non-functional requirements may affect the overall architecture of a system rather than the individual components.
 - For example, to ensure that **performance** requirements are met, you may have to organize the system to **minimize communications between components**.

Types of nonfunctional requirement



General non-functional classifications

- ❖ Product requirements – concern product behaviour.
- ❖ Organizational requirements – derived from policies / procedures in customer's or developer's organization (e.g., process constraints).
- ❖ External requirements – derived from factors external to the product and its development process (e.g., interoperability).

Metrics for specifying non functional requirements

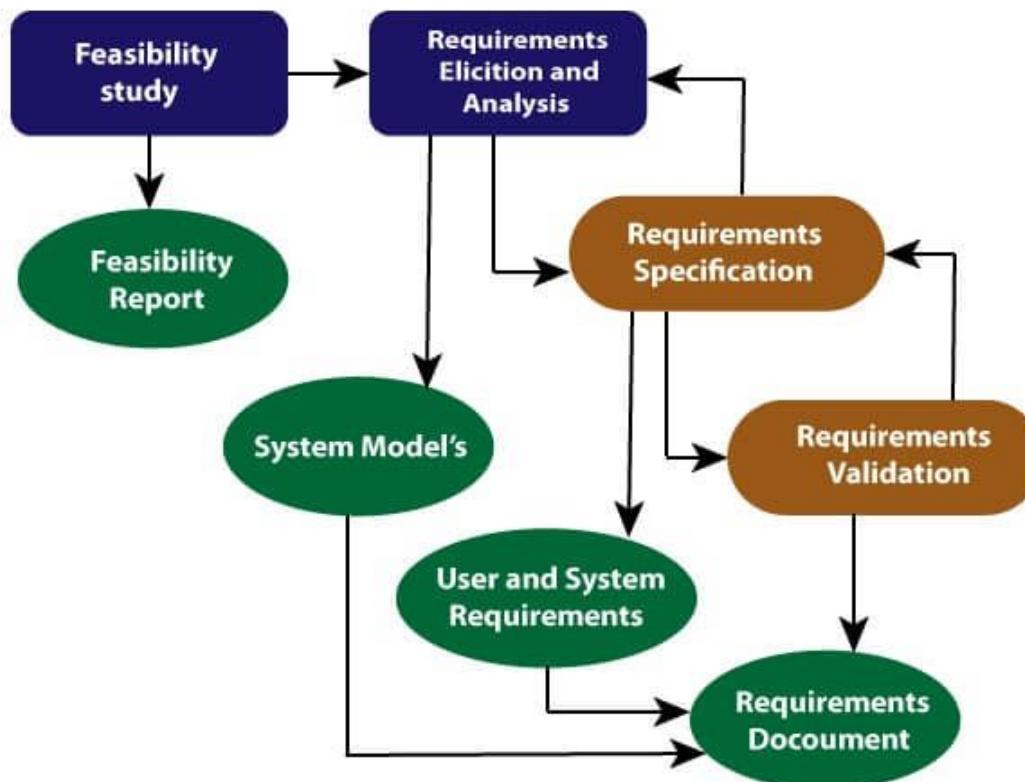
Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Requirements completeness and consistency

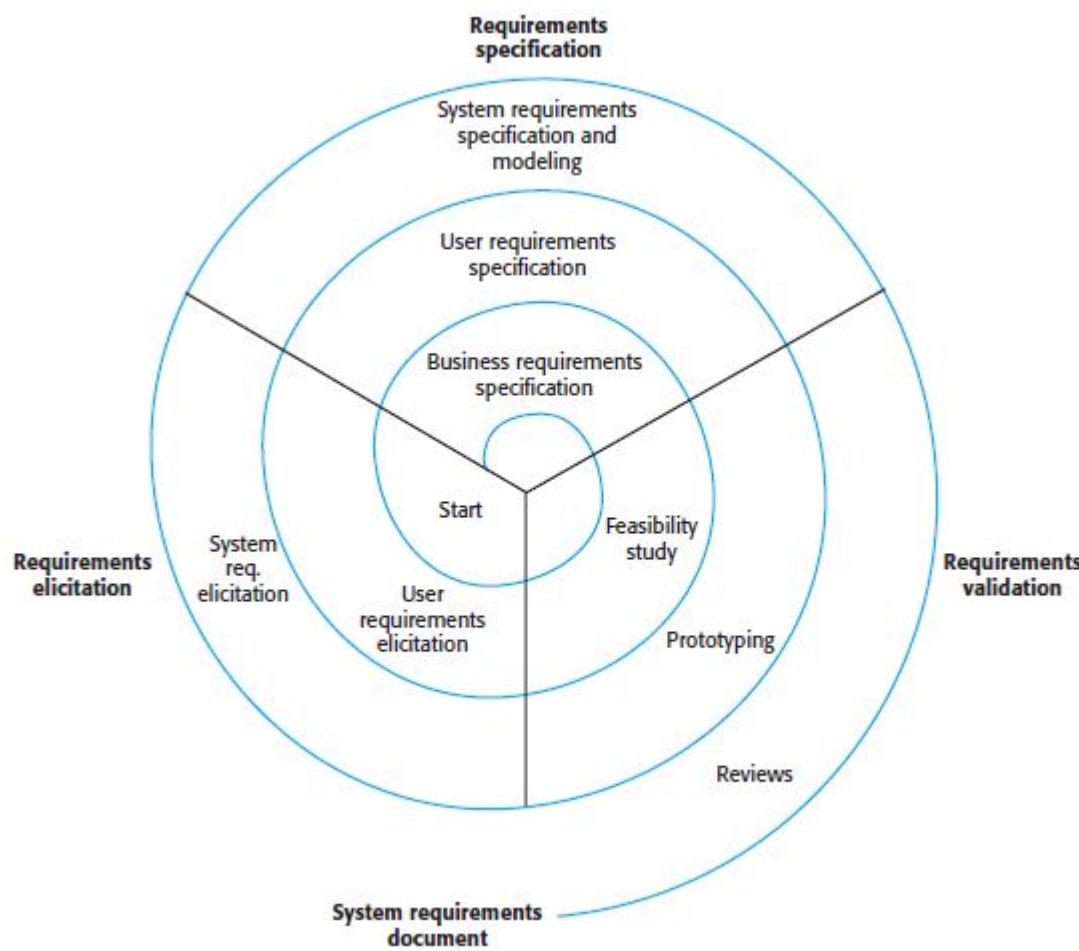
- In principle, requirements should be both **complete** and **consistent**.
- Complete
 - They should **include descriptions of all facilities required**.
- Consistent
 - There should be **no conflicts or contradictions in the descriptions of the system facilities**.
- In practice, because of system and environmental complexity, it is **impossible to produce a complete and consistent** requirements document.

Requirements Engineering processes

Requirements Engineering processes



A spiral view of the requirements engineering process



Feasibility study

Feasibility study

- » A feasibility study decides whether or not the proposed system is worthwhile
- » A short focused study that checks
 - If the system contributes to organizational objectives
 - If the system can be engineered using current technology and within budget
 - If the system can be integrated with other systems that are used

Feasibility types

- » Technology and system feasibility
- » Economic feasibility
 - Cost-based study
 - Time-based study
- » Legal feasibility
- » Operational feasibility
- » Schedule feasibility
- » Resource feasibility
- » Cultural feasibility

Requirements elicitation and analysis

Requirements elicitation

- ❖ Find about application domain, services and operational constraints
- ❖ Requirements elicitation is essentially a **communications process**, involving different stakeholders.
- ❖ Do at the **start of the project**, an element of elicitation carry out through out the life cycle.
 - new requirements may be proposed during design and implementation
 - Existing requirements may be modified and deleted.

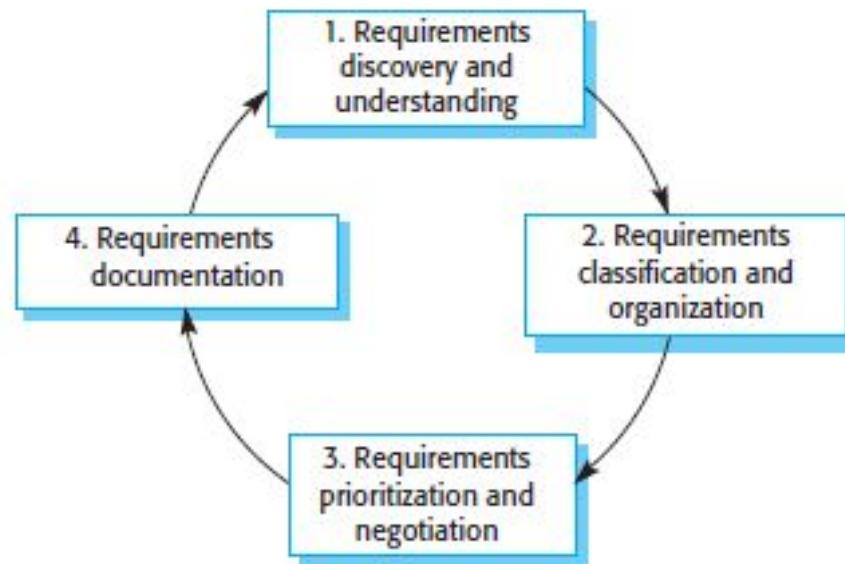
What is fact finding?

- ❖ Focusing on a study of organization.
 - Identification of relevant users across different levels.
 - Identification of analysts with relevant domain and technical expertise.
 - Determination of scope or reconfirmation of the scope.
 - Identification of similar systems
- ❖ Finally it will gives these outputs;
 - Statement of problem context
 - Overall objectives
 - Boundary and constraints

Requirements elicitation

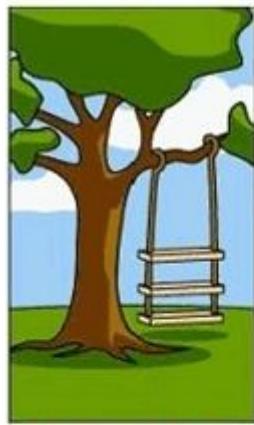
- ❖ Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide.
- ❖ Stages include:
 - Requirements discovery,
 - Requirements classification and organization,
 - Requirements prioritization and negotiation,
 - Requirements documentation.

Requirements elicitation and analysis

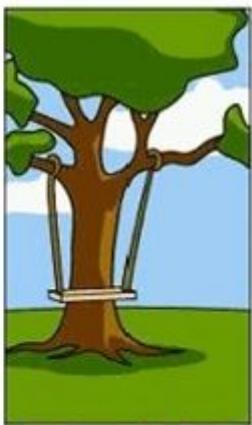


Problems of requirements elicitation

- ❖ Stakeholders **don't know what they really want.**
- ❖ Stakeholders **express** requirements in their **own terms.**
- ❖ Different stakeholders may have **conflicting requirements.**
- ❖ **Organisational and political factors** may **influence** the system requirements.
- ❖ The **requirements change during the analysis process.**
New stakeholders may emerge and the business environment may change.



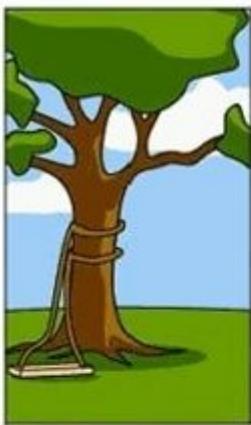
How the customer
explained it



How the project leader
understood it



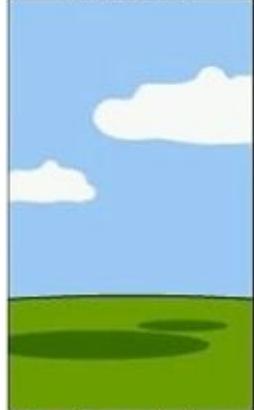
How the engineer
designed it



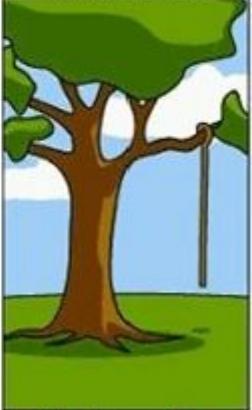
How the programmer
wrote it



How the sales
executive described it



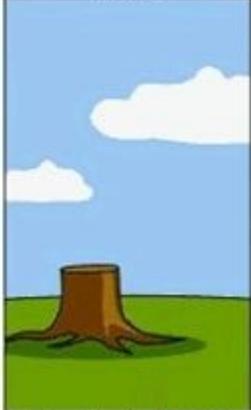
How the project was
documented



What operations
installed



How the customer
was billed



How the helpdesk
supported it



What the customer
really needed

Process activities

- ❖ Requirements discovery
 - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- ❖ Requirements classification and organisation
 - Groups related requirements and organises them into coherent clusters.
- ❖ Prioritisation and negotiation
 - Prioritising requirements and resolving requirements conflicts.
- ❖ Requirements specification
 - Requirements are documented and input into the next round of the spiral.

Requirements Discovery

- From different user expectation we capture the requirements
- With the scope of the system understood, the next task is to gather the requirements in more detail. The requirements gathering exercise consists of
 - Creating **a wish list** for each user category across all levels of the user community
 - **Classify wish lists** according to functional, and non-functional, environment, and design constraints

Requirements Discovery

- ❖ There are two fundamental approaches to requirements discovery:
 - **Interviewing**, where you talk to people about what they do.
 - **Observation or ethnography**, where you watch people doing their job to see what artifacts they use, how they use them, and so on.
- ❖ You should use a mix of interviewing and observation to collect information and, from that, you derive the requirements.
- ❖ Meeting, Q&A, simulations and group discussions also used.

- ❖ At times there are certain aspects about which information is not yet available though the aspects are identified.
- ❖ Such Items are also included as "**To be Determined**" (TBD).
- ❖ So that they remain visible and are noticed and resolved before the requirements phase is considered complete.

Interviewing

- ❖ Formal or informal interviews with stakeholders are part of most RE processes.
- ❖ Types of interview
 - **Closed interviews**: where stakeholders answer a predetermined list of questions
 - **Open interviews**: where various issues are explored with stakeholders (no pre defined ones)
- ❖ For “Effective interviewing”
 - **Be open-minded**, avoid fixed ideas about the requirements and are willing to listen to stakeholders.
 - **Prompt the interviewee to get discussions going** by using a springboard question, a requirements proposal, or by working together on a prototype system.

Interviews in practice

- ❖ Normally a **mix of closed and open-ended interviewing**.
- ❖ Interviews are **good** for getting an **overall understanding of what stakeholders do** and how they might interact with the system.
- ❖ **Interviews are not good for understanding domain requirements**
 - Domain requirements may be expressed using special domain terminologies, and software engineers often find it difficult to understand and it's easy for them to misunderstand.
 - Sometimes stakeholders won't tell you some requirements because they assume it's so fundamental and it doesn't worth mentioning, or they find it difficult to explain, which won't be taken into consideration in the requirements..

Problems with interviews

- ❖ Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- ❖ Interviews are not good for understanding domain requirements



Ethnography

- ❖ Ethnography is an observational technique that can be **used to understand operational processes** and help **derive requirements for software to support these processes**.
- ❖ An analyst engages himself or herself in the working environment where the system will be used.
- ❖ The day-to-day work is observed, and notes are made of the actual tasks in which participants are involved
- ❖ Ethnography can be combined with the development of a system prototype

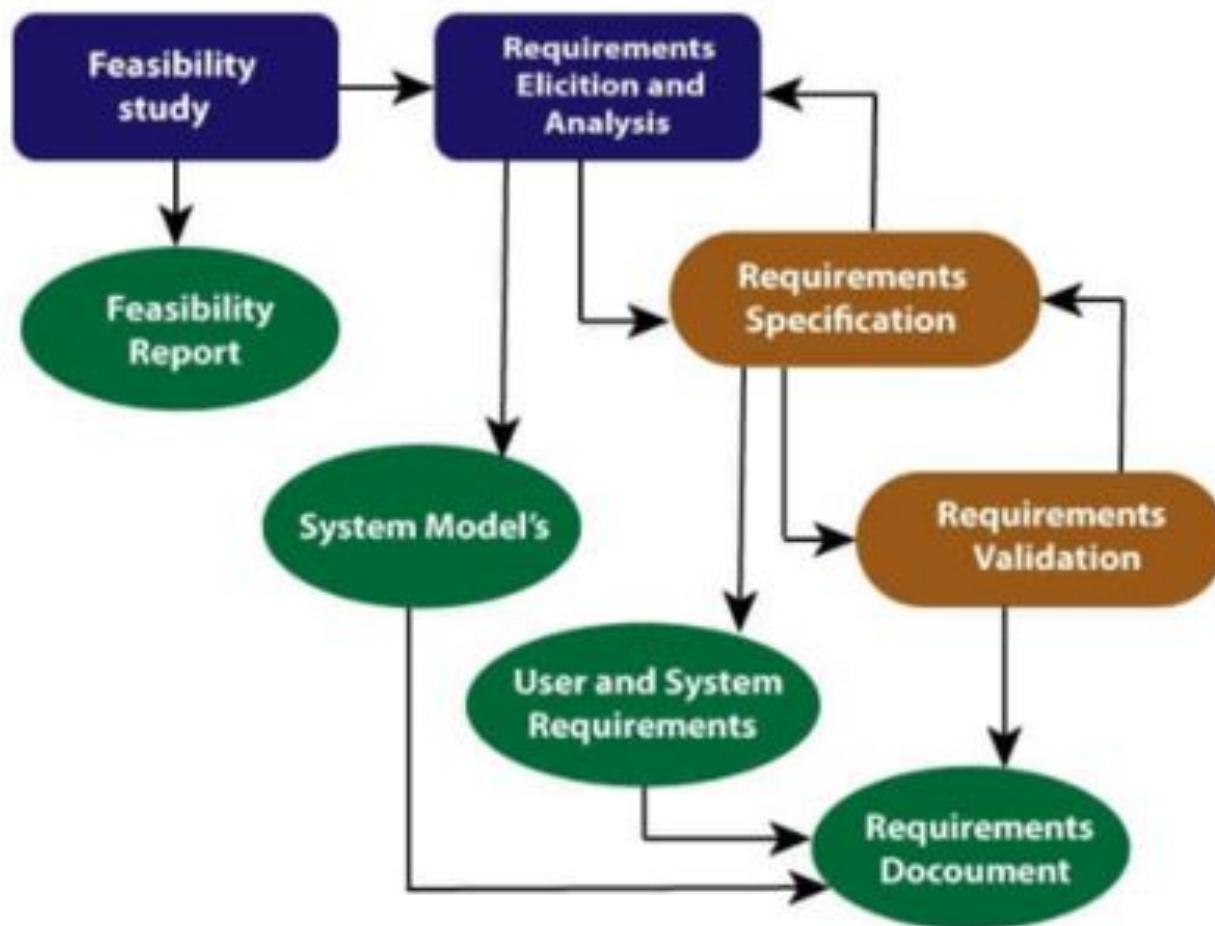
Requirement Classification & Organization

- It's very important to organize the overall structure of the system.
- Putting related requirements together, and decomposing the system into sub components of related requirements. Then, we define the relationship between these components.
- What we do here will help us in the decision of identifying the ***most suitable architectural design patterns.***

Requirements Prioritization & Negotiation

- ❖ This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiations until you reach a situation where some of the stakeholders can settle.
- ❖ Prioritizing your requirements will help you later to **focus on the essentials and core features of the system**, so you can meet the user expectations.
- ❖ It can be achieved by giving every piece of function a priority level. So, functions with higher priorities need **higher attention and focus**.

Requirements Engineering processes



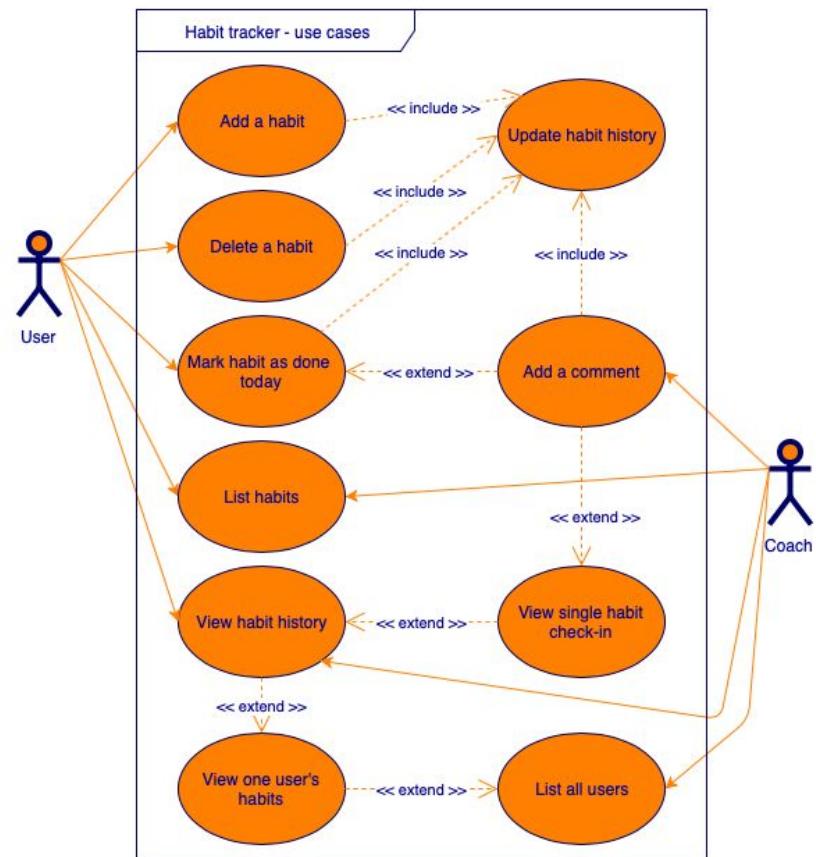
Requirement Analysis

- ❖ Starts in parallel in the elicitation process
- ❖ Helps to identify
 - Inconsistencies, errors and other issues
 - Provide basis for the design
 - Missing parts in requirements
 - Relationships in requirements
- ❖ Proposed several models – depends on views
 - Use case
 - Sequence diagram
 - Component diagram
 - Class diagram
 - Develop prototype

Use cases and scenarios

- ❖ The use cases and scenarios are two different techniques, but, usually they are used together.
- ❖ Use cases identify interactions between the system and its users or even other external systems (using graphical notations).
- ❖ A scenario is an instance of a use case describing a concrete set of actions

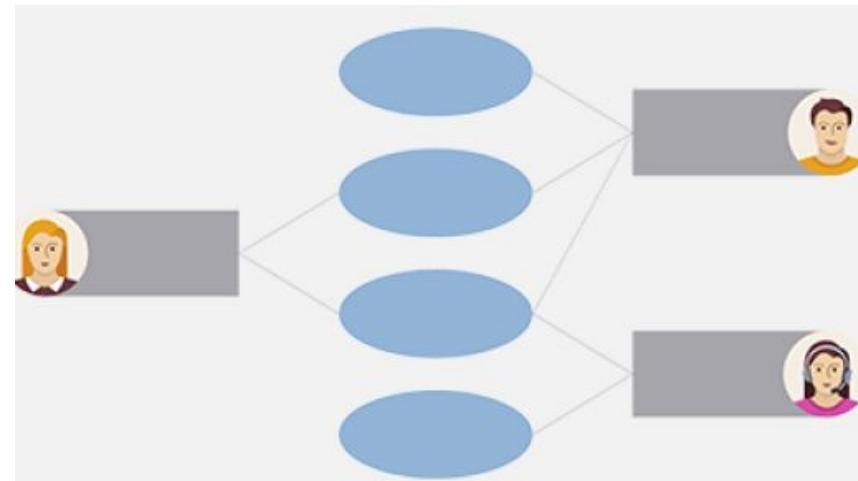
Use Cases



Graphical Notations: Use cases

- ❖ In the Unified Modeling Language (UML), a use case diagram is used to **summarize the details of your system's users** (also known as actors) and their interactions with the system.
- ❖ An effective use case diagram can help your team discuss and represent:

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system



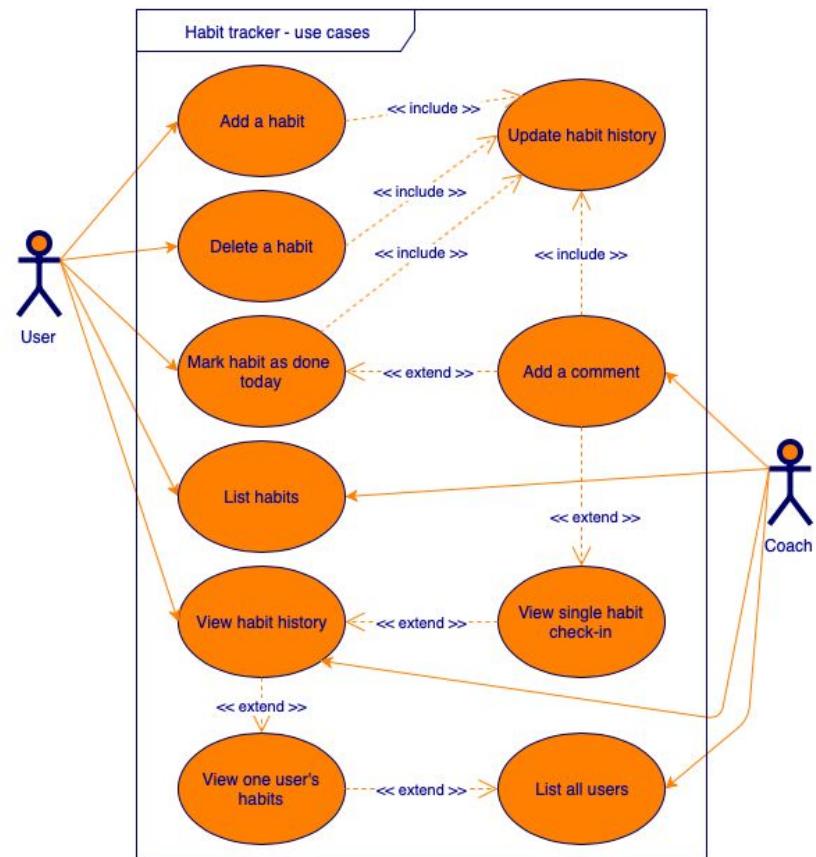
Use cases

- ❖ Use cases deal only in the **functional requirements for a system**.
- ❖ Importance of Use Case Diagrams
 - To **identify functions and how roles interact with them** – The primary purpose of use case diagrams.
 - For a **high-level view** of the system – Especially useful when presenting to managers or stakeholders. You can highlight the roles that interact with the system and the functionality provided by the system without going deep into inner workings of the system.
 - To identify **internal and external factors** – This might sound simple but in large complex projects a system can be identified as an external role in another use case.

Use Case Diagram Objects

- ❖ Use case diagrams consist of 4 objects.
 - Actors
 - Use case
 - System
 - Relationships

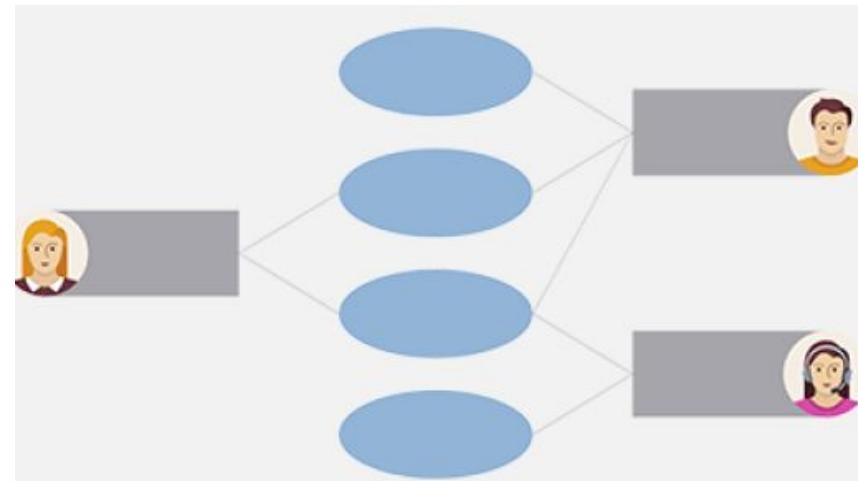
Use Cases



Graphical Notations: Use cases

- ❖ In the Unified Modeling Language (UML), a use case diagram is used to **summarize the details of your system's users** (also known as actors) and their interactions with the system.
- ❖ An effective use case diagram can help your team discuss and represent:

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system



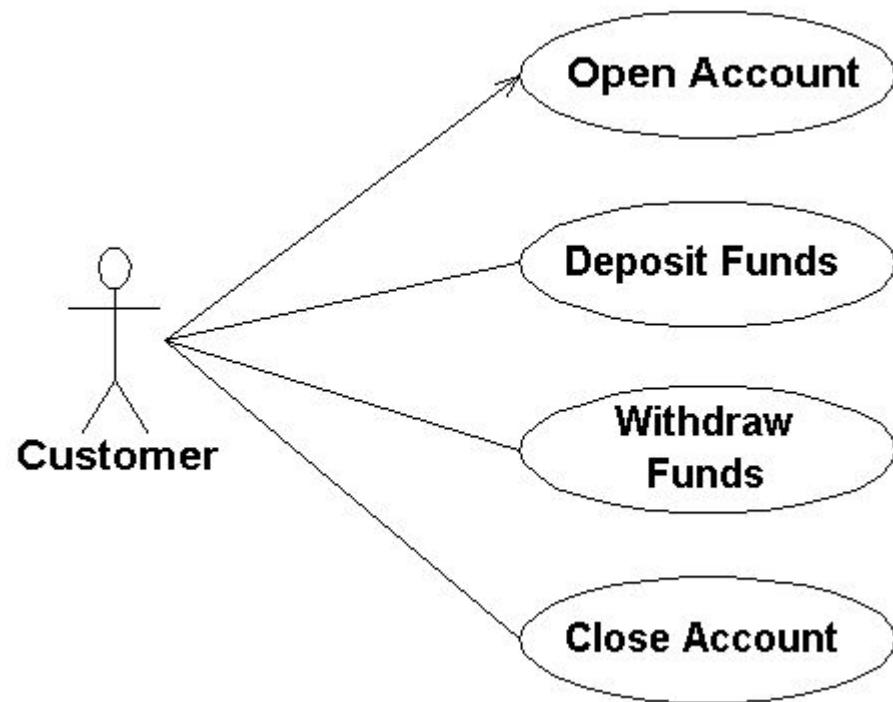
Use cases

- ❖ Use cases deal only in the **functional requirements for a system**.
- ❖ Importance of Use Case Diagrams
 - To **identify functions and how roles interact with them** – The primary purpose of use case diagrams.
 - For a **high-level view** of the system – Especially useful when presenting to managers or stakeholders. You can highlight the roles that interact with the system and the functionality provided by the system without going deep into inner workings of the system.
 - To identify **internal and external factors** – This might sound simple but in large complex projects a system can be identified as an external role in another use case.

Use Case Diagram Objects

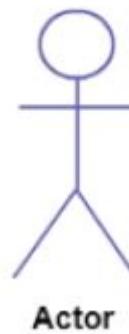
- ❖ Use case diagrams consist of 4 objects.
 - Actors
 - Use case
 - System
 - Relationships

Simple Use Case Diagram



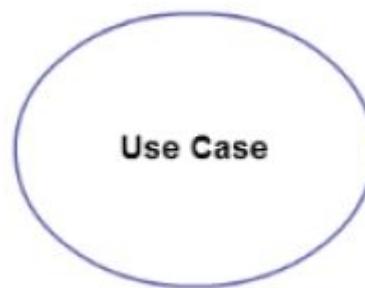
Actors

- ❖ Actor in a use case diagram is any entity that **performs a role in one given system.**
- ❖ This could be a person, organization or an external system and usually drawn like skeleton shown below.



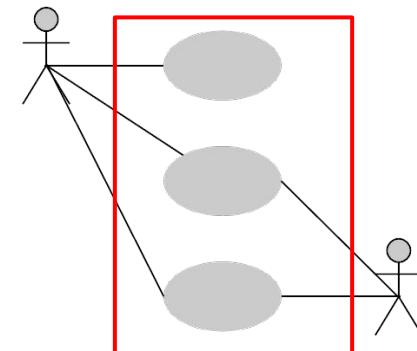
Use case

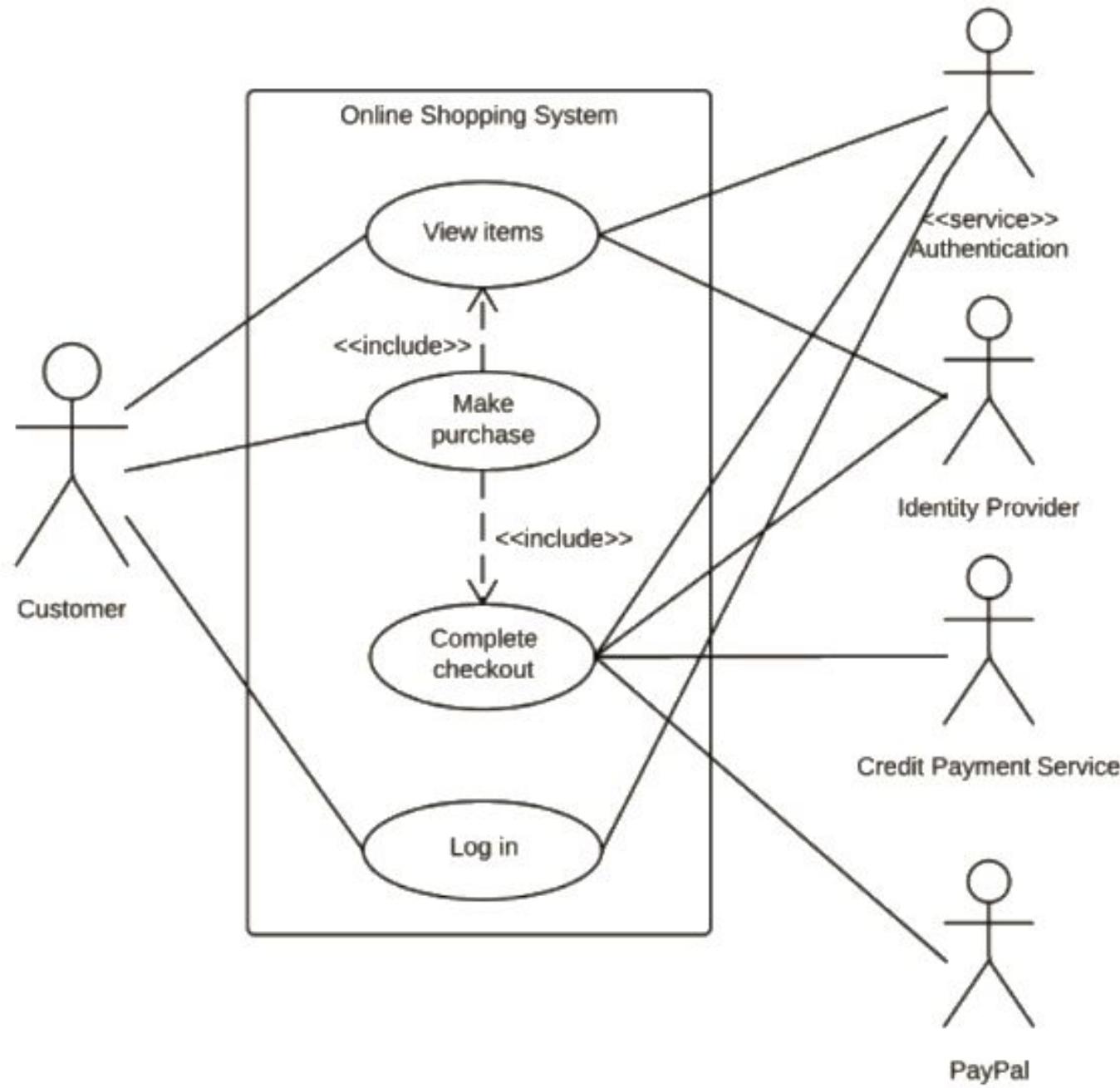
- ❖ A use case represents a **function** or an action within the system.
- ❖ It's drawn as an oval and named with the function.



System

- ❖ The system is used to define the **scope of the use case and drawn as a rectangle.**
- ❖ This is an optional element but useful when you're visualizing large systems.
- ❖ For example, you can create all the use cases and then use the system object to define the scope covered by your project.



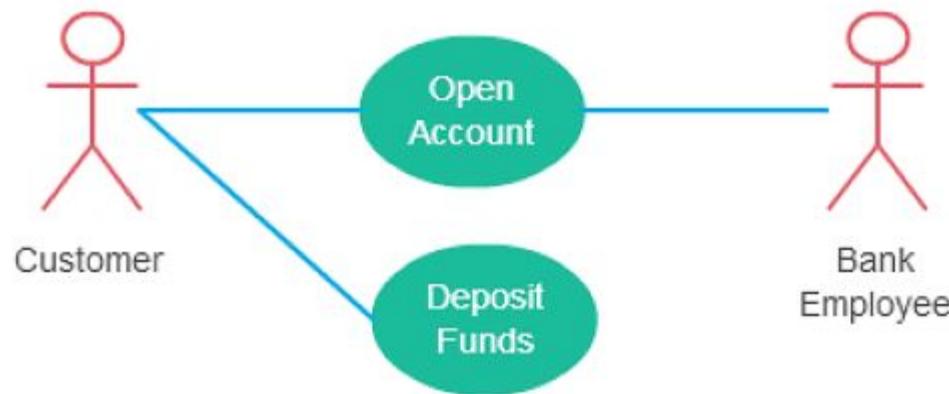


Relationships

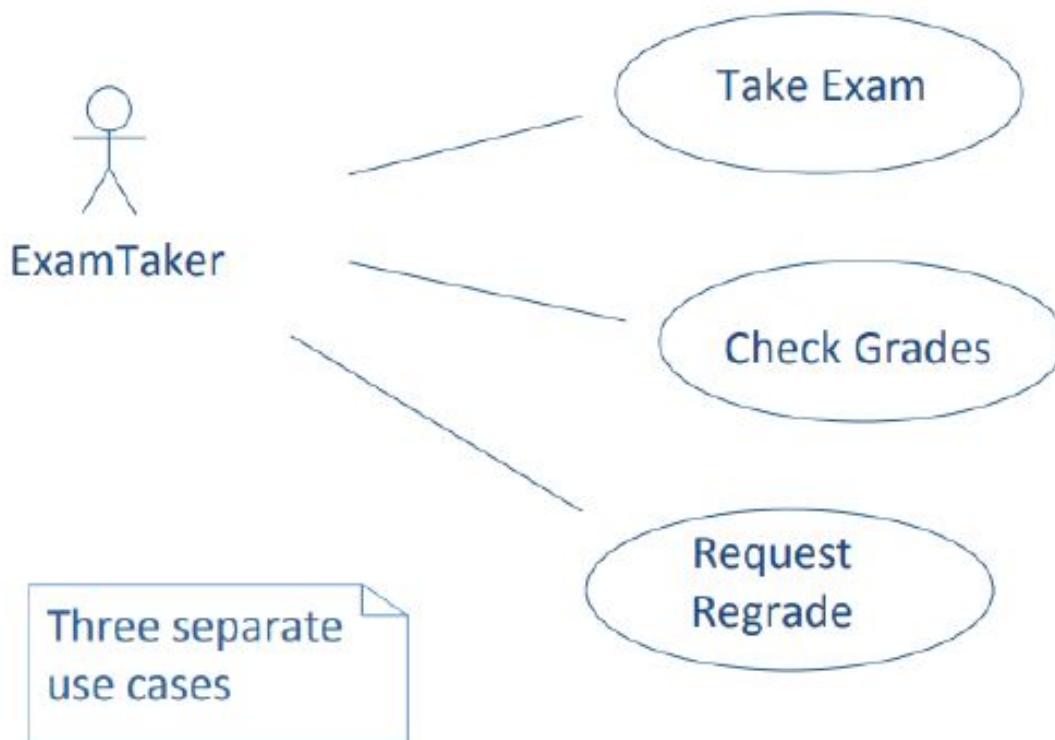
- ❖ There are four types of relationships in a use case diagram. They are
 - ↳ **Association** between an actor and a use case
 - ↳ **Generalization** of an actor
 - ↳ **Extend** relationship between two use cases
 - ↳ **Include** relationship between two use cases

Relationships

- ❖ Association between an actor and a use case
 - An actor must be associated with at least one use case.
 - An actor can be associated with multiple use cases.
 - Multiple actors can be associated with a single use case

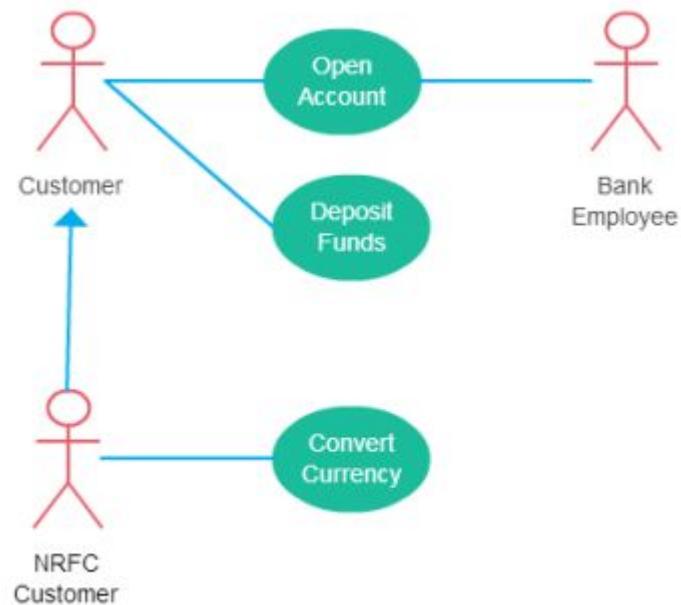


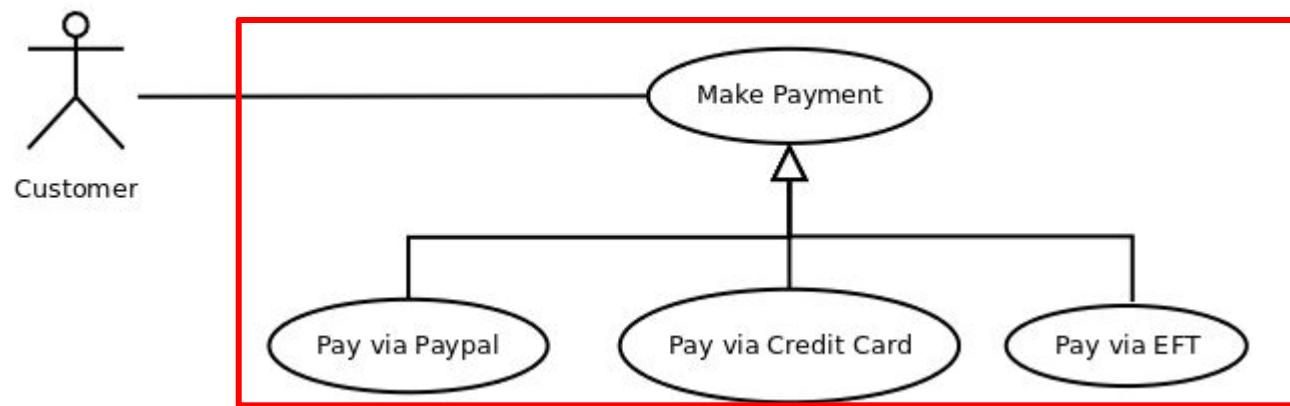
Relationships - Exam System



Generalization

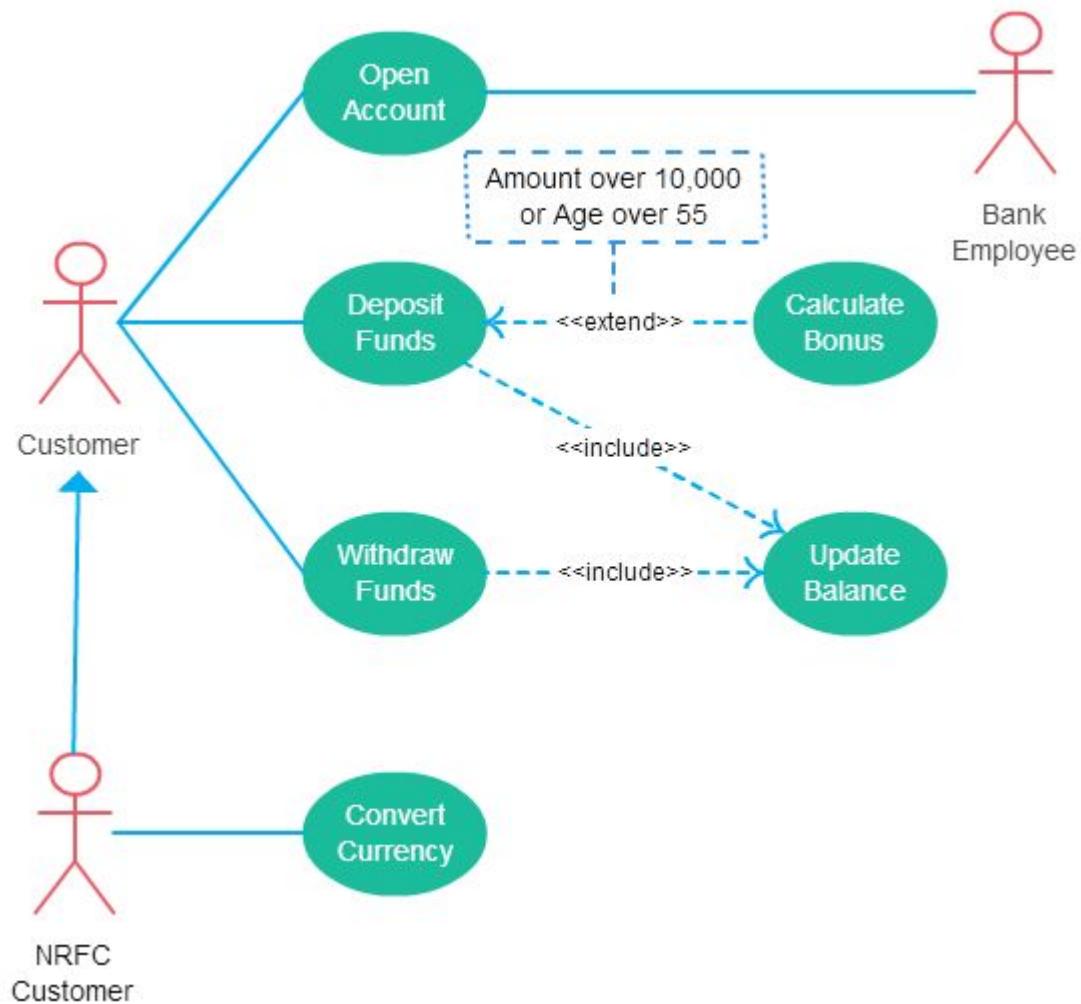
- ❖ Generalization of an actor
 - One actor can **inherit the role of the other actor**.
 - The **descendant** inherits **all** the use cases of the **ancestor**.
 - The descendant has one or more use cases that are specific to that role.

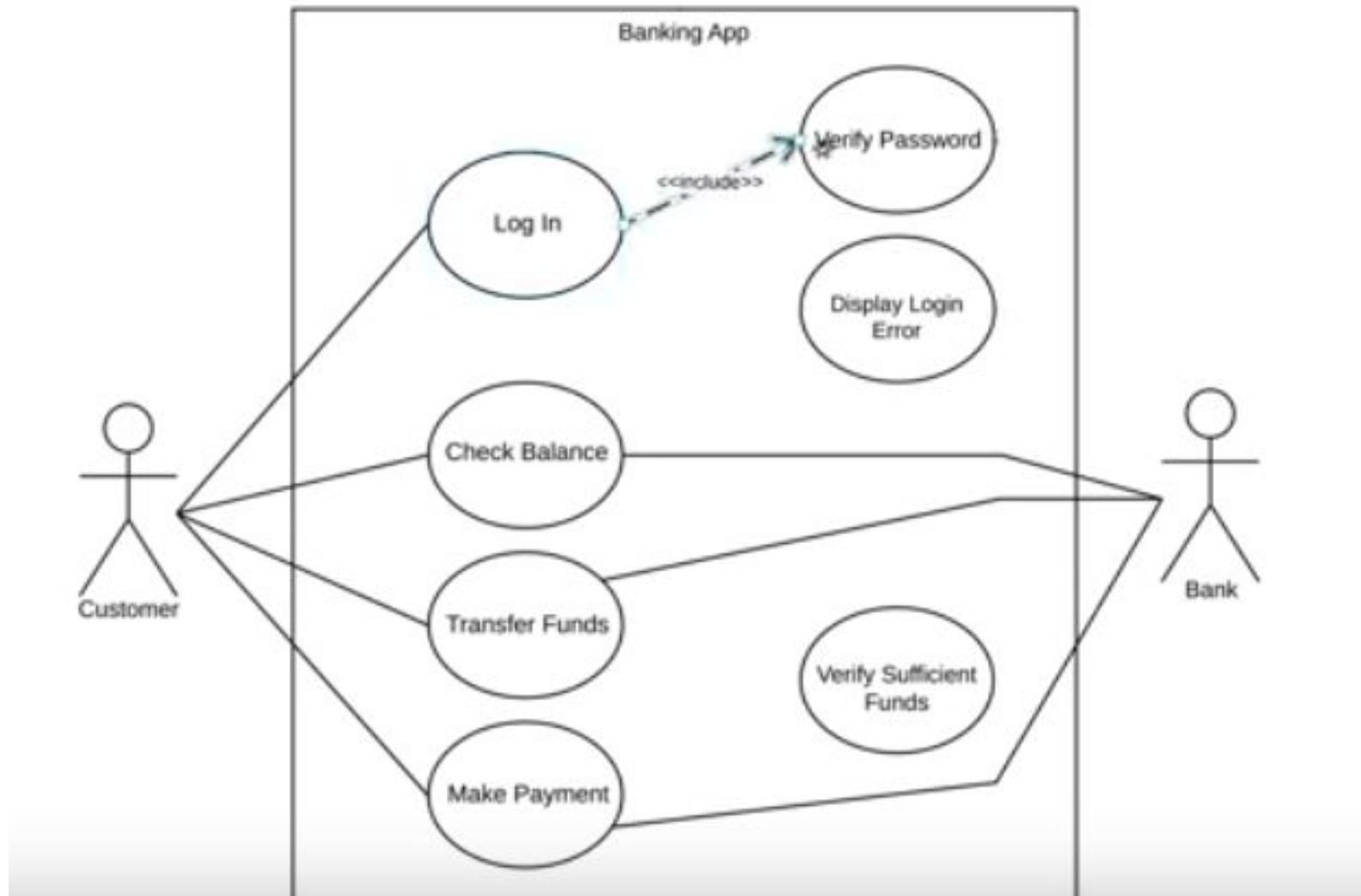




Include relationship

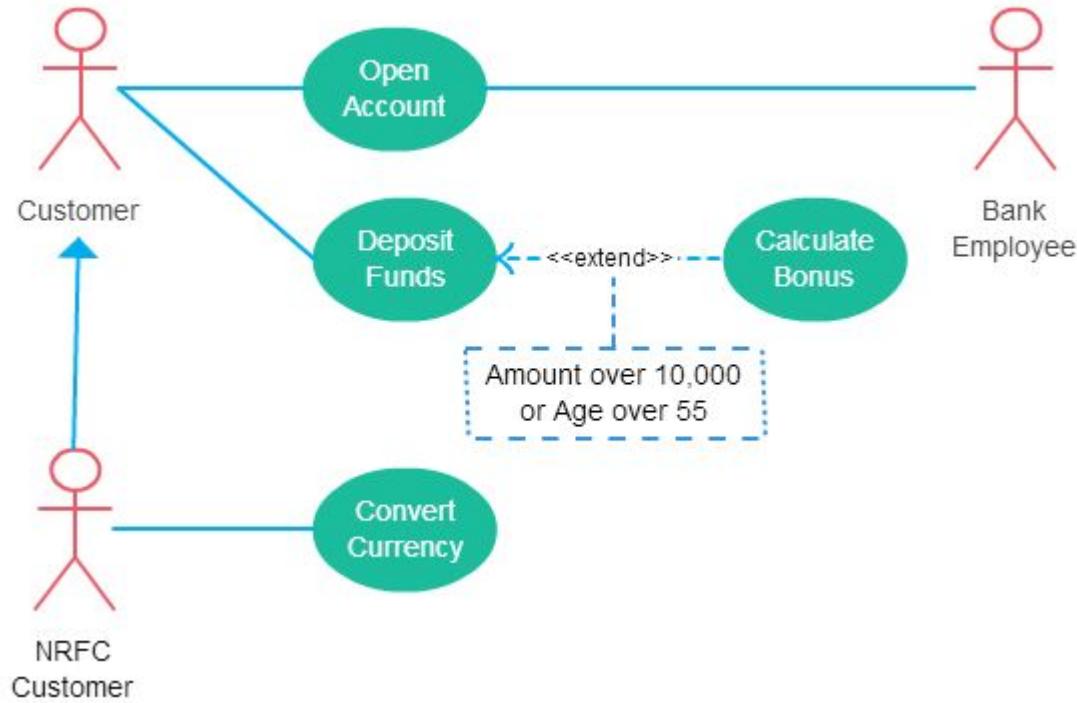
- ❖ Include relationship show that the behavior of the included use case is part of the including (base) use case.
- ❖ The main reason for this is to reuse the common actions across multiple use cases.

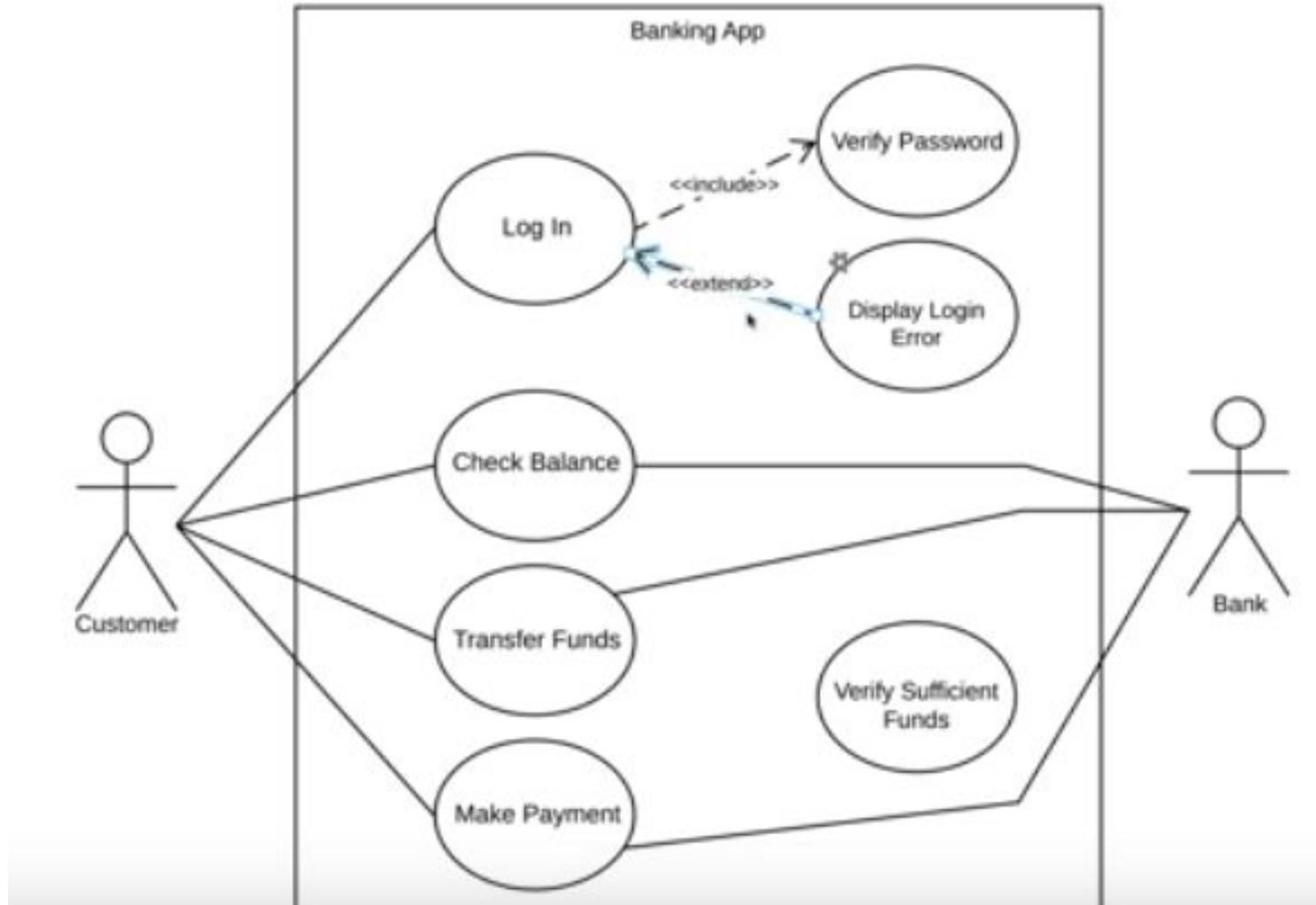


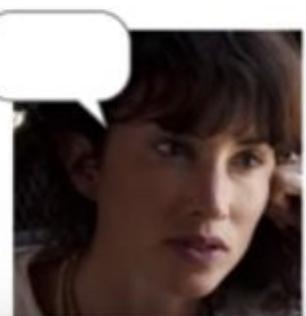
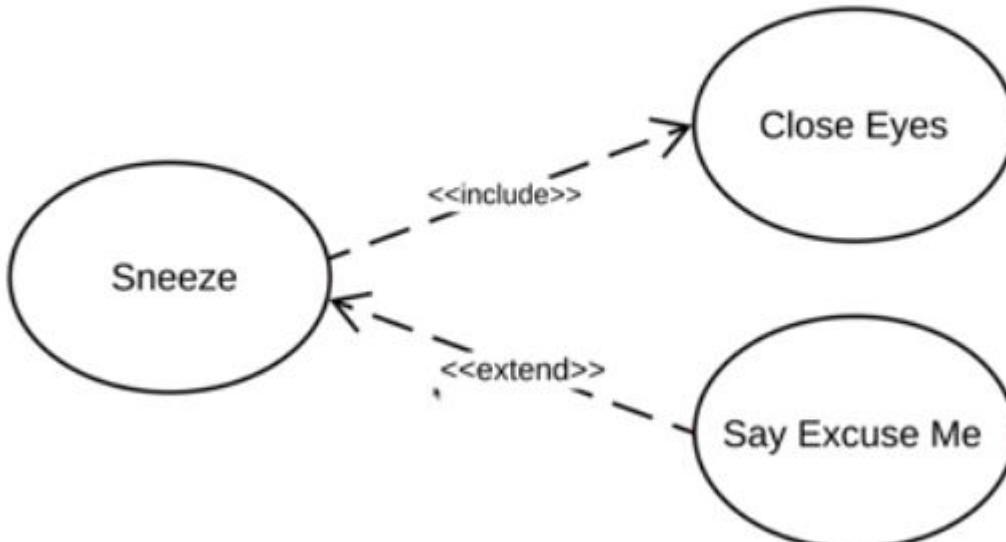


Exclude relationship

- ❖ It extends the base use case and adds more functionality to the system
- ❖ The extending use case is dependent on the extended (base) use case
- ❖ The extending use case is usually optional and can be triggered conditionally
- ❖ The extended (base) use case must be meaningful on its own.







Scenarios

- ❖ A structured form of user story
- ❖ Scenarios should include
 - A description of the **starting situation**;
 - A description of the **normal flow of events**;
 - A description of **what can go wrong**;
 - Information about **other concurrent activities**;
 - A description of the **state when the scenario finishes**.

Name	Course Registration
Actors	Student and University System
Description	It shows how a student can register for a course and view personal info
Pre-condition	The student is logged in
Post-condition	The student registered his/her course list for the semester.
Actions(Main Scenario)	<ol style="list-style-type: none"> 1. Student will press on "Course Registration" from Home Page. 2. Select desired courses for the next semester. 3. Enter personal info. 4. Press on "Register". 5. Confirmation message upon success. 6. Student can view his/her personal info.
Exceptions	#3 User entered invalid input, thus, an error message will be displayed

Requirements specification

Requirements specification

- ❖ Requirements specification is the process of **writing down the user and system requirements in a requirements document.**
- ❖ The user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent
- ❖ User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document.
- ❖ System requirements may also be written in natural language, but other notations based on forms, graphical, or mathematical system models can also be used

❖ Most important work products

- Customers/users for understanding **what** they are supposed to get
- Project managers to estimate and plan the project to deliver the system.
- Designers and programmers to know what to build.
- Testers to prepare testing activities
- Maintenance teams for understanding the system that they will maintain
- Trainers to prepare training material for training the end users
- Users to understand the proposed system and prepare for the change over.

Ways of writing a system requirements specification

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. UML (unified modeling language) use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want, and they are reluctant to accept it as a system contract. (I discuss this approach, in Chapter 10, which covers system dependability.)

Guidelines for writing requirements

- ❖ Invent a **standard format** and use it for all requirements.
- ❖ Use language in a consistent way. Use **shall** for **mandatory** requirements, **should** for **desirable** requirements.
- ❖ Use **text highlighting** to identify key parts of the requirement.
- ❖ **Avoid** the use of **computer jargon**.
- ❖ **Include** an **explanation** (rationale) of why a requirement is necessary.

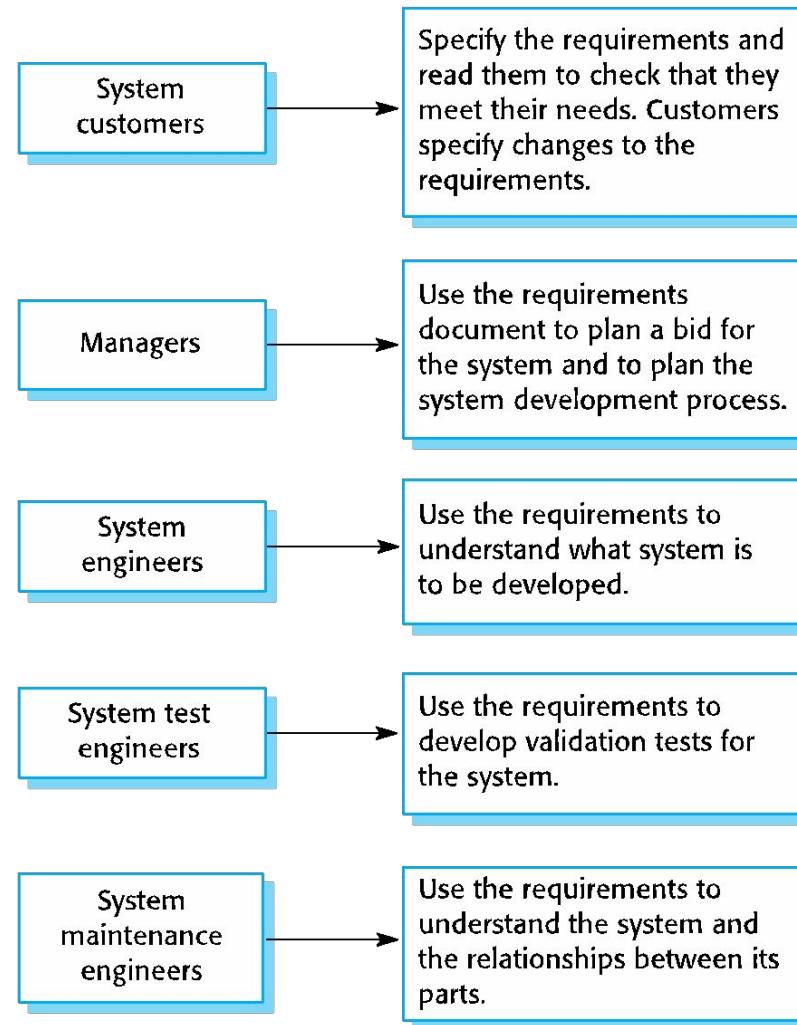
The software requirements document

- ❖ The software requirements document is the official statement of what is required of the system developers.
- ❖ Should include both a definition of user requirements and a specification of the system requirements.
- ❖ It is **NOT a design document**. As far as possible, it should **set of WHAT the system should do rather than HOW it should do it**.

Structure of the SRS

- ❖ Preface
- ❖ Introduction
- ❖ Glossary
- ❖ User requirements definition
- ❖ System architecture
- ❖ System requirements specification
- ❖ System models
- ❖ System evolution
- ❖ Appendices

Users of a requirements document



Requirements Documentation- SRS

Characteristics

Software Requirement
Specification

Completeness

- ❖ Requirements from all the stakeholders
- ❖ More than the system functionality
 - Focusing on user tasks, problems, bottlenecks and improvements required
- ❖ Ranking/Prioritizing Requirements
 - Customers identifies most important ones
 - Scope Matrix
- ❖ NO TBDs – complete all missing requirements

Clarity

- ❖ The documented requirements should lead to only a single interpretation independent of the person or the time
- ❖ Minimize the use of natural languages
- ❖ Use requirement specification language
 - Takes more time and not practical

Readability

- ❖ Consistently number chapters, sections and sub sections and individual requirements
- ❖ Align text to the left margin rather than “justify both margins”
- ❖ Use white space to reduce eye strain
- ❖ Use highlighting features
- ❖ Do not use colors as far as possible
- ❖ Create a table of contents
- ❖ Use hyperlinks within the document to easy navigation

Correctness

- every requirement stated must be actual required by the system.

Consistency

- There should not be any conflicts in the requirements

Modifiability

- labels and references to figures, tables, diagrams and appendices in the SRS

Traceability

- ❖ Labeling (or numbering) of requirements provides a mechanism to trace each requirement through the design into the test plans, test cases and source code.
 - **Backward traceability** to the documentation/and other work-products created prior to the requirements phase. This will depend upon how well referencing and labeling has been provided in the earlier documents/ work products
 - **Forward traceability** – This will depend upon how each requirement is labeled or numbered. Forward traceability is extremely important as this is one of the way of tracing a requirement to its implementation.

Common Problems with SRS

- ❖ Making bad assumptions
- ❖ Writing implementation (How) instead of requirements (What)
- ❖ Using wrong language – Requirements should be easy to read and understand
- ❖ Unverifiable requirements
- ❖ Over specifying or missing requirements
- ❖ Using incorrect terms – identification of shall and should
 - Are
 - Is
 - Was
 - Must – do not belong in a requirement

Requirements Validation

Requirements Validation

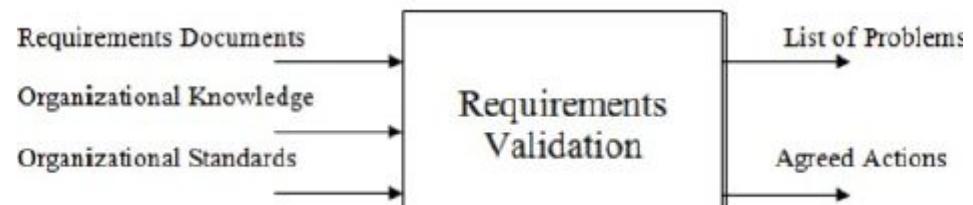
- ❖ Requirements validation is the process of checking that requirements define the system that the customer really wants.
- ❖ It overlaps with elicitation and analysis, as it is concerned with finding problems with the requirements.
- ❖ Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

Requirements checking

- ❖ Validity - Does the system provide the functions which best support the customer's needs?
- ❖ Consistency - Are there any requirements conflicts?
- ❖ Completeness- Are all functions required by the customer included?
- ❖ Realism - Can the requirements be implemented given available budget and technology
- ❖ Verifiability- Can the requirements be checked?

Requirements validation techniques

- ❖ Requirements reviews
 - Systematic manual analysis of the requirements.
- ❖ Prototyping
 - Using an executable model of the system to check requirements.
- ❖ Test-case generation
 - Developing tests for requirements to check testability



Requirements reviews

- ❖ Regular reviews should be held while the requirements definition is being formulated.
- ❖ Both client and contractor staff should be involved in reviews.
- ❖ Reviews may be formal (with completed documents) or informal.
- ❖ Good communications between developers, customers and users can resolve problems at an early stage.

❖ Continuous Review

- Verification ensure that the requirements conform to the characteristics of excellent requirement statements (complete, correct, feasible, necessary, prioritized, unambiguous, verifiable) and of excellent requirements specification (complete, consistent, modifiable, traceable).

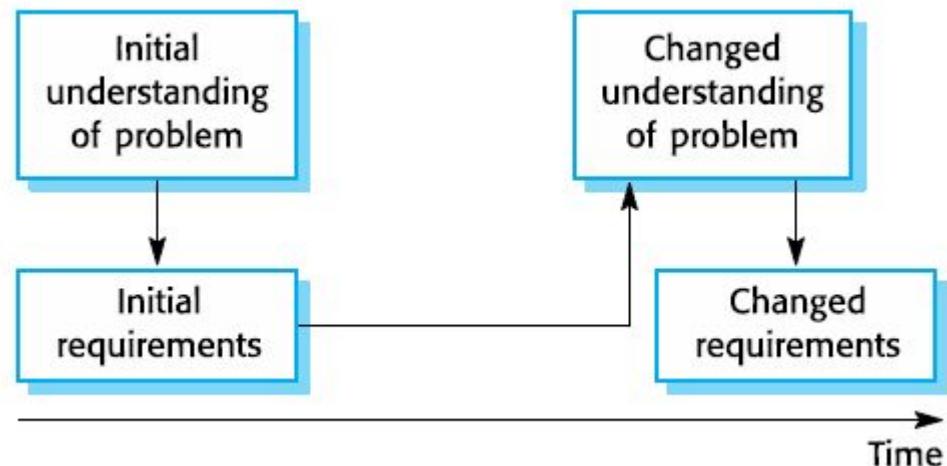
❖ Phase end review

- Do the phase wise review
- Has a group of stakeholders and they do it and get necessary actions

Requirements change

Changing Requirements

- ❖ The business and technical environment of the system always changes after installation.
- ❖ The people who pay for a system and the users of that system are rarely the same people.
- ❖ Large systems usually have a diverse user community



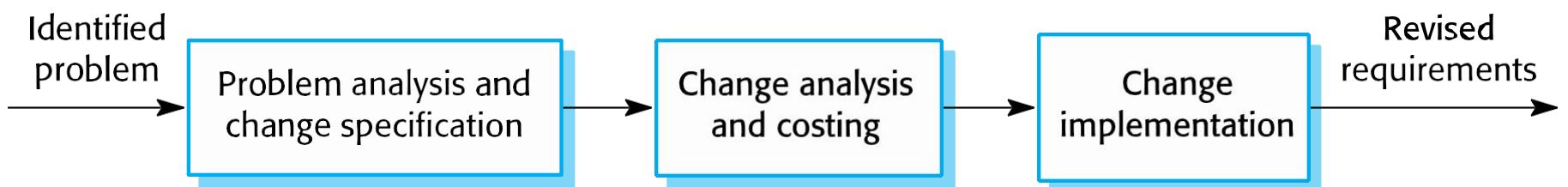
Requirements Management (RM)

- RM is the process of agreeing on requirements and then controlling change and communicating to relevant stakeholders.
 - It is a continuous process throughout a project
 - New requirements emerge as a system is being developed
 - Need to establish a formal process for making change proposals and linking these to system requirements
- Two main aspects of RM
 - *Change Management*
 - *Maintaining Requirements Traceability*

Requirements management planning/decisions

- Requirements identification
- A change management process
- Traceability policies
- Tool support

Requirements change management process



Change Management Procedure

Initiation → Execution → Closing

- ① • **Initiation** – A change to the requirements is initiated by filling up the relevant parts of the CRF. A change to requirements must always be expressed as a change to the SRS. The initiator has to describe the benefits of the change and how important the change is from the initiator's point of view.
 - ↙ *Change Request form*
- The **CRF** is handed over the configuration controller or configuration manager. The project manager and the configuration manager hands over the CRF to various people to get different view points on the change, like, calendar time, effort, additional resource etc.

Change Management Procedure

- The impact analysis above is collected by the configuration manager who calls for a meeting of the **CCB**. The CCB holds the configuration manager, the project manager, the chief designer, and some others from the customer or user side. Anyone of the following decisions can be taken.
 - Approved for change
 - Rejected
 - Partially approved
 - Keep pending for next CCB meeting (may be once a week or month)
 - Perform further impact analysis.

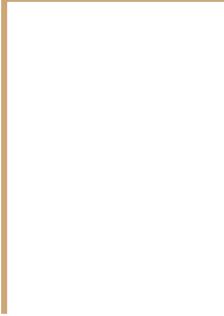
↗ *Change Control Board / Configuration Control Board*

Change Management Procedure

- Once approved or partially approved then the CM makes sure that the Implementation Plan is updated for implementing the CRF and includes the baselined items that need to be changed now and what quality controls that be taken for each item that will be changed.
- ② • **Execution – The changes are made according to the Implementation Plan.** This includes all the quality control steps that were planned in the Implementation Plan for the CRF.
- ③ • **Closing** – The CRF is marked as closed after verifying that all changes that were required have been made.

Requirements Traceability

- ❖ Requirements traceability is the means of tracing down each individual requirement in the SRS to the various work-products.
 - It helps to confirm that all requirements are designed and tested.
 - It often happens that certain requirements get missed out from the design or test plan due to oversight. The cross reference identifies such oversight immediately.
 - It helps to confirm that every feature in downstream work products supports a requirement. Designers and developers sometimes add features, some of which may not support any requirement. Such features are a waste of time and effort. The traceability cross-reference identifies such unnecessary design features.
- ❖ Requirements could be traced forward or backward.

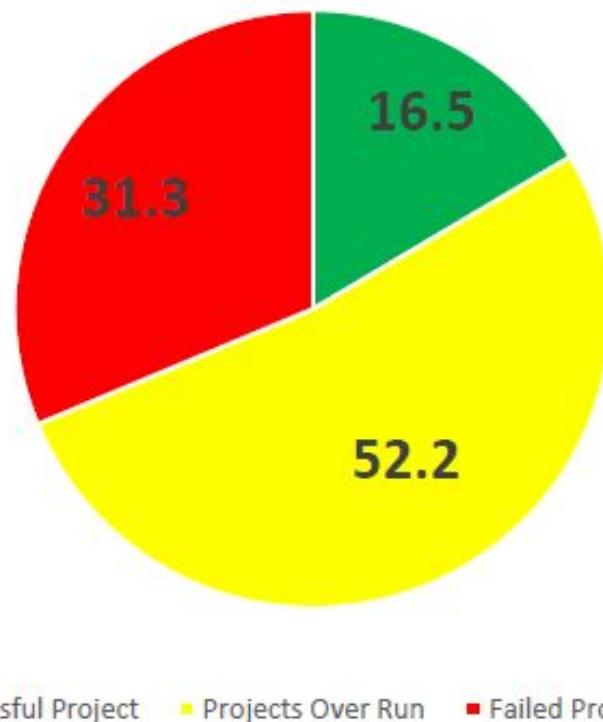


Agile Software Development



Status of Software Projects

Sandish CHOS Report 2019



Successful project key factors

- User involvement
- Executive management support
- Clear Statement of Requirements
- Proper planning
- Realistic expectations

Project Failure factors

- Incomplete Requirements
- Lack of user involvement
- Lack of resources
- Unrealistic expectations
- Changing Requirements & Specifications
- Lack of planning
- Lack of IT management
- Technical illiteracy

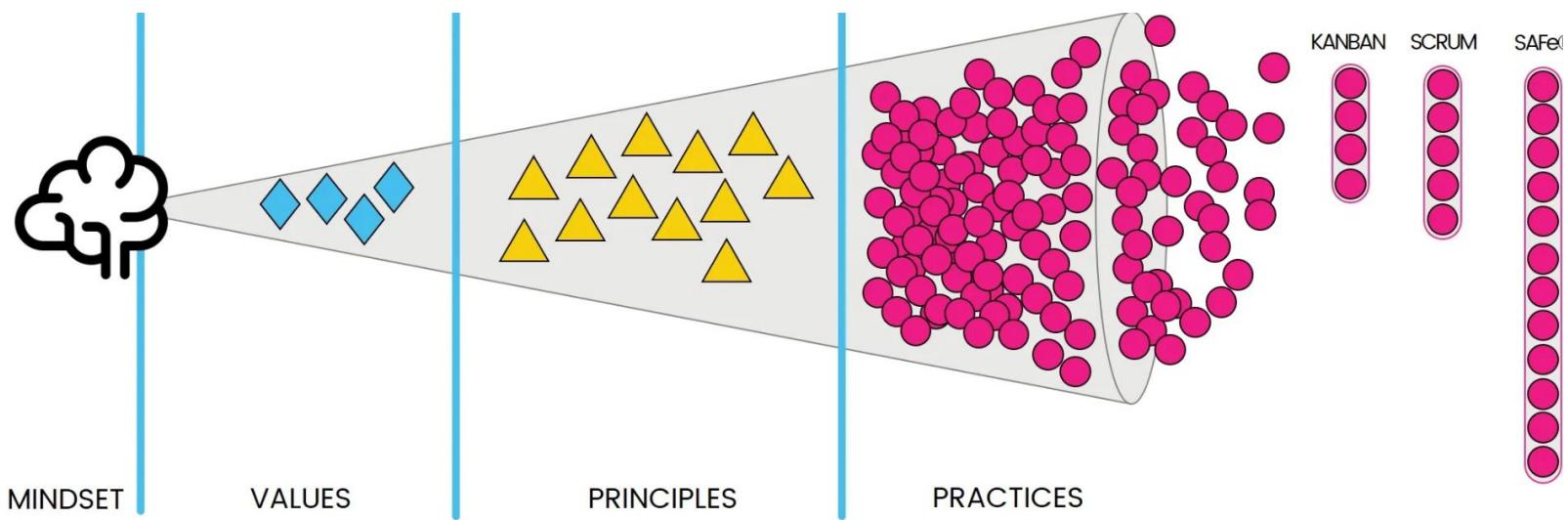
- ▼ Pervasiveness of Change?
 - ▶ Changes in Requirements
 - ▶ Changes in Design, Implementation,
 - ▶ Changes in Technology
 - ▶ Changes in Team
 - ▶ Changes in users/client contacts
- ▼ S/W engineers must be **quick to accommodate** the rapid changes

Agile

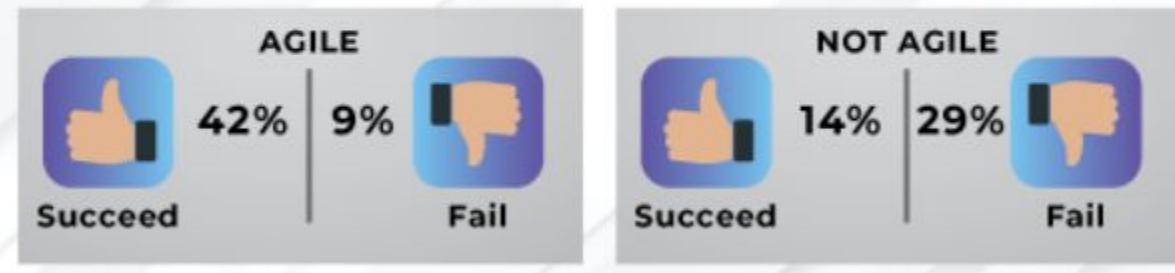
What is Agile?

- ▼ It's NOT a
 - ▼ Methodology
 - ▼ Specific way of developing software
 - ▼ Framework or process

Agile



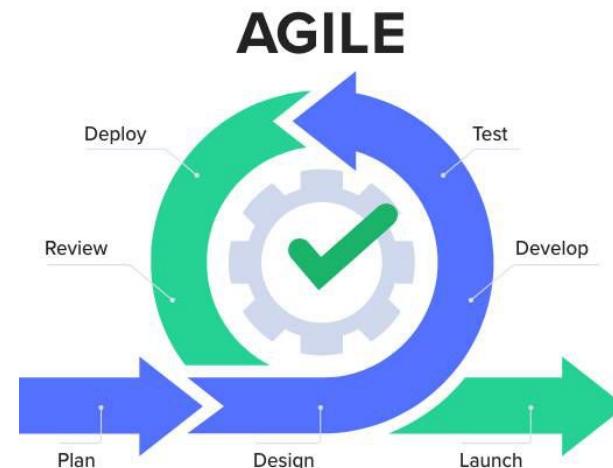
REDUCE FAILURE



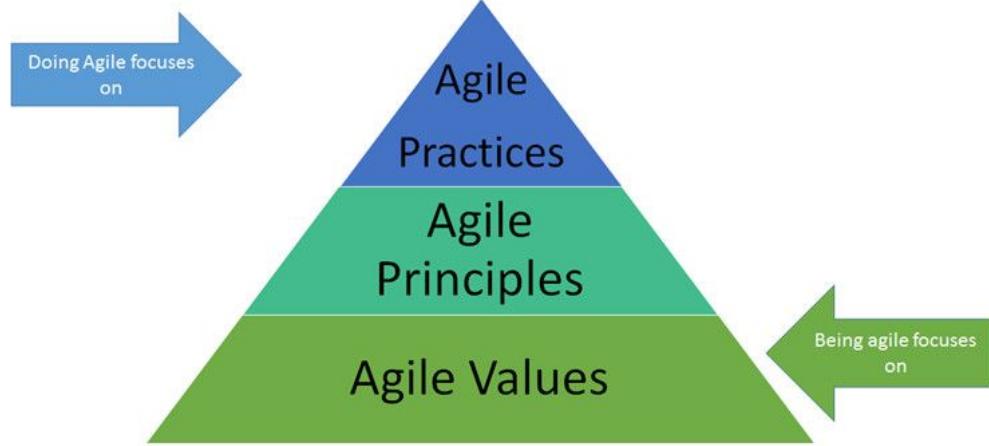
What is Agile?

Philosophy + a set of Guidelines for software development

“Agile is a mind set defined by **Values**, guided by **Principles** and manifested through many different **Practices**”



Agile Values, Principles and Practices



- 4 Values
- 12 Principles
- Many practices,
grouped with as
methodologies
 - E.g. XP, SCRUM,



Agile Software development

- ❖ Based on Learn & Adapt rather than rigid processes
- ❖ Evolving systems in short iterations
 - Each release is a working system
 - Focus on business value
 - Design for change
- ❖ Leveraging human strengths
 - Engage, align, and empower the team
 - Get power from each member



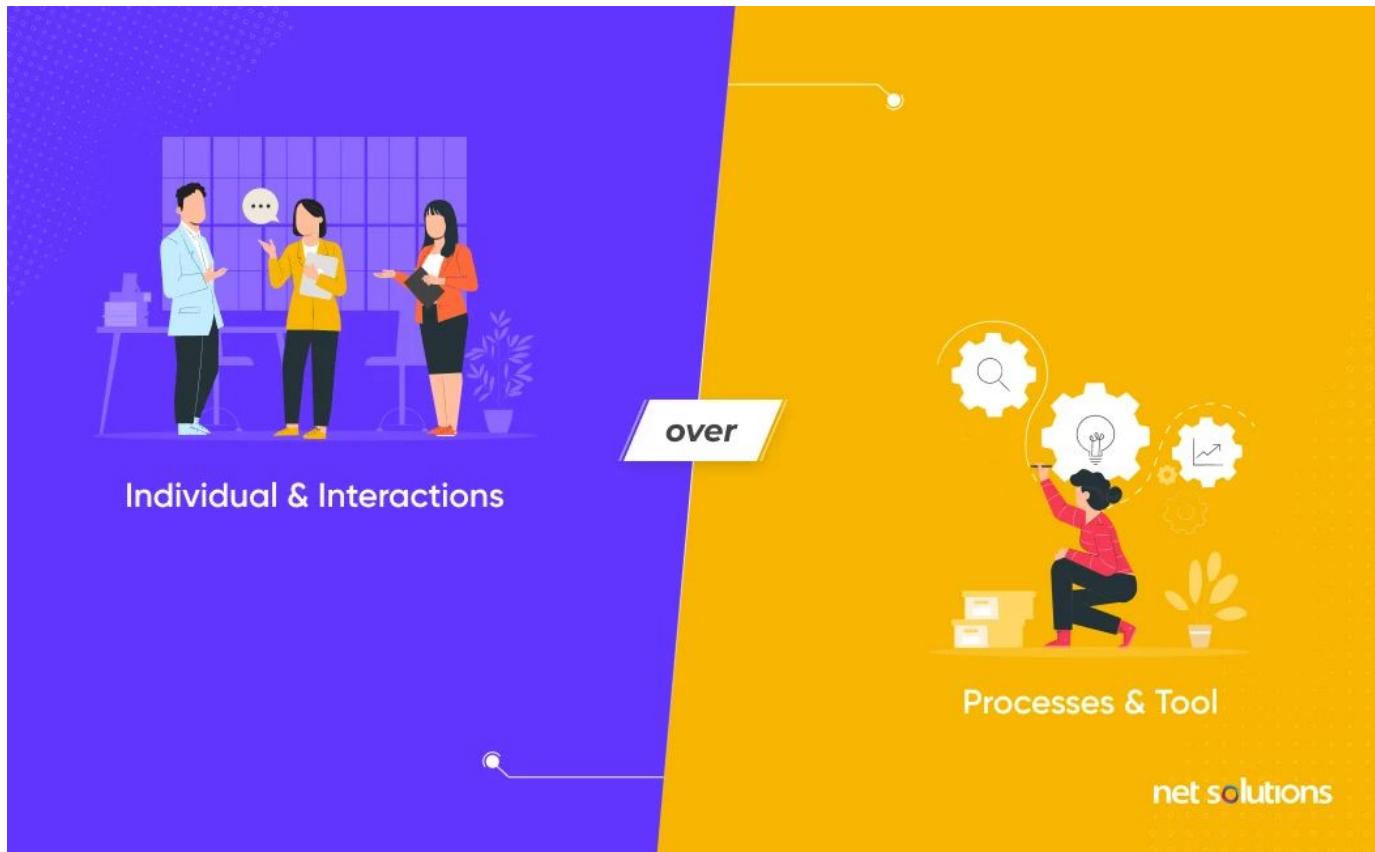
Agile vs Waterfall

	Agile Methods	Waterfall Methods
Approach	Adaptive	Predictive
Success Measurement	Business Value	Conformation to plan
Management Style	Decentralized	Autocratic
Perspective to Change	Change Adaptability	Change Sustainability
Culture	Leadership-Collaboration	Command-Control
Documentation	Low	Heavy
Emphasis	People-Oriented	Process-Oriented
Cycles	Numerous	Limited
Upfront Planning	Minimal	Comprehensive
ROI	Early in Project	End of Project
Team Size	Small/Creative	Large

Agile manifesto

- ⌚ The Agile Manifesto is a **document** that identifies values and principles
- ⌚ Its authors believe software developers should **use to guide** their work.
- ⌚ Formally called the *Manifesto for Agile Software Development*

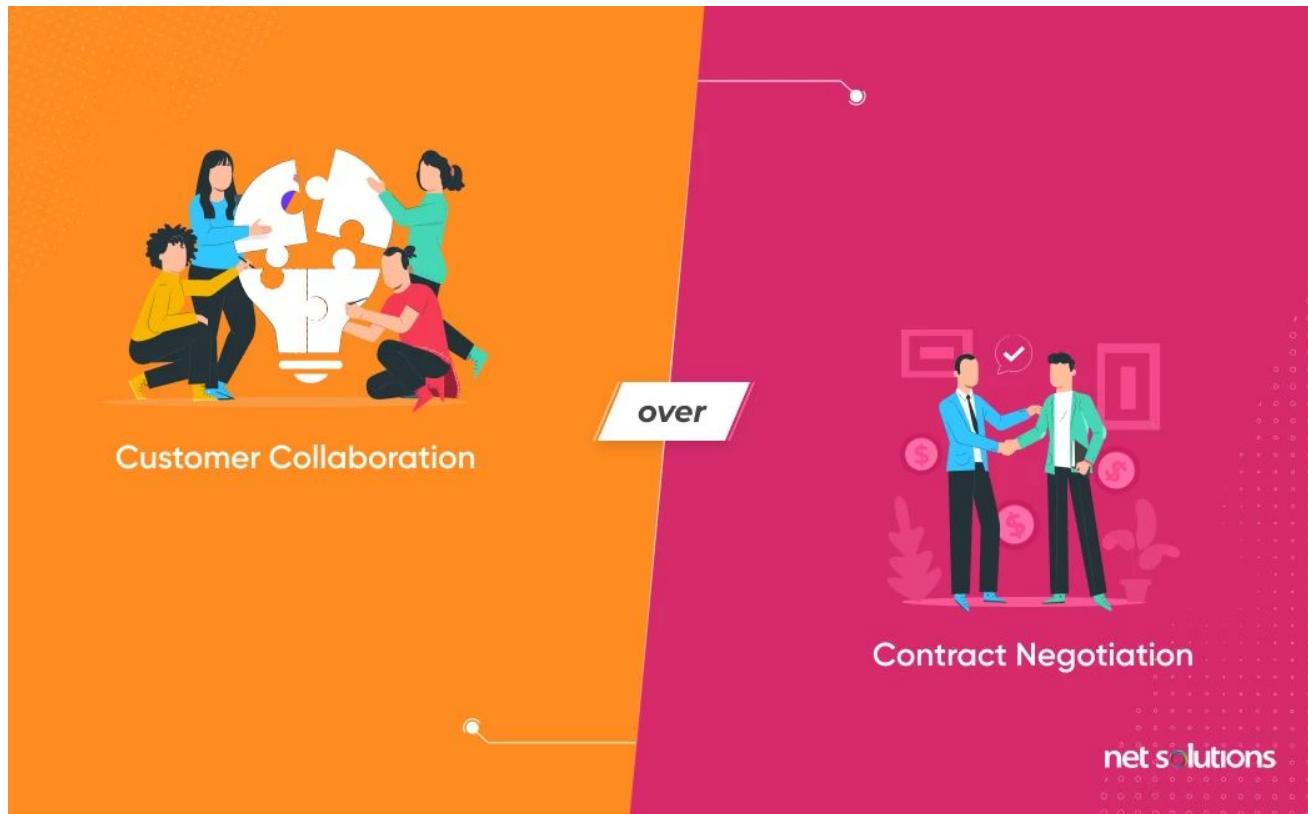
Individuals and interaction over processes and tools



Working software over comprehensive documentation

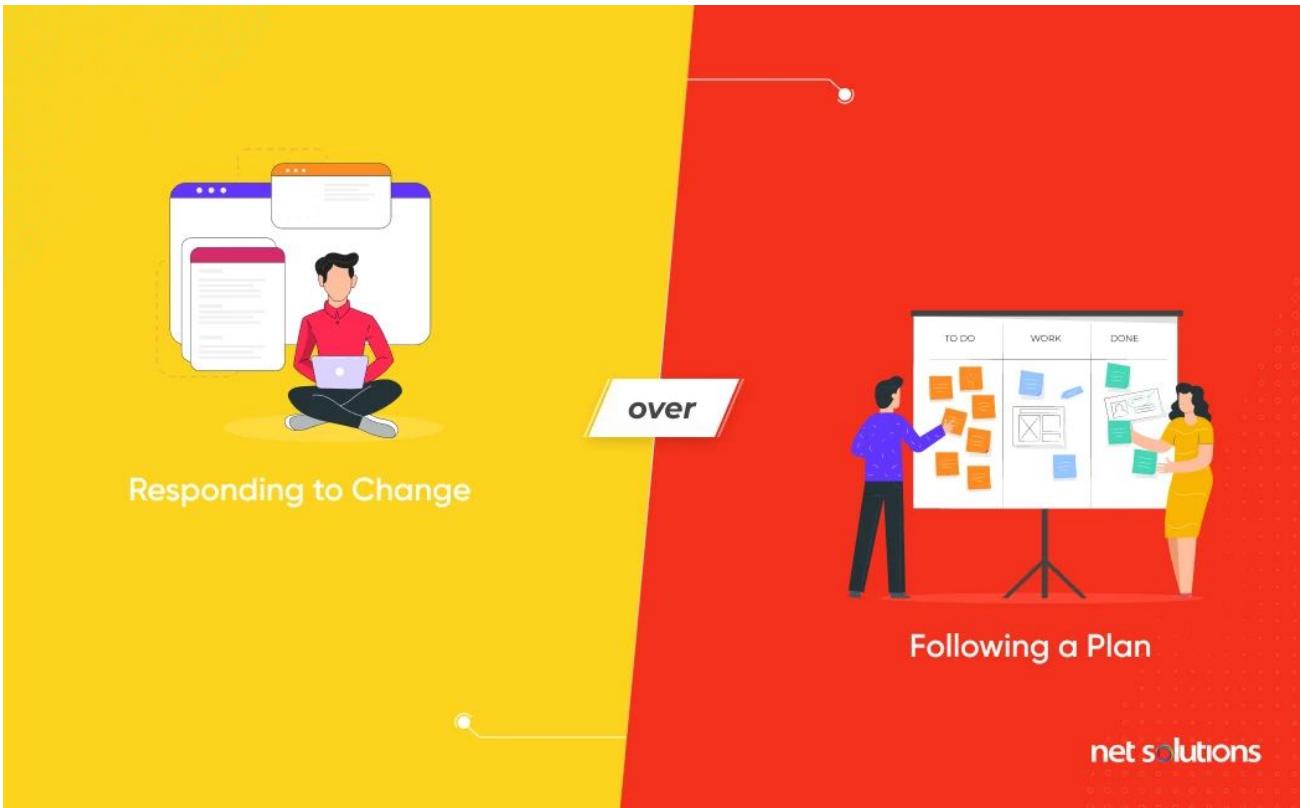


Customer collaboration over contract negotiation



net solutions

Responding to change over following a plan

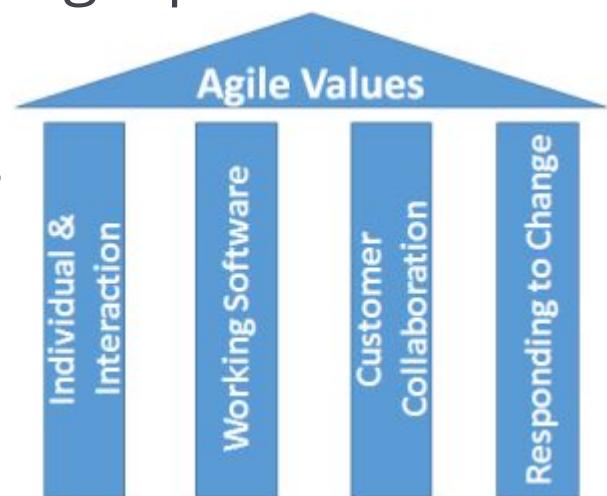


net solutions

Agile manifesto - value

- ▶ Individuals and **interactions** over processes and tools
- ▶ **Working software** over comprehensive documentation
- ▶ **Customer collaboration** over contract negotiation
- ▶ **Responding to change** over following a plan

▶ These are the base of Agile and we keep balance



12 Agile Principles



Focus on customer satisfaction



Work with a motivated team



Enhance agility



Accept changes



Have face-to-face interactions with the team



Have simplicity



Deliver work frequently



Focus on the working product



Maintain a self-organizing team



Work as a team



Ensure sustainable development



Reflect and update

12 AGILE PRINCIPLES

- | | | | | | |
|-----------|---|-----------|---|-----------|---|
| 01 | Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. | 02 | Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. | 03 | Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. |
| 04 | Business people and developers must work together daily throughout the project. | 05 | Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. | 06 | Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. |
| 07 | Working software is the primary measure of progress. | 08 | The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. | 09 | Continuous attention to technical excellence and good design enhances agility. |
| 10 | Simplicity – the art of maximizing the amount of work not done – is essential. | 11 | The best architectures, requirements, and designs emerge from self-organizing teams. | 12 | At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. |

12 AGILE PRINCIPLES

01 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

02 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive

03 Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale

1. Early and continuous delivery of valuable software

together daily throughout the project.

them the environment and support they need, and trust them to get the job done.

sponsors, developers, and users should be able to maintain a constant pace indefinitely.

07 Working software is the primary measure of progress.

08 The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

09 Continuous attention to technical excellence and good design enhances agility.

10 Simplicity – the art of maximizing the amount of work not done – is essential.

11 The best architectures, requirements, and designs emerge from self-organizing teams.

12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

12 AGILE PRINCIPLES

01 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

02 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

03 Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale.

04 Business people and developers must work together daily throughout the project.

2. Welcome Changing Requirements

them to get the job done.

maintain a constant pace indefinitely.

07 Working software is the primary measure of progress.

08 The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

09 Continuous attention to technical excellence and good design enhances agility.

10 Simplicity – the art of maximizing the amount of work not done – is essential.

11 The best architectures, requirements, and designs emerge from self-organizing teams.

12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

12 AGILE PRINCIPLES

01 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

04 Business people and developers must work together daily throughout the project.

07 Working software is the primary measure of progress.

10 Simplicity – the art of maximizing the amount of work not done – is essential.

02 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

05 Build projects around motivated individuals who will do the best work they can in a environment where they have the support they need, and trust them to get the job done.

08 The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

11 The best architectures, requirements, and designs emerge from self-organizing teams.

03 Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale.

users should be able to maintain a constant pace indefinitely.

09 Continuous attention to technical excellence and good design enhances agility.

12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

3. Deliver working software frequently

12 AGILE PRINCIPLES

01 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

02 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

03 Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale.

04 Business people and developers must work together daily throughout the project.

05 Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

06 Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace

4. Business people and developers must work together

Information and training
development team is face-to-face conversation.

10 Simplicity – the art of maximizing the amount of work not done – is essential.

11 The best architectures, requirements, and designs emerge from self-organizing teams.

12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

12 AGILE PRINCIPLES

01 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

02 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

03 Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale.

04 Business people and developers must work together daily throughout the project.

05 Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

06 Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace.

07 Working software is the primary measure of progress.

5. Build projects around motivated individuals

development team is face-to-face conversation.

10 Simplicity – the art of maximizing the amount of work not done – is essential.

11 The best architectures, requirements, and designs emerge from self-organizing teams.

12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

12 AGILE PRINCIPLES

- | | | | | | |
|-----------|---|-----------|---|---|---|
| 01 | Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. | 02 | Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. | 03 | Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale. |
| 04 | Business people and developers must work together daily throughout the project. | 05 | Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. | 06 | Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. |
| 07 | Working software is the primary measure of progress. | 08 | The most effective means of conveying information is face-to-face conversation. | 6. Promote sustainable development | |
| 10 | Simplicity – the art of maximizing the amount of work not done – is essential. | 11 | The best architectures, requirements, and designs emerge from self-organizing teams. | 12 | At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. |

12 AGILE PRINCIPLES

- | | | | | | |
|-----------|---|-----------|---|-----------|---|
| 01 | Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. | 02 | Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. | 03 | Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale. |
| 04 | Business people and developers must work together daily throughout the project. | 05 | Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. | 06 | Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. |
| 07 | Working software is the primary measure of progress. | 08 | The most efficient and effective method of conveying information to and within a | 09 | Continuous attention to technical excellence and good design enhances agility. |

7. Working software to measure the progress

work not done – is essential.

emerge from self-organizing teams.

ular intervals, the team reflects on how to become more effective, then tunes a adjusts its behavior accordingly.

12 AGILE PRINCIPLES

- | | | | | | |
|-----------|---|-----------|---|-----------|---|
| 01 | Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. | 02 | Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. | 03 | Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale. |
| 04 | Business people and developers must work together daily throughout the project. | 05 | Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. | 06 | Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. |
| 07 | Working software is the primary measure of progress. | 08 | The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. | 09 | Continuous attention to technical excellence and good design enhances agility. |
| 10 | Simplicity – the art of maximizing the amount of work not done – is essential. | | | | |

8. Face-to-face conversation

accordingly.

ear
e
a

12 AGILE PRINCIPLES

01 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

02 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

03 Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

04 Business people and developers must work together daily throughout the project.

05 Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

06 Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

07 Working software is the primary measure of progress.

08 The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

09 Continuous attention to technical excellence and good design enhances agility.

10 Simplicity – the art of maximizing the amount of work not done – is essential.

11 The best architecture, requirements, and designs emerge from self-organizing teams.

more effective, then tunes and adjusts its behavior accordingly.

9. Technical excellence and good design.

12 AGILE PRINCIPLES

01 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

02 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

04 Business people and developers must work together daily throughout the project.

05 Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

07 Working software is the primary measure of progress.

08 The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

10 Simplicity – the art of maximizing the amount of work not done – is essential.

11 The best architectures, requirements, and designs emerge from self-organizing

03 Deliver working software frequently, from a couple of weeks to a couple of months with a preference to the shorter timescale.

06 Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

09 Continuous attention to technical excellence and good design enhances agility.

10. Maximizing the amount of work not done.

12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

12 AGILE PRINCIPLES

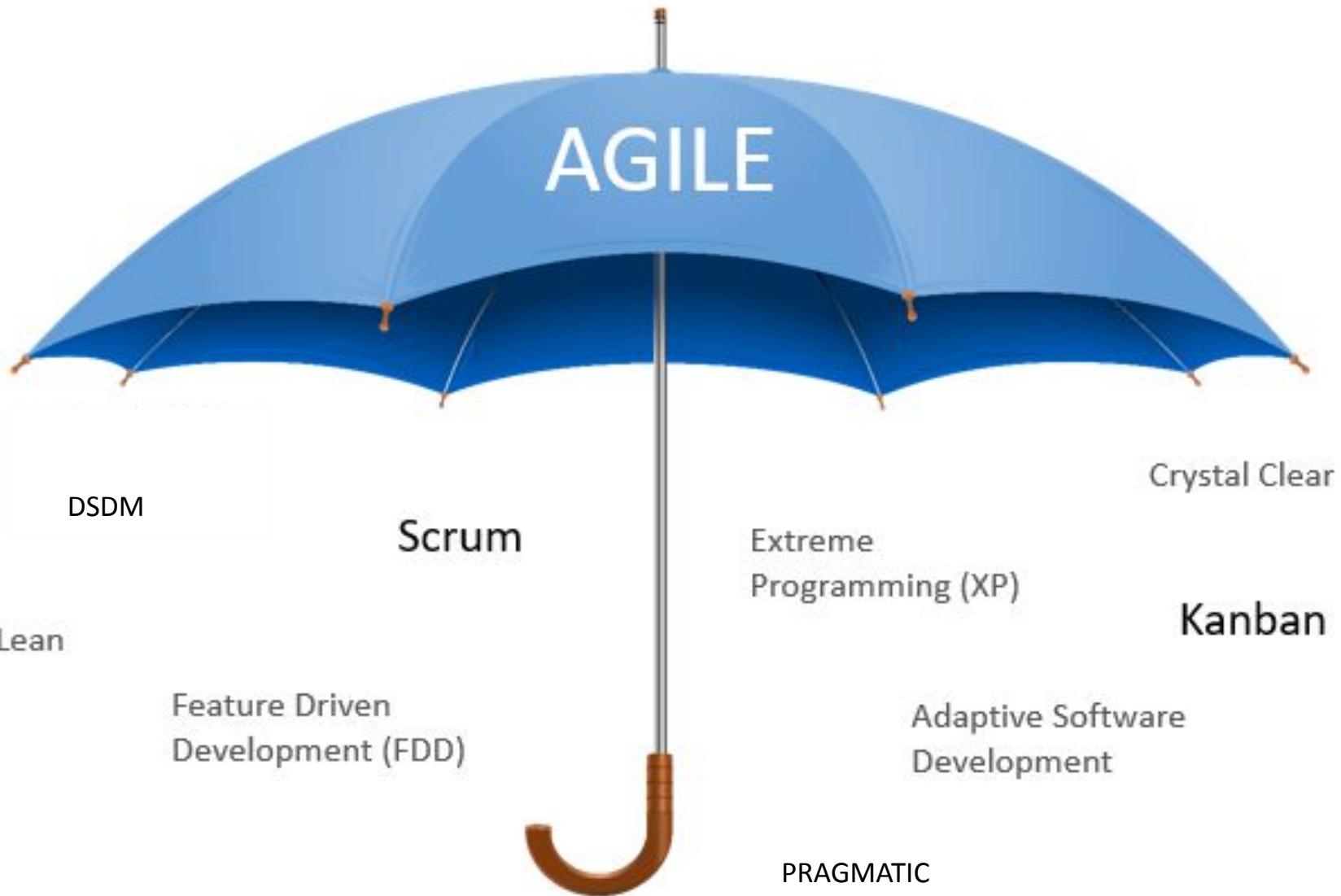
- | | | | | | |
|-----------|---|-----------|---|-----------|---|
| 01 | Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. | 02 | Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. | 03 | Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. |
| 04 | Business people and developers must work together daily throughout the project. | 05 | Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. | 06 | Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. |
| 07 | Working software is the primary measure of progress. | 08 | The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. | 09 | Continuous attention to technical excellence and good design enhances agility. |
| 10 | Simplicity – the art of maximizing the amount of work not done – is essential. | 11 | The best architectures, requirements, and designs emerge from self-organizing teams. | 12 | At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior. |

11. Self-organizing teams

12 AGILE PRINCIPLES

- | | | | | | |
|-----------|---|-----------|---|-----------|---|
| 01 | Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. | 02 | Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. | 03 | Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. |
| 04 | Business people and developers must work together daily throughout the project. | 05 | Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. | 06 | Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. |
| 07 | Working software is the primary measure of progress. | 08 | The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. | 09 | Continuous attention to technical excellence and good design enhances agility. |
| 10 | Simplicity – the art of maximizing the amount of work not done – is essential. | 11 | The best architectures, requirements, and designs emerge from self-organizing teams. | 12 | At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior. |

Agile development techniques



The most widely-used Agile methodologies

- ❖ **Agile Scrum Methodology**
- ❖ Lean Software Development
- ❖ Kanban
- ❖ **Extreme Programming (XP)**
- ❖ Crystal
- ❖ Dynamic Systems Development Method (DSDM)
- ❖ Feature Driven Development (FDD)

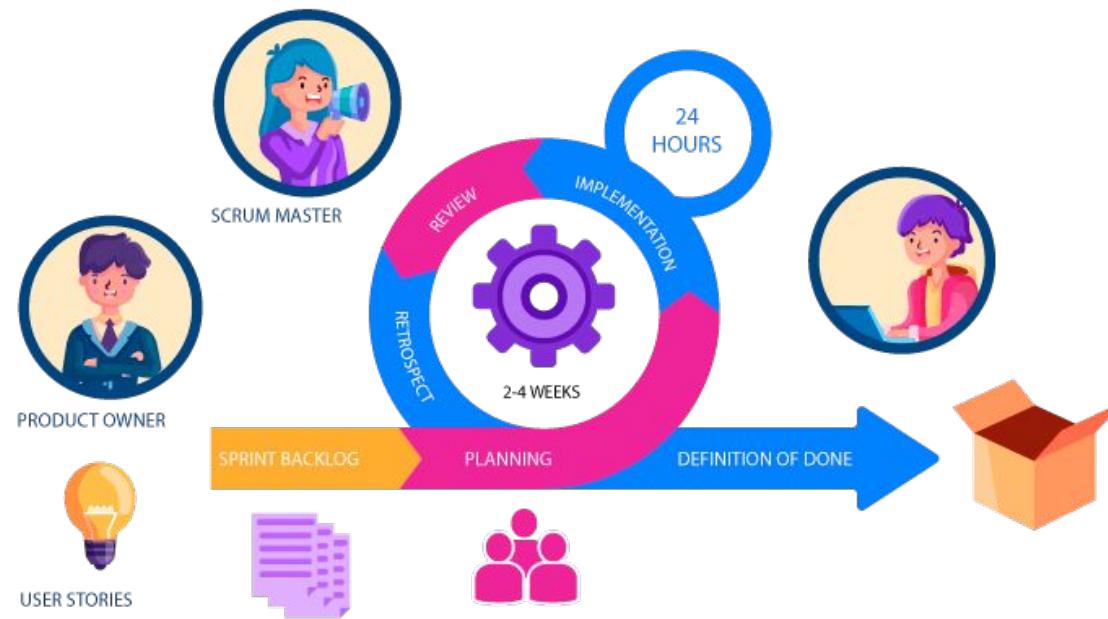
SCRUM

- ❖ Scrum is a **lightweight management framework** with broad applicability for managing and controlling iterative and incremental projects of all types
- ❖ Very popular one
- ❖ Scrum has been proven to scale to multiple teams across very large organizations.

Extreme programming (XP)

- ❖ Promotes high customer involvement, rapid feedback loops, continuous testing, continuous planning, and close teamwork to deliver working software at very frequent intervals, typically every 1-3 weeks, 40hr week.
- ❖ It primarily focuses on **Engineering practices**

SCRUM METHODOLOGY

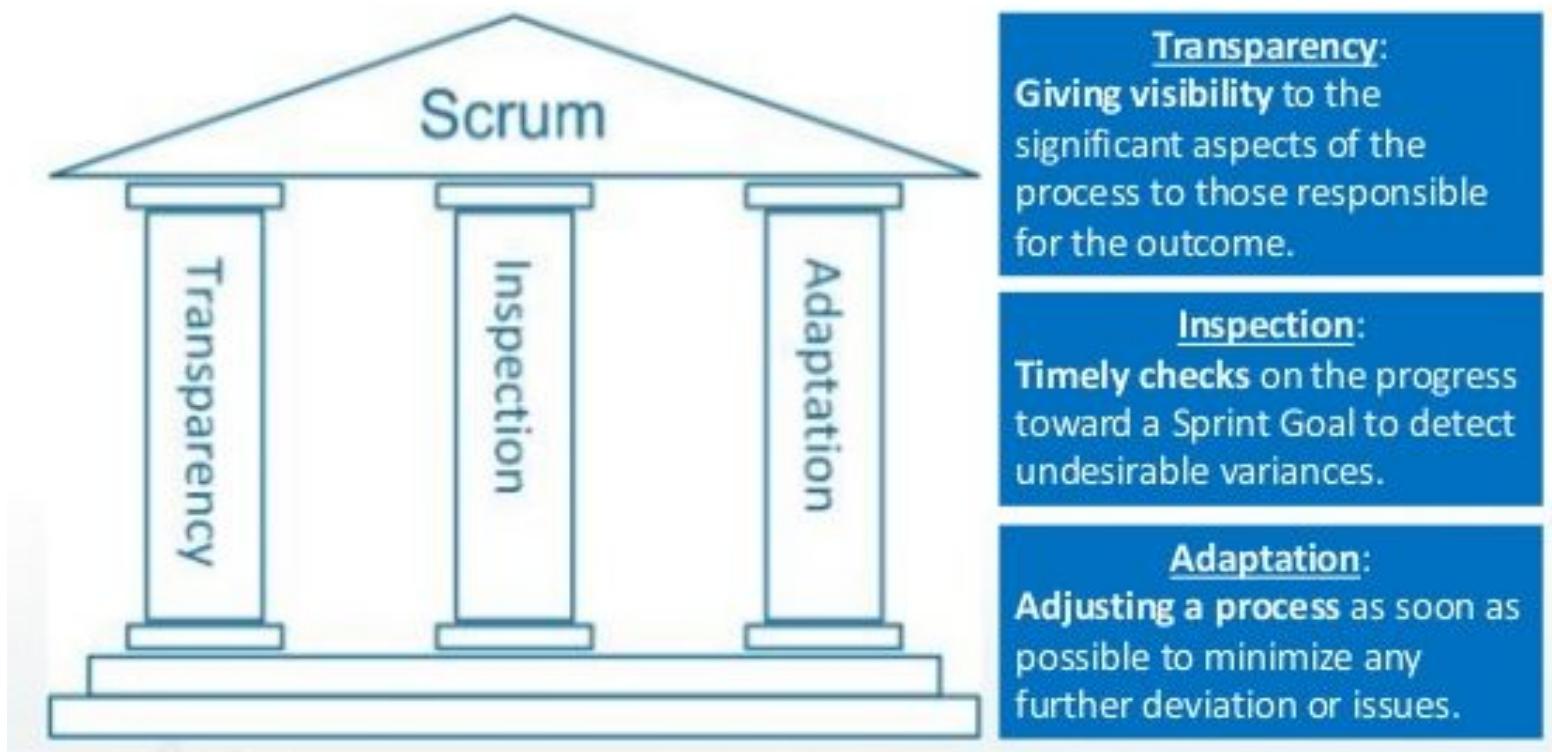


What is Scrum?

- ⌚ Scrum is an Agile framework for completing complex projects.
- ⌚ An iterative incremental process for software development.
- ⌚ An approach that controls the chaos of changing requirements.
- ⌚ Team based development.
- ⌚ Whole team travel as a unit to achieve the task.



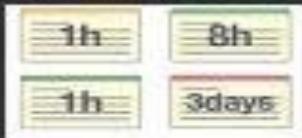
Scrum Three pillars



Scrum



PRODUCT BACKLOGS



RELEASE BACKLOGS



SPRINTS



TEAM ROLES

This is the most popular Scrum video of all time.



Great for an introduction to the Scrum process or for a quick refresher.



BURNDOWN CHARTS



DAILY STANDUPS

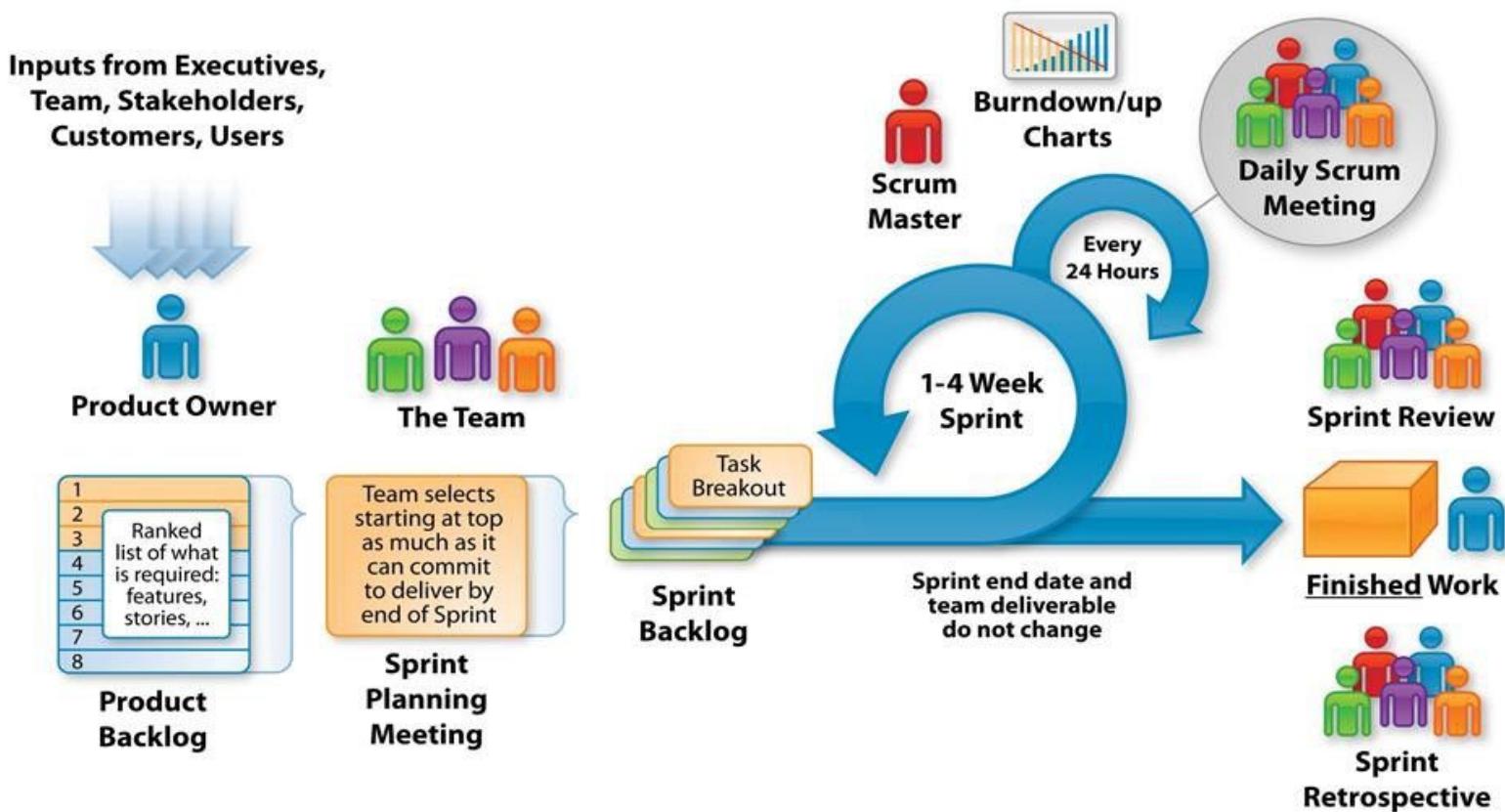


RETROSPECTIVES



ESTIMATIONS

Scrum



The Scrum Roles

- ❖ Product owner
- ❖ Scrum master
- ❖ Developer
- ❖ Tester
- ❖ Customer
- ❖ Executives

The Scrum Roles



Product Owner

Owns the backlog, strives to prioritize, and takes all the key product decisions. This person turns all user requirements into actionable work for the development team.



The Development Team

Cross-functional team of developers, QA testers, designers, and other technical members who are needed for development of the product.

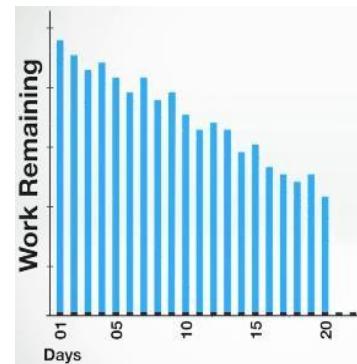
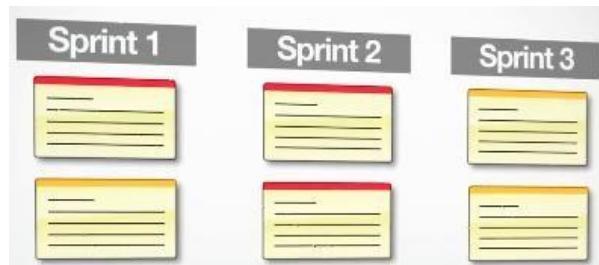
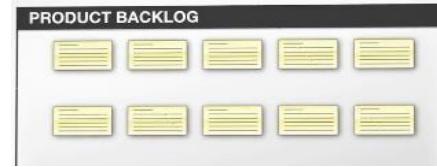


Scrum Master

Ensures the entire team has everything they need to ship a user story on time. This person communicates progress and ensures there aren't any roadblocks.

User Story:

As a (role), I want (feature),
so that (benefit).



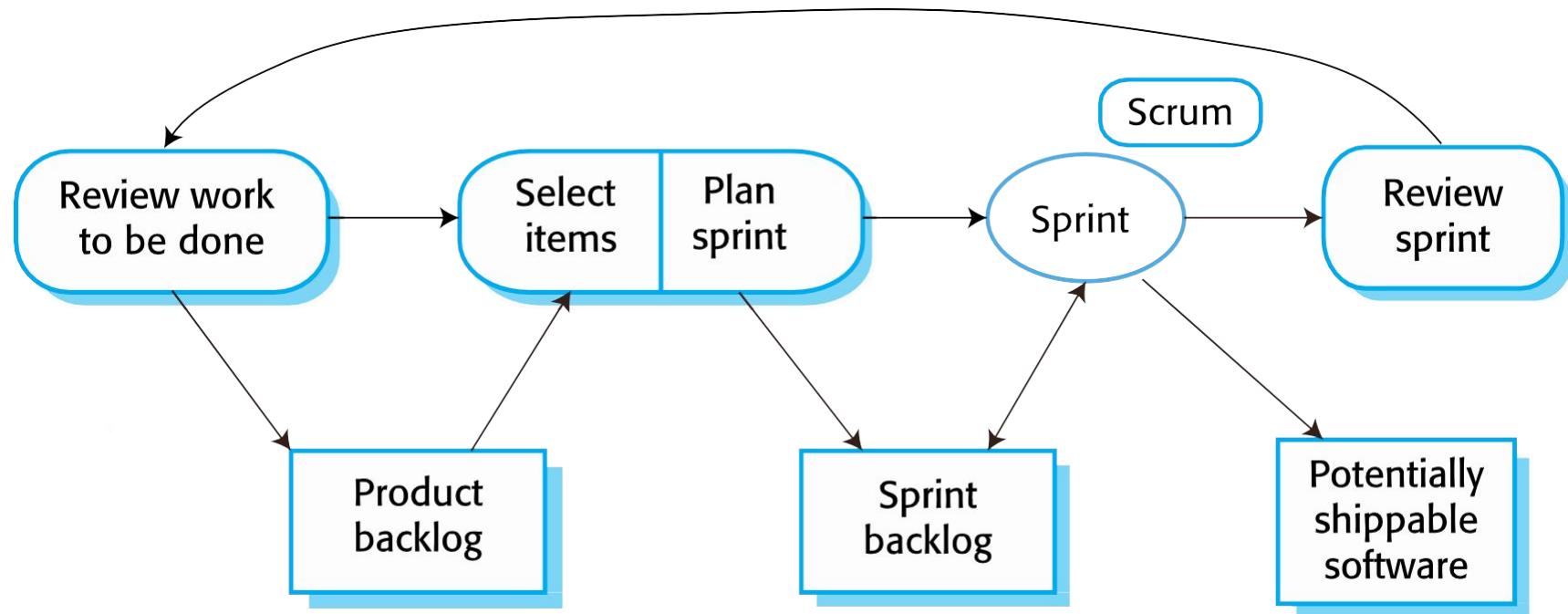
Scrum terminology

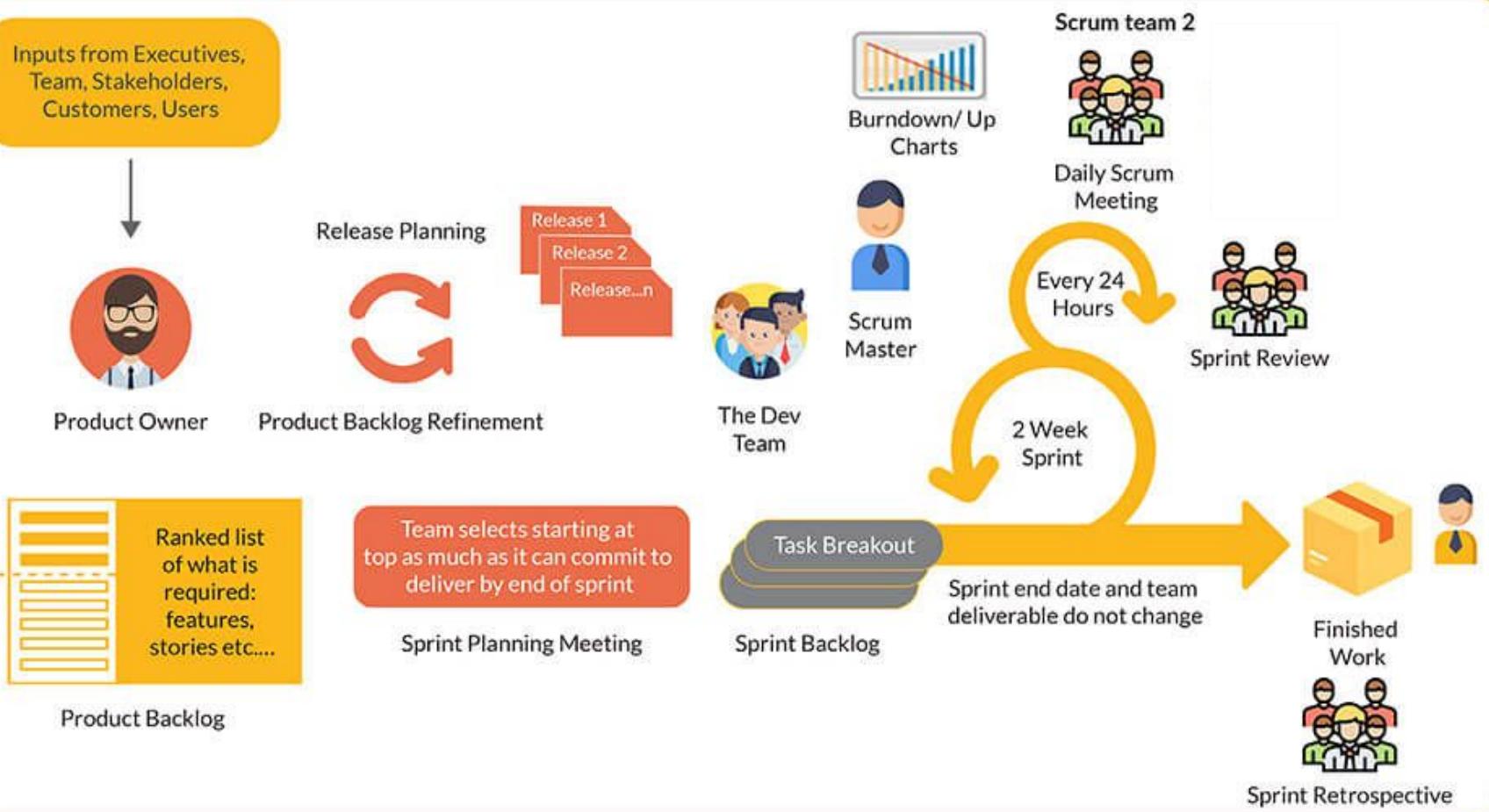
Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a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

Scrum terminology

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

Scrum sprint cycle





Daily Scrum Meeting

↻ Goal

- Enable the Team to share progress with each other
- Identify what's blocking or slowing down the Team (not problem solving)
- Enable the Team to check and familiarize daily

↻ Team stands in a circle and each member reports only 3 things

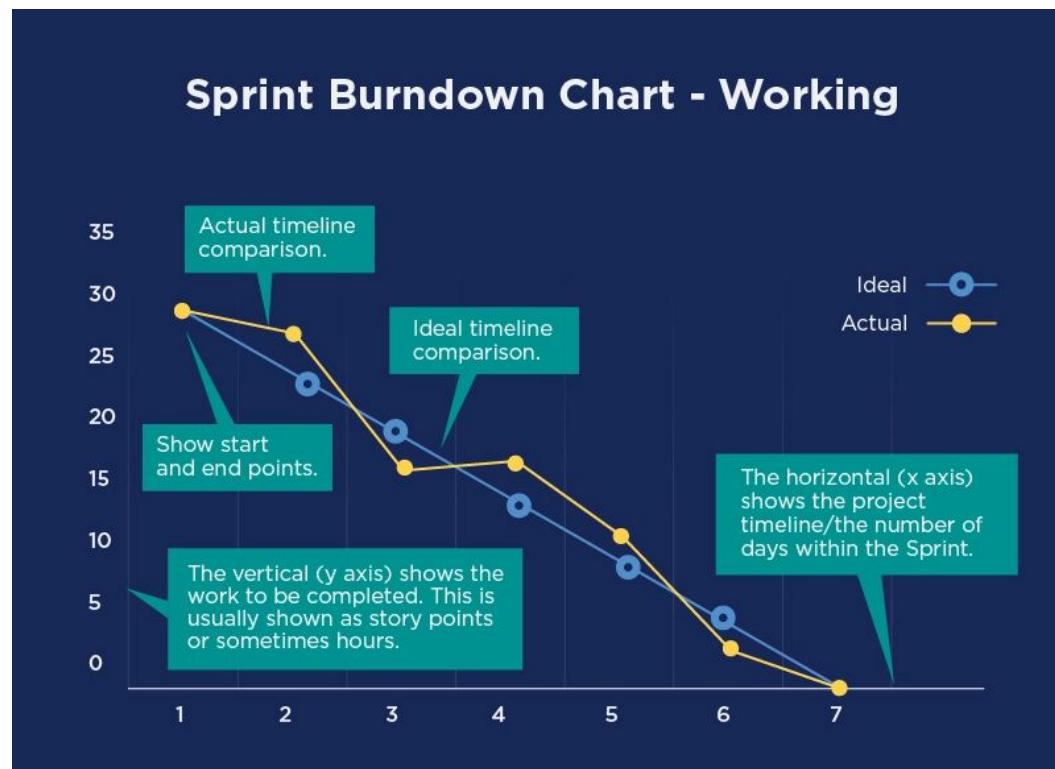
↻ Time-boxed to 15 mins





Burndown chart

- ❖ The Team updates and reviews burn down chart daily
- ❖ Shows teams progress through the Sprint



Phases

❖ Pre-game

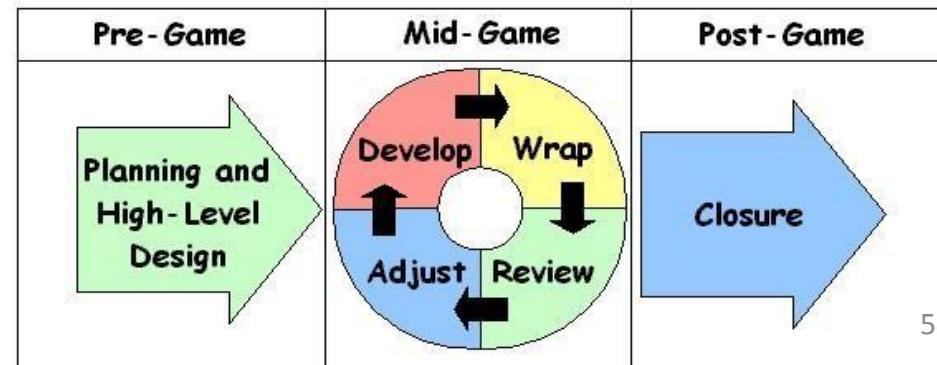
- Planning – define system/Product Backlog
- Architecture - high level design of system

❖ Sprints

- Iterative cycles (sprints) - new or enhanced functionality

❖ Post-game

- Requirements satisfied - no new items or issues



Scrum sprints

- ❖ Scrum consists of a series of Sprints (2-4 weeks)
 - Each Sprint involves:
 - Sprint planning - *The purpose of **sprint planning** is to define what can be delivered in the **sprint** and how that work will be achieved*
 - Sprint preparation
 - Daily Activities
 - Sprint Demo
 - Sprint Retrospective – a recurring meeting dedicated to discussing what went well and what can be improved in a sprint. It also gives a chance to recover from a sprint and prepare for the next one.



SCRUM PROCESS

SPRINT 0



SPRINT CYCLE



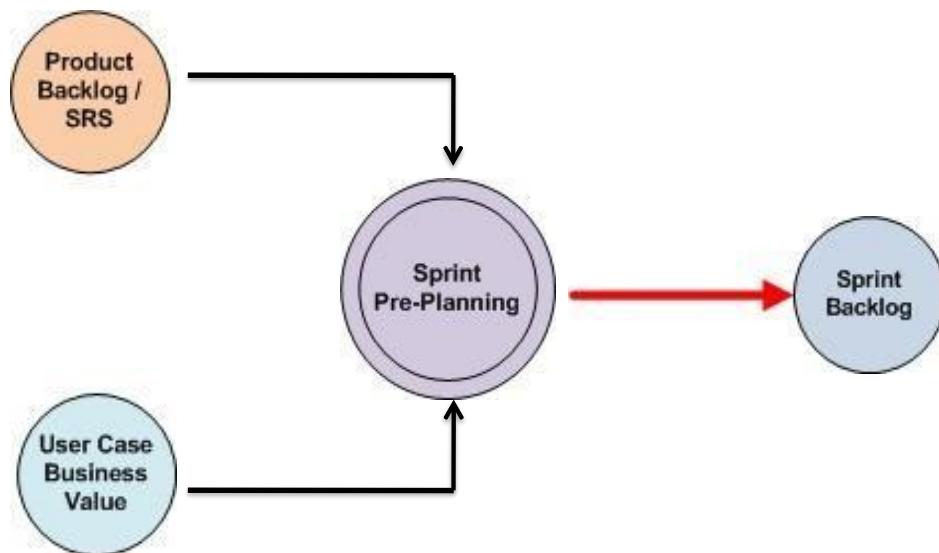
SPRINT PRE-PLANNING

Sprint 0

- ❖ Do initial planning and understand the project in terms of business and technical requirement
- ❖ Create setup for the project base
- ❖ Activities :
 - Understand requirement
 - Document, review and refine requirement
 - Estimate effort
- ❖ Deliverables:
 - SRS
 - High level architecture
 - Deployment model

Sprint Pre-Planning

Sprint Pre-Planning involves **prioritizing requirements** so that the **most important Business Value** is **delivered early** in the form of a working software.



The Scrum sprint cycle

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

The Sprint cycle

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

Teamwork in Scrum

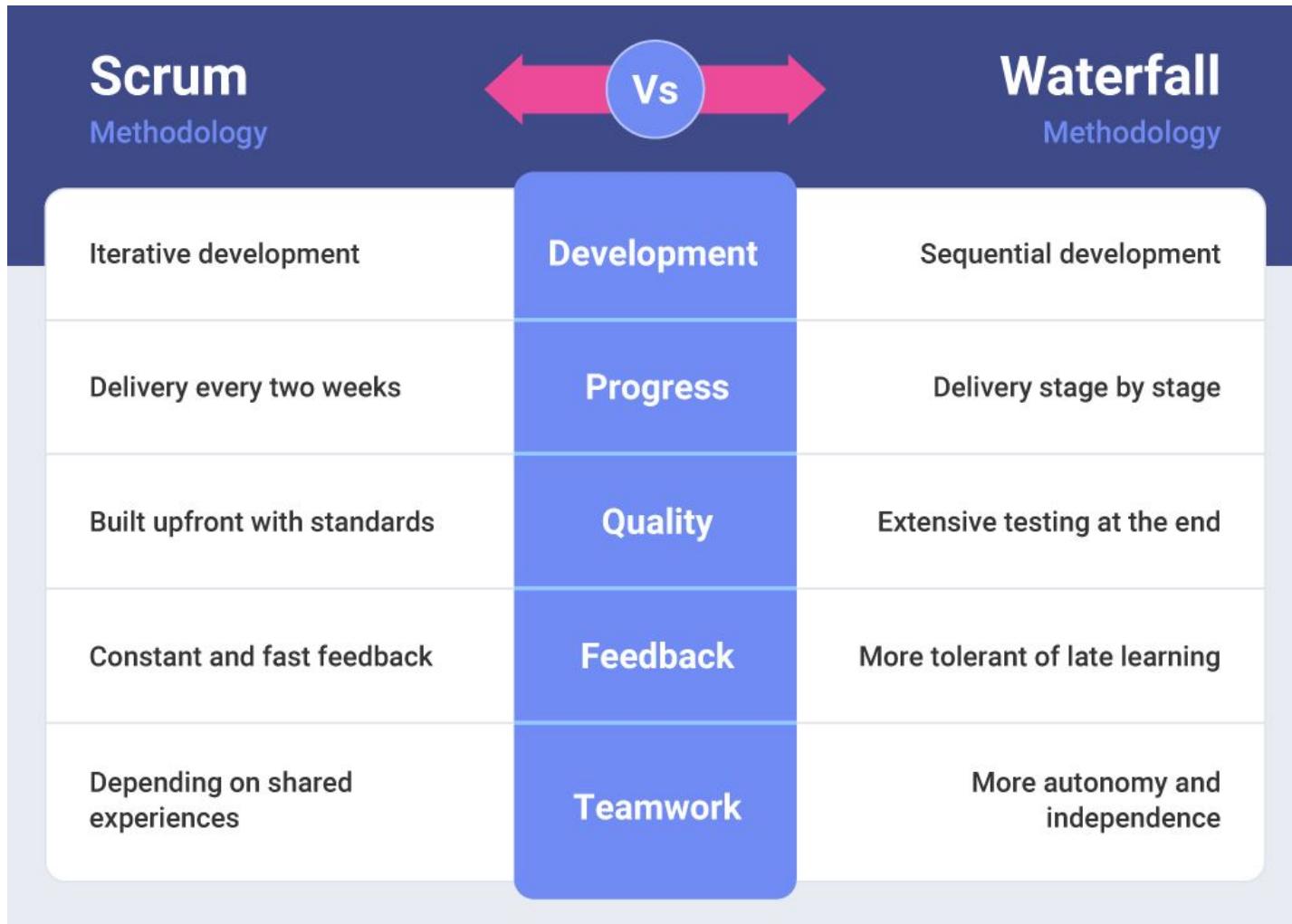
- ❖ The '**Scrum master**' is a **facilitator** who arranges **daily meetings**, tracks the backlog of **work to be done**, **records decisions**, **measures progress** against the backlog and communicates with customers and management outside of the team.
- ❖ The whole team attends short daily meetings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

Scrum benefits

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

Scrum pitfalls

- ❖ The wrong product owner
- ❖ Incorrect organized team
- ❖ Do scrum without having a understand
- ❖ In sufficient co-operation
- ❖ Too little documents





Extreme programming

Extreme programming

- ❖ Widely used methodology
- ❖ Lightweight mechanism
- ❖ Emphasize the team work
- ❖ Four factors/values
 - simplicity
 - Communication
 - Feedback
 - Courage

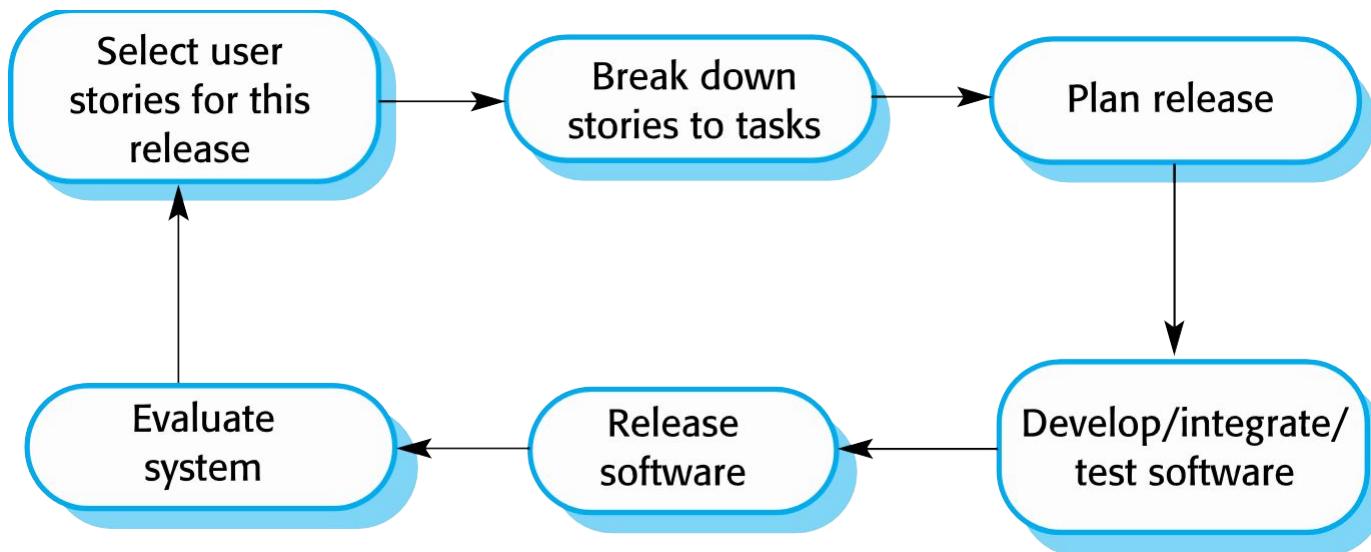
XP Phases

- ❖ Exploration – looking around and try to get aware
- ❖ Planning – set priority to stories and plan 1st release
- ❖ Iteration to release
- ❖ Productionizing – extra testing before release
- ❖ Maintenance – no more services are pending

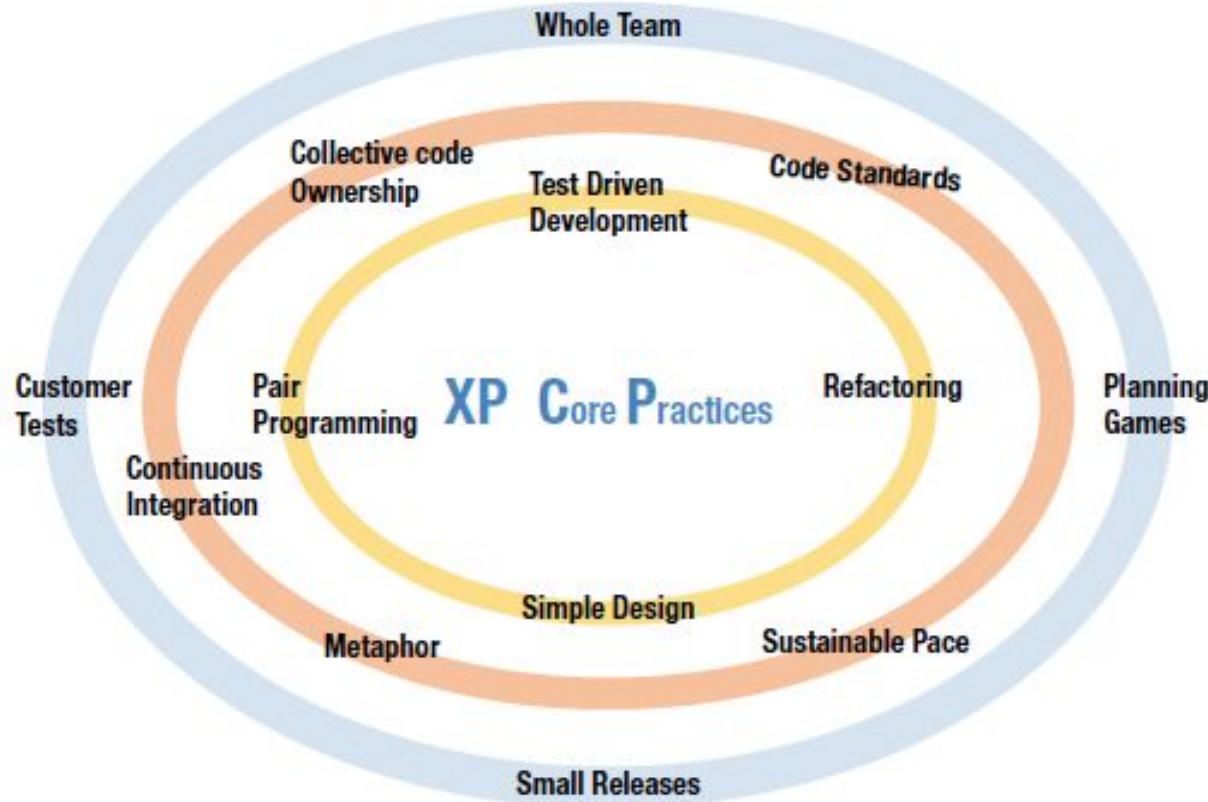
XP Roles

- ❖ **Programmer** - writes tests and then code
- ❖ **Customer** - writes stories and functional tests
- ❖ **Tester** - helps customer write tests and runs them
- ❖ **Tracker** - gives feedback on estimates and process on iterations
- ❖ **Leader**- person responsible for whole process
- ❖ **Consultant** - supplies specific technical knowledge needed
- ❖ **Manager** - makes decisions

The extreme programming release cycle



XP Practices



User Stories

- ⇒ Small, much smaller than other user requirements such as use cases or scenarios
- ⇒ Written by users
- ⇒ 1-3 sentences
- ⇒ Used for estimates
- ⇒ Used to generate acceptance tests (Testable)
- ⇒ There are some challenges
- ⇒ Grouping done as a team
- ⇒ If no. of weeks > 3 then request to split the story

- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Parking passes can be paid via PayPal.
- Professors can input student marks.
- Students can obtain their current seminar schedule.
- Students can order official transcripts.
- Students can only enroll in seminars for which they have prerequisites.
- Transcripts will be available online via a standard browser.

A 'prescribing medication' story

Prescribing medication

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

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

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

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

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

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

Examples of task cards for prescribing medication

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

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

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

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

Small Releases

- A simple system into production early and update frequently in a very small cycles.
- Small in terms of functionality
- Releases of the system are frequent and incrementally add functionality to the first release

Metaphor

- ❖ Explain the things that everyone can understand
- ❖ Common description that mentioned
 - How the system works
 - It's architecture
 - Not rely to the technical things
 - Shared vocabulary

Planning Game

- ❖ Release Planning
 - Define set of features required for the next release
- ❖ Iteration Planning
 - Customer chooses the stories for the iteration and programmers estimate
 - what to do now?
 - What to do next?

40-Hour Week

- ❖ programmers keep a tight schedule
- ❖ 40 hours per week = 8 hours * 5
- ❖ No overtime

On Site Customer

- ❖ The customer stays with XP team (as a team member)
 - write acceptance tests
 - run them
 - answers questions
 - make decisions for programmers
 - accept releases

XP Design

- ❖ KIS Principle (**Keep It Simple**)
- ❖ Design provides the implementation guidelines when stories are written
- ❖ Nothing less and Nothing More
- ❖ Still we have challenges

Simple Design

- ❖ Mean understanding
- ❖ The right design for the software is one that
 - Runs all the tests
 - Has no duplicated logic.
 - States every intention important to the programmers.
 - Has the fewest possible classes and methods

Refactoring

- ❖ Change the internal structure of codes without changing external behavior of class/module
- ❖ Why?
 - Clean code
 - Remove bugs
 - Improve design
- ❖ If size too big, then split it and do it and later combined them

Coding

- ❖ Design + coding goes together
- ❖ Once preliminary design is done, develop unit tests for each story to be included in the release before starting coding
- ❖ Pair programming + Coding standards
- ❖ Integration – part of construction

Pair Programming

- ⬇ Two Developers, One monitor, One Keyboard
- ⬇ Developing the code together
- ⬇ One “drives” and the other thinks
- ⬇ Switch roles as needed



Collective Ownership

- ❖ Whole team owns the code
- ❖ Anybody can change any part they need to change
- ❖ Everyone is responsible for the whole of the system.
- ❖ Increases speed

Continuous Integration

- ❖ If it finish, then combine
- ❖ As soon as the work on a task is complete, it is integrated into the whole system.
- ❖ After any such integration, all the unit tests in the system must pass.

Coding Standard

- ❖ All team members follow a common standard
- ❖ Easy to understand/readable for all members
- ❖ Will facilitate refactoring

Testing

- ❖ Unit Test before developing code
- ❖ Use a framework for automated testing

Extreme programming practices

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

Extreme programming practices

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

WHAT ARE
**THE PROS
AND CONS
OF EXTREME
PROGRAMMING (XP)?**





Fundamentals of Software Engineering

Course Code : CS2002

Ms. Mathangi Krishnathasan
[\(tmk@ucsc.cmb.ac.lk\)](mailto:tmk@ucsc.cmb.ac.lk)

Course Structure

- 3 Credit course
- Assessment plan: Continuous Assignments (Quizzes + In-Class Assignments)
- Course Evaluation :
 - Exam paper: (70%)
 - Assignments: (30%)

Lecture Notes / Reading Materials / Notices: published in LMS

Recommended Reading

- Main Reference: Ian Sommerville, Software Engineering, 10th Edition, Pearson, 2017.
- Ref 2: R Pressman, Software Engineering - A Practitioners Approach, 7th Edition, McGraw Hill.

Course Content

- System Modeling
- Architectural design
- Design and implementation
- Software Testing
- Software Evolution

System Modeling – Part I

CHAPTER - 05

Intended Learning Outcomes

After successfully completing this course, students should be able to:

- LO1. Understand how graphical models can be used to represent software systems
- LO2. Understand why several models are required to fully represent the system.
- LO3. Identify the fundamental system modeling perspectives.
- LO4. Understand the principal diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling.
- LO5. Explain the model-driven engineering, where an executable system is automatically generated system models.

Lesson Outline

- Introduction to System Modeling
- Context Models
- Interaction Models
- Structural Models
- Behavioral Models
- Model-driven Engineering

Introduction to 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.
- Use to represent a system using some kind of graphical notation, which is mostly based on diagram types in the Unified Modeling Language (UML).
- Help the analyst to understand the functionality of the system.
- Models are used to communicate with customers.

System Modeling

- ❑ Models are used during different stages of Software Development Life Cycle.
 - Requirement Engineering - Help to derive the detailed requirements for a system.
 - Designing - Describe the system to engineers implementing the system.
 - After Implementation - Document the system's structure and operation.

System Models

- **Models of the existing system**
 - Used to clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses with stakeholders during requirements engineering.
- **Models of the new system**
 - Used help explain the proposed requirements to system stakeholders.
 - Engineers use these models to discuss design proposals and to document the system for implementation.
- In a **model-driven engineering process**, it is possible to generate a complete or partial system implementation from the system model.

System Models

- System model is not a complete representation of system.
- It purposely leaves out detail to make it easier to understand.
- A model is an abstraction of the system being studied rather than an alternative representation of that system.
- An abstraction deliberately simplifies a system design and picks out the most salient characteristics.

System Perspectives

Different models are used to represent 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 Perspectives

- When developing system models, you can often be flexible in the way that the graphical notation is used. Details of a model depend on how you intend to use it.
- There are three ways in which graphical models are commonly used.
 - Stimulate discussion about an existing or proposed system - The models may be incomplete (as long as they cover the key points of the discussion), and they may use the modeling notation informally.
 - Document the existing system - They do not have to be complete, as you may only need to use models to document some parts of a system. However, these models have to be correct - they should use the notation correctly
 - Detailed system description that can be used to generate a system implementation – The models are used as part of a model-based development process, the system models have to be both complete and correct.

Unified Modeling Language [UML]

- The UML is universally accepted as the standard approach for developing models of software systems.
- Diagrams defined in the Unified Modeling Language which has become a standard language for object-oriented modeling
- The UML has 13 diagram types and so supports the creation of many different types of system model. But 5 diagram types are commonly used.

UML Diagram Types

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

Context Models

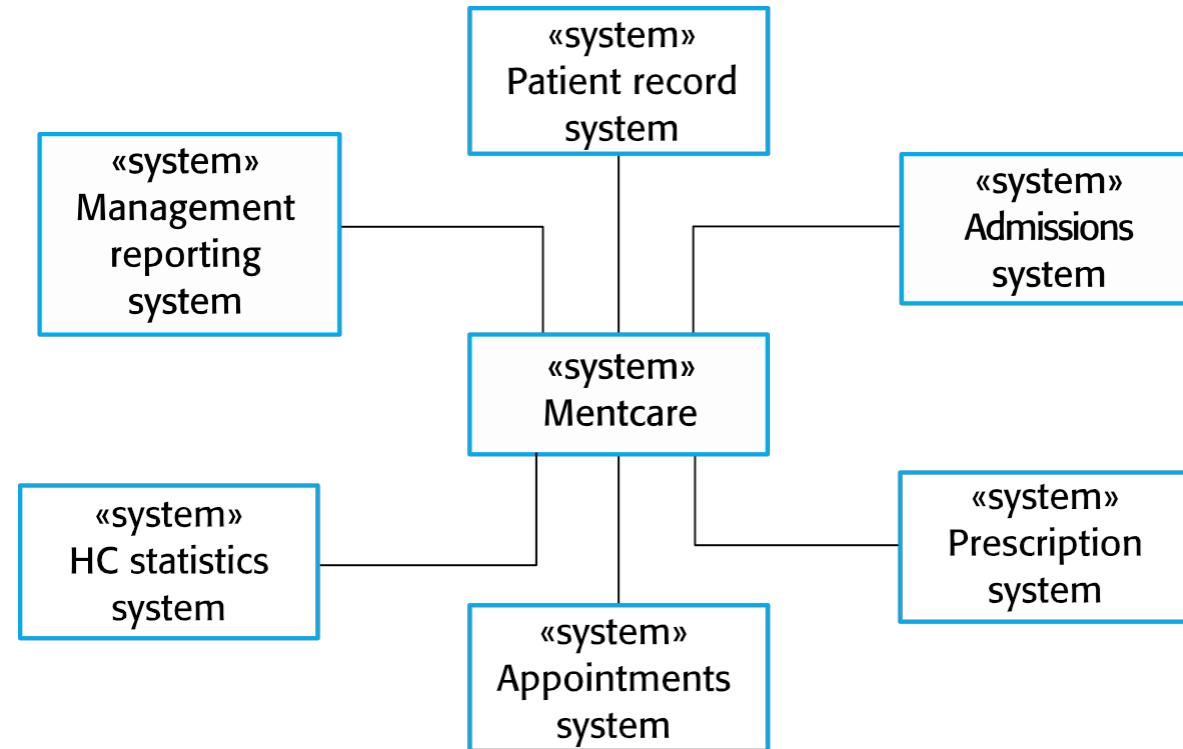
Context Models

- Context models are used to illustrate the operational context of a system - they show what is and is not part of the system being developed.
- At an early stage in the specification of a system, you should decide on the system boundaries, that is, on what is and is not part of the system being developed.
- This involves working with system stakeholders to decide what functionality should be included in the system and what processing and operations should be carried out in the system's operational environment.

System Boundaries

- It is used to decide on the processes that should be implemented in the software being developed and the processes that are manual or support by different systems.
- It helps to look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented.
- These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.
- In some cases, boundary between the system and its environment is clear and in other cases there is more flexibility.
- Social and organizational concerns may affect the decision on where to position system boundaries.

The Context of the Mentcare System

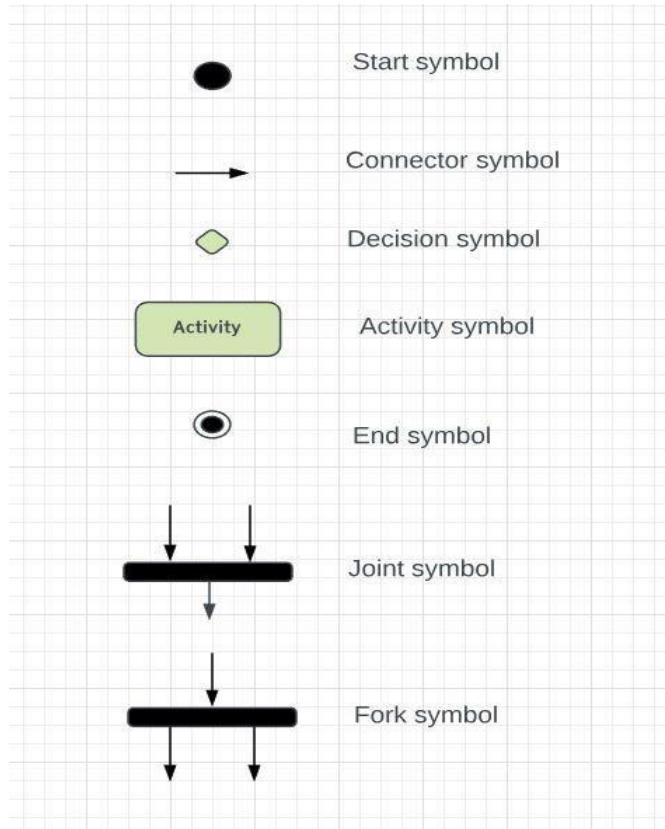


Context Models and Process Models

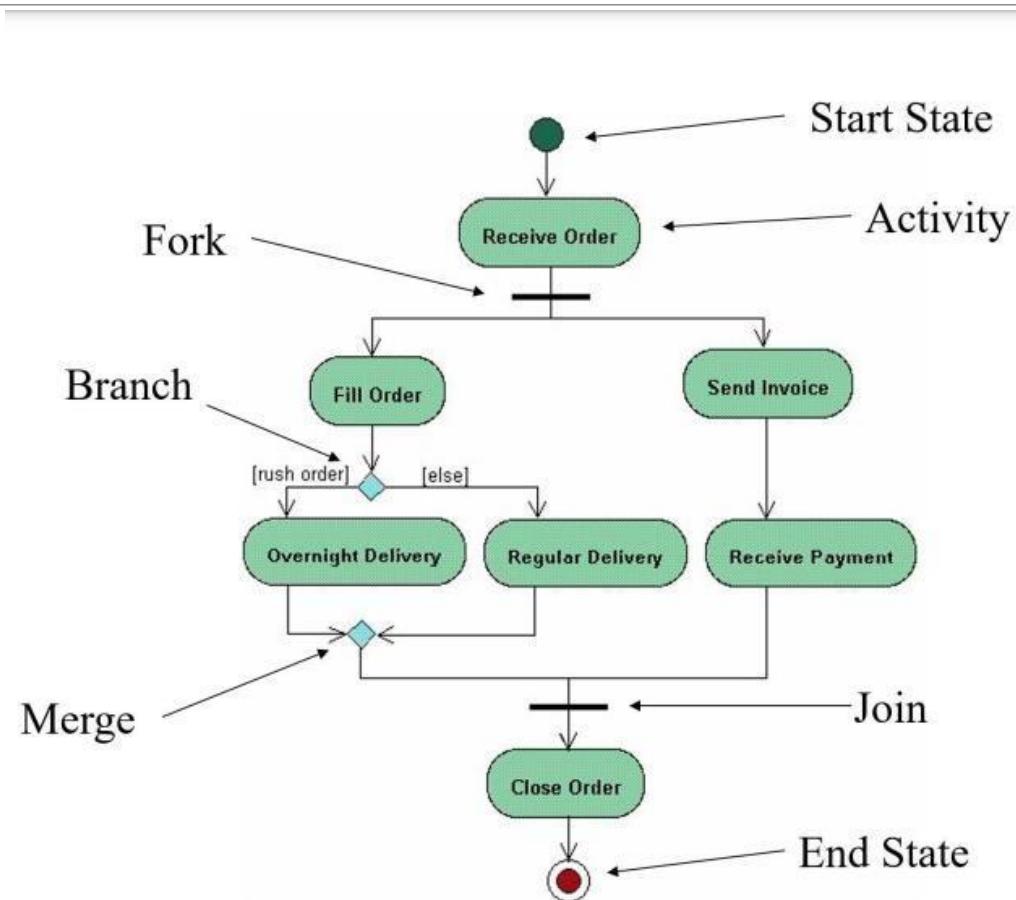
- Context models show environment including several other automated systems.
- They don't show relationships between the systems in the environment and the system that is being specified.
- Simple context models are used along with other models, such as business process models to describe the human and automated process in which the particular systems are used.
- UML activity diagrams may be used to show business process in which systems are used.

Activity Diagram

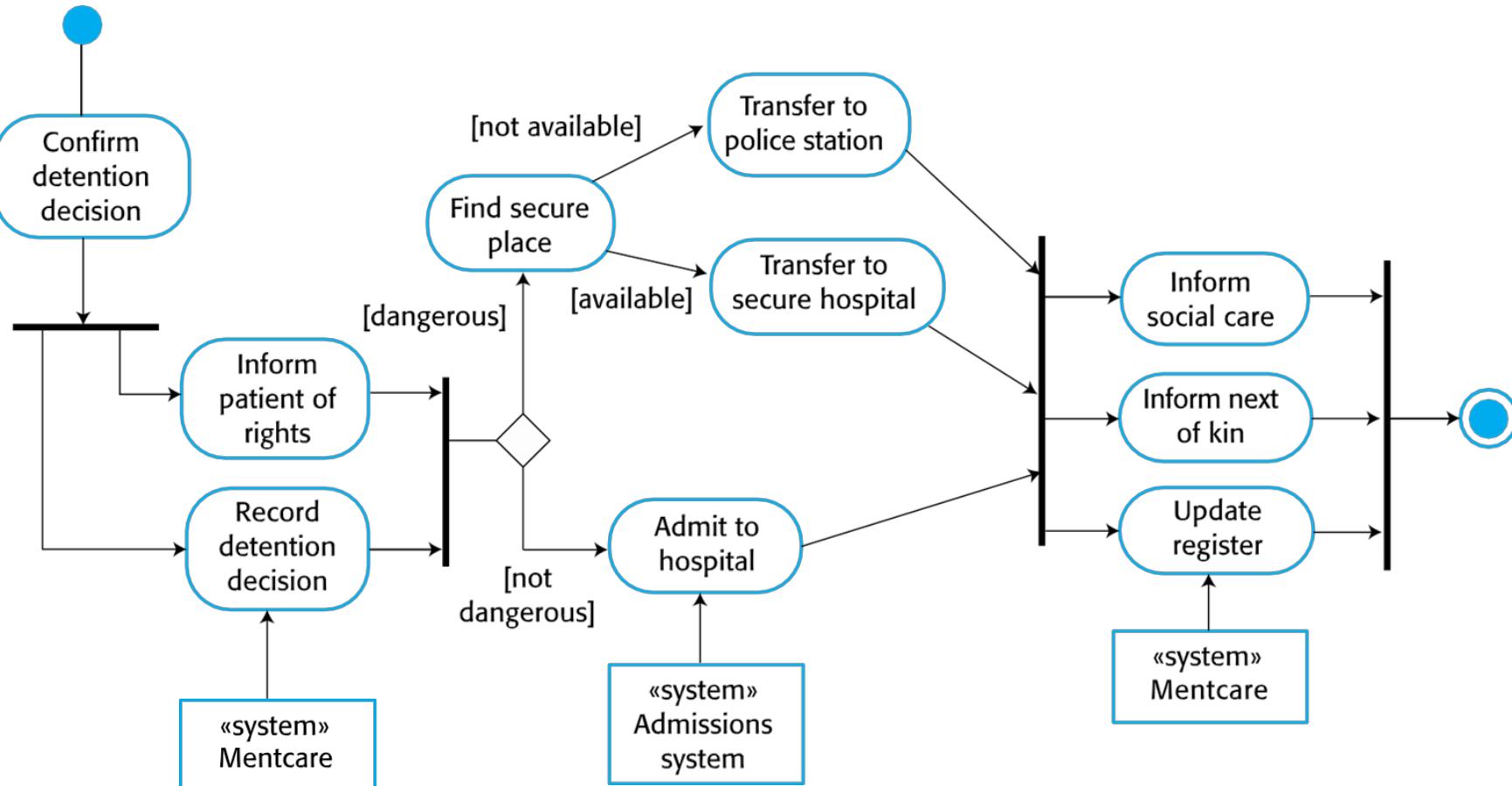
Activity Diagrams show the activities in a process and the flow of control from one activity to another. They don't show relationships between the systems in the environment.



Activity Diagram - Example



A process model of involuntary detention



Interaction Models

Interaction Models

- All system involve in different kind of interaction.
- User interaction which involves user inputs and outputs, interaction between the software being developed and other systems in its environment, or interaction between the components of a software system.
 - Modeling user interaction is important as it helps to identify user requirements.
 - Modeling system to system interaction highlights the communication problems that may arise.
 - Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

Interaction modeling

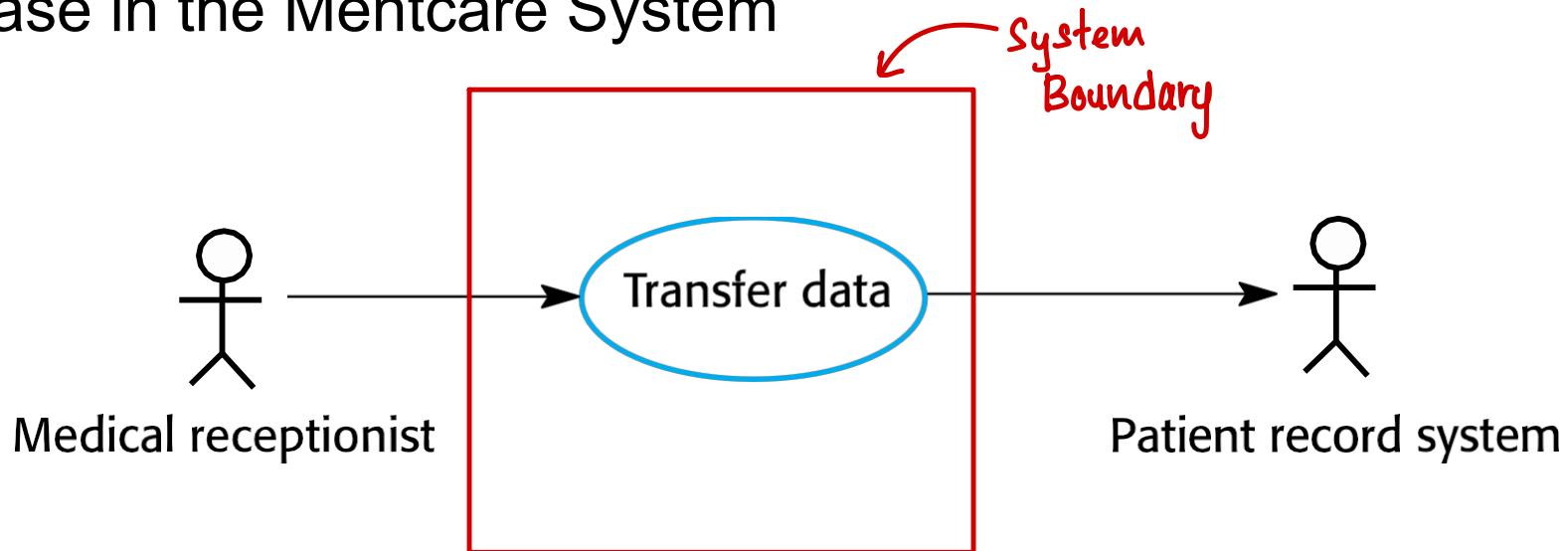
- There are two approaches to interaction modeling.
 - Use case modeling : Used to model interaction between a system and external agents(human users or other systems).
 - Sequence Diagrams :Used to model interaction between system components, although external agents may also be included.
- Use case diagrams and sequence present interaction at different levels of detail and so it will be used together.

Use case modeling

- A use case can be taken as a simple description of what a user expect from a system in a specific interaction.
- Each use case represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.
- Use case diagrams give a simple overview of an interaction, and you need to add more detail for complete interaction description. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram.

Transfer-data use case

A use case in the Mentcare System



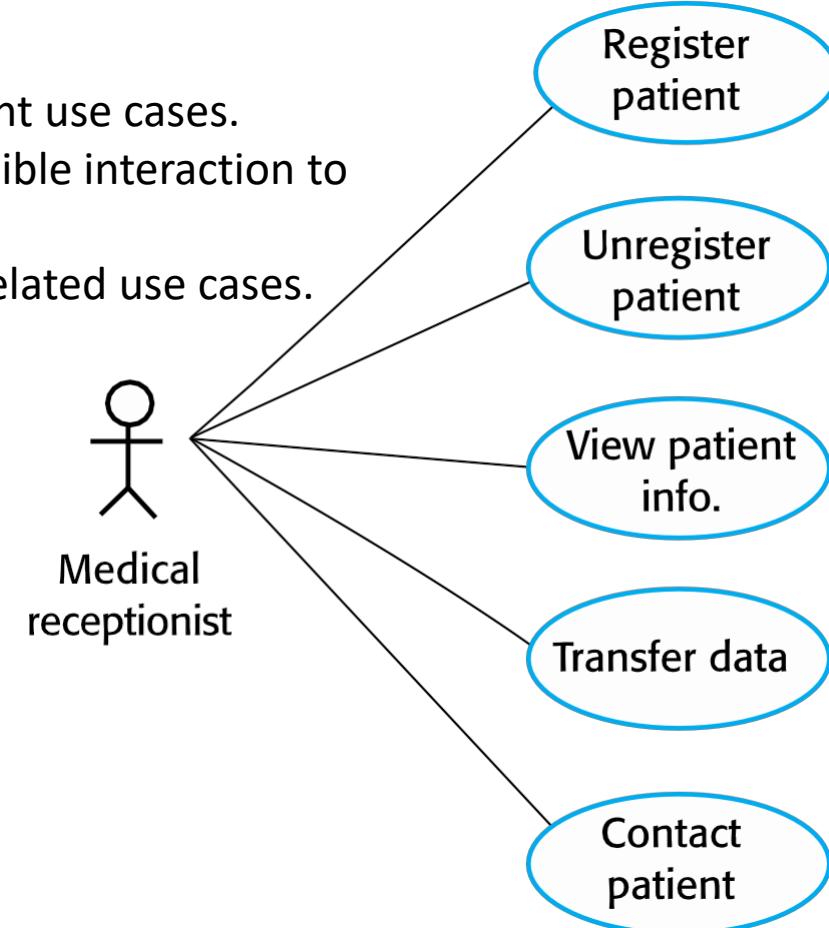
Tabular description of the Transfer-data use case

A use case in the Mentcare System

Mentcare system: Transfer data	
Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Use cases in the Mentcare system involving the role 'Medical Receptionist'

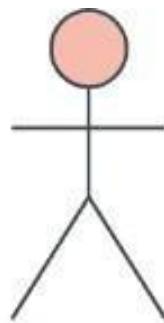
Composite use case diagrams show a number of different use cases.
But sometimes , it may be impossible to include all possible interaction to one use case diagram because of number of use cases.
In such cases, you may develop several diagrams with related use cases.



Sequence diagrams

- Sequence diagrams in the UML are primarily used to model the interactions between
 - the actors and the objects in a system
 - the objects themselves
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Interactions between objects are indicated by annotated arrows(call to objects, parameters, return values).

Sequence diagrams - Notations



Actor
symbol

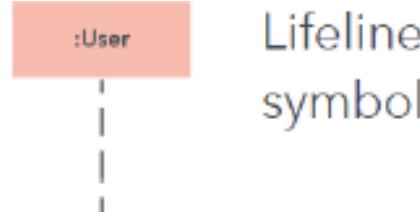
Shows entities
that interact with
or are external to
the system.

P: PatientInfo

Object
symbol

Represents a
class or object in
UML. The object
symbol
demonstrates
how an object will
behave in the
context of the
system.

Sequence diagrams - Notations



Lifeline
symbol

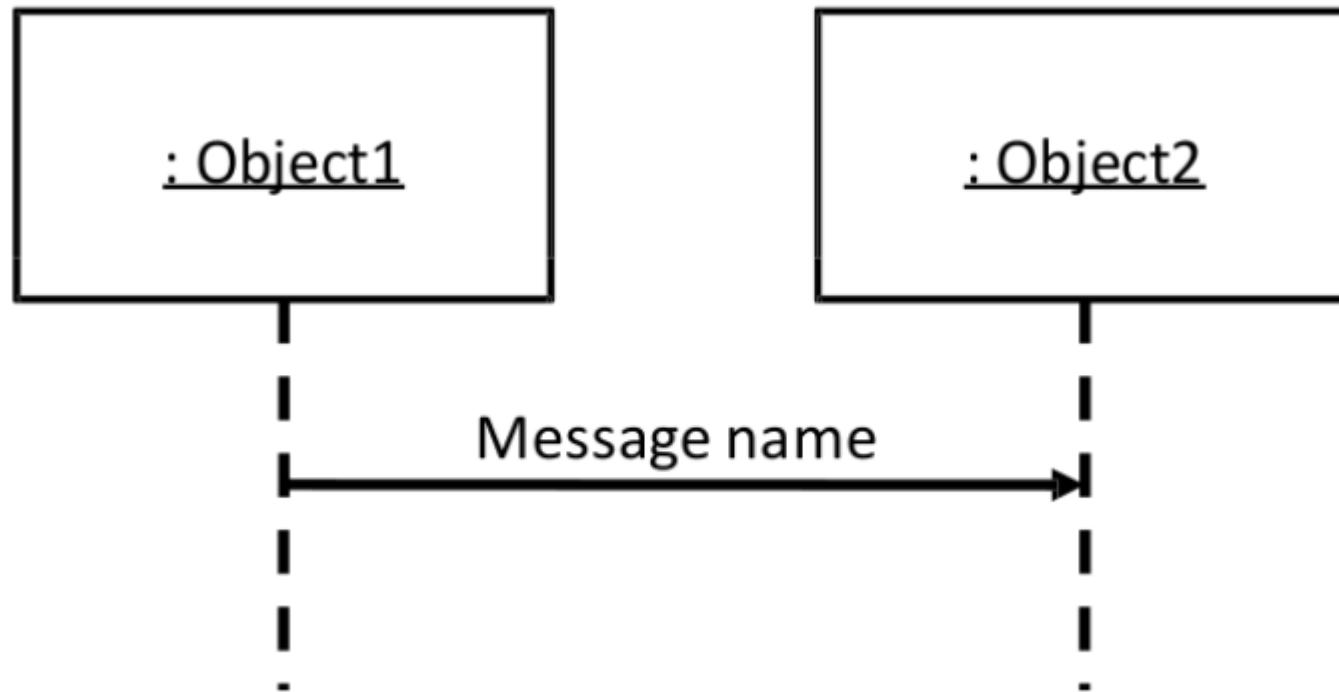
- Should be arranged horizontally across the top of the diagram.
- No two lifeline notations should overlap each other.
- Represent different objects or parts that interact with each other in the system during the sequence.
- Lifelines may begin with a labeled rectangle shape or an actor symbol
- Indicates the existence of an object



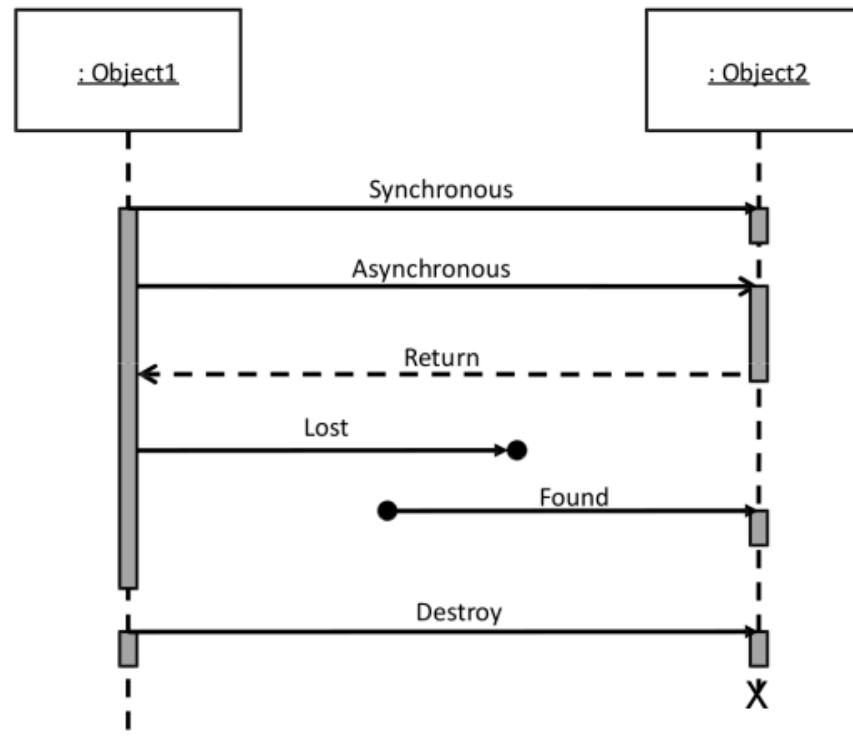
Activation
box

Represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.

Sequence diagrams – Message



Sequence diagrams – Message



Sequence diagrams - Messages

 Reply
message
symbol

- Represented by a dashed line with a lined arrowhead, these messages are replies to calls.

 Synchronous
message
symbol

- Represented by a **solid line with a solid arrowhead**.
- This symbol is used when a **sender must wait for a response to a message before it continues**.
- The diagram should show both the call and the reply.

 Asynchronous
message
symbol

- Represented by a **solid line with a lined arrowhead**.
- Asynchronous messages **don't require a response before the sender continues**.
- Only the call should be included in the diagram

Simplifying the Complexity

- Interaction Use

- Reuse a sequence

- Control Constructs

- Control statements - Loops

Interaction Use

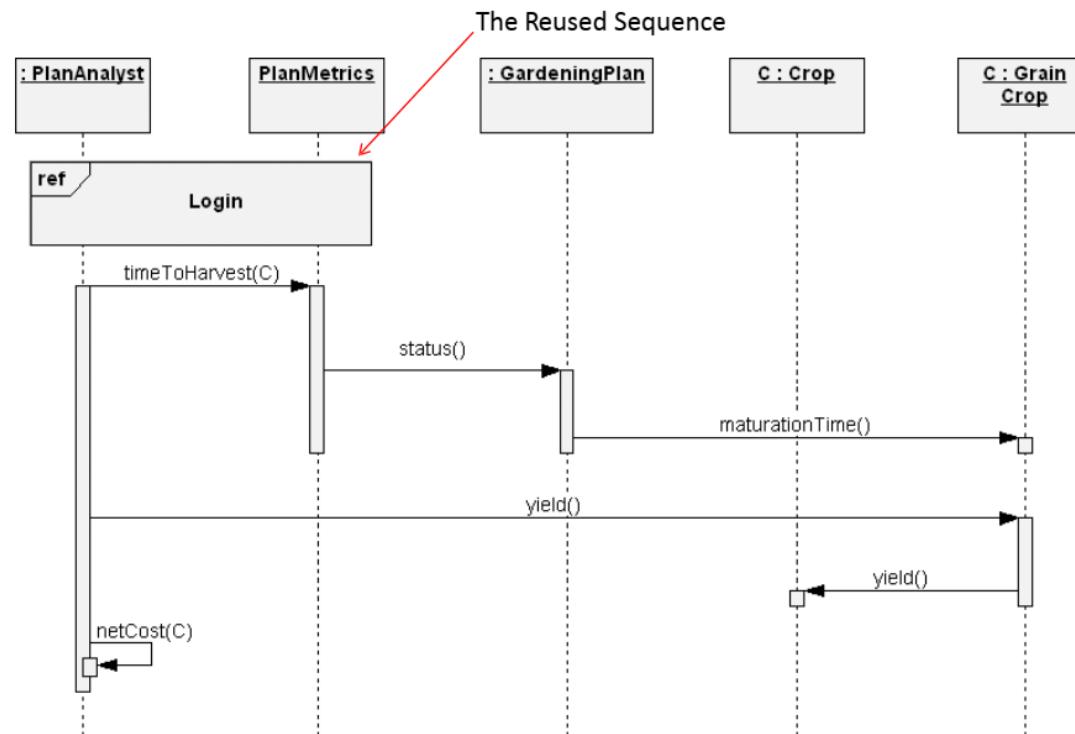
To simplify the complex sequence diagrams

- A way to indicate that the reuse of an interaction which is defined in elsewhere
 - Example: Login sequence

Indicated as a Frame labeled as “ref”

Sequence is applied where the Frame is inserted

Interaction Use



Control Constructs

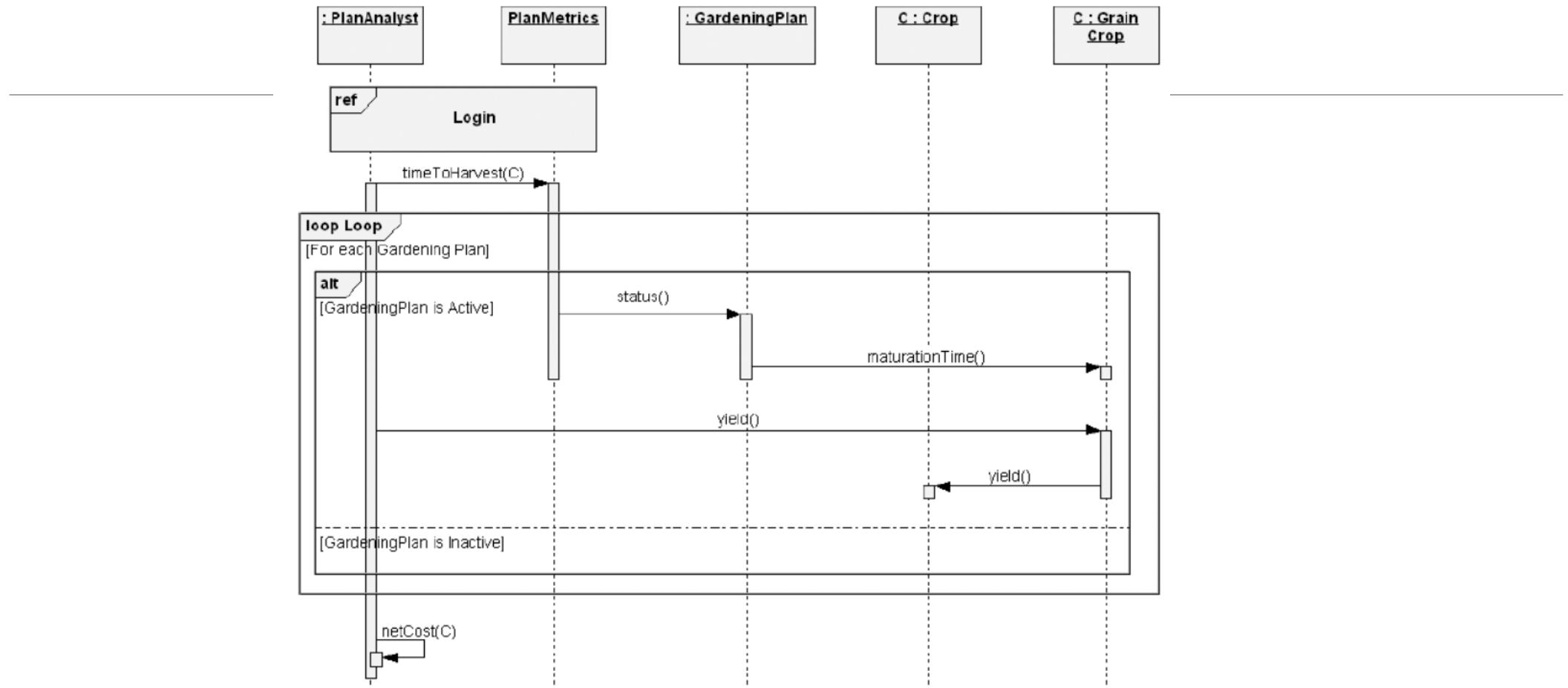
To indicate flow control constructs on a sequence diagram

Interaction operators

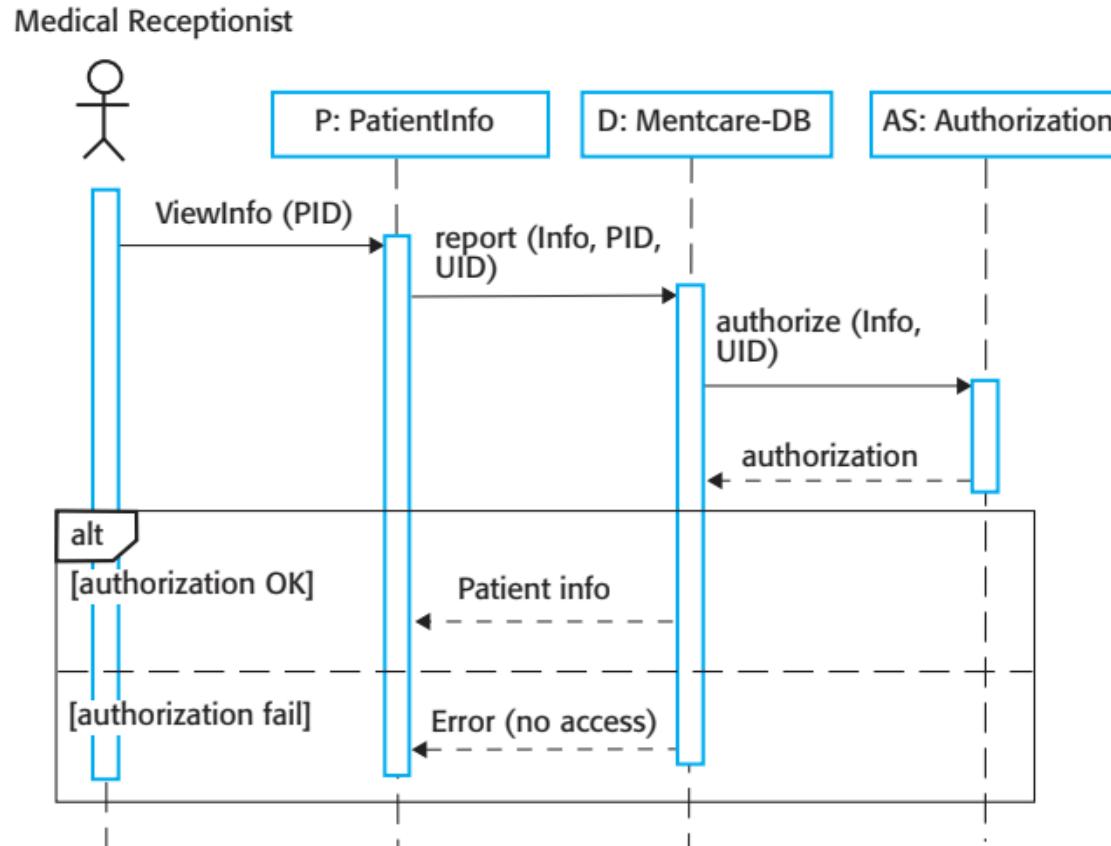
- Loop
 - Controlled by a condition
- Alt
 - A Selection

Ex: gardening plans which are currently in the active state

Control Constructs

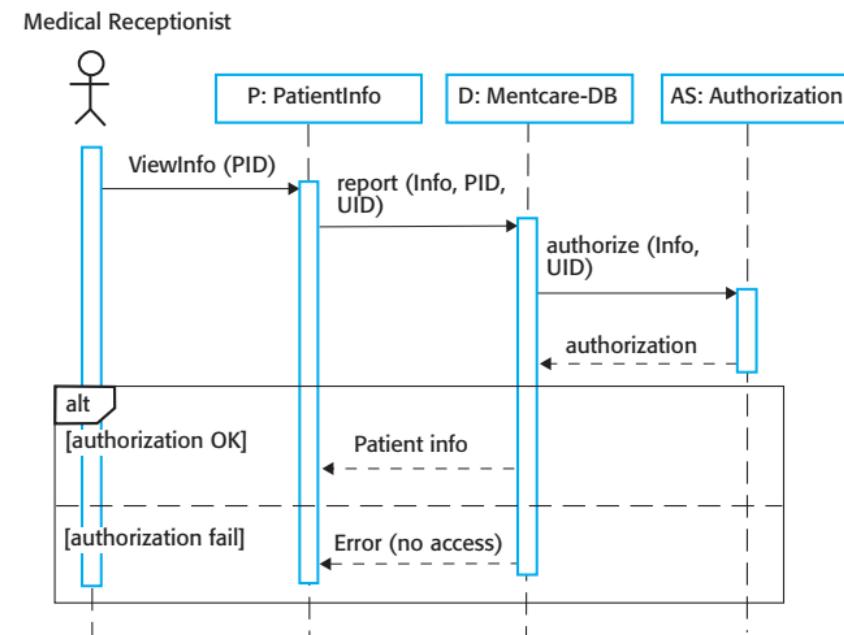


Sequence diagram for View patient information



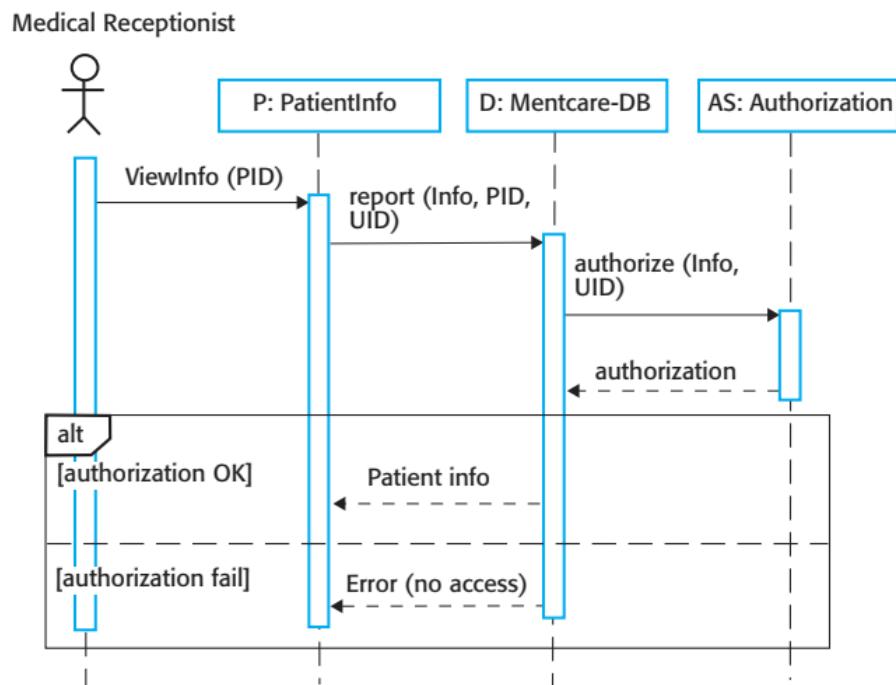
Sequence diagram for View patient information

The medical receptionist triggers the ViewInfo method in an instance P of the Patient Info object class, supplying the patient's identifier, PID to identify the required information. P is a user interface object, which is displayed as a form showing patient information.



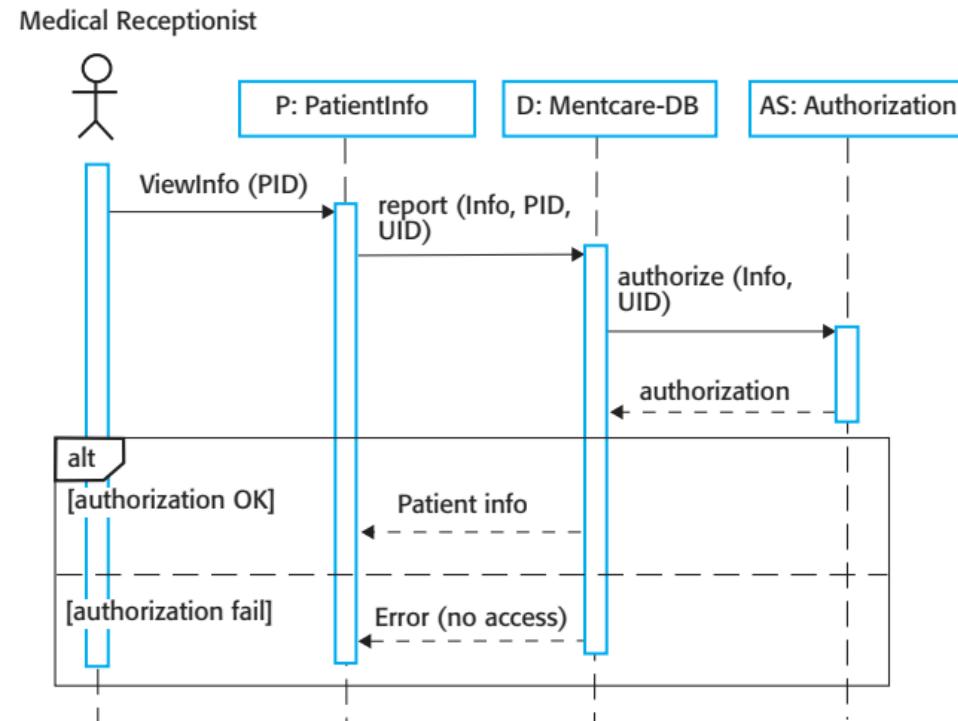
Sequence diagram for View patient information

The instance P calls the database to return the information required, supplying the receptionist's identifier to allow security checking. (At this stage, it is not important where the receptionist's UID comes from.)



Sequence diagram for View patient information

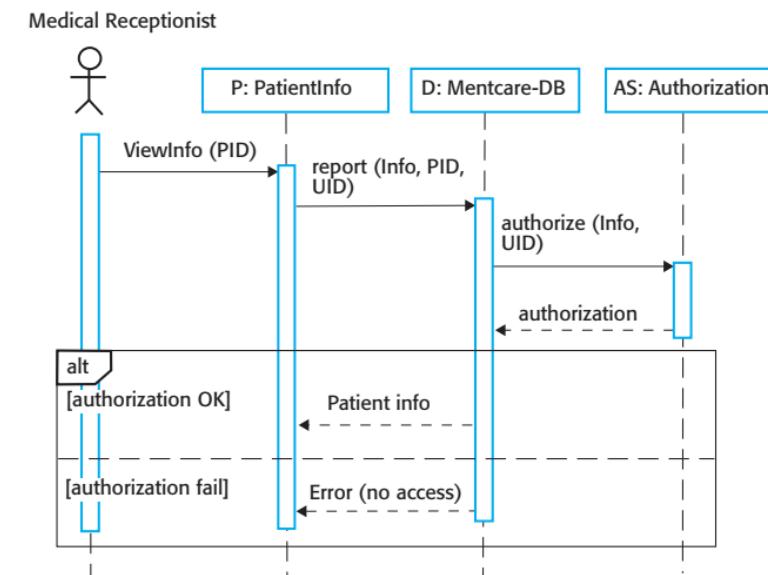
The database checks with an authorization system that the receptionist is authorized for this action



Sequence diagram for View patient information

If authorized, the patient information is returned and is displayed on a form on the user's screen.

If authorization fails, then an error message is returned. The box denoted by "alt" in the top-left corner is a choice box indicating that one of the contained interactions will be executed. The condition that selects the choice is shown in square brackets



Sequence Diagram

In class Activity : Maintaining the Climatic Plan

- There are four entities involved for maintaining a climatic plan. The “Gardening Planner” is responsible for maintaining the climatic plan. It will also record all the activities for maintaining the climatic plan. The “Environment Planner” sets the optimal environmental parameters for the green house. (In this example we’ll consider the temperature parameter only.)
- To maintain the optimal temperature, the Environment planner iteratively checks the current temperature.
 - If the observed temperature is below the optimal level, it’ll increase the temperature by invoking the heating functionality of the “Heater” entity.
 - Else if the observed temperature is above the optimal level, the environment controller will invoke the cooling functionality of the “Cooler” entity to reduce the temperature.



Fundamentals of Software Engineering

Course Code : CS2002

Ms. Mathangi Krishnathasan
[\(tmk@ucsc.cmb.ac.lk\)](mailto:tmk@ucsc.cmb.ac.lk)

System Modeling – Part II

CHAPTER - 05

Lesson Outline

- Structural Models
- Behavioral Models
- Model-driven Engineering

Structural models

Structural models

- Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- Static models
 - show the structure of the system design
- Dynamic models
 - show the organization of the system when it is executing
- Structural models are created when discussing and designing the system architecture.
- Class diagrams are used for modeling the static structure of the object class in a software system.

Class diagrams

- Class diagrams are used when developing an object-oriented system model to show the **classes** in a system and the **associations between** these classes.
- An object class can be thought of as a general definition of one kind of system object.
- An association is a **link between** classes that indicates that there is some relationship between these classes.



- Class diagrams in the UML can be expressed at different levels of detail.

Class and Objects

- What is an object ?
- An object in real world can be defined as any of the following,
 - A tangible and/or visible thing
 - Something toward which action is directed
- Class is the template for creating objects.
- Basic elements of a class diagram.
 - Classes
 - Association between classes.

Class Diagram

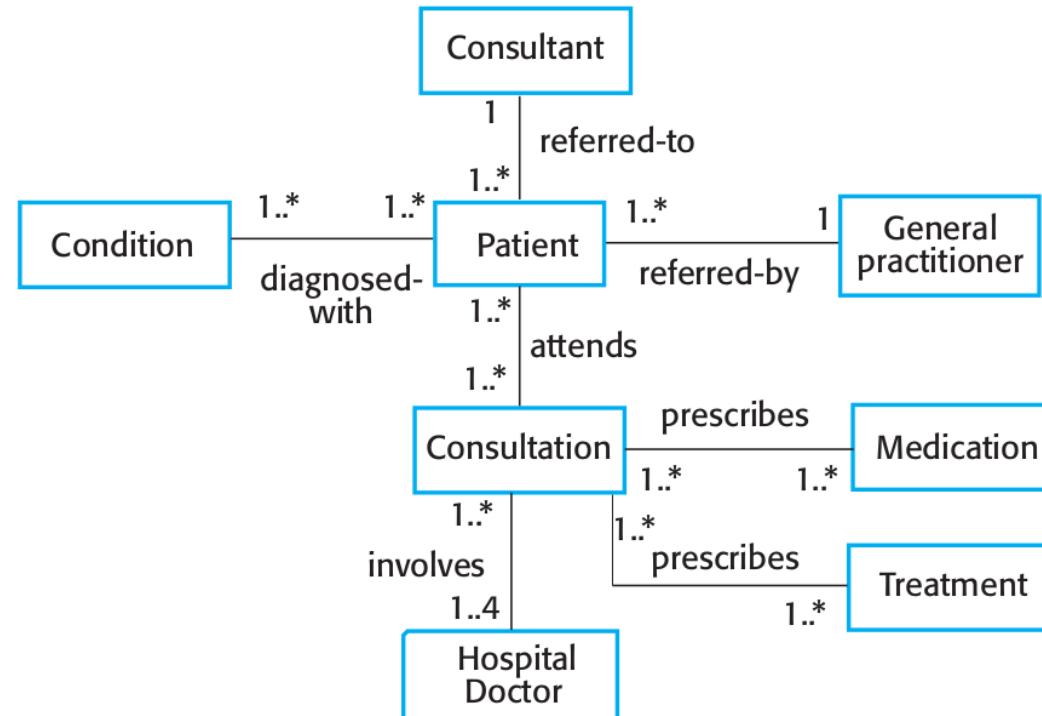
- Class diagrams can be represented in at different levels of details.
- A class is simply represented as a box with the name of the class inside.



- Important feature of class diagram is it has the ability to show how many objects are involved in the association.
 - possible to define the exact number of objects that are involved (e.g., 1..4) or, by using a *, indicate that there are an indefinite number of objects involved in the association.

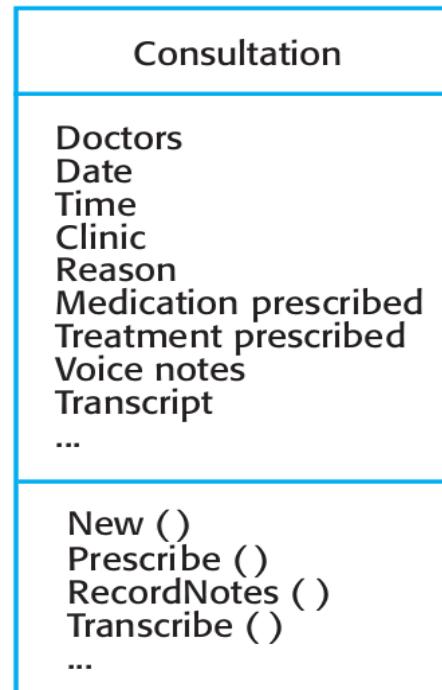
Class Diagram

In this example, name associations are shown to indicate the type of relationship that exists.



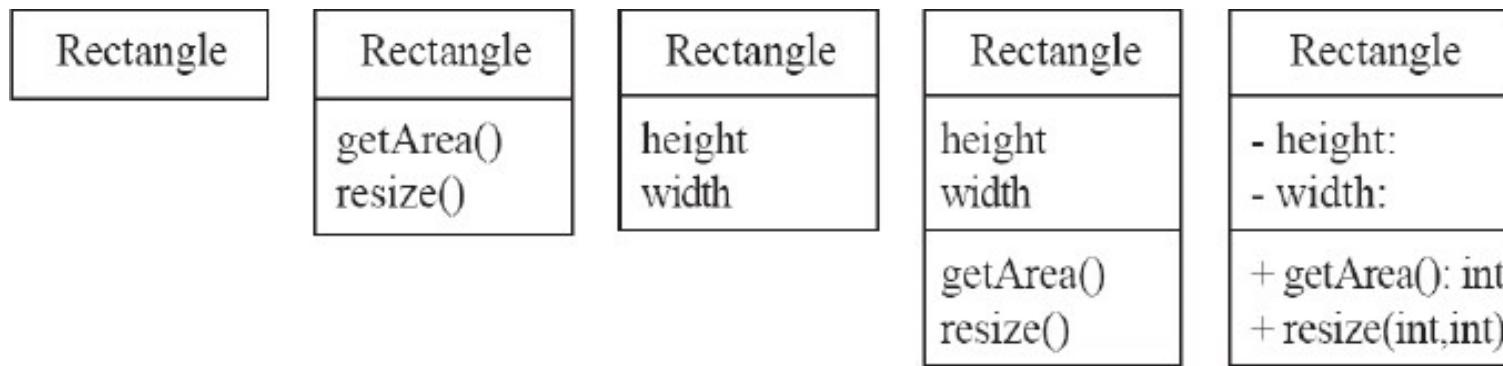
Class Diagram

To define objects in more detail, information about their attributes (the object's characteristics) and operations (the object's functions) can be added to the diagram.



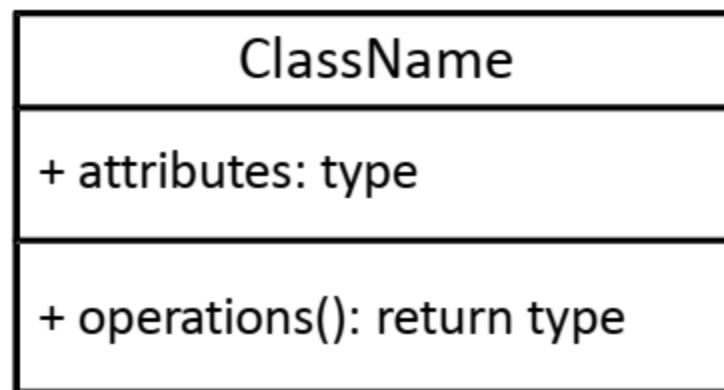
Class Diagram

Class diagrams can be represented in at different levels of details.



Class Notation

- The representation of a Class.
 - Class name
 - Attributes - represent characteristics of the objects.
 - Operations - represent the object's functions
- General Notation



Class

- Attribute
 - General format
- *Visibility attributeName : Type = DefaultValue*
- Operation
 - General Format
- *Visibility operationName(parameterName: parameterType):returnType*
- Visibility

Access Right	public (+)	private (-)	protected (#)	Package (~)
Members of the same class	yes	yes	yes	yes
Members of derived classes	yes	no	yes	yes
Members of any other class	yes	no	no	in same package

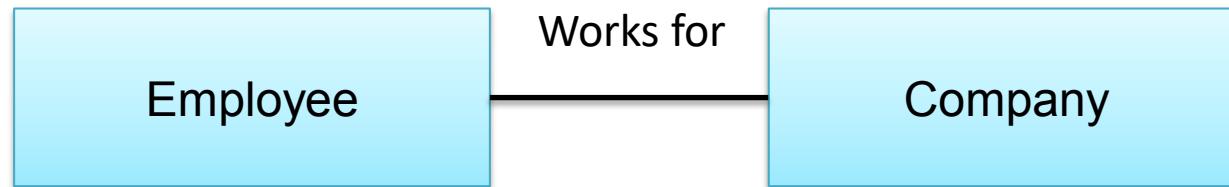
Class Diagram: Relationships

- A class can collaborate with other classes in a variety of ways.
- Different types of class relationships:
 - Association
 - Generalization
 - Aggregation
 - Composition

Association

- An association is used to show how two classes are related to each other
 - represented by a solid line between classes.
 - Possible to have more than one association between the same pair of classes.
 - It should be named to indicate the role played by the class attached at the end of the association path.

•



Associations and Multiplicity

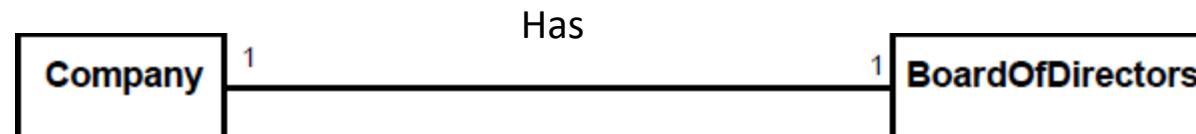
- Symbols indicating multiplicity are shown at **each end of the association**.
- If a multiplicity is not specified, by default ONE is considered as a default multiplicity.



Analyzing and validating associations

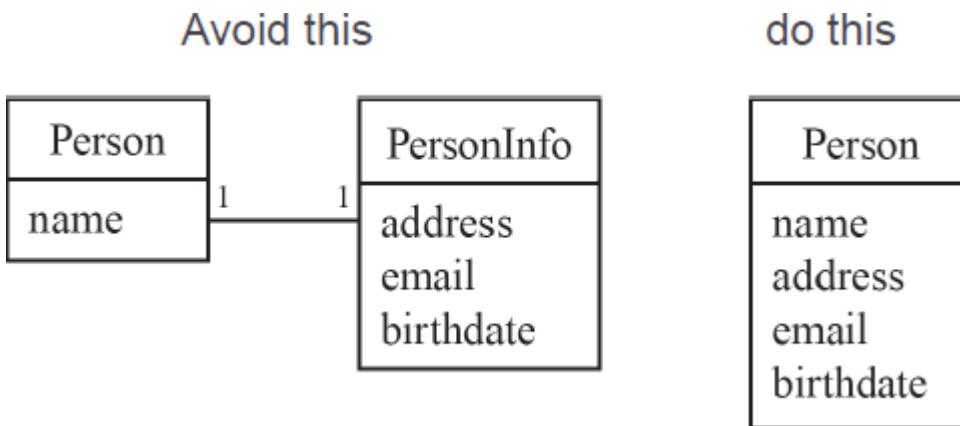
- **One-to-one**

- For each company, there is exactly one board of directors.
- A board is the board of only one company.
- A company must always have a board.
- A board must always be of some company.



Analyzing and validating associations

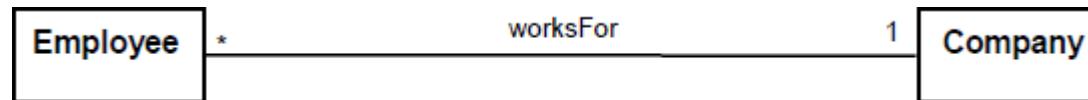
- Avoid unnecessary one-to-one associations



Analyzing and validating associations

- **Many-to-one**

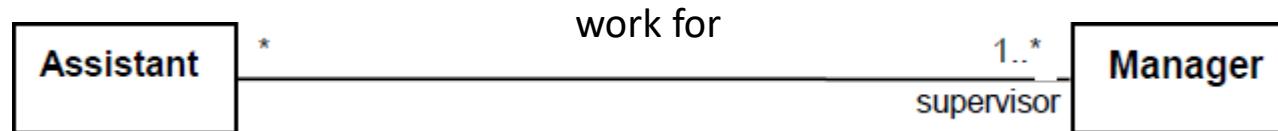
- A company has many employees,
- An employee can only work for one company.
- It is not possible to be an employee unless you work for a company



Analyzing and validating associations

- **Many-to-many**

- An assistant can work for many managers.
- A manager can have many assistants.
- There should be at least one manager for each assistants.



Associations : Multiplicity

Here are some examples of multiplicity notations in associations

- Exactly one - 1
- Zero or one - 0..1
- Many - 0..* or *
- One or more - 1..*
- Exact Number - e.g. 3..4 or 6
- Or a complex relationship - e.g. 0..1, 3..4, 6.* would mean any number of objects other than 2 or 5.

Aggregation

- Objects in the real world are often made up of different parts.
- **A special kind of an association that represents a ‘part-whole’ relationship is aggregation.**
- Denotes a whole(aggregate)/part hierarchy.
 - one object (the whole) is composed of other objects (the parts).
 - The same (shared) part could be included in several composites.
- In a relationship between X and Y, an aggregation implies a relationship that X can exist independently of the Y.
- If composite is deleted, shared parts may still exist.

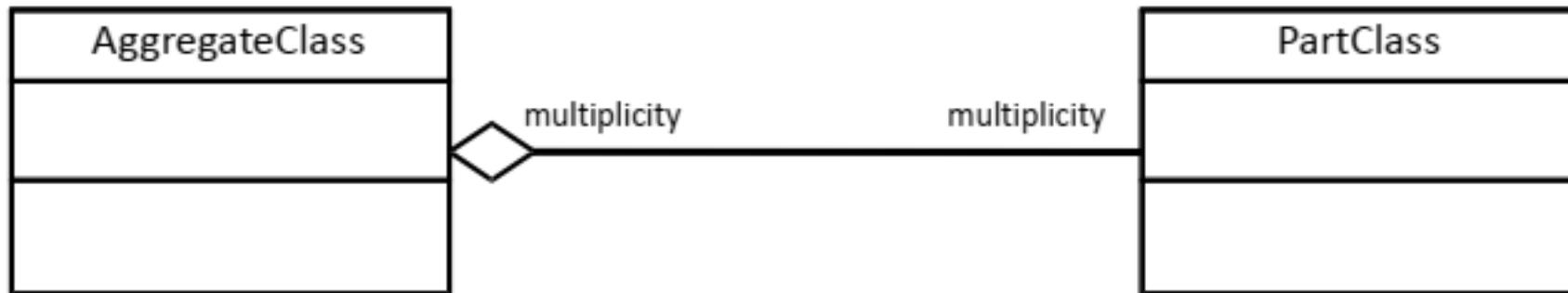
Aggregation : Notation

To define aggregation, a diamond shape is added to the link next to the class that represent the whole.

Notation :

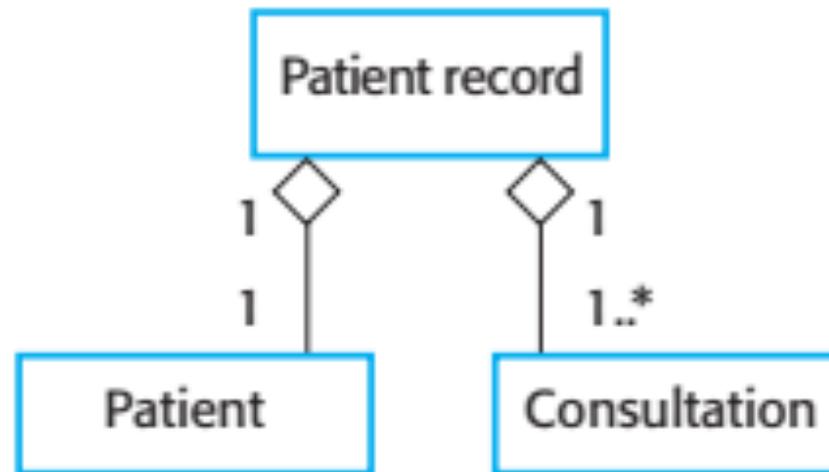


Applying Notation :



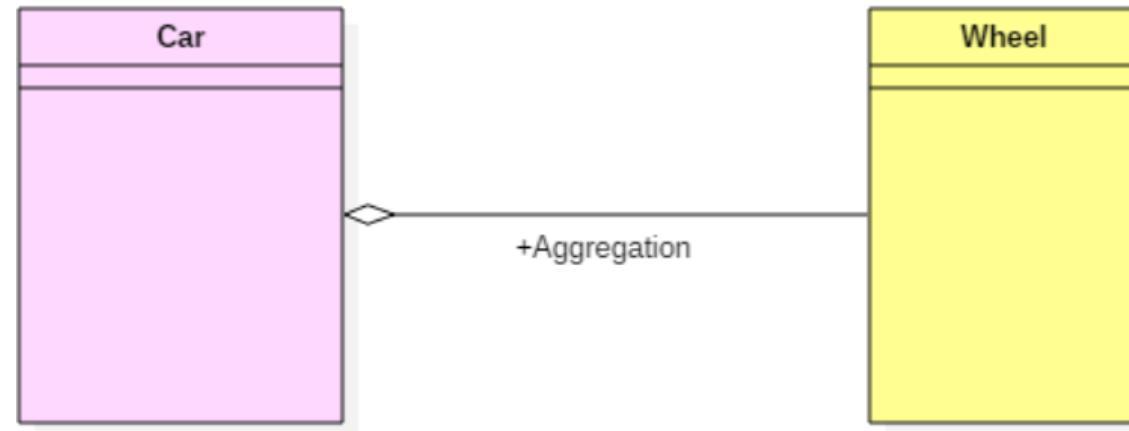
Aggregation : Example

- Patient record is an aggregate of patient and an indefinite number of consultations.
- Patient record maintains personal patient information as well as an individual record for each consultation with a doctor.



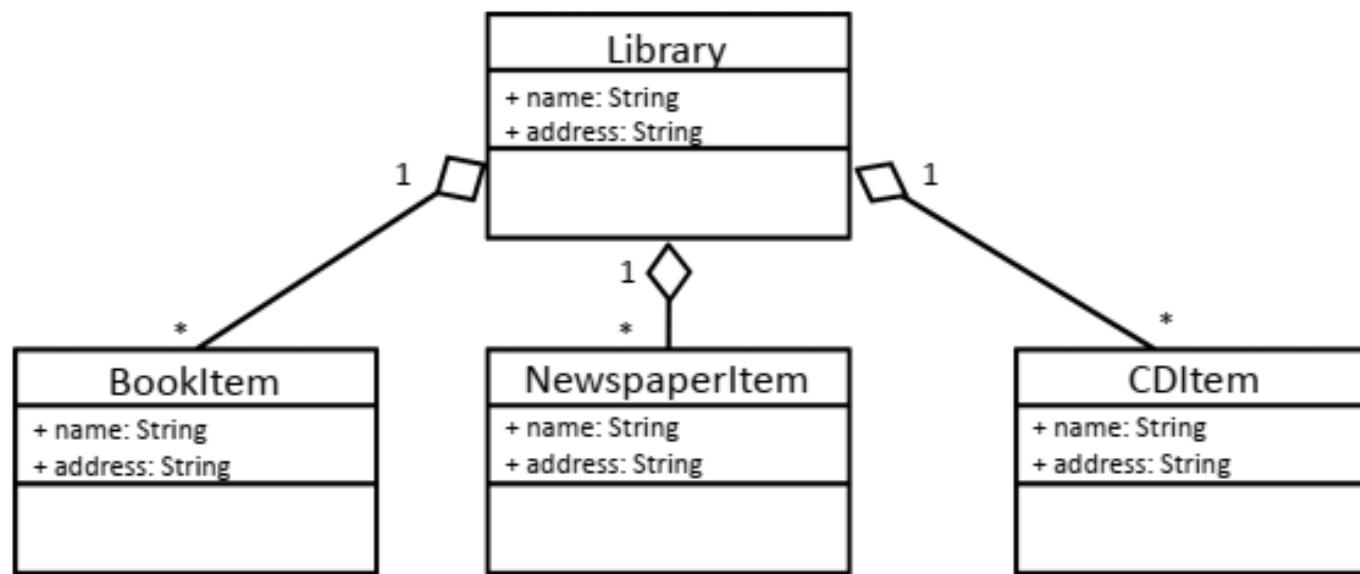
Aggregation : Example 1

- A car needs a wheel to function correctly, but a wheel doesn't always need a car.
- It can also be used with the bike, bicycle, or any other vehicles but not a particular car.
- Here, the wheel object is meaningful even without the car object.

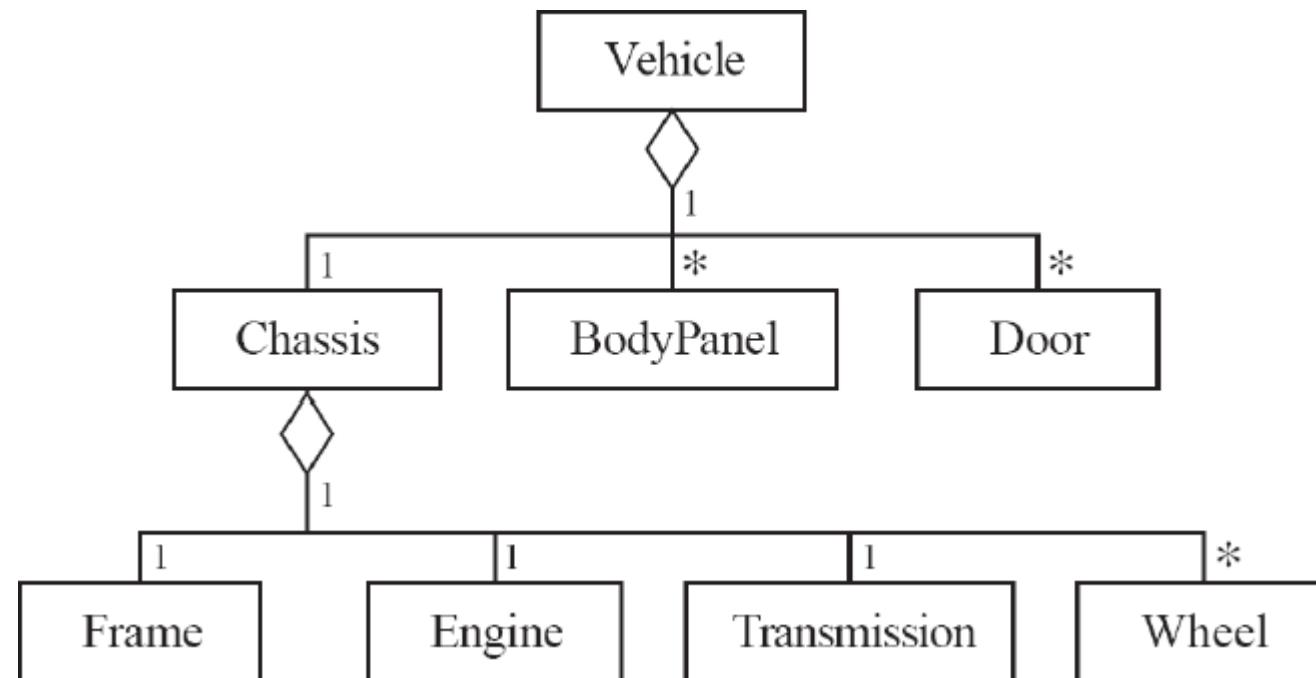


Aggregation : Example 2

- Library has book item
- Library has newspaper item
- Library has CD item



Aggregation hierarchy

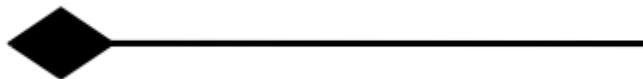


Composition

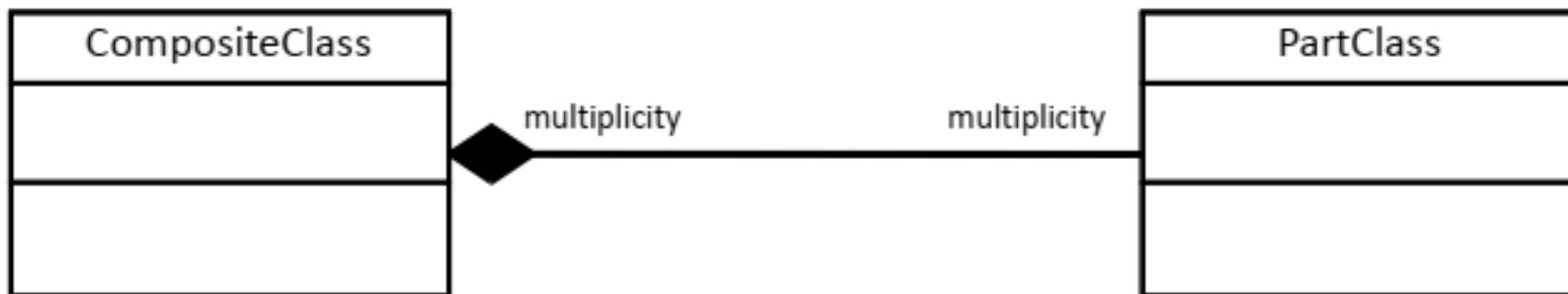
- A **composition** is a strong form of aggregation.
- It is also a whole/part relationship.
- Part could be included in at most one composite (whole) at a time.
- In a relationship between X and Y, an aggregation implies a relationship that X **can not** exist independently of the Y.
- if a composite (whole) is deleted, all of its composite parts are "normally" deleted with it.

Composition : Notation

Notation :



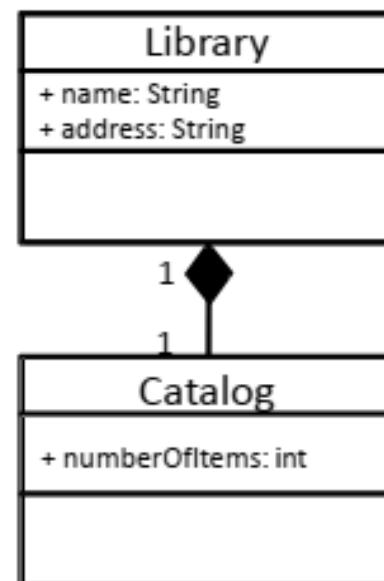
Applying Notation :



Composition : Example

The Catalog is a part of the Library

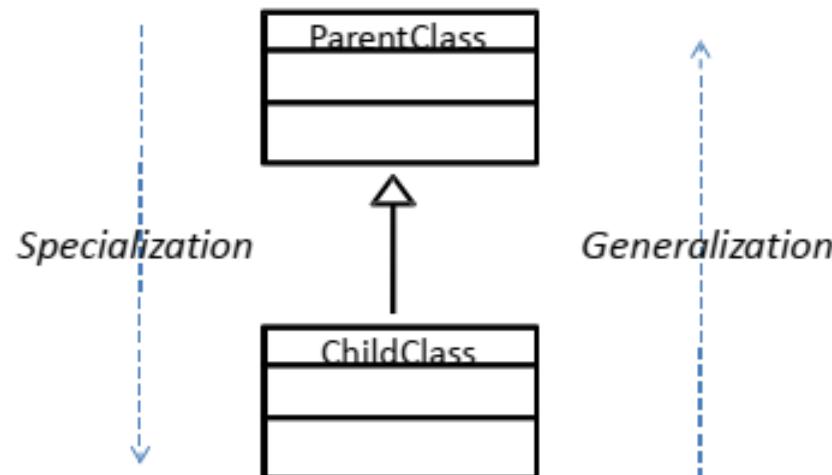
If the Library is destroyed, the Catalog will also be destroyed. (In other words, the catalog can not remain as an individual entity without the Library)



Generalization

- The generalization relationship occurs between two entities or objects, such that one entity is the parent, and the other one is the child.
- “Is a” Relationship – Sub class & Super class Relationship
- Common information will be maintained in one place only.

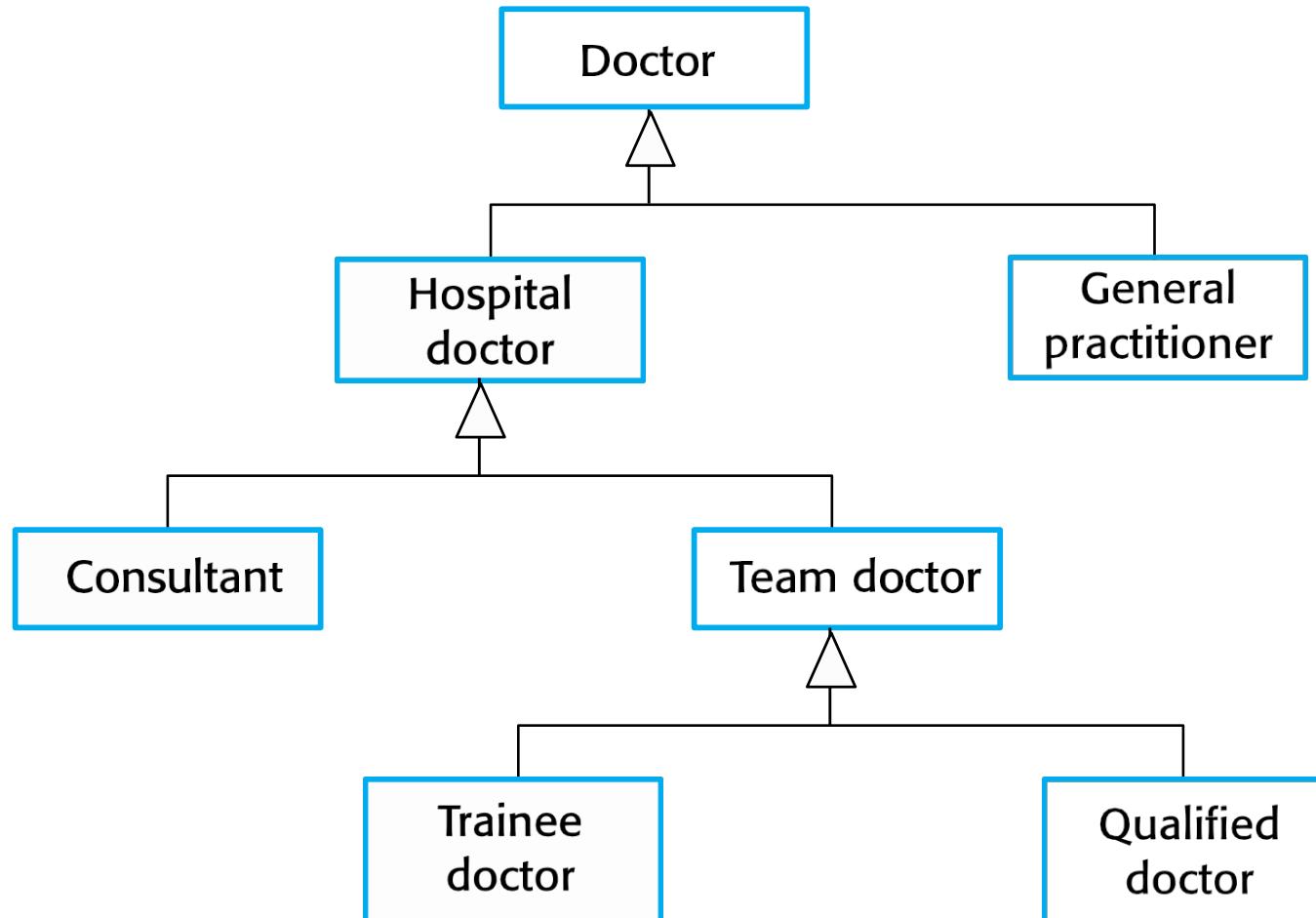
Notation:



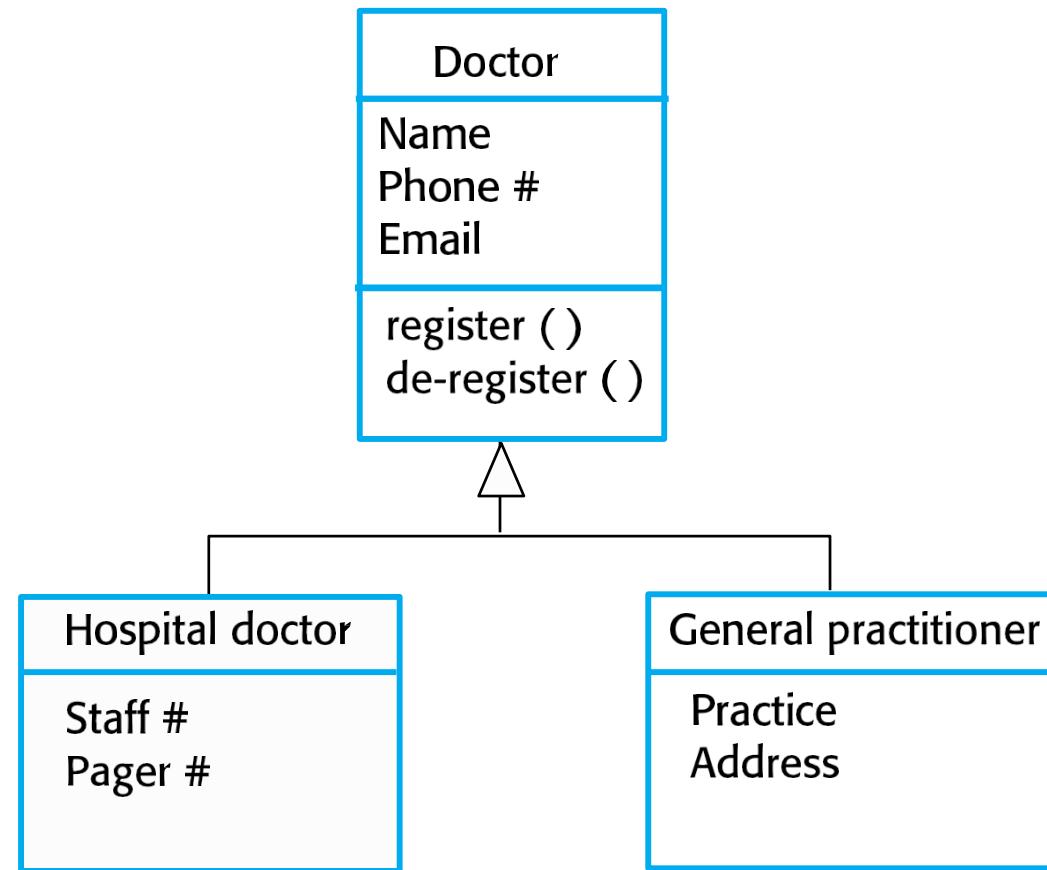
Generalization

- In object-oriented languages, such as Java, generalization is implemented using the inheritance mechanisms.
- The child class inherits the functionality of its parent class.
- It is a good design practice.
- If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change. You can make the changes at the most general level.

A generalization hierarchy

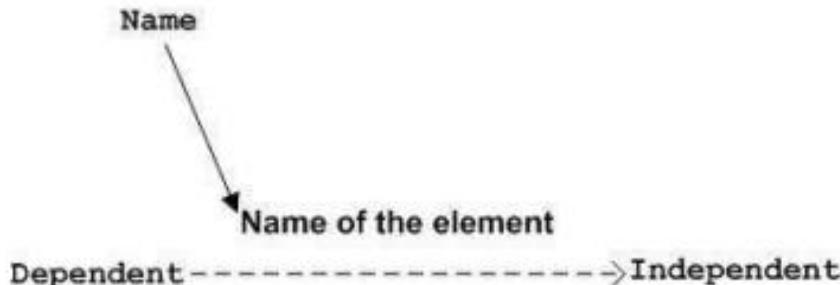


Generalization - Example



Dependency

- Describes the
 - dependent elements and
 - the direction of dependency
- Represented by
 - a dotted arrow as shown below.
 - the arrow head represents the independent element and the other end represents the dependent element



Association, Aggregation and Composition

- Association can be considered as any logical relationship between two classes.
- Both aggregation and composition are the refinements of the association relationship.
- Both aggregation and composition relationships are considered as “part of” relationship.

- **Aggregation is a weak form of the “part of” relationship**

Aggregate and the parts are loosely coupled If the aggregate is destroyed, the part can still be remained.

- **Composition is a strong form of the “part of” relationship**

Composite and the parts are strongly coupled if the composite is destroyed, the parts will also be destroyed.

Behavioral models

Behavioral models

- Models the dynamic behavior of a system when it is executing.
- Behaviour models show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- These stimuli may be either data or events.
 - **Data** : if data becomes available and that has to be processed by the system. The availability of the data triggers the processing.
 - **Events** : An event happens then that triggers system processing.

Data-driven modeling

- Many business systems are data-processing systems that are primarily driven by data.
- They are controlled by the data input to the system, with relatively little external event processing.
- Data-driven models show the **sequence of actions involved in processing input data and generating an associated output.**
 - For example, a phone billing system will accept information about calls made by a customer, calculate the costs of these calls, and generate a bill for that customer.

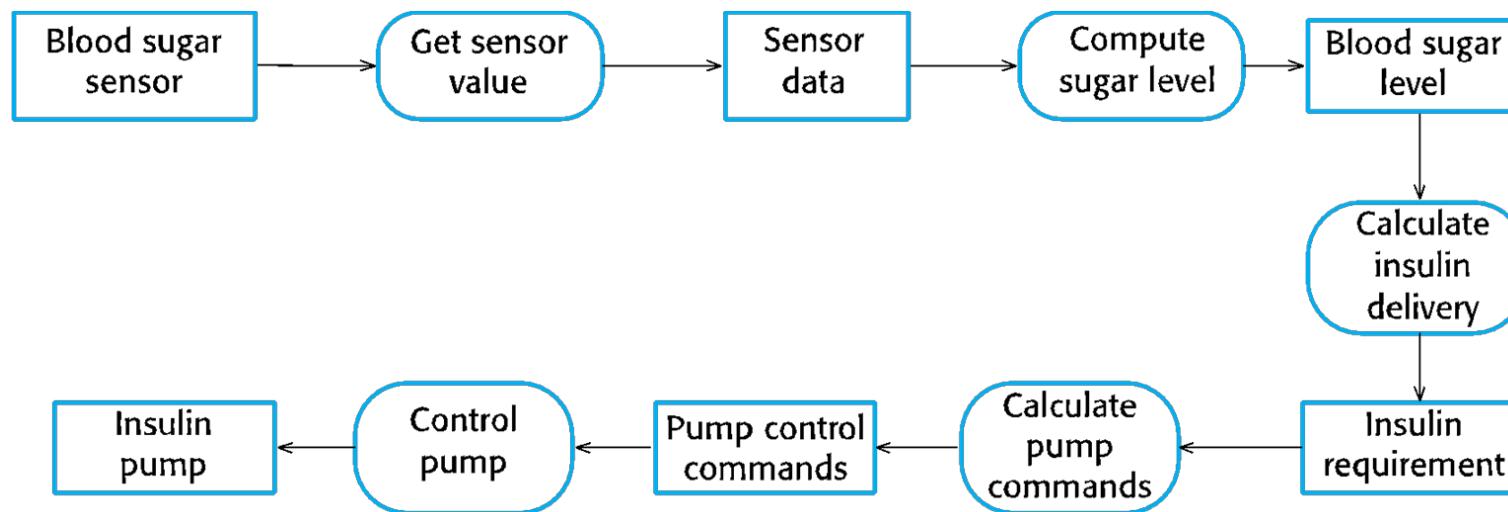
Data-driven modeling

- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- They are particularly **useful during the analysis of requirements** as they can be used to show end-to-end processing in a system
- Data-flow models are used in tracking and documenting how data associated with a particular process moves through the system.
- Data-flow diagrams can be represented in the UML using the activity diagram or sequence diagram.

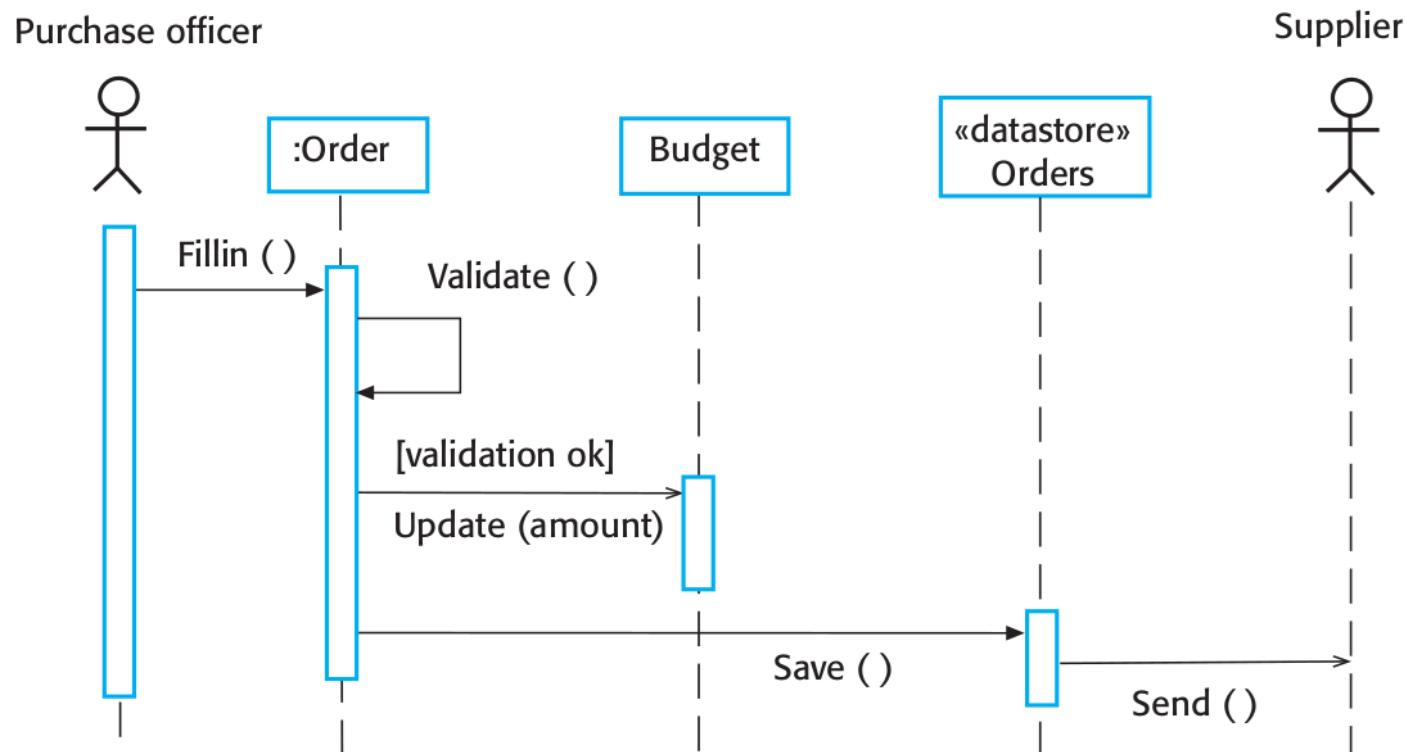
Data-flow diagrams using UML Activity diagram.

An activity model of the insulin pump's operation

- The processing steps are represented as activities (rounded rectangles), and the data flowing between these steps are represented as objects (rectangles).



Data-flow diagrams using UML Sequence diagram.



A sequence model of processing an order and sending it to a supplier

Event-driven modeling

- Event-driven modeling shows how a system responds to external and internal events.
- It is based on the assumption that a **system has a finite number of states and that events** (stimuli) may cause a transition from one state to another.
- Real-time systems are often event-driven, with minimal data processing.
 - Eg: A landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.
- The UML supports event-based modeling using state diagrams, which show system states and events that cause transitions from one state to another.

State diagram : Notations

- Notations
 - Rounded rectangle: System State
 - Labeled Arrow: Stimuli (force the transition from one state to another)



Initial state



Final state

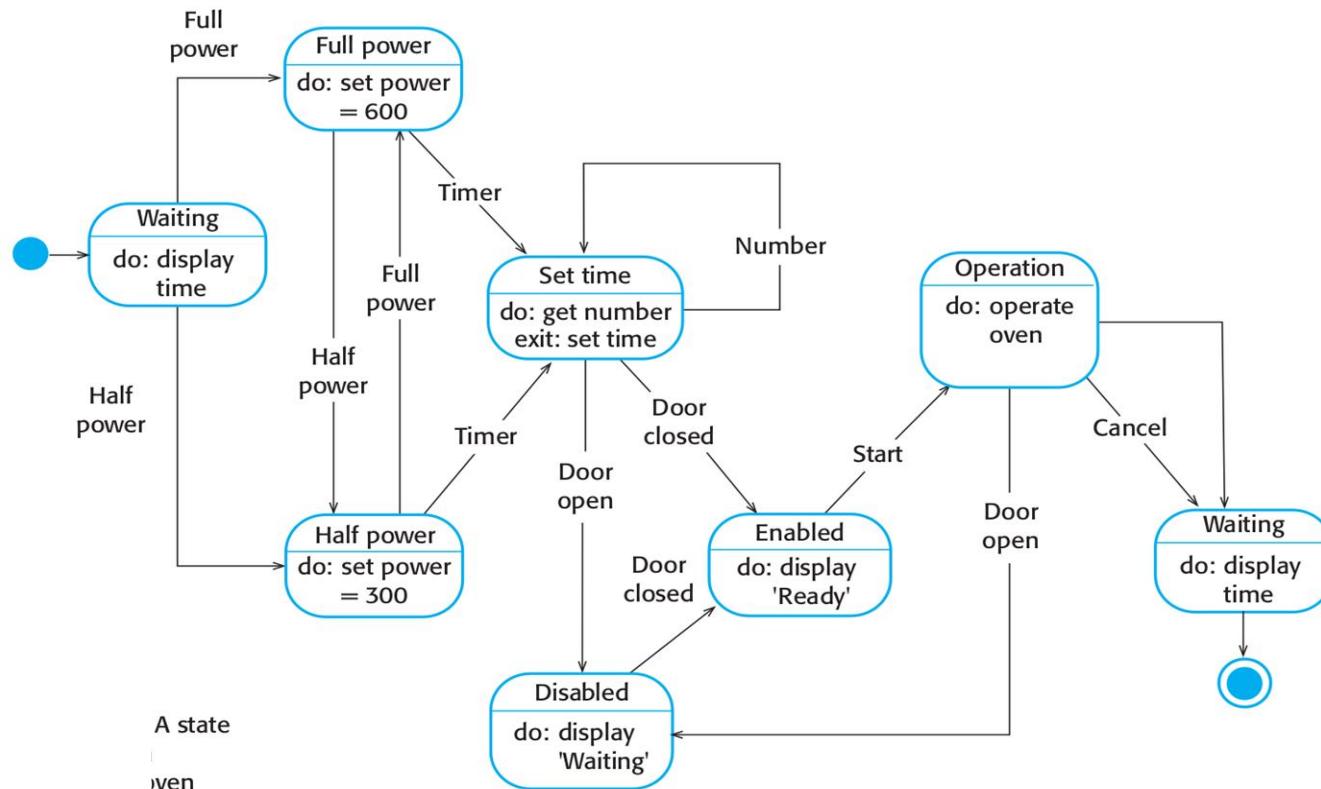


State



Transition

Event-flow diagrams using UML State diagram.

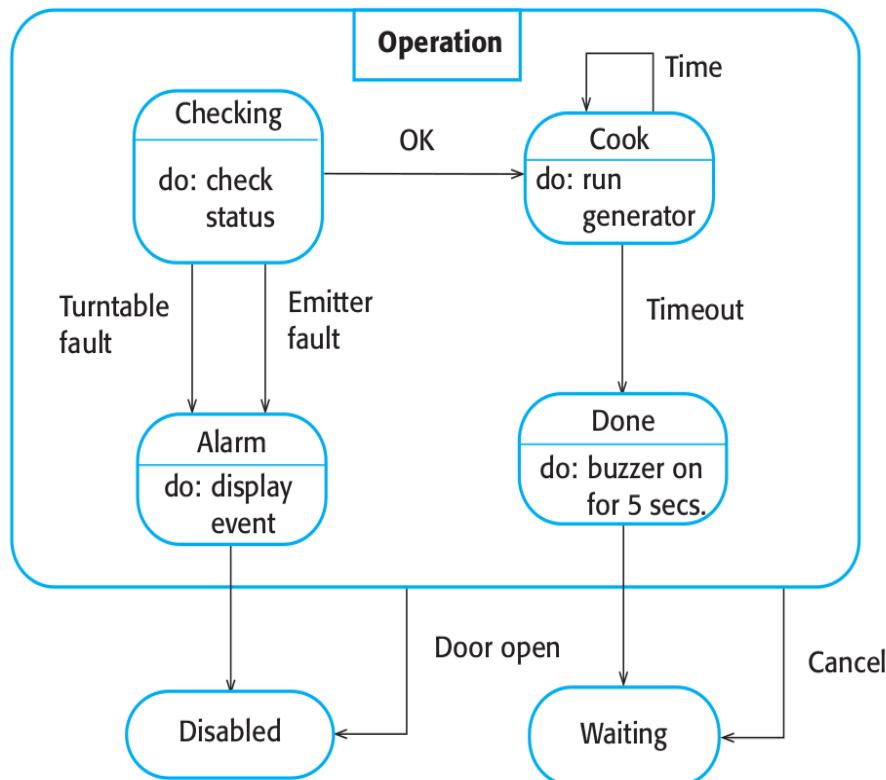


A simple microwave oven illustration.

Problem in state-based modeling

- The problem with state-based modeling is that the number of possible states increases rapidly.
- For large system models, the detail needs to be hidden in the models.
 - One way to do this is by using the notion of a “**superstate**” that encapsulates a number of separate states.
 - This superstate looks like a single state on a high-level model but is then expanded to show more detail on a separate diagram.

Event-flow diagrams using UML State diagram.



Superstate microwave oven illustration.

State Diagrams

- State models of a system provide an overview of event processing, these models have to be extended with a more detailed description of the stimuli and the system states.
- A table is used to list the states and events that stimulate state transitions along with a description of each state and event.

Half power	The oven power is set to 300 watts. The display shows "Half power."
Full power	The oven power is set to 600 watts. The display shows "Full power."

Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.

Model-driven Engineering

- Model-driven engineering (MDE) is an approach to software development where the models rather than programs are the principal outputs of the development process.
- Experts in MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.
- Model-driven engineering was developed from the idea of model-driven architecture (MDA).
- MDA focuses on the design and implementation stages of software development, whereas MDE is concerned with all aspects of the software engineering process.
- Model-based requirements engineering, software processes for model-based development, and model-based testing are part of MDE but are not considered in MDA.

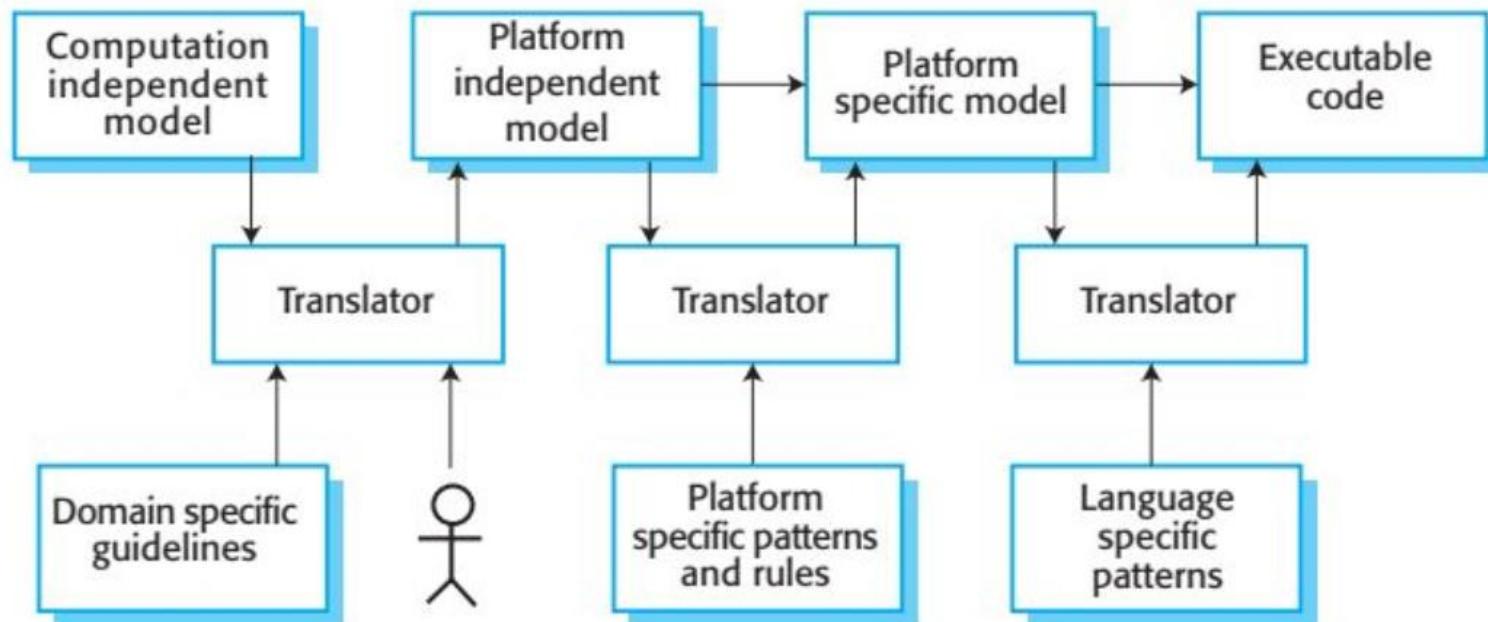
Benefits Model-driven Engineering

- Model-based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementation.
- This reduces the likelihood of errors and speeds up the design and implementation process.
- It allows for the creation of reusable, platform-independent application models. By using powerful tools, system implementations can be generated for different platforms from the same model.

Model-driven Architecture

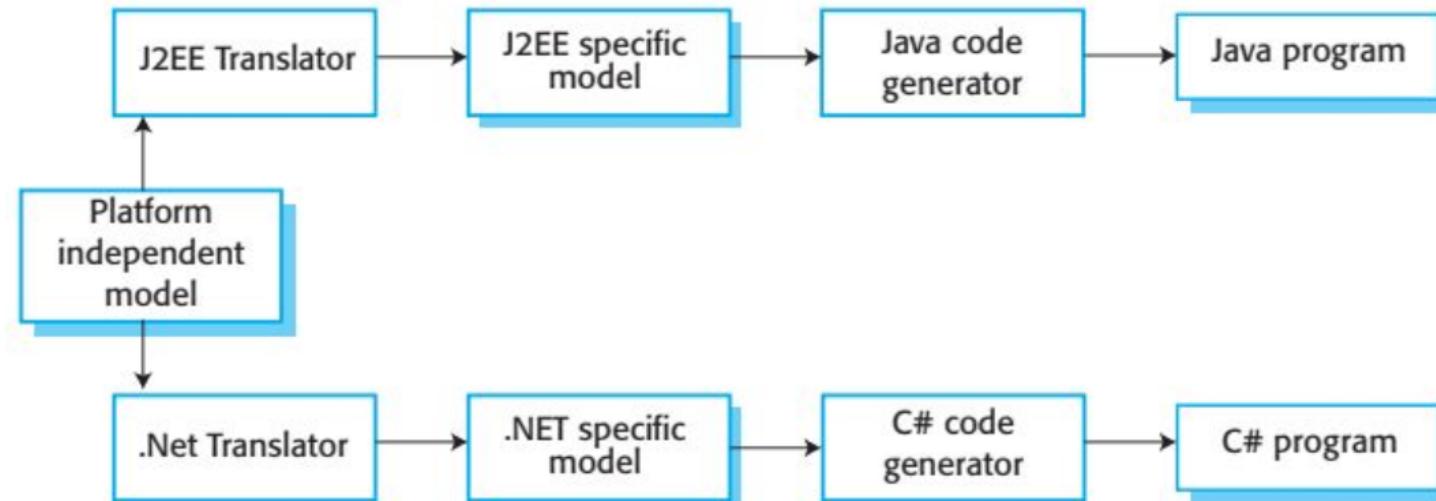
- Model-driven architecture is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- Models at different levels of abstraction are created. The MDA method recommends that three types of abstract system model should be produced:
 - A computation independent model (CIM) - develop several different CIMs, reflecting different views of the system or for different domains.
 - A platform-independent model (PIM) - model the operation of the system without reference to its implementation.
 - Platform-specific models (PSM) - transformations of the platform independent model with a separate PSM for each application platform. there may be layers of PSM, with each layer adding some platform specific detail.
- Fundamental to MDA is the notion that transformations between models can be defined and applied automatically by software tools. but In practice, completely automated translation of models to code is rarely possible.

Model-driven Architecture



Model-driven Architecture

Multiple model platform-specific models



Model-driven Architecture

There are several other reasons why MDA has not become a mainstream approach to software development.

- Models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are useful for discussions are the right abstractions for implementation.
- For most complex systems, implementation is not the major problem— requirements engineering, security and dependability, integration with legacy systems and testing are all more significant. Consequently, the gains from the use of MDA are limited.
- The arguments for platform independence are only valid for large, long-lifetime systems, where the platforms become obsolete during a system's lifetime. For software products and information systems that are developed for standard platforms, such as Windows and Linux, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.



Architectural Design

Course Code : CS2002

Ms. Mathangi Krishnathasan
[\(tmk@ucsc.cmb.ac.lk\)](mailto:tmk@ucsc.cmb.ac.lk)

Intended Learning Outcomes

- Explain why the architectural design of software is important.
- Discuss the decisions that have to be made about the system architecture during the architectural design process.
- Explain the 4+1 view of System Architecture
- Explain the common architectural patterns that are often used to organize the system architecture.

Lesson Outline

- Architectural design
- Architectural design decisions
- Architectural views
- Architectural patterns

What is architecture?

- Before you create the system, before you go on detailing of the system you need to have an overall idea of the system.
- For that we used **architectural model**.

Why do we need an Architecture?

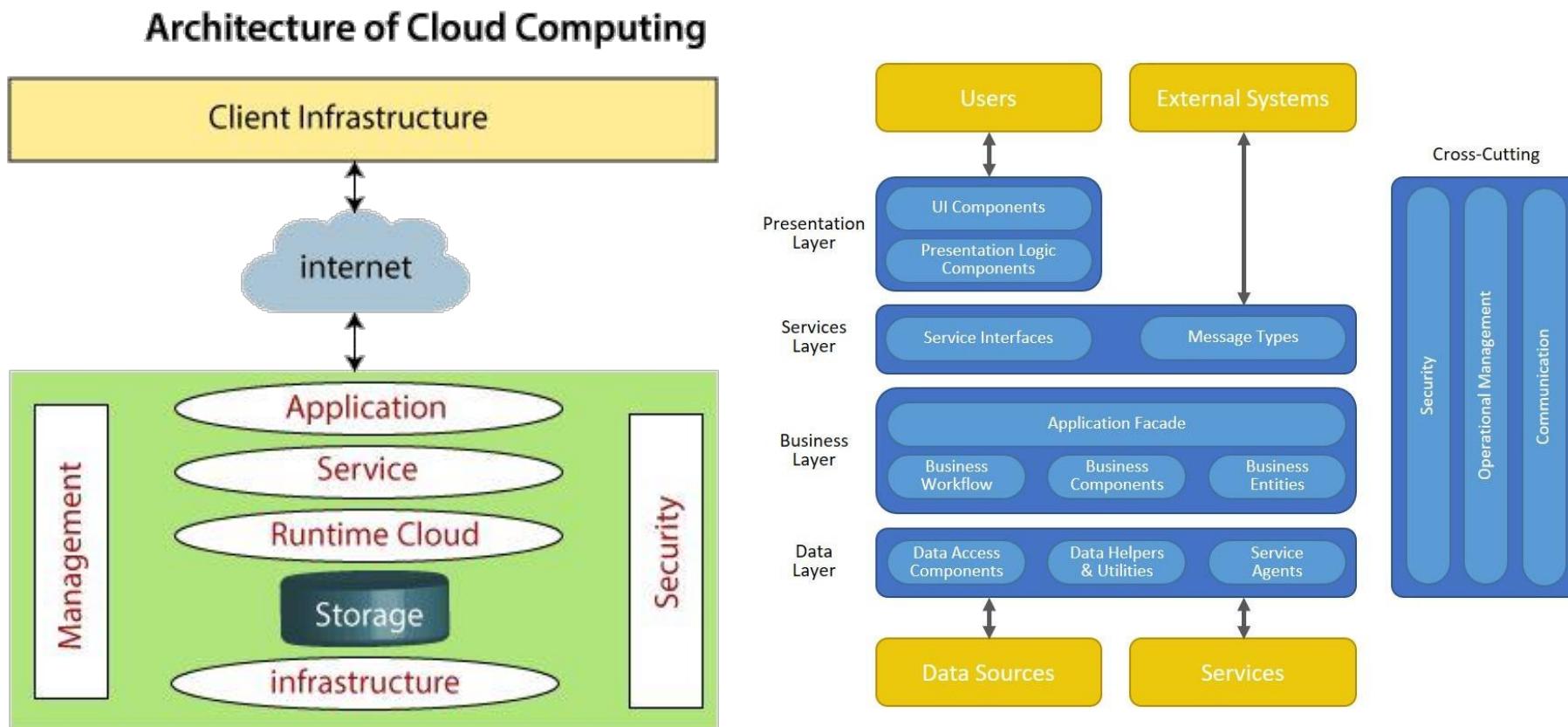
- Understand the system
 - Complexity
- Organize the development
 - Architectural partitioning
- Reuse
- Evolution
 - Changes and dependencies



What is a Software Architecture?

- How a software system should be organized?
- Identifies the main structural components in a system and its relationships.
- Overall system design
 - Subsystems
 - Interconnections
 - Collaborations
- Architectural model -The output of the architectural design process
 - how the system is organized as a set of communicating components.

What is architecture?



Architectural design

- Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
- Architectural design is the first stage in the software design process.
- Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.
- The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

Agility and architecture

- Overall systems architecture design should be done at an early stage of agile processes.
- Incremental development of architectures is not usually successful.
- Refactoring the system architecture is usually expensive because it affects so many components in the system
- Software architectures can be at two levels of abstraction.
 - Architecture in the small - At this level, we are concerned with the way that an individual program is decomposed into components.
 - Architecture in the large - This is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.

Non-functional requirements had the most significant effect on the system's architecture.

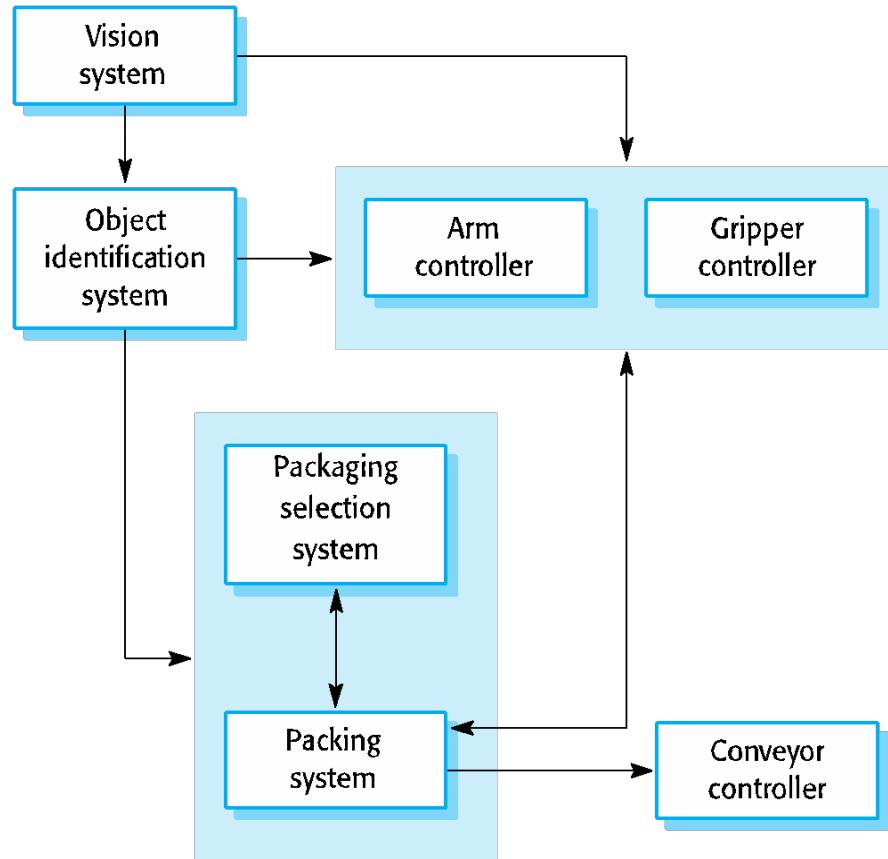
Advantages of Architecture and Documentation

- Stakeholder communication
 - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
 - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
 - The architecture may be reusable across a range of systems
 - System architecture is often the same for similar requirements.

Architectural representations

- Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
 - Each box in the diagram represents a component.
 - Boxes within boxes indicate that the component has been decomposed to subcomponents.
 - Arrows mean that data and or control signals are passed from component to component in the direction of the arrows.

The architecture of a packing robot control system - Box and Line Diagram



Architectural design decisions

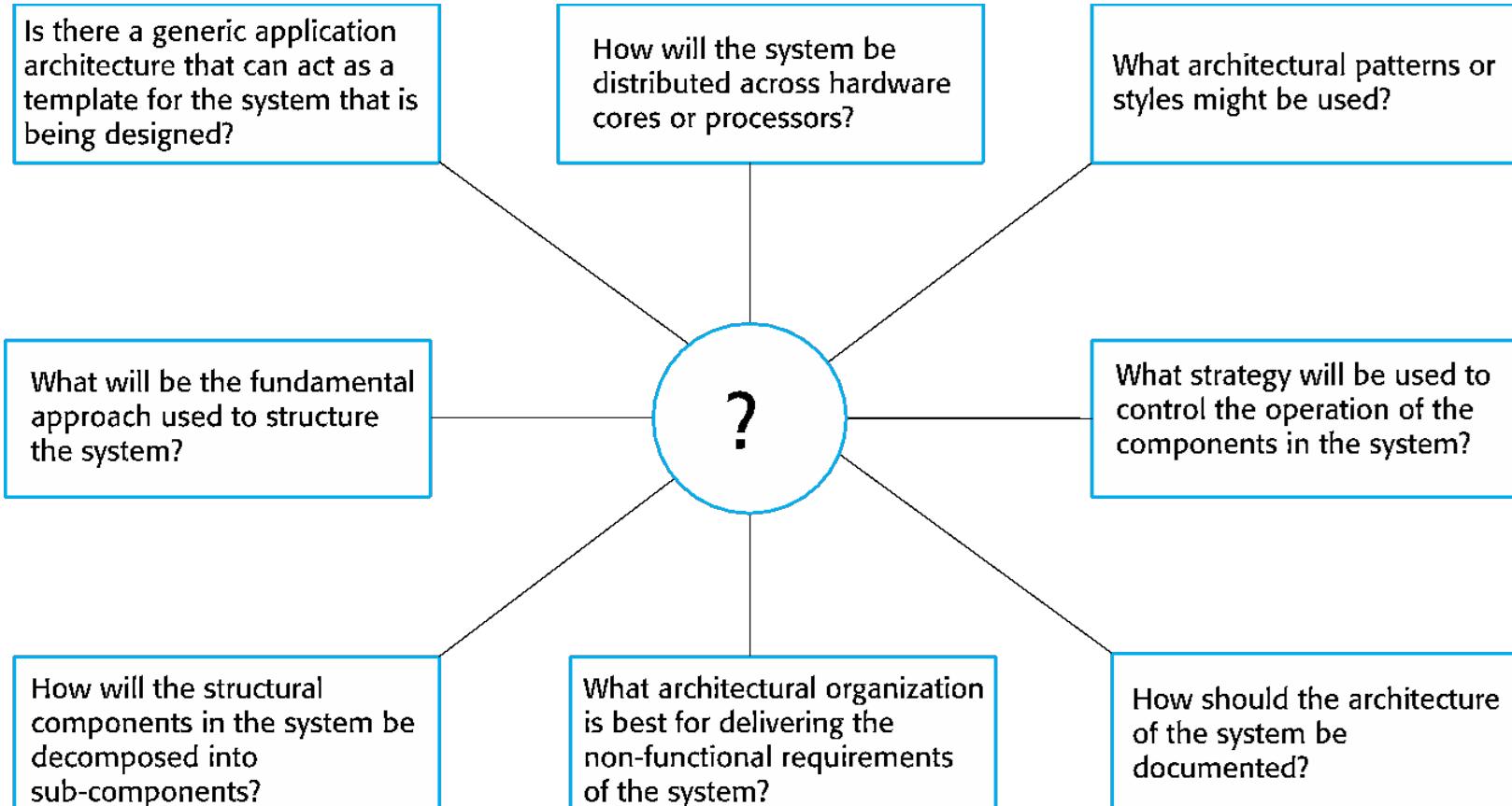
Architectural design decisions

- Architectural design is a creative process in which design to satisfy the functional and non-functional requirements of a system.
- Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain.
- Architectural design process depending on
 - the **type of system** being developed
 - the experience of the architect
 - **system requirements.**

Architectural design decisions

- Architectural design is considered as a series of decisions to be made rather than a sequence of activities.
- During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process.
- Based on their knowledge and experience, they have to consider some fundamental questions and made decisions based on it

Architectural design decisions



Architecture and non functional system characteristics

- Performance

Localize critical operations within a small number of components, with these components deployed on the same computer rather than distributed across the network to minimize communications. May need to use large rather than fine-grain components.

- Security

Use a layered architecture with critical assets in the inner layers and high- level security validations applied to these layers.

- Safety

Localize safety-critical features in a small number of subsystems.

Architecture and non functional system characteristics

- Availability

The architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system(Fault tolerance).

- Maintainability

Use fine-grain, replaceable components.

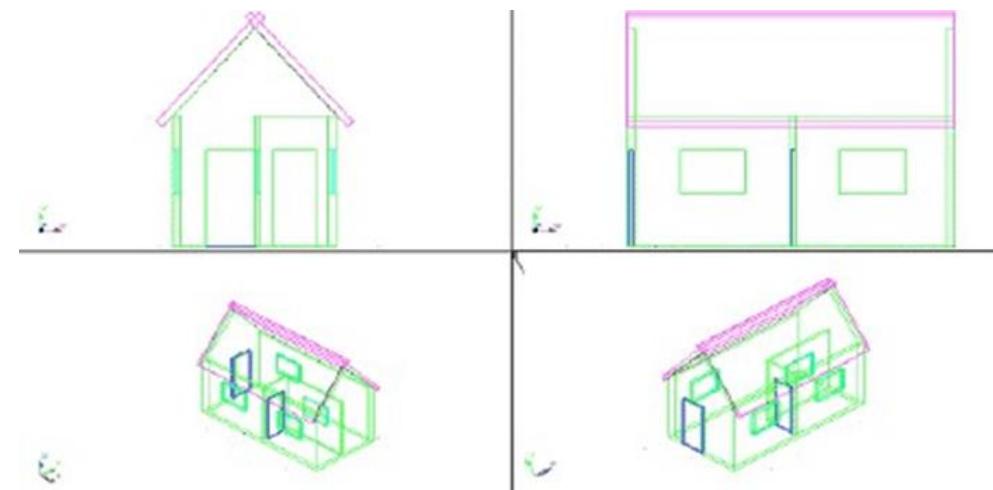
Architectural views

Architectural views

- What views or perspectives are useful when designing and documenting a system's architecture?
- What notations should be used for describing architectural models?
- Each architectural model only shows one view or perspective of the system.
 - It might show **how a system is decomposed into modules**, how the **run-time processes interact** or the **different ways in which system components are distributed across a network**.
- For both design and documentation, you usually need to present multiple views of the software architecture.

What is a View Model?

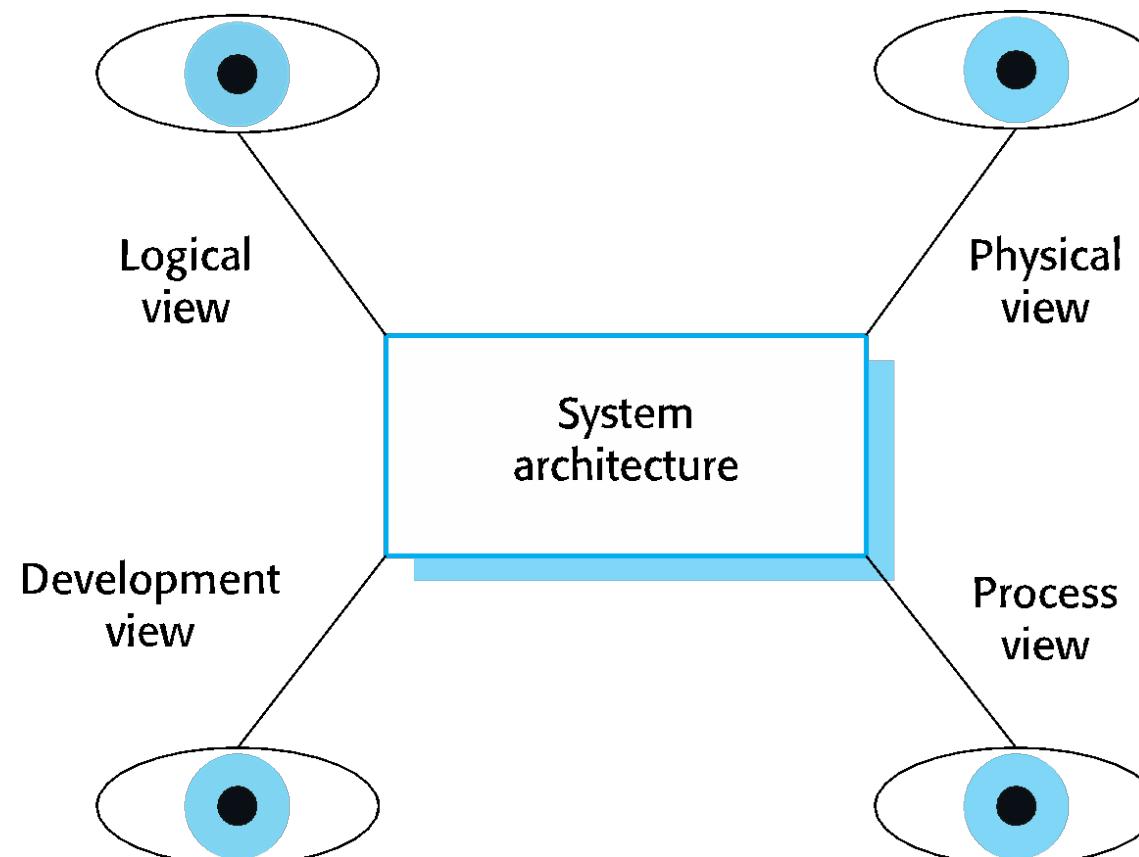
- Defines set of views to be used in designing software architecture.
- View is a representation of the whole system from **different perspectives**.



4 + 1 VIEW MODEL OF SOFTWARE ARCHITECTURE

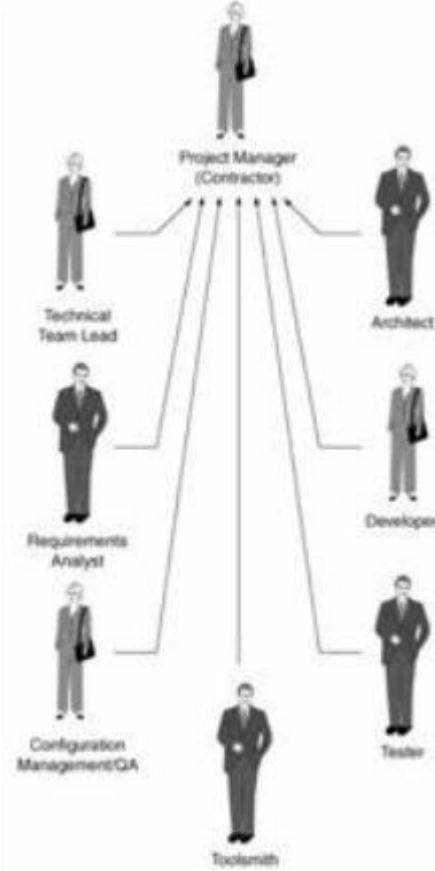
Designed by : Philippe Kruchten

4 + 1 view model of software architecture

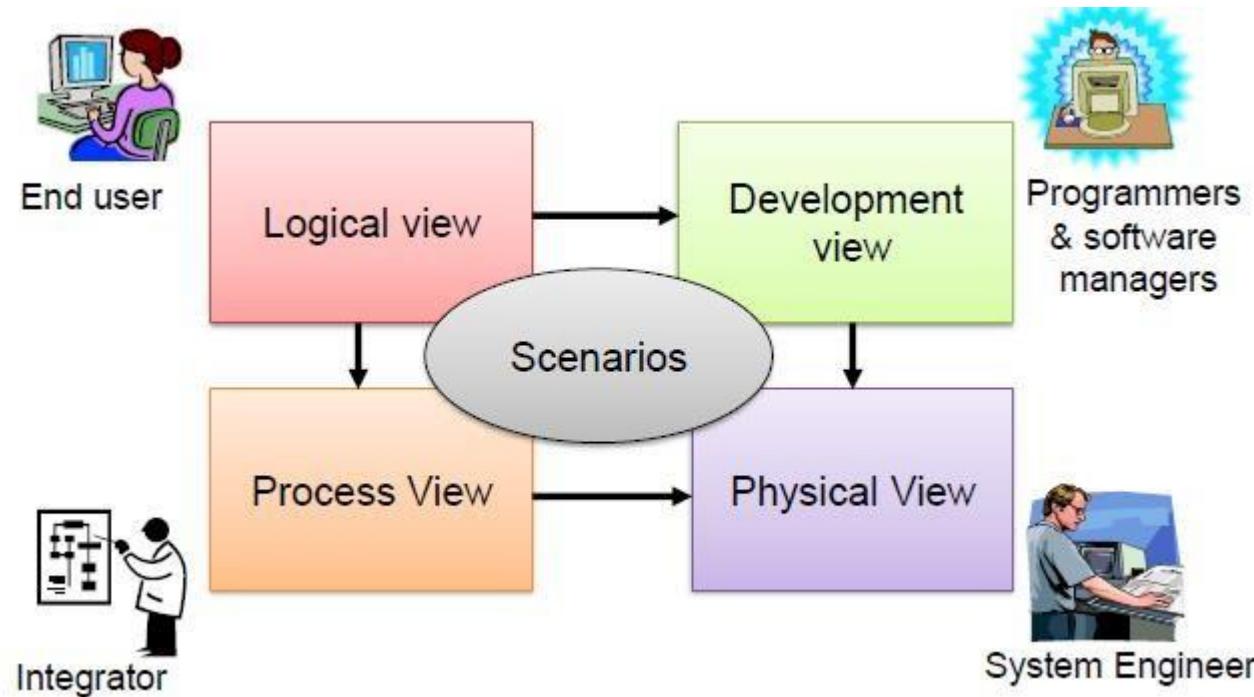


Intent of 4 + 1 View Model

- To separate different aspects of a software system into different views of the system.
- Why?
 - Different stakeholders have different interest of the system
 - It explains four fundamental architectural views, which can be linked through common use cases or scenarios.



4+1 View Model of Architecture



4 + 1 view model of software architecture

- Explains the system from the viewpoint of different stakeholders.
 - A logical view - shows the key abstractions in the system as objects or object classes.
 - A process view - shows how, at run-time, the system is composed of interacting processes.
 - A development view - shows how the software is decomposed for development.
 - A physical view - which shows the system hardware and how software components are distributed across the processors in the system.
- Plus one view to represent use cases / scenarios

Logical view

- **Viewer:** End-user
- **Considers:** Functional requirements
 - What the system should provide in terms of services to its users.
 - Shows the components of the system and their interaction.

Process View

(The process decomposition)

- **Viewer:** Integrators
- **Considers:** Non -functional requirements (concurrency, performance, scalability)
- Processes / workflow of a system and how processes communicate with each other.
- The process view focuses on the system's runtime behavior and deals with the system's dynamic elements. It explains the system processes and how they communicate with each other.

Development View

(Subsystem decomposition)

- **Viewer:** Programmers and Software Managers
- **Considers:** software module organization (Hierarchy of layers, software management, reuse, constraints of tools)
 - Represents the building block view of the system.
 - The implementation view is another name for this view.

Physical View

(Mapping the software to the Hardware)

- **Viewer:** System Engineers
- **Considers:** Non-functional requirements regarding to underlying hardware (Topology, Communication)
 - System execution environment
 - The deployment view is another name for this view.
 - The **deployment diagram** is one of the UML diagrams used to depict the physical perspective.

Scenarios

(Putting it all together)

- **Viewer:** All users of other views and Evaluators.
- **Considers:** System consistency, validity
 - Validation and illustration to show the design is complete.
 - The use case view is another name for this view.
- Sequences of interactions between objects and processes are described in the scenarios.

Views and Corresponding UML diagram

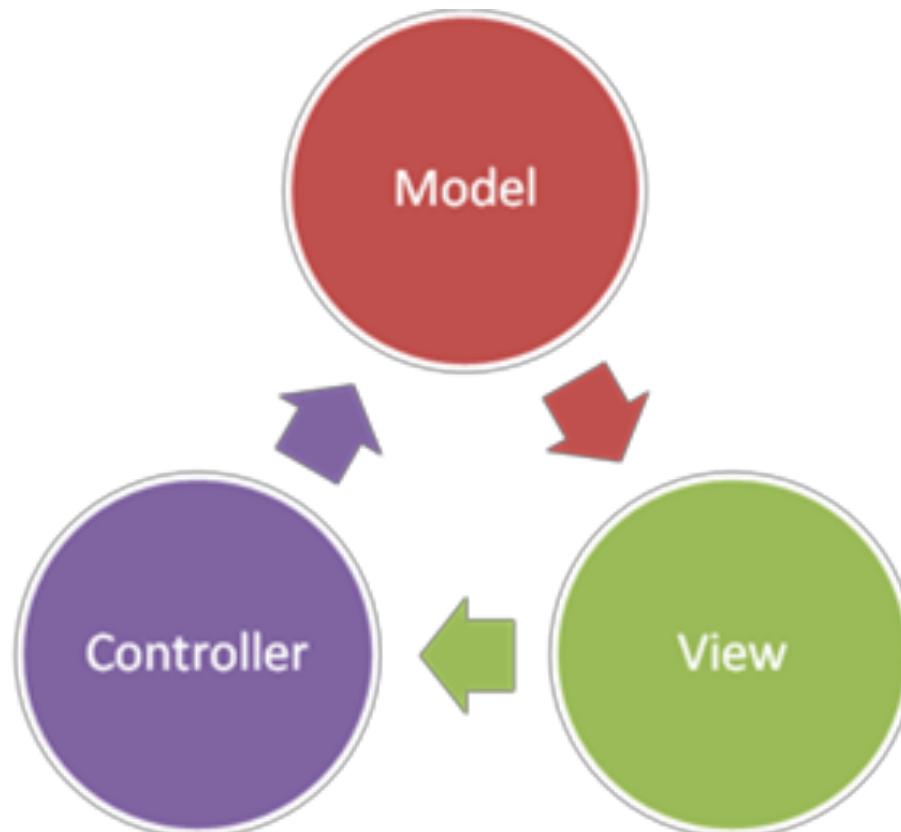
- Class diagrams and state diagrams are examples of UML diagrams that are used to depict the logical view.
→ Implementation view
- The package diagram is one of the UML diagrams used to depict the development view.
→ Deployment view
- The deployment diagram is one of the UML diagrams used to depict the physical view.
- The sequence diagram, communication diagram, and activity diagram are all UML diagrams that can be used to describe a process view.

Architectural patterns

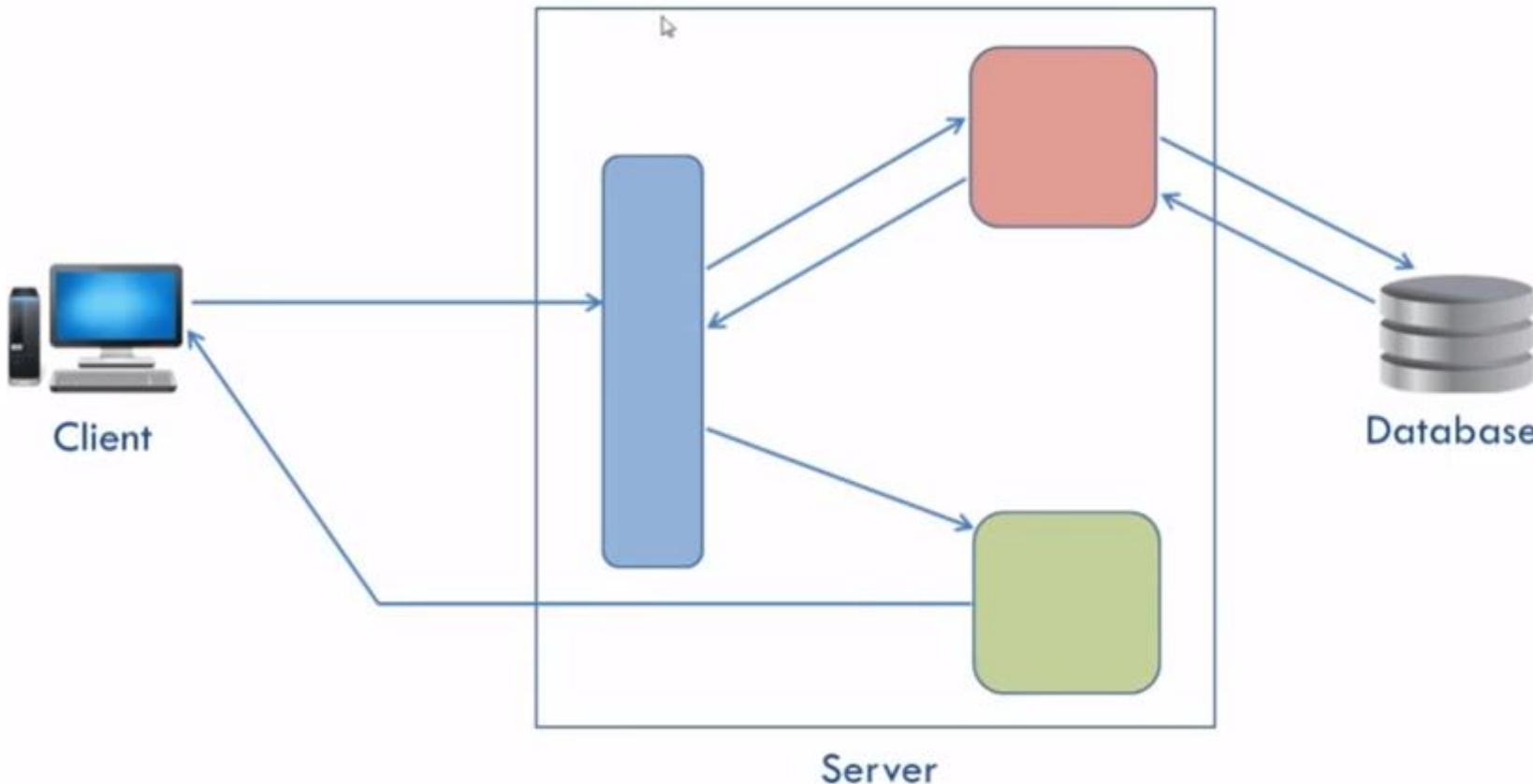
Architectural patterns

- A general, reusable solution to a commonly occurring problem in software architecture within a given context.
- Patterns are a means of **representing, sharing and reusing** knowledge.
- An architectural pattern is a **abstract description of good design practice, which has been tried and tested in different environments.**
- Patterns should include information about when they are and when they are not useful. Strengths and weakness of patterns also included in the architecture pattern.
- Patterns may be represented using tabular and graphical descriptions.
- The notions of separation and independence are fundamental to architectural design because they allow changes to be localized.

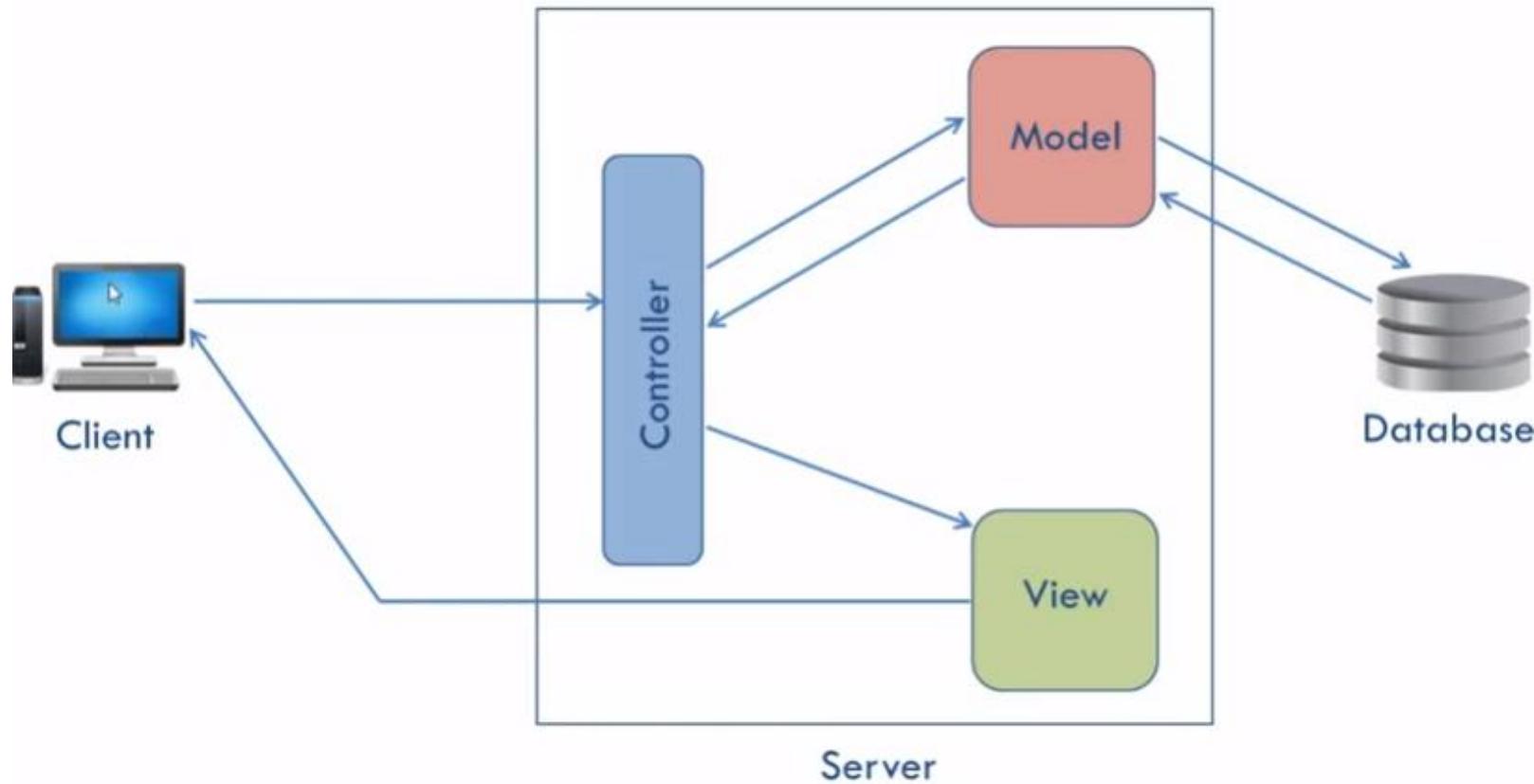
The Model-View-Controller (MVC) pattern



The Model-View-Controller (MVC) pattern



The Model-View-Controller (MVC) pattern



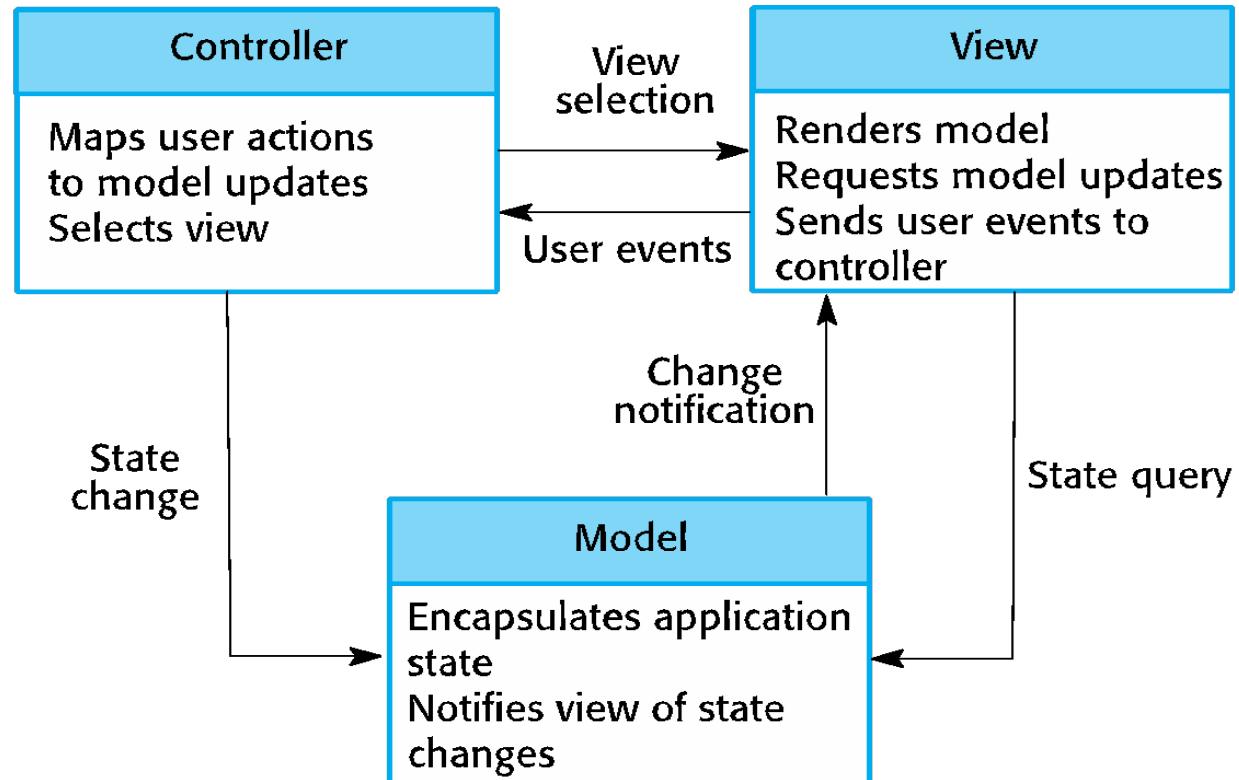
The Model-View-Controller (MVC) pattern

- **Model** : Manages the system data and associated operations on that data
- **View** : Defines and manages how the data is presented to the user.
- **Controller** : manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model

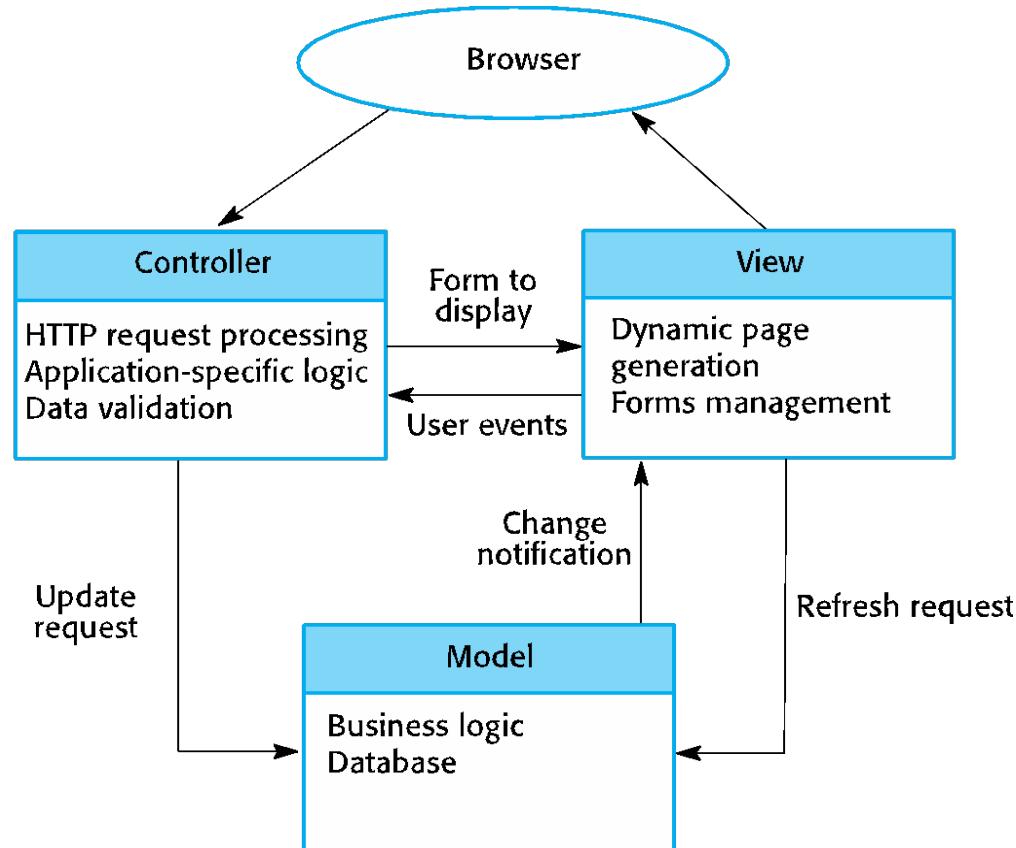
The Model-View-Controller (MVC) pattern - Summary

Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways, with changes made in one representation shown in all of them.
Disadvantages	May involve additional code and code complexity when the data model and interactions are simple.

The organization of the Model-View-Controller



Web application architecture using the MVC pattern



Layered architecture

- The Layered Architecture pattern is a way of achieving separation and independence.
- Organizes the system into a set of layers each of which provide a set of services.
- The system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.
- This layered approach supports the incremental development of systems. When a layer is developed, some of the services provided by that layer may be made available to users.

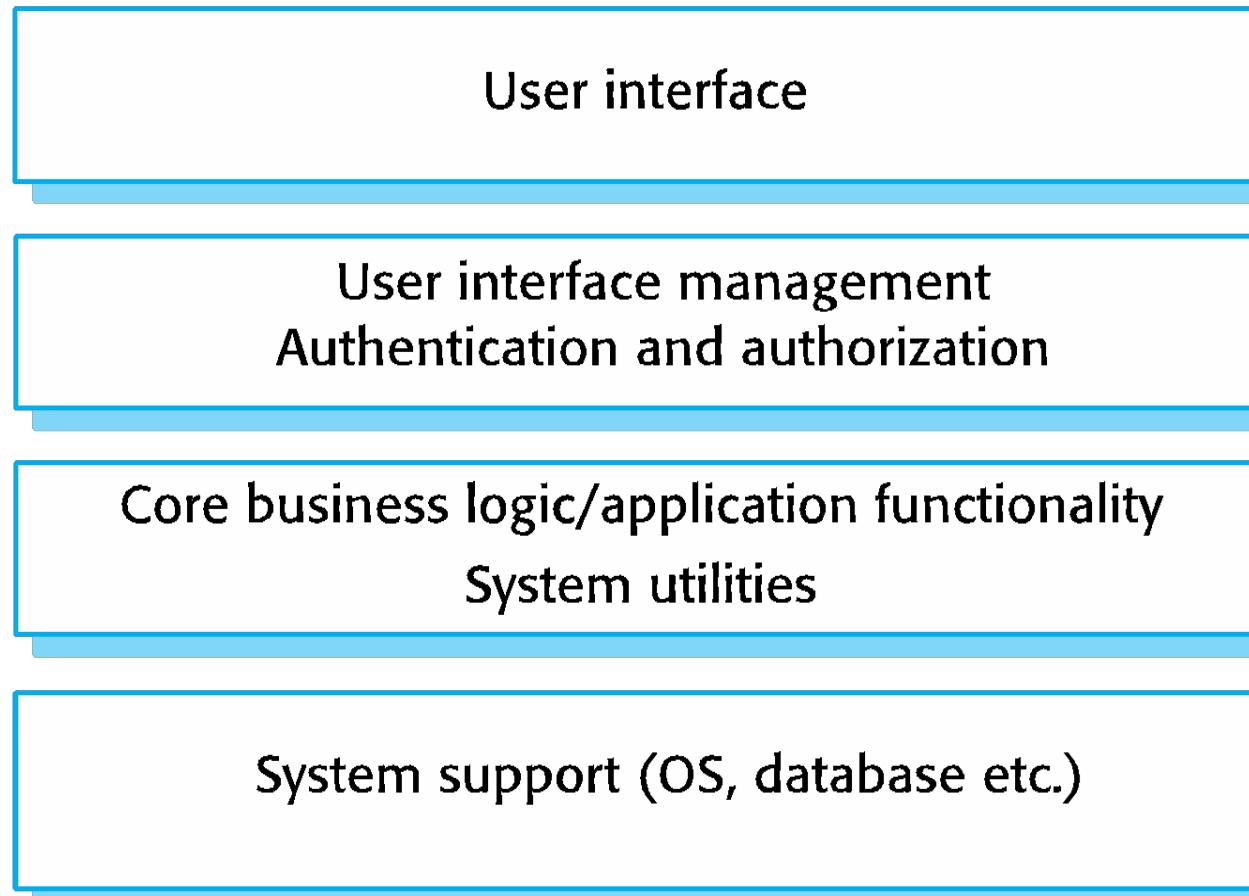
Layered architecture

- If interface is not changes, a new layer with extended functionality can replace an existing layer without changing other parts of the system.
- Furthermore, when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected.
- As layered systems localize machine dependencies, this makes it easier to provide multi-platform implementations of an application system. Only the machine dependent layers need be reimplemented to take account of the facilities of a different operating system or database.

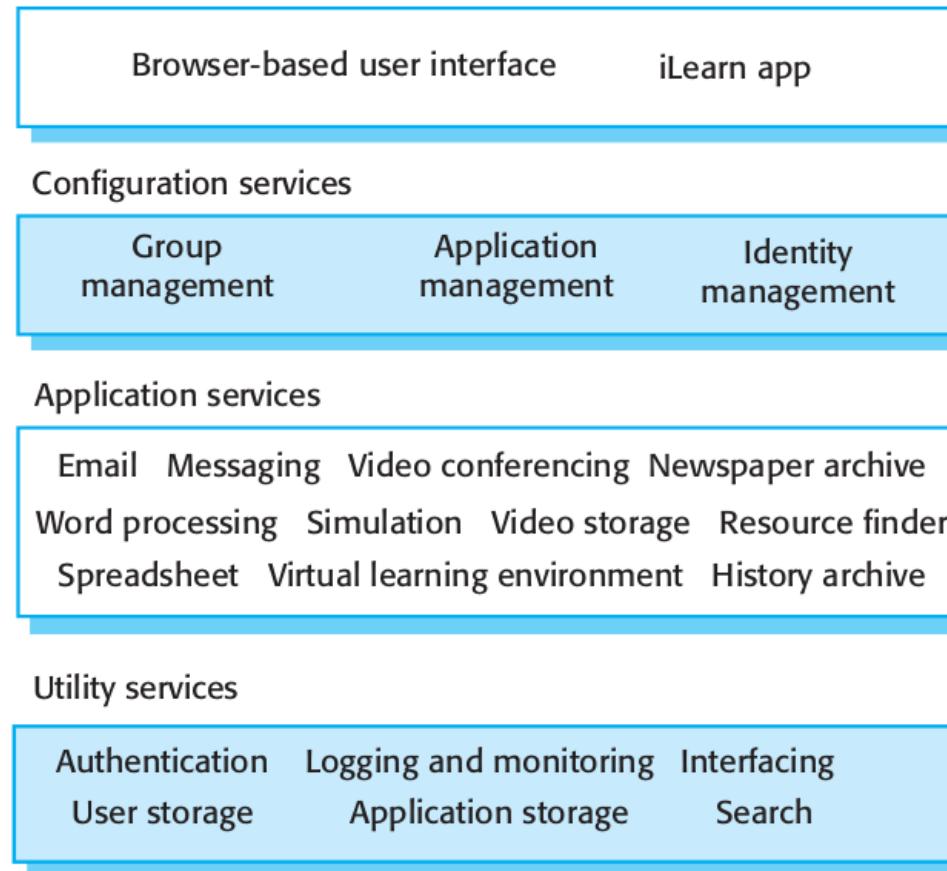
The Layered architecture pattern - Summary

Description	Organizes the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest level layers represent core services that are likely to be used throughout the system.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multilevel security.
Advantages	Allows replacement of entire layers as long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult, and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

A generic layered architecture



Example - iLearn System



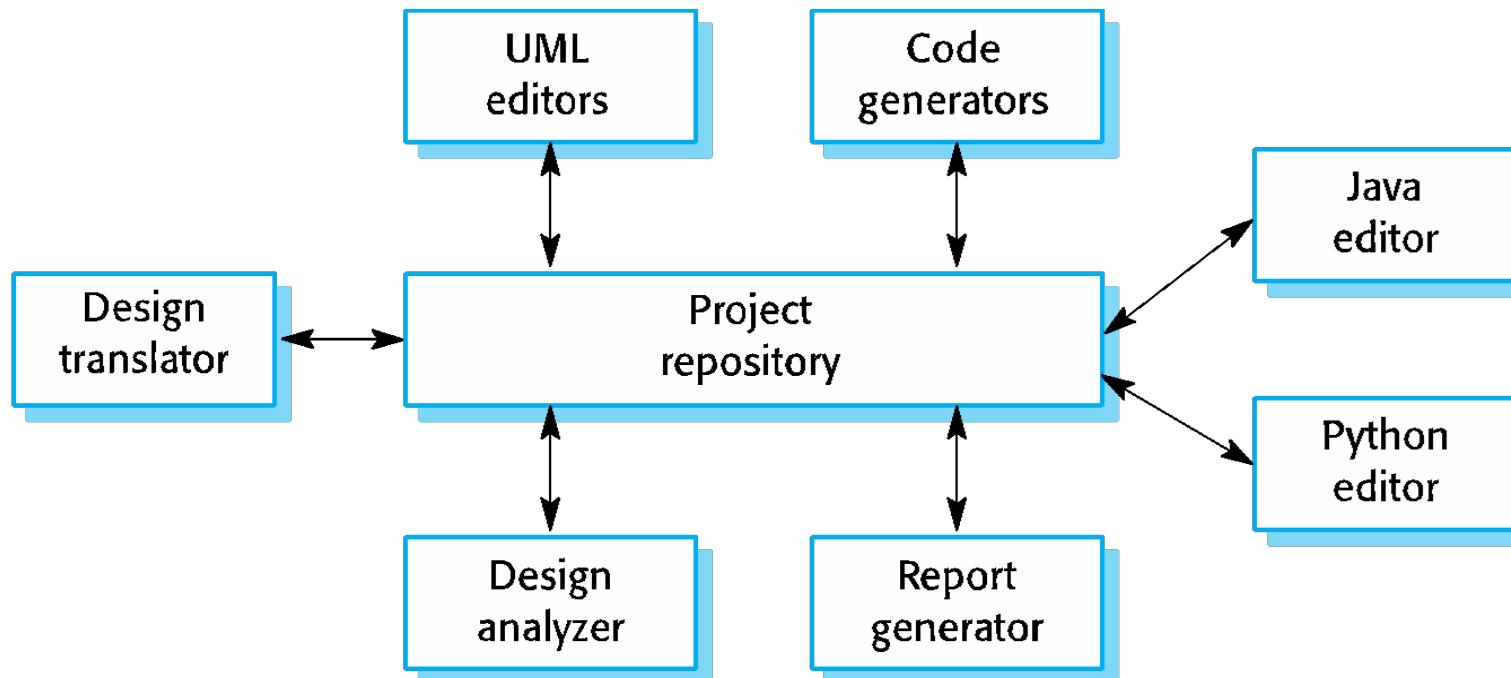
Repository architecture

- Describes how a set of interacting components can share data.
- Sub-systems must exchange data. This may be done in two ways:
 - Shared data is held in a central database or repository and may be accessed by all sub-systems;
 - Each sub-system maintains its copy of data.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.
- Examples of this type of system include command and control systems, management information systems, Computer- Aided Design (CAD) systems, and interactive development environments for software.

The Repository Architecture

Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent; they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

The Repository Architecture of IDE



The Repository Architecture

- It may be difficult or impossible to integrate new components if their data models do not fit the agreed schema.
- In practice, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, this involves maintaining multiple copies of data. Keeping these consistent and up to date adds more overhead to the system.
- The repository is passive and control is the responsibility of the components using the repository in the repository architecture.
- The Repository pattern is concerned with the static structure of a system and does not show its runtime organization.

Client-server architecture

- Model of the run time organization of the distributed systems.
- Client–server architectures are usually thought of as distributed systems architectures, but the logical model of independent services running on separate servers can be implemented on a single computer.
- Clients have to know the names of the available servers and the services they provide.
- However, servers do not need to know the identity of clients or how many clients are accessing their services.

Client-server architecture

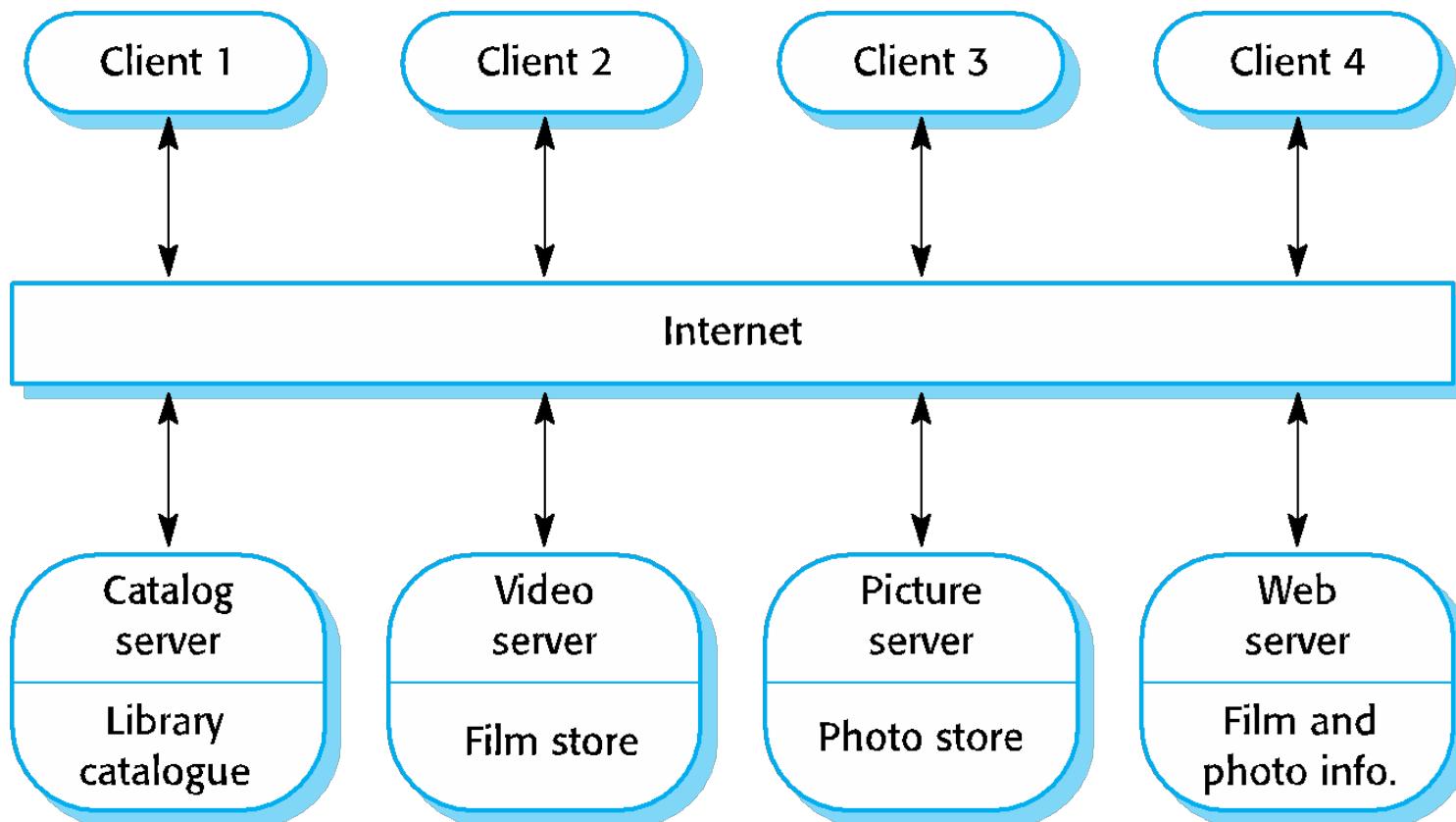
The major components of this model are:

- **A set of servers** that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server that offers programming language compilation services. Servers are software components, and several servers may run on the same computer.
- **A set of clients** that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
- **A network** that allows the clients to access these services. Client–server systems are usually implemented as distributed systems, connected using Internet protocols.

The Client–server architecture - Summary

Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

A client–server architecture for a film library



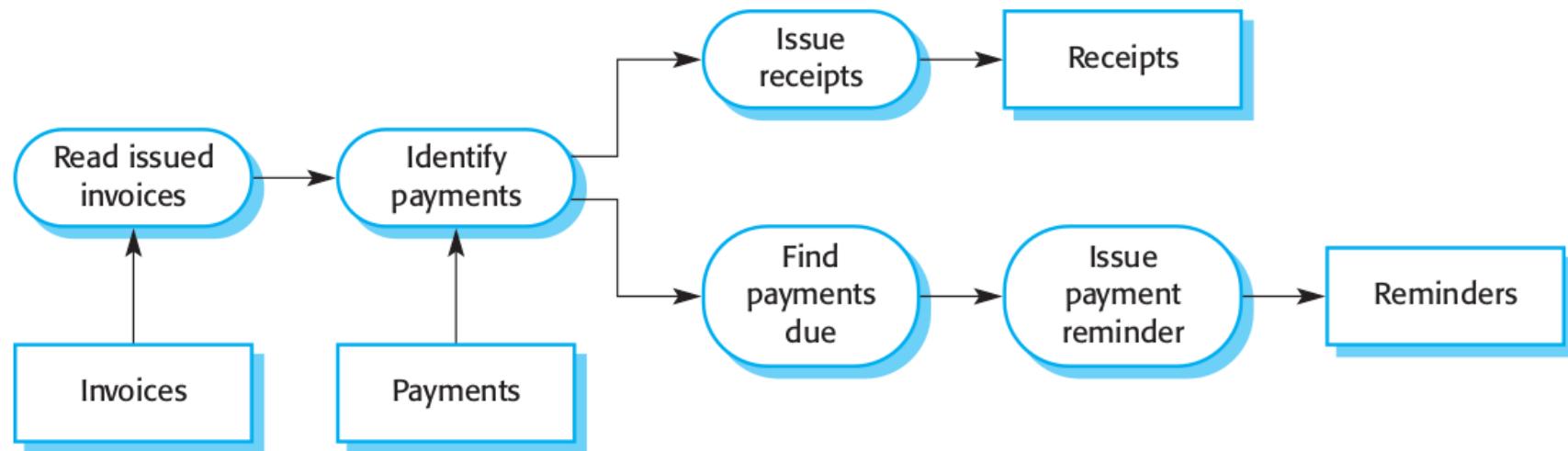
Pipe and filter architecture

- Model of the run time organization of the system where the functional transformations process their inputs to produce outputs.
- Data flows from one to another and transforms as it moves through the sequence.
- Each processing step is implemented as a transform.
- Input date flows through these transforms until converted to outputs. Transformations may execute sequentially / parallel.
- The data can be processed by each transform item by item or in a single batch.
- Not really suitable for interactive systems.

The pipe and filter pattern - Summary

Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
When used	Commonly used in data-processing applications (both batch and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse architectural components that use incompatible data structures.

An example of the pipe and filter architecture used in a payments system



Key points

- A software architecture is a description of how a software system is organized.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.
- Commonly used Architectural patterns include model-view-controller, layered architecture repository, client–server, and pipe and filter.



Design and Implementation – Part I

Course Code : CS2002

Ms. Mathangi Krishnathasan
[\(tmk@ucsc.cmb.ac.lk\)](mailto:tmk@ucsc.cmb.ac.lk)

Intended Learning Outcomes

- Explain the most important concepts in the Object-Oriented Design process.
- Discuss what is a design pattern and how these can be reused in designing software.
- Identify issues specific to the implementation.
- Explain the open-source development.

Lesson Outline

- Object-oriented design using the UML
- Design patterns
- Implementation Issues
- Open Source Development

Design and Implementation

- One of the stage in the software engineering process
- Develop an executable software system.
- Software design and implementation activities are invariably inter-leaved.
 - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
 - Implementation is the process of realizing the design as a program.

Object-Oriented Design using the UML

Object Oriented Design

- An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state.
- The representation of the state is private and cannot be accessed directly from outside the object.
- Designed using several interacting objects.
- Design object classes and the relationship between these classes.

An Object-Oriented Design Process

- Objects include both data and operations to manipulate that data.
- Changing the implementation of an object or adding services should not affect other system objects.
- Objects are created dynamically from the class definitions.
- Advantages of OOD?
 - Improves understandability
 - Increase maintainability

Process Stages

- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
 - Understand and define the context and the external interactions with the system.
 - Design the system architecture.
 - Identify the principal system objects.
 - Develop design models.
 - Specify object interfaces.

System Context and Interactions

- The relationships between the software that is being designed and its **external environment** is essential for deciding how to provide the required system functionality and how to **structure the system to communicate with its environment**.
- **Establish the boundaries of the system** helps you to decide
 - what features are implemented in the system being designed
 - what features are in other associated systems.

Context and interaction models

- A **system context model** is a structural model that demonstrates the other systems in the environment of the system being developed.
- An **interaction model** is a dynamic model that shows how the system interacts with its environment as it is used.

Context Models

- The context model of a system may be represented using associations.
- A simple block diagram can be used to document the environment of the system showing the entities in the system and their associations.

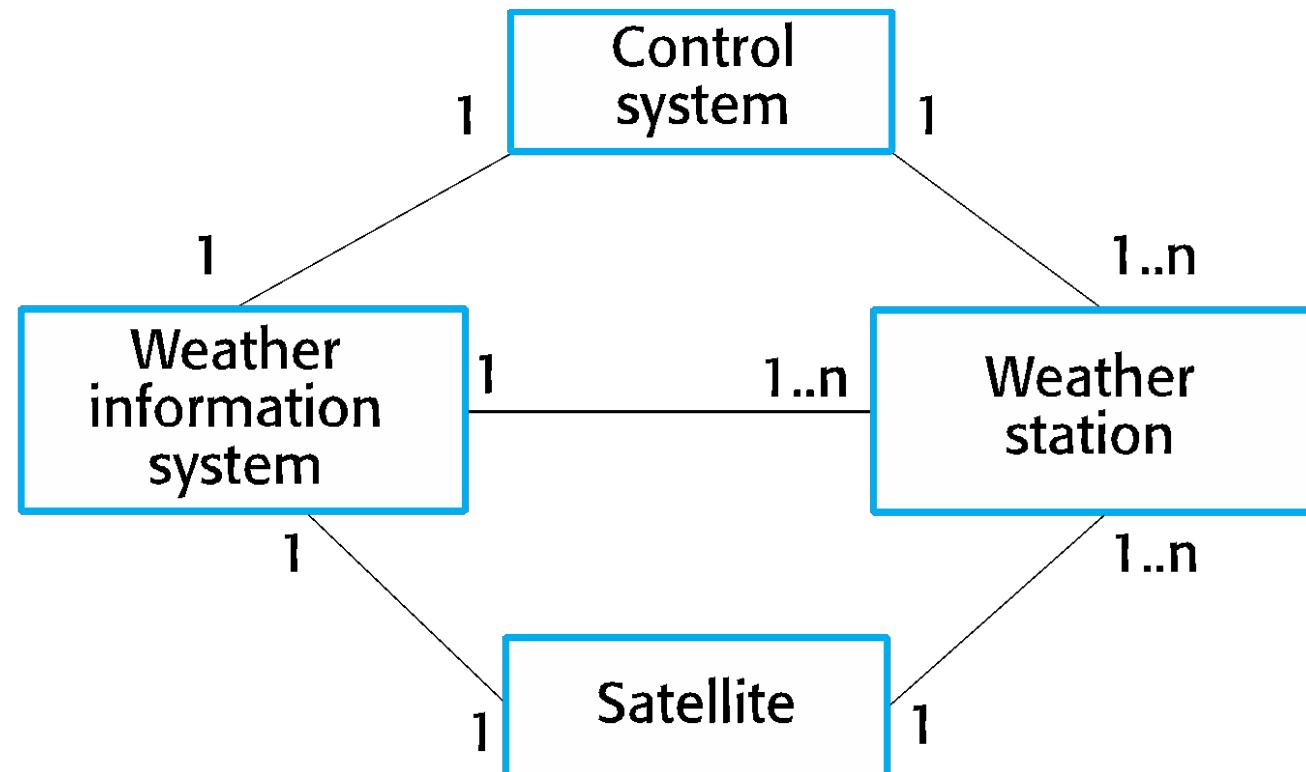
Interaction Models

- When you model the interactions of a system with its environment, you should use an abstract approach that does not include too much detail.
- One way to depict the interactions is to use a use case model.
- Use case represents an interaction with the system.

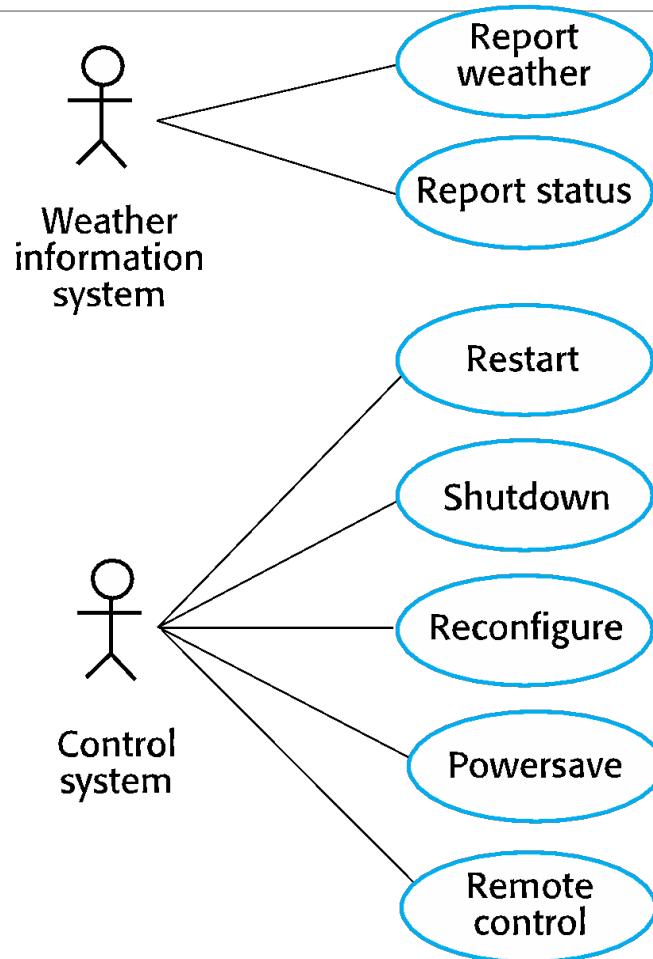
A wilderness weather station

- To help monitor climate change and to improve the accuracy of weather forecasts in remote areas, the government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.
- These weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.
- Wilderness weather stations are part of a larger system which is a weather information system that collects data from weather stations and makes it available to other systems for processing.

System context for the weather station



Weather station use cases



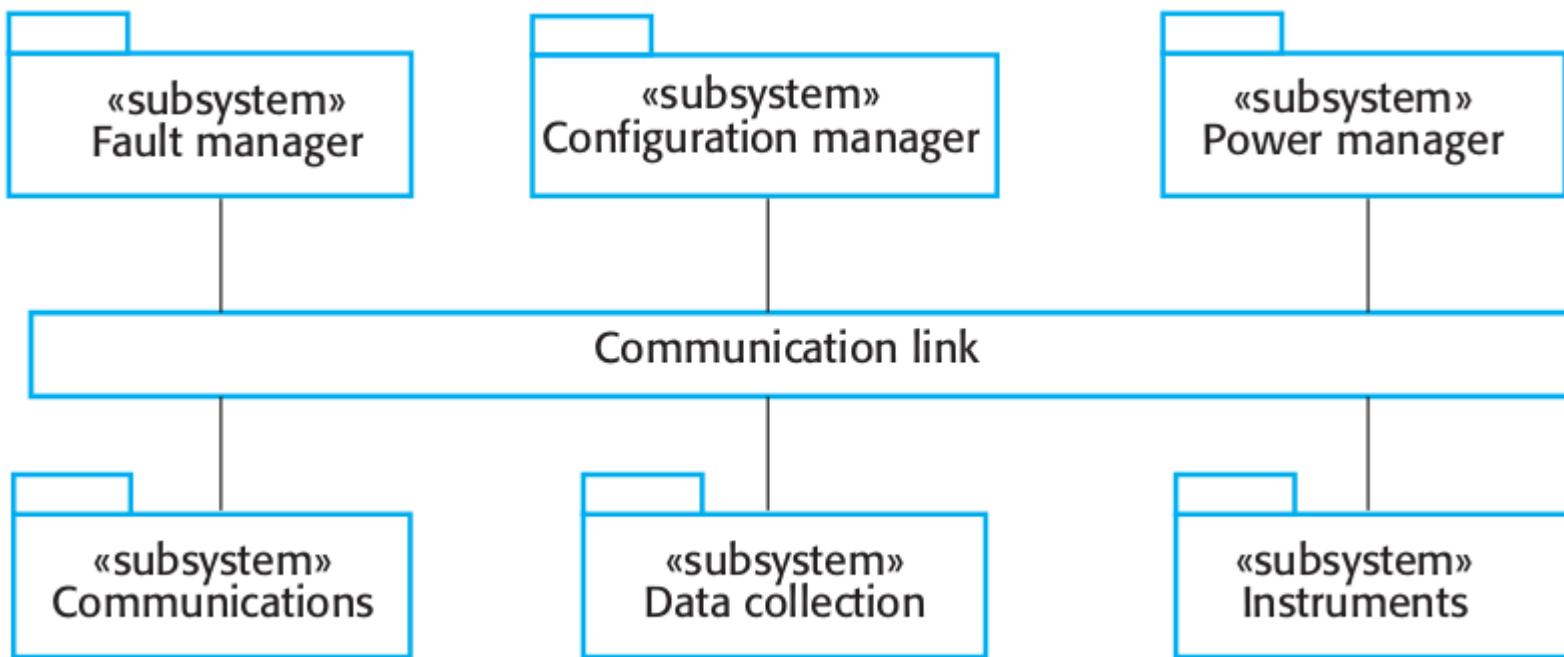
Weather station use case description : Report Weather

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future.

Architectural design

- Identify the major components that make up the system and their interactions.
- Organize the components using an architectural pattern such as a layered or client-server model.
- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.
- This “listener model” is a commonly used architectural style for distributed systems.
 - Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them.

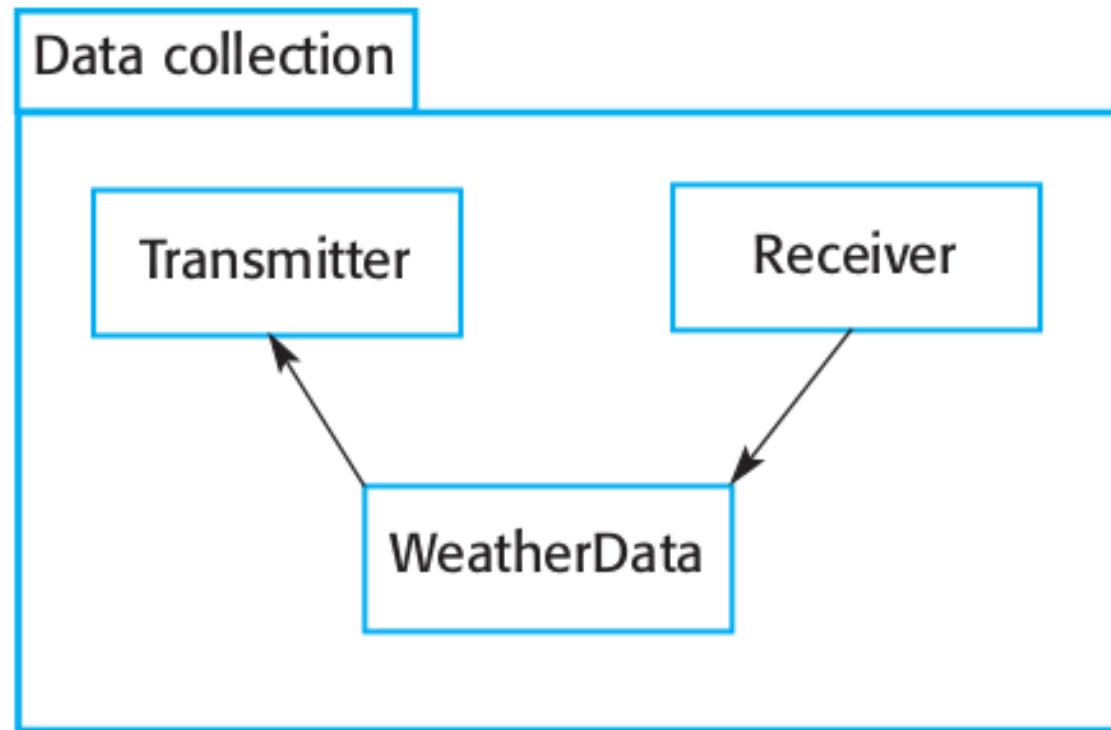
High-level architecture of the weather station - Listener Model



High-level architecture of the weather station - Listener Model

- When the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way.
- The key benefit of this architecture is that it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem.

Architecture of data collection subsystem



Object class identification

- Identifying object classes is often a difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification

- Use a grammatical approach based on a natural language description of the system.
- Base the identification on tangible things in the application domain.
- Use a scenario-based analysis.

The objects, attributes and methods in each scenario are identified.

Approaches to identification - Weather station

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future.

Approaches to identification - Weather station

- The **Ground thermometer, Anemometer, and Barometer** objects are application domain objects, and the **Weather Station** and **Weather Data** objects have been identified from the system description and the scenario (use case) description.
- The **Weather Station** object class provides the basic interface of the weather station with its environment.
- The **Weather Data** object class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.
- The **Ground thermometer , Anemometer , and Barometer** object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware.
- These objects operate autonomously to collect data at the specified frequency and store the collected data locally.
- This data is delivered to the **Weather Data** object on request.

Object classes - Weather station

WeatherStation
identifier
reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

WeatherData
airTemperatures
groundTemperatures
windSpeeds
windDirections
pressures
rainfall
collect ()
summarize ()

Ground thermometer
gt_Ident
temperature
get ()
test ()

Anemometer
an_Ident
windSpeed
windDirection
get ()
test ()

Barometer
bar_Ident
pressure
height
get ()
test ()

Design models

- Design models show the objects or object classes and relationships between these entities.
- These models are the bridge between the system requirements and the implementation of a system.
- There are two kinds of design model:
 - **Structural models** : describe the static structure of the system in terms of object classes and relationships.
 - **Dynamic models** : describe the dynamic structure of the system and show the expected runtime interactions between the system objects.

Examples of design models

- **Subsystem models** that show logical groupings of objects into coherent subsystems.
- **Sequence models** that show the sequence of object interactions.
- **State machine models** that show how individual objects change their state in response to events.

Subsystem models

- Shows how the design is organized into logically related groups of objects.
- Detailed object models show the objects in the systems and their associations.(inheritance, generalization, aggregation and etc.)
- The actual organization of objects in the system may be different.

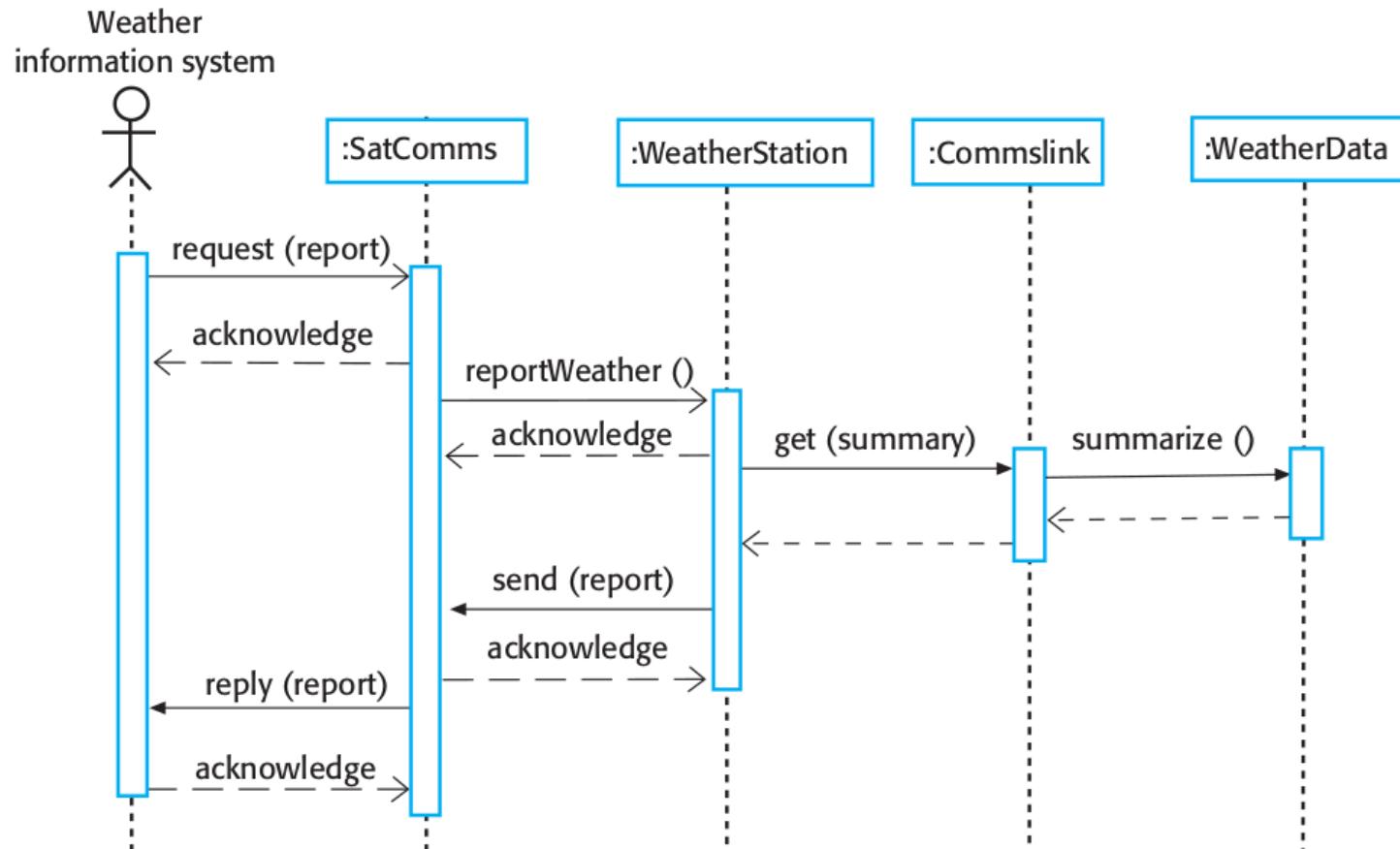
Sequence models

- Sequence models show the sequence of object interactions that take place.
- When documenting a design, you should produce a sequence model for each significant interaction.

Sequence Diagram :

- Objects are arranged horizontally across the top.
- Time is represented vertically so models are read top to bottom.
- Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction.
- A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

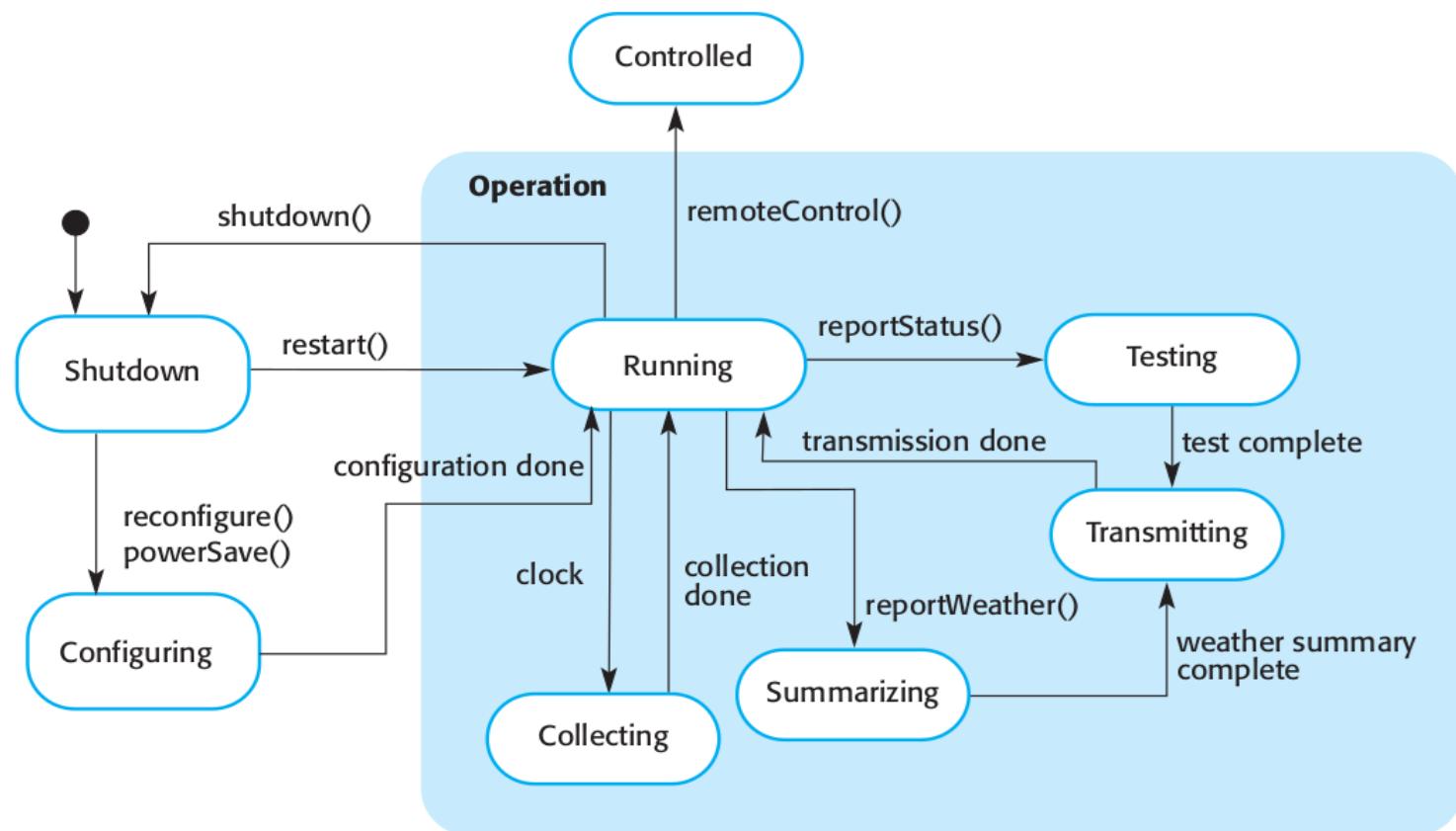
Sequence diagram describing data collection



State diagrams

- State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- State diagrams are useful high-level models of a system or an object's run-time behavior.
- State diagrams for all of the objects in the system are NOT required.
 - Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

Weather station state diagram



Interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Interfaces can be specified in the UML using the same notation as a class diagram.
- However, there is no attribute section, and the UML stereotype «interface» should be included in the name part.
- Objects may have several interfaces which are viewpoints on the methods provided.
- Attributes are **not defined** in an interface specification.

Weather Station Interfaces

**«interface»
Reporting**

weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport

**«interface»
Remote Control**

startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string

Design patterns

Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Patterns

- Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.
- A **design pattern** is a general repeatable solution to a commonly occurring problem in software **design**. A **design pattern** isn't a finished **design** that can be transformed directly into code.
- It is a **description** or **template** for **how to solve a problem** that can be **used in many different situations**

Object-oriented Concepts

- Abstraction
- Inheritance
- Polymorphism
- Encapsulation

Object-Oriented Concepts (OOP)

- **Abstraction**

- Abstraction is about hiding unwanted details while showing most essential information.
- A mechanism to reduce and filter out details so that one can focus on a few concepts at a time.
- focuses on **what an object does**, instead of how an object is represented or “how it works.”
- Often used for managing large and complex programs
 - E.g. - Human Being's can talk, walk, hear, eat, but the details of the muscles mechanism and their connections to the brain are hidden from the outside world.

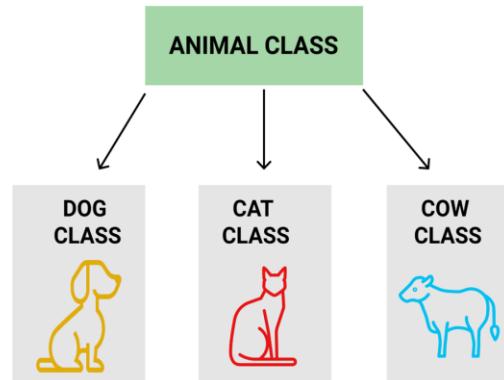
Object- Oriented Concepts

- **Inheritance**

- Methods and/or attributes defined in an object class can be inherited or reused by another object class.
- some individuals in the room might be classified as STUDENTS and TEACHERS.

Thus, STUDENT and TEACHER object classes are members of the object class

PERSON



Object-oriented Concepts

- **Polymorphism**
 - Allows different forms of the same service to be defined.
 - Sometimes an operation has the same name in different classes.

eg. You can open a door, you can open a window, and you can open a newspaper, a bank account or a conversation.
 - In each case, you are performing a different operation.
 - In object-orientation, each class “knows” how that operation is supposed to take place.

Object- Oriented Concepts

- **Encapsulation**
 - Packaging of several items together into one unit (both attributes and behavior of the object)
 - The only way to access or change an object's attribute is through that object's specific behavior.
 - Objects encapsulates what they do.
That is, they hide the inner workings of their operations
 - **from the outside world**
 - **and from other objects**
 - Also known as information hiding

Pattern Elements

- Name
 - A meaningful pattern identifier.
- Problem description.
- Solution description.
 - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- Consequences
 - The results and trade-offs of applying the pattern.

Types of Design Patterns

Creational Patterns

- These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.

Structural Patterns

- These design patterns concern class and object composition.

Behavioral Patterns

- These design patterns are specifically concerned with communication between objects.

The Observer pattern

Name

- Observer

Description

- Separates the display of object state from the object itself.

Problem description

- Used when multiple displays of state are needed.

Solution description

- See slide with UML description.

Consequences

- Optimizations to enhance display performance are impractical.

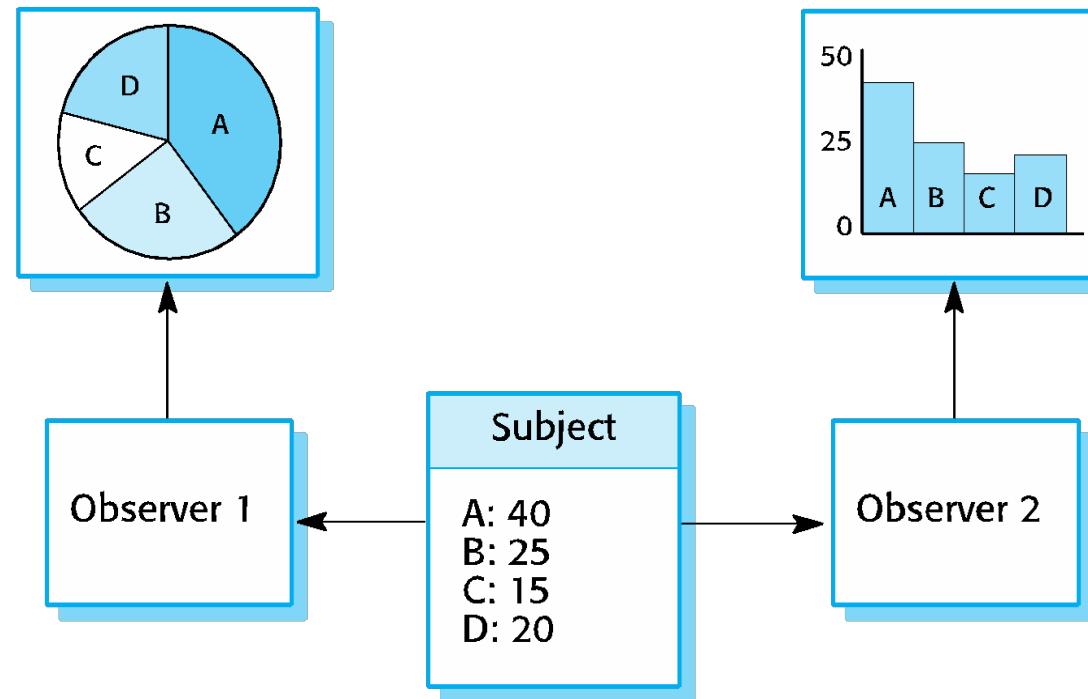
The Observer pattern (1)

Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

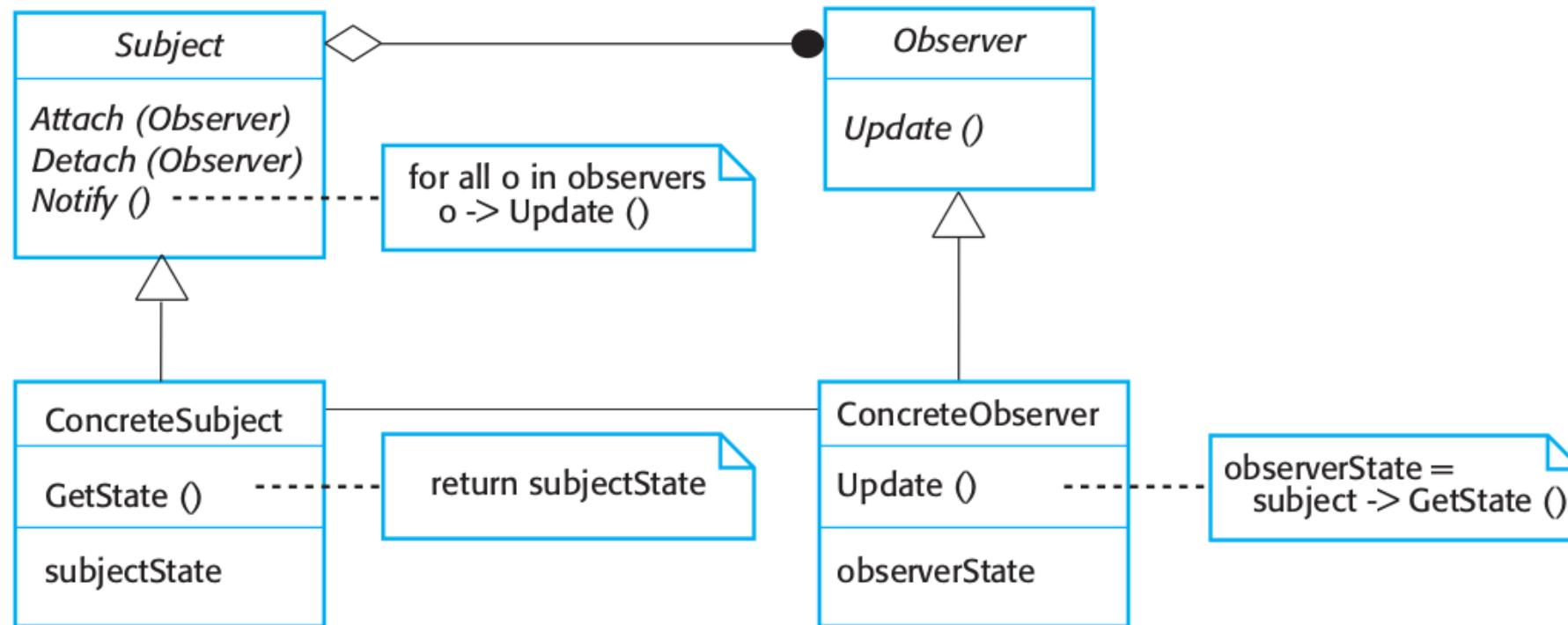
The Observer pattern (2)

Pattern name	Observer
Solution description	<p>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.</p> <p>The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.</p>
Consequences	<p>The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.</p>

Multiple displays using the Observer pattern



A UML model of the Observer pattern



Design problems

- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
 - Tell several objects that the state of some other object has changed (Observer pattern).
 - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
 - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
 - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Key points

- Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.



Design and Implementation – Part II

Course Code : CS2002

Ms. Mathangi Krishnathasan
[\(tmk@ucsc.cmb.ac.lk\)](mailto:tmk@ucsc.cmb.ac.lk)

Intended Learning Outcomes

- Explain the most important concepts in the Object-Oriented Design process.
- Discuss what is a design pattern and how these can be reused in designing software.
- Identify issues specific to the implementation.
- Explain the open-source development.

Lesson Outline

- Object-oriented design using the UML
- Design patterns
- **Implementation Issues**
- **Open Source Development**

Implementation Issues

- A critical stage of software development process is system implementation, where you create an executable version of the software.
- Some aspects of implementation that are particularly important to software engineering are as follows,
 - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
 - **Configuration management** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
 - **Host-target development** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Reuse

- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
 - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.
- A reuse-based approach is now widely used for web-based systems of all kinds, scientific software, and, increasingly, in embedded systems engineering.

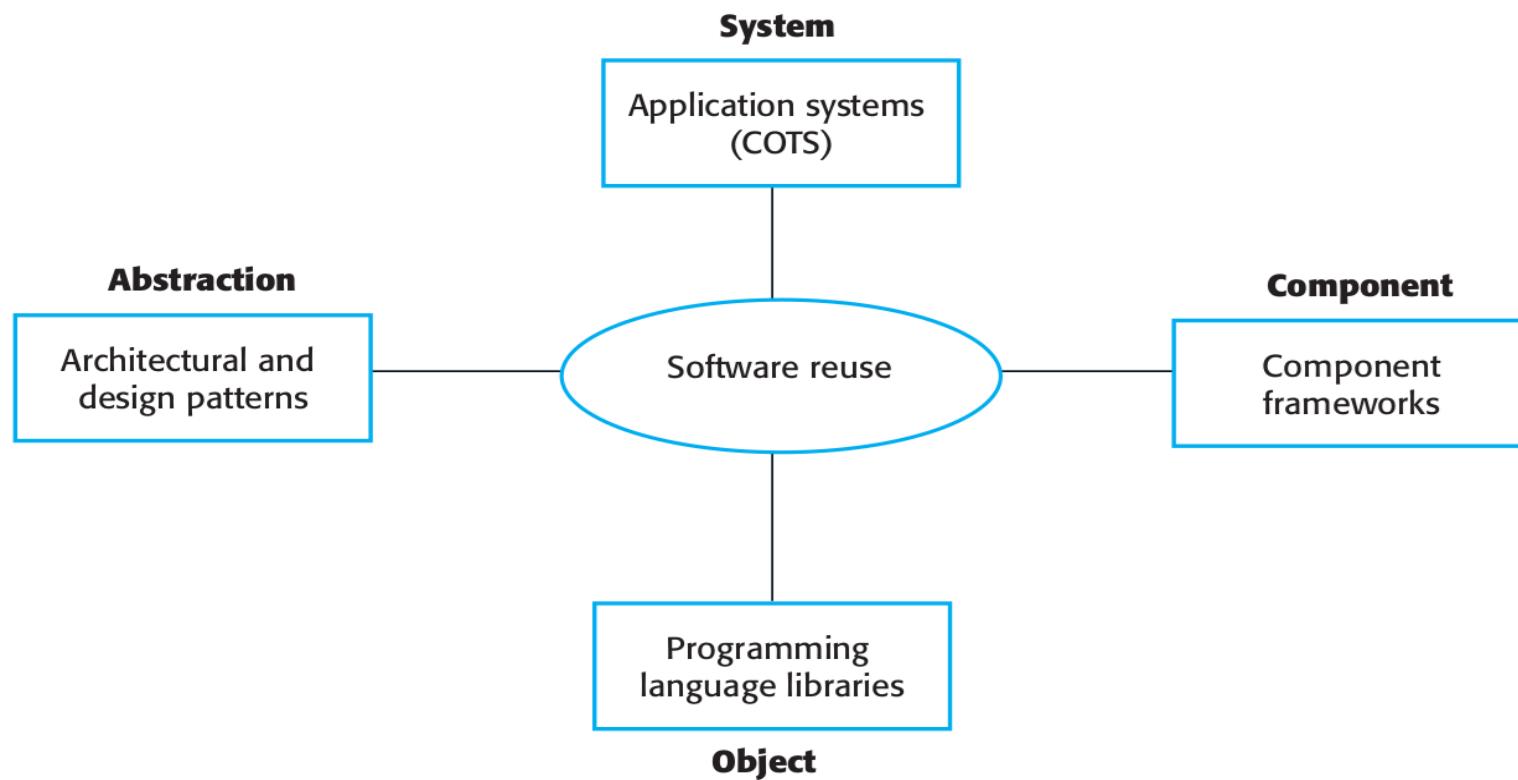
Reuse Levels

- The abstraction level
 - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
 - Design patterns and architectural patterns are ways of representing abstract knowledge for reuse.
- The object level
 - At this level, you directly reuse objects from a library rather than writing the code yourself.
 - To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need.
 - Eg: Java Mail library objects and methods can be used to process email messages in a Java program

Reuse Levels

- The component level
 - Components are collections of objects and object classes that operate together to provide related functions and services.
 - You often have to adapt and extend the component by adding some code of your own.
 - An example of component-level reuse is where you build your user interface using a framework.
- The system level
 - At this level, you reuse entire application systems.
 - This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code or by using the system's own configuration interface.

Software Reuse



Reuse Costs

- By reusing existing software, you can develop new systems more quickly, with fewer development risks and at lower cost.
- As the reused software has been tested in other applications, it should be more reliable than new software. However, there are some costs associated with reuse:
 - The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
 - Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
 - The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
 - The costs of integrating reusable software elements with each other and with the new code that you have developed.

Configuration Management

- In software development, change happens all the time, so change management is absolutely essential.
- It is a process of managing a changing software system.
- The aim of configuration management is to **support the system integration process** so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- There are four fundamental configuration management activities.

Configuration Management Activities

- **Version management :**

- Provide support to keep track of the different versions of software components.
- Version management systems include facilities to coordinate development by several programmers.
- They stop one developer from overwriting code that has been submitted to the system by someone else.

- **System integration :**

- Provide support to help developers define what versions of components are used to create each version of a system.
- This description is then used to build a system automatically by compiling and linking the required components.

Configuration Management Activities

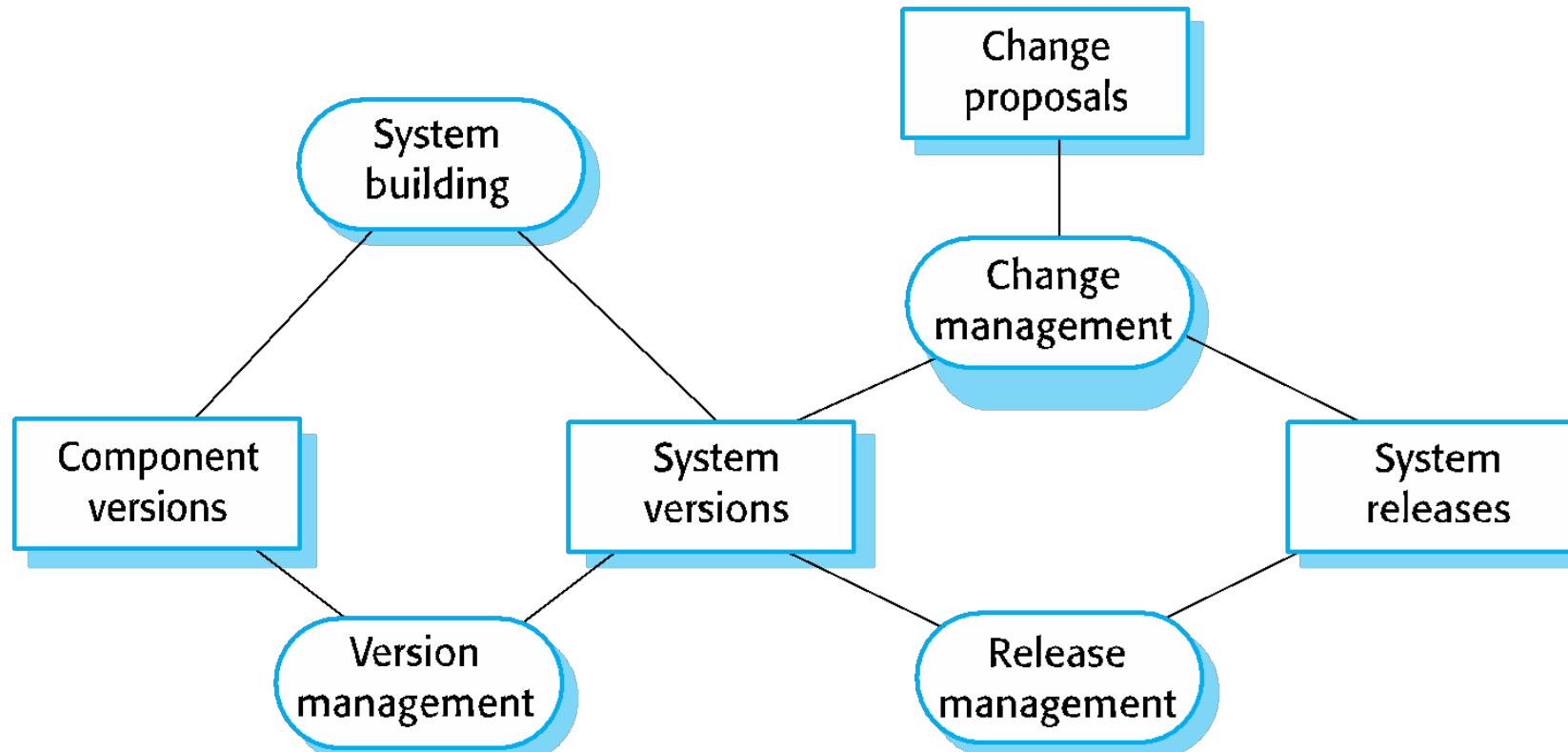
- **Problem tracking :**

- Provide support to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

- **Release management :**

- Release management, where new versions of a software system are released to customers.
 - Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

Configuration Management



Configuration Management

- Software configuration management tools support each of the above activities. These tools are usually installed in an integrated development environment, such as Eclipse.
- Version management may be supported using a version management system such as Subversion or Git which can support multi-site, multi-team development.
- System integration support may be built into the language or rely on a separate toolset such as the GNU build system.
- Bug tracking or issue tracking systems, such as Bugzilla, are used to report bugs and other issues and to keep track of whether or not these have been fixed.

Host-target Development

- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
 - A platform is more than just hardware.
 - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment. Eg: if you develop in Java, the target environment is the Java Virtual Machine.

Host-target Development

- Sometimes, the development platform and execution platform are the same, making it possible to develop the software and test it on the same machine.
- However, particularly for embedded systems and mobile systems, the development and the execution platforms are different.
- You need to either move your developed software to the execution platform for testing or run a simulator on your development machine.

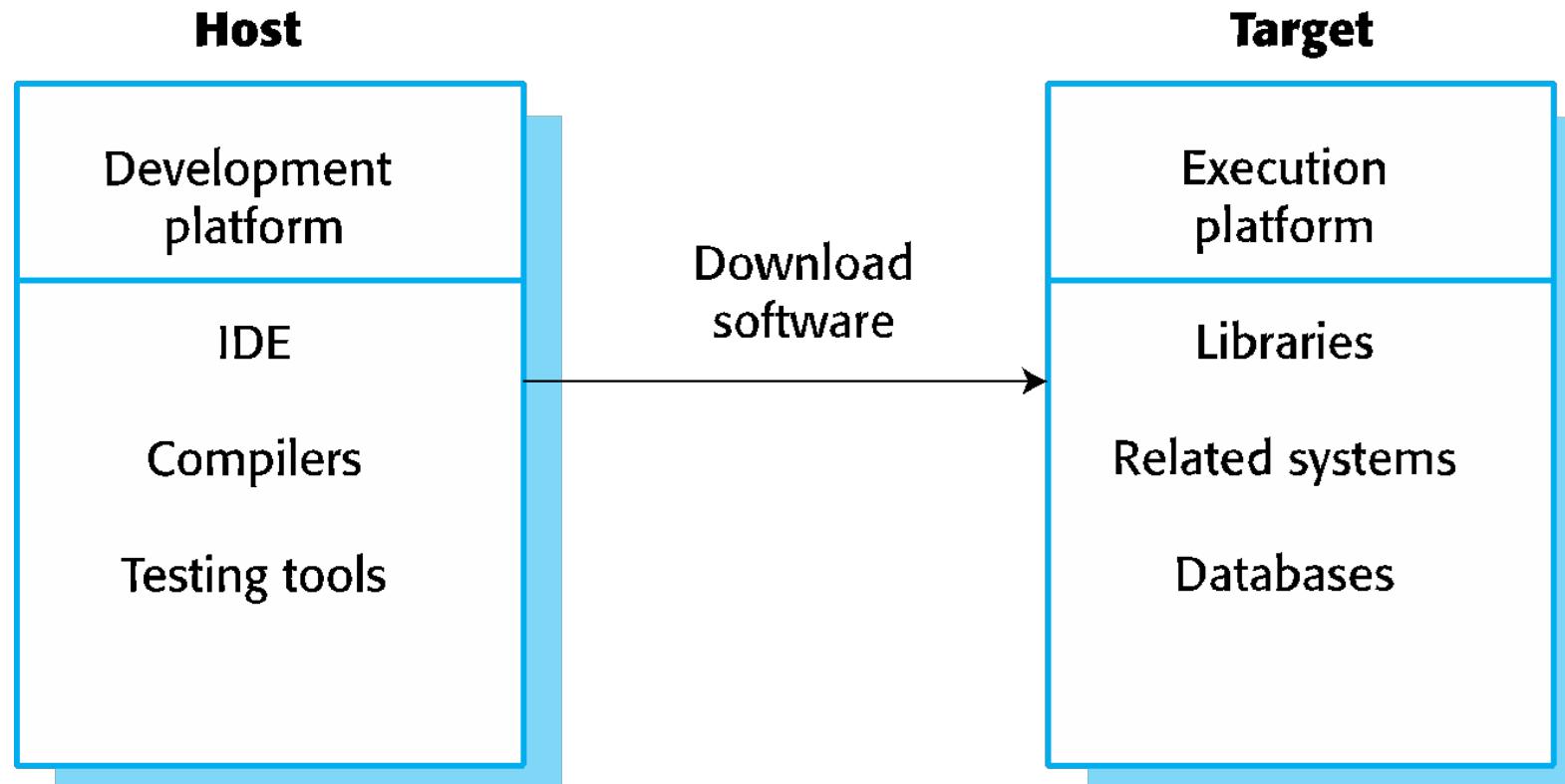
Host-target Development - Simulators

- Simulators are often used when developing embedded systems.
- You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed.
- Simulators speed up the development process for embedded systems as each developer can have his or her own execution platform with no need to download the software to the target hardware.
- However, simulators are expensive to develop and so are usually available only for the most popular hardware architectures.

Development Platform Tools

- A software development platform should provide a range of tools to support software engineering processes.
 - An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
 - A language debugging system.
 - Graphical editing tools, such as tools to edit UML models.
 - Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
 - Tools to support refactoring and program visualization.
 - Configuration management tools to manage source code versions and to integrate and build systems.

Host-target Development



Host-target Development

- Software development tools are now usually installed within an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development within some common framework and user interface.
- Generally, IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially or may be an instantiation of a general-purpose IDE, with specific language-support tools.
- A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together. The best-known general-purpose IDE is the Eclipse environment

Host-target Development - Deployment

- As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform.
- Issues that you have to consider in making this decision are:
 - The hardware and software requirements of a component
 - The availability requirements of the system
 - Component communications

Open - source Development

Open-source development

- An approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process.
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code.
- Open-source software extended this idea by using the Internet to recruit a much larger population of volunteer developers.
- Important and universally used open-source products are Java, the Eclipse IDE, and the mySQL database management system.

Open-source development - Benefits

- In principle at least, any contributor to an open-source project may report and fix bugs and propose new features and functionality. However, in practice, successful open-source systems still rely on a core group of developers who control changes to the software.
- It is usually cheap or even free to acquire open-source software. You can normally download open-source software without charge. However, if you want documentation and support, then you may have to pay for this, but costs are usually fairly low.
- Widely used open-source systems are very reliable. They have a large population of users who are willing to fix problems themselves rather than report these problems to the developer and wait for a new release of the system.
- Bugs are discovered and repaired more quickly than is usually possible with proprietary software.

Open-source Development - Issues

- For a company involved in software development, there are two open-source issues that have to be considered:
 - Should the product that is being developed make use of open source components?
 - Should an open source approach be used for the software development?
- The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.

Open-source business

- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.
- Some companies believe that adopting an open-source approach will reveal confidential business knowledge to their competitors and so are reluctant to adopt this development model.
- However, if you are going to work for a small company and they open source their software, this may reassure customers that they will be able to support the software if the company goes out of business.

Open-source licensing

- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
 - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
 - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
 - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

Open-source License Models

- The GNU General Public License (GPL). This is a so-called ‘reciprocal’ license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

License management - Bayersdorfer 2007

- Establish a system for maintaining information about open-source components that are downloaded and used.
- Be aware of the different types of licenses and understand how a component is licensed before it is used.
- Be aware of evolution pathways for components.
- Educate people about open source.
- Have auditing systems in place.
- Participate in the open source community.

Software Architecture Vs Software Design

What is the difference between software architecture and software design?

Software Architecture	Software Design
Architecture faces towards strategy, structure and purpose, towards the abstract.	Design faces towards implementation and practice, towards the concrete.
The architecture of a system is its 'skeleton'. It's the highest level of abstraction of a system.	Software design is about designing the individual modules / components.
<u>Architecture</u> is the bigger picture: the choice of frameworks, languages, scope, goals, and high-level methodologies	<u>Design</u> is the smaller picture: the plan for how code will be organized; how the contracts between different parts of the system will look
Architecture is "what" we're building	Design is "how" we're building
Architecture comprises the frameworks, tools, programming paradigms, component-based software engineering standards, high-level principles.	Design is an activity concerned with local constraints, such as design patterns, programming idioms, and refactorings.



Fundamentals of Software Engineering

Course Code : CS2002

Ms. Mathangi Krishnathasan
[\(tmk@ucsc.cmb.ac.lk\)](mailto:tmk@ucsc.cmb.ac.lk)

Software Testing – Part I

CHAPTER - 08

Intended Learning Outcomes

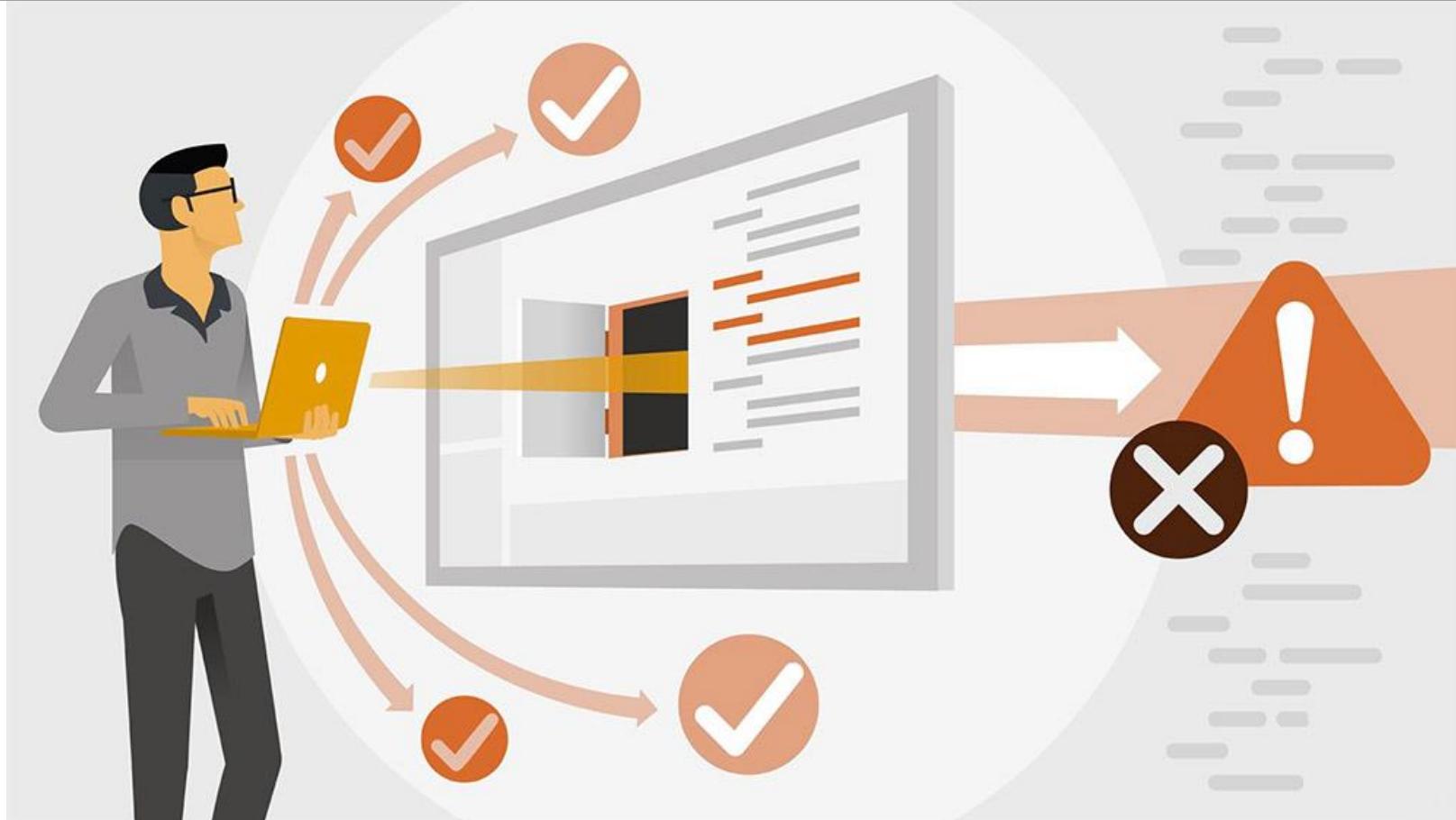
After successfully completing the lesson students should be able to

- Define what is software testing and why it is important.
- Identify the stages of testing / testing process.
- Identify how to select test cases that will be geared to discover program defects.
- Explain test-first development that design tests before writing the code & run these automatically.
- Compare and contrast different types of testing techniques.

Lesson Outline

- **Introduction to Testing**
- **Development testing**
- Test-driven development
- Release testing
- User testing

What is Software Testing?



What is Software Testing?

- Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do. (IBM)
- Another definition: Software testing is a process of executing a program or application with the intent of finding the **software bugs**.

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



Why Software Testing is Important?

- To deliver the best application / high quality product.
- Additional points
 - lower maintenance cost
 - Increase accuracy, consistency and reliability.
 - Increase customer confidence / satisfaction.
 - Loss of money
 - Loss of time
 - Business reputation
 - Harm or death
 - some safety-critical systems could result in injuries or deaths if they don't work properly (e.g. flight traffic control software or Pacemaker)

Software Testing ...

- Intendent to show that **a program does what it is intended to do** and to **discover program defects before it is put into use.**
- Execute a program using artificial / actual data.
- Check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance.

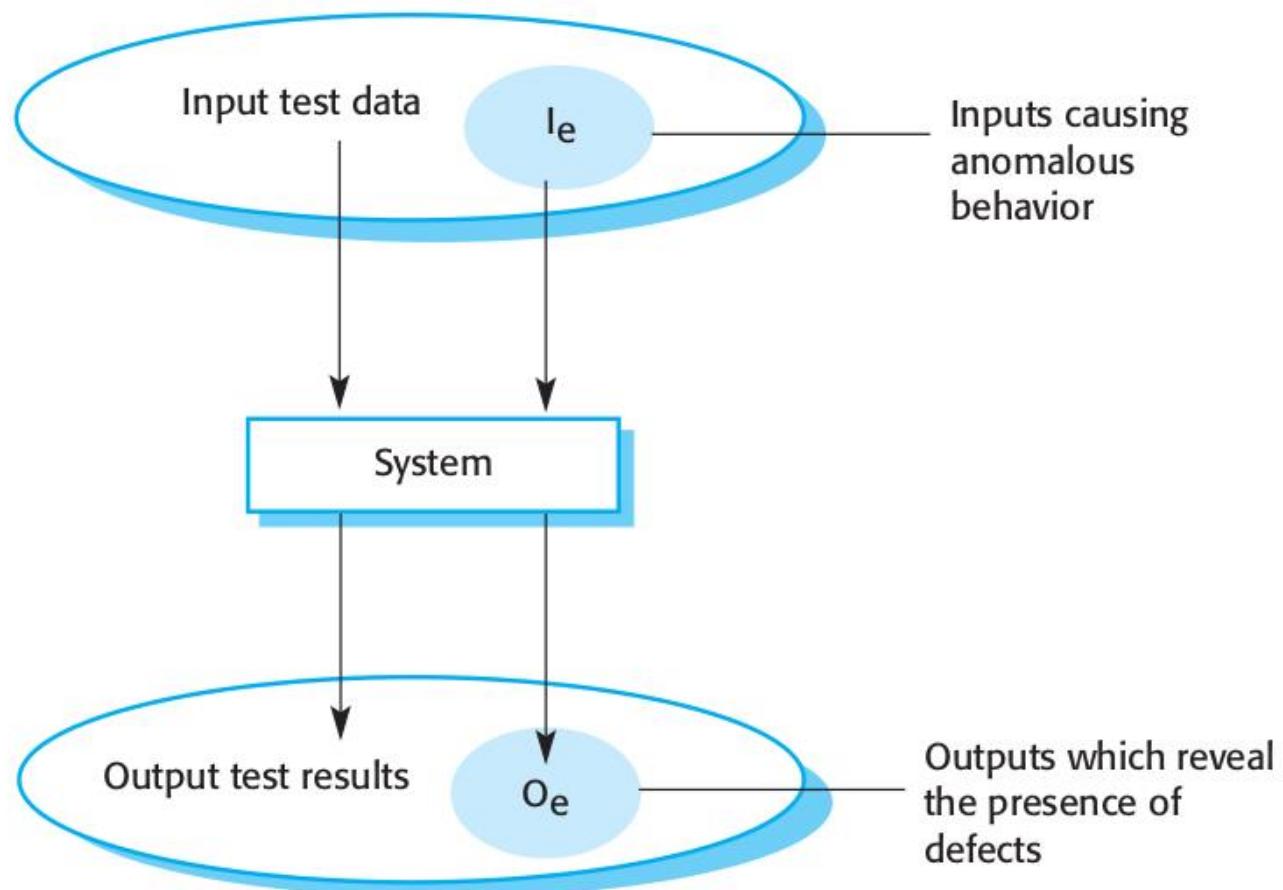
Main Goals - Software testing

- To demonstrate to the developer and the customer that the software meets its requirements.
 - A successful test shows that the system operates as intended.
 - Expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not confirm to its specification(Identify Defects)
 - The test cases are designed to expose defects.

Validation Testing and defect Testing

- The first one validation testing, where you expect the system to perform correctly using a set of test cases that reflect the system's expected use.
- The second is defect testing, where the test cases are designed to expose defects.
- there is no definite boundary between these two approaches to testing.
- During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements

Validation Testing and Defect Testing



Software Testing

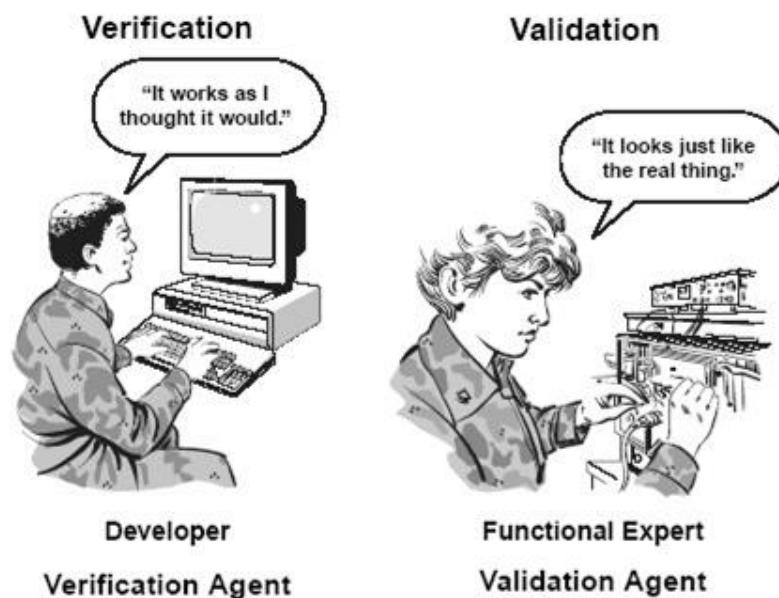
- Testing is part of a broader process of software verification and validation (V & V).
- Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software.
- These checking processes start as soon as requirements become available and continue through all stages of the development process.

Validation vs Verification

- Validation:
 - **"Are we building the right product"**
 - The aim of software validation is to ensure that the software meets the customer's expectations.
- Verification:
 - **"Are we building the product right"**
 - Software verification is the process of checking that the software meets its stated functional and non-functional requirements.

Validation and Verification (V & V) Goals

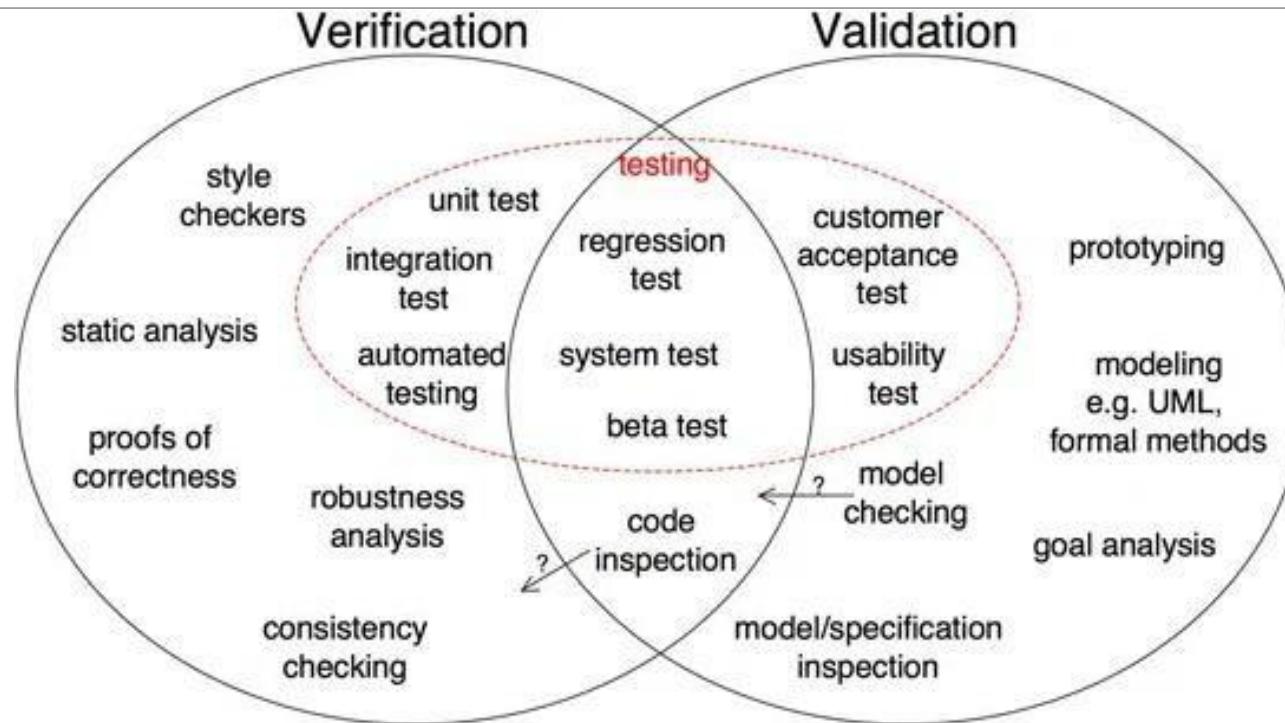
The goal of verification and validation processes is to establish confidence that the software system is “**fit for purpose**.”



Validation and Verification (V & V) Goals

- The level of required confidence depends on
 - **Software purpose:** The more critical the software, the more important it is that it is reliable.
 - **User expectations:** Users may have low expectations of certain kinds of software. When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery. However, as a software product becomes more established, users expect it to become more reliable.
 - **Marketing environment:** In a competitive environment, the company may decide to release a program before it has been fully tested and debugged because it wants to be the first into the market. If a software product or app is very cheap, users may be willing to tolerate a lower level of reliability.

V & V TECHNIQUES



<http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation>

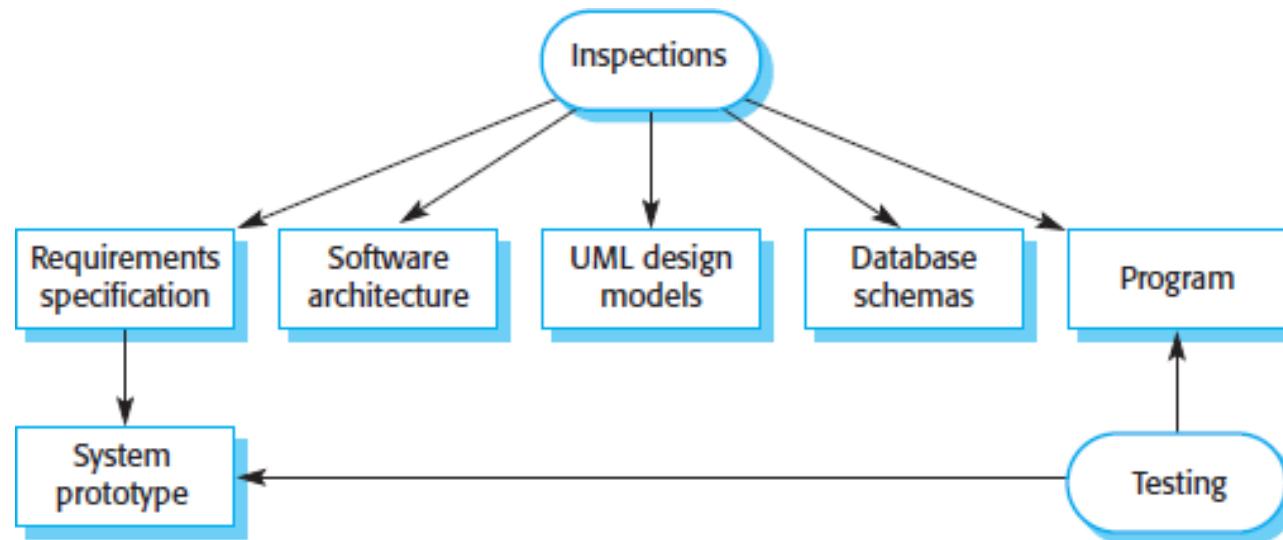
V & V TECHNIQUES

- Software inspections & Review (Static)
 - Concerned with analysis of the static system representation to discover problems.
- Software testing(Dynamic)
 - Concerned with exercising and observing product behavior
 - The system is executed with test data and its operational behavior is observed.

Software inspections & Reviews

- The verification and validation process may involve software inspections and reviews.
- These are “static” V & V techniques in which you don’t need to execute the software to verify it.
- An effective technique to **discover programming errors**.
- It can be used **before implementation**.
- Its applicable to any representation of the system.
 - Requirements
 - Designs
 - Configuration data

Inspections and Testing



Software inspections & Reviews

- Actions of Inspections and reviews
 - Analyze and check the system requirements
 - Design model
 - Design the program source code
 - Propose system tests.
- Inspections mostly focus on the source code of a system, but any readable representation of the software, such as its requirements or a design model, can be inspected.
- When you inspect a system, you use knowledge of the system, its application domain, and the programming or modeling language to discover errors.

Advantages of Inspections

- Errors may mask (hide) other errors. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error.
 - Because inspection doesn't involve executing the system, No need to worry about interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- An inspection can consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.
 - It can look for inefficiencies, inappropriate algorithms and poor programming styles that could make the system difficult to maintain and update.

Inspections vs Software testing

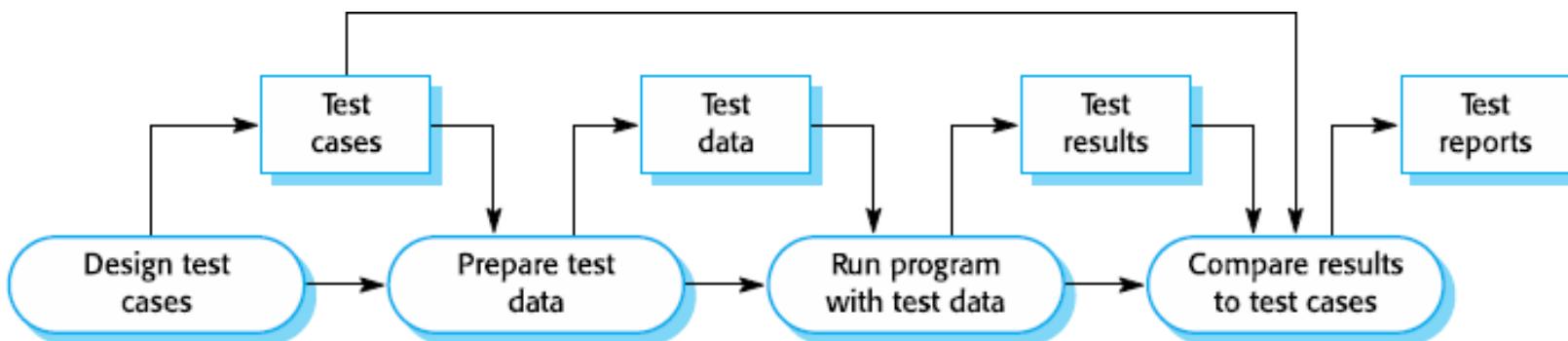
- Complementary but not opposing. Inspections cannot replace software testing.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.
- Inspections are **NOT good for discovering defects that arise due to unexpected interaction** (different parts of a program/ Timing errors / system performance)

TESTING & TESTING PROCESS



A model of the software testing process

Abstract model of the traditional testing process, as used in plan-driven development.



What is a Test Case?

- Set of conditions or variables under which a tester will determine whether the system is working correctly or not.
- Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested.
- Test data are the inputs that have been devised to test a system.
- Test data can sometimes be generated automatically, but automatic test case generation is impossible.
- People who understand what the system is supposed to do must be involved to specify the expected test results. However, test execution can be automated.

Example - Testcase

	A	B	C	D	E	F	G	H	I	J	K
1	Test Case ID		BU_001	Test Case Description		Test the Login Functionality in Banking					
2	Created By		Mark	Reviewed By		Bill		Version		2.1	
3											
4	QA Tester's Log			Review comments from Bill incorporate in version 2.1							
5											
6	Tester's Name		Mark	Date Tested		1-Jan-2017		Test Case (Pass/Fail/Not)		Pass	
7											
8	S #	Prerequisites:				S #	Test Data				
9	1	Access to Chrome Browser				1	Userid = mg12345				
10	2					2	Pass = df12@434c				
11	3					3					
12	4					4					
13											
14	Test Scenario	Verify on entering valid userid and password, the customer can login									
15											
16	Step #	Step Details		Expected Results		Actual Results		Pass / Fail / Not executed / Suspended			
17											
18	1	Navigate to http://demo.guru99.com		Site should open		As Expected		Pass			
19	2	Enter Userid & Password		Credential can be entered		As Expected		Pass			
20	3	Click Submit		Cutomer is logged in		As Expected		Pass			
21	4										
22											
23											

What is a Test Case?

- Mostly corresponds to a use-case.
- Test case contains
 - ID
 - Description (statement being tested)
 - Input to the test
 - Expected output from the system
- Test Automation
 - Test data can be generated automatically.
 - Automatic test case generation is hard.
 - Test execution can be automated.
 - The test results are automatically compared with the predicted results.

Automated Testing

- In automated testing, the tests are encoded in a program that is run each time the system under development is to be tested.
- This is faster than manual testing, especially when it involves regression testing—re-running previous tests to check that changes to the program have not introduced new bugs.
- Unfortunately, testing can never be completely automated as automated tests can only check that a program does what it is supposed to do.
- It is practically impossible to use automated testing to test systems that depend on how things look (e.g., a graphical user interface), or to test that a program does not have unanticipated side effects.

Software Testing Stages

- **Development testing:** The system is tested during development to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.
- **Release testing:** A separate testing team test a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of the system stakeholders.
- **User testing:** Users or potential users of a system, test the system in their own environment. then group decides if the software can be marketed, released and sold.
 - Acceptance testing is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.



Fundamentals of Software Engineering

Course Code : CS2002

Ms. Mathangi Krishnathasan
[\(tmk@ucsc.cmb.ac.lk\)](mailto:tmk@ucsc.cmb.ac.lk)

Software Testing – Part II

CHAPTER - 08

Intended Learning Outcomes

After successfully completing the lesson students should be able to

- Define what is software testing and why it is important.
- Identify the stages of testing / testing process.
- Identify how to select test cases that will be geared to discover program defects.
- Explain test-first development that design tests before writing the code & run these automatically.
- Compare and contrast different types of testing techniques.

Lesson Outline

- Introduction to Testing
- **Development testing**
- **Test-driven development**
- **Release testing**
- **User testing**

DEVELOPMENT TESTING



Development Testing

- Includes all testing activities that are **carried out by the team developing the system**.
- For critical systems, a more formal process may be used, with a separate testing group within the development team.
- This group is responsible for developing tests and maintaining detailed records of test results.
- There are three stages of development testing.
 - **Unit testing** - Individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
 - **Component testing** – Several individual units are integrated to create composite components. Component testing should focus on testing the component interfaces that provide access to the component functions.
 - **System testing** – Some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

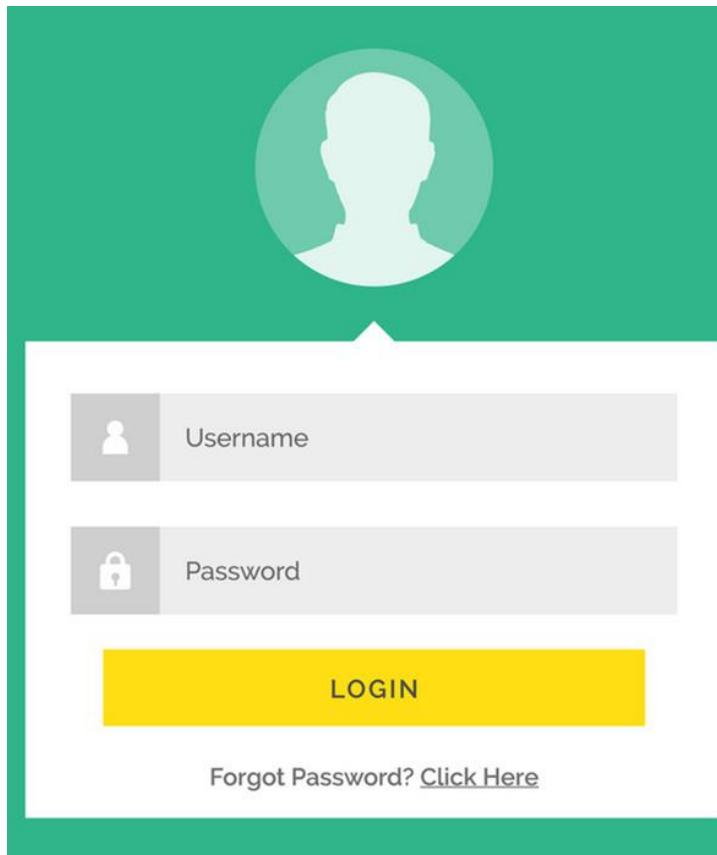
UNIT TESTING



What is Unit Testing?

- Unit testing is the process of testing program components, such as methods or object classes.
- Tests should be calls to these routines with different input parameters.
- Isolate each part of the system and show that the individual parts are correct.
i.e. Individual units or components of a software are tested.
- Units testing should designed in a way to provide coverage to all the features of object:
 - Individual functions or methods within an object
 - Object classes with several attributes
 - All possible states of object

Individual Functions / Methods



Unit Testing - Object Class Testing

- When you are testing object classes, you should design your tests to provide coverage of all of the features of the object.
- This means that you should test all operations associated with the object; set and check the value of all attributes associated with the object; and put the object into all possible states.
- Hence it will all events that cause a state change.

Object class testing - Example

- You need a test that checks if **identifier** property has been properly set up. You need to define test cases for all of the methods associated with the object such as **reportWeather** and **reportStatus**.
- Ideally, you should test methods in isolation, but, in some cases, test sequences are necessary.
 - For example, to test the method that shuts down the weather station instruments (`shutdown`), you need to have executed the `restart` method.

WeatherStation
identifier
reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

Object class testing - Example

- To test the states of the weather station, you can use a state model.
- Using this model, you identify sequences of state transitions that have to be tested and define event sequences to force these transitions.
- In principle, you should test every possible state transition sequence, although in practice this may be too expensive.
- Examples of state sequences that should be tested in the weather station include:

- Shutdown → Running → Shutdown
- Configuring → Running → Testing → Transmitting → Running
- Running → Collecting → Running → Summarizing → Transmitting → Running

WeatherStation
identifier
reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

Automated testing

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.



Automated Testing

- They can then run all of the tests that you have implemented and report often through some graphical unit interface (GUI).
- An entire test suite can often be run in a few seconds, so it is possible to execute all tests every time you make a change to the program.
- An automated test has three parts:
 - A setup part
 - A call part
 - An assertion part

Automated testing - parts

- A **setup part**, where you initialize the system with the test case, namely the inputs and expected outputs.
- A **call part**, where you call the object or method to be tested.
- An **assertion part** where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

Mock Objects

- Sometimes, the object that you are testing has dependencies on other objects that may not have been implemented or whose use slows down the testing process.
 - If an object calls a database, this may involve a slow setup process before it can be used. In such cases, you may decide to use mock objects.
- Mock objects are objects with the same interface as the external objects being used that simulate its functionality.
 - For example, a mock object simulating a database may have only a few data items that are organized in an array.
 - They can be accessed quickly, without the overheads of calling a database and accessing disks.
- Similarly, mock objects can be used to simulate abnormal operations or rare events.
 - For example, if your system is intended to take action at certain times of day, your mock object can simply return those times, irrespective of the actual clock time.

Choosing unit test cases

- Testing is expensive and time consuming, so it is important that you choose effective unit test cases
 - The test cases should show, when used as expected, the component that you are testing does what it is supposed to do.
 - If there are defects in the component, these should be revealed by test cases.
- Two types of unit test cases
 - Normal operation of a program and should show that the component works as expected.
 - Abnormal inputs to check that these are properly processed and do not crash the component.

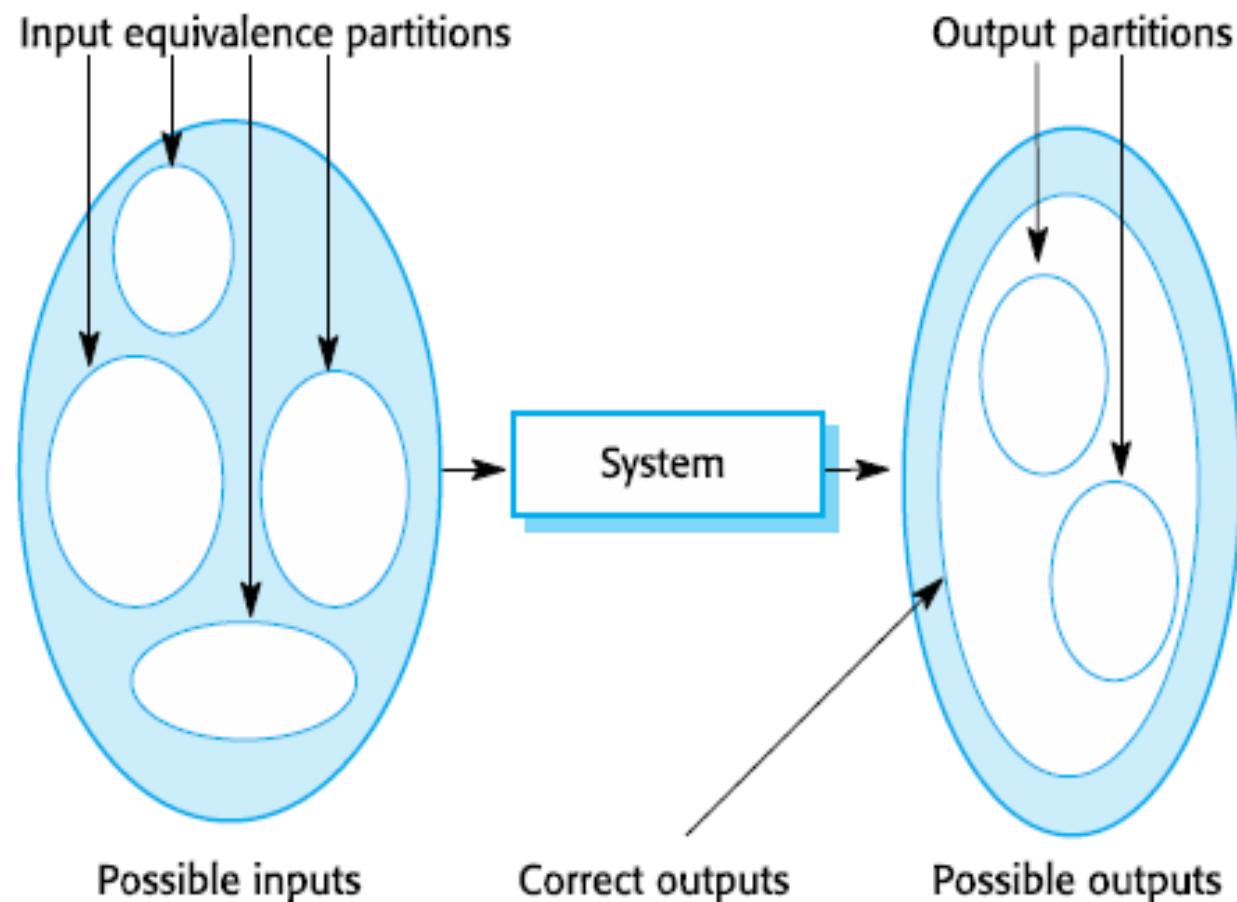
Test cases selection strategies

- Two strategies that can be effective in helping you choose test cases are,
 - **Partition testing**
 - identify groups of inputs that have common characteristics and should be processed in the same way.
 - You should choose tests from within each of these groups.
 - **Guideline-based testing**
 - Use testing guidelines to choose test cases.
 - Guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Partition testing/ Equivalence partitioning

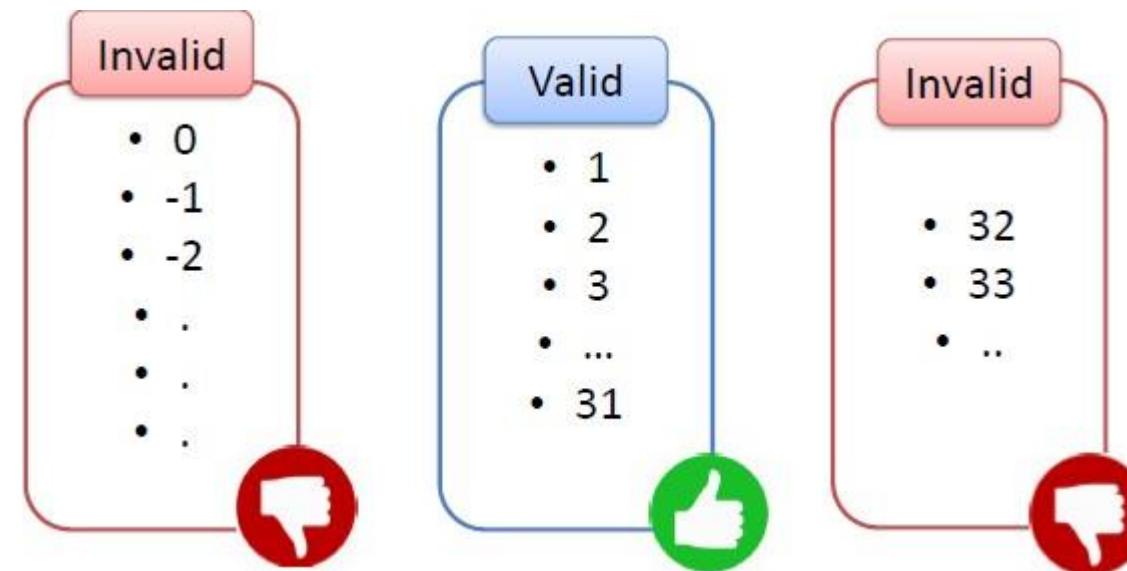
- Divide input test data into partitions.
- **Assumption:** Any input within the partition is equivalent - the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.
- Test each partition.
- One systematic approach to test-case design is based on identifying all input and output partitions for a system or component.

Equivalence partitioning



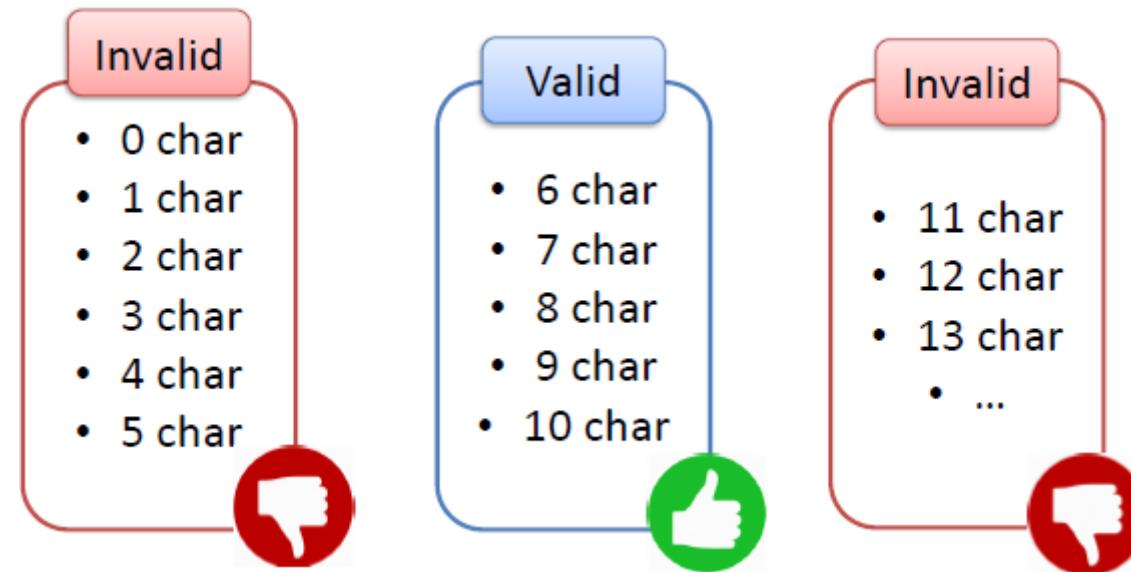
Equivalence partitioning – Example 1

- Date Field (Integer values 1 to 31 is accepted)
- Identify the equivalence partitioning.



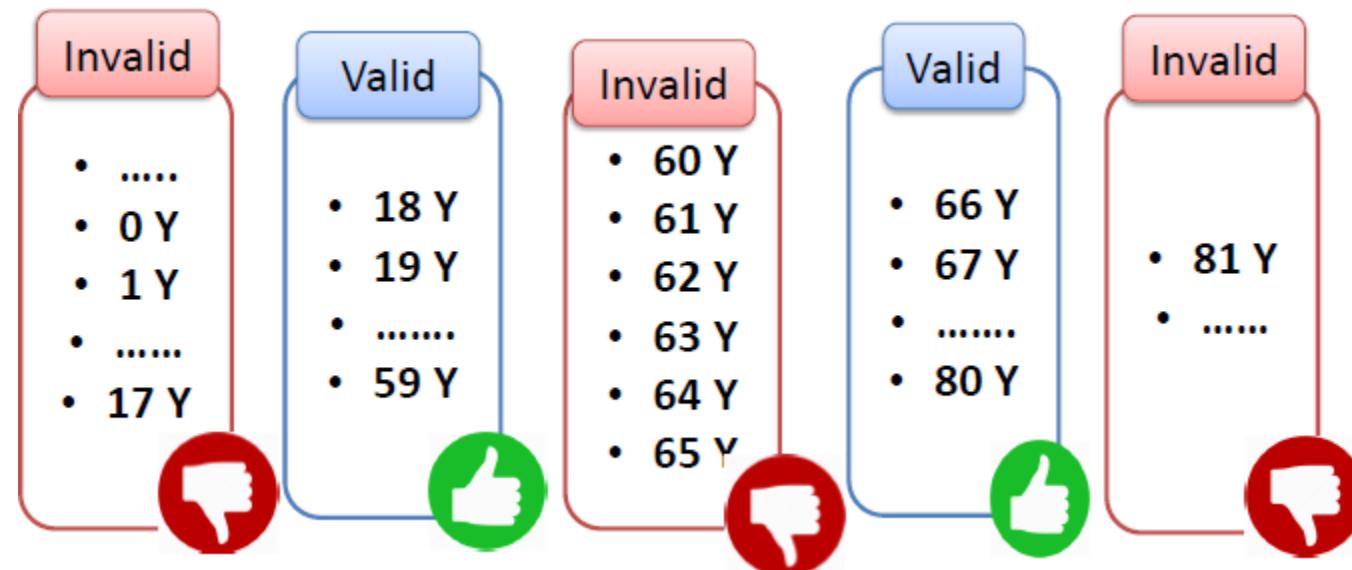
Equivalence partitioning – Example 2

- User Name (6 –10 Characters)
- Identify the equivalence partitioning



Equivalence partitioning – Example 3

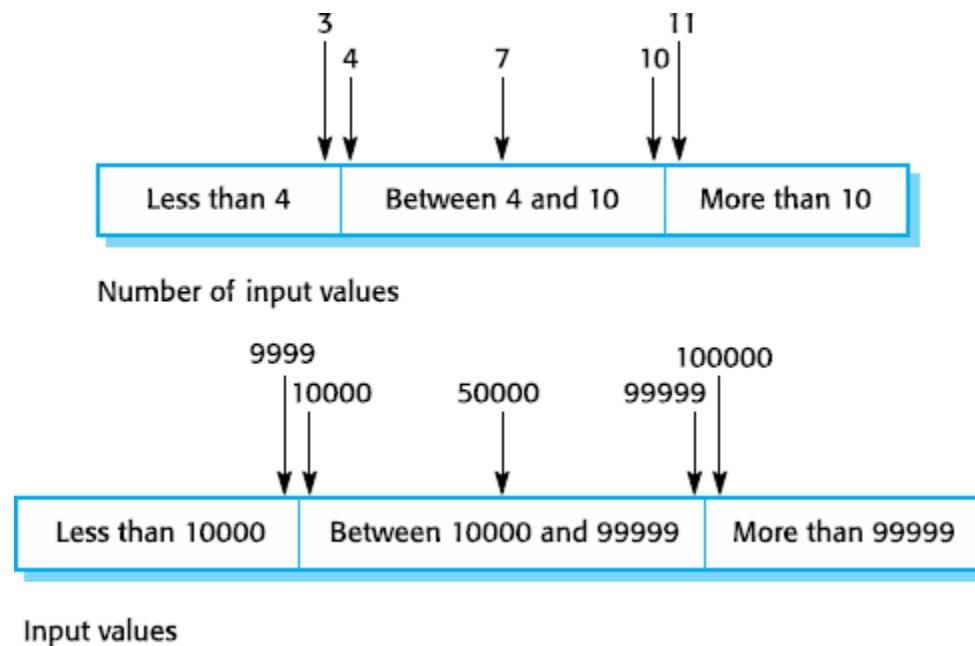
- Age(18 –80 Years except 60 –65 Years)
- Identify the equivalence partitioning



Exercise

- Suppose a program specification states that the program accepts 4 to 10 inputs which are five-digit integers greater than ten thousand (10,000)
- Identify the input partitions and possible test input values.
 - A good rule of thumb for test-case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition.
 - The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system. You test these by choosing the midpoint of the partition. Boundary values are often atypical (e.g., zero may behave differently from other non-negative numbers) and so are sometimes overlooked by developers.

Equivalence partitions



Testing guidelines (sequences)

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the sequence are accessed.
- Test with sequences of zero length.

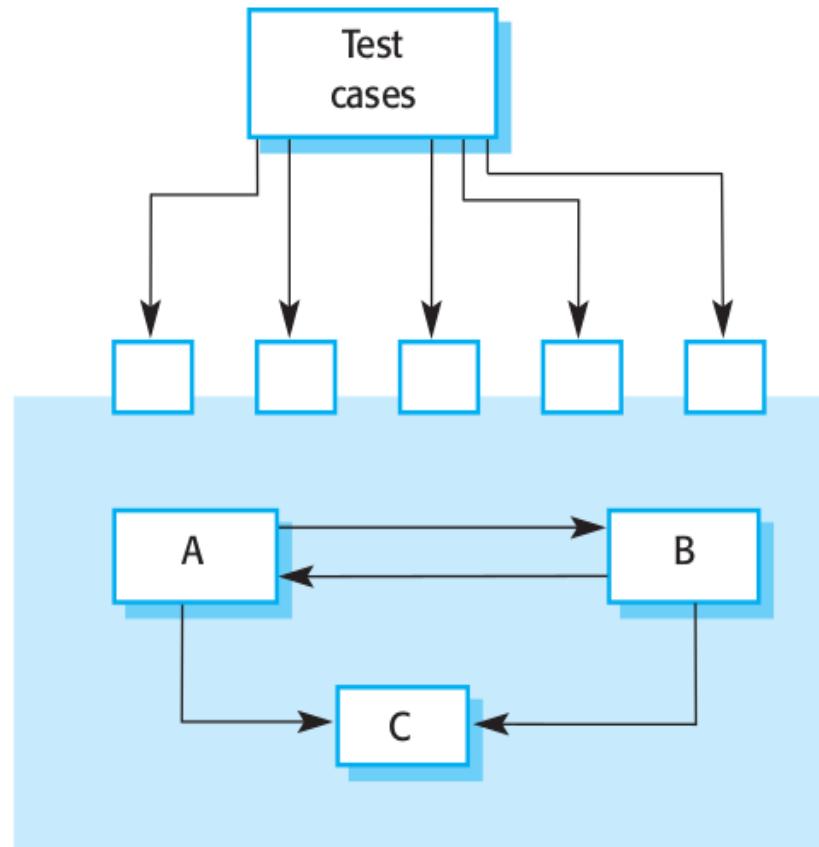
General testing guidelines

- Choose inputs that force the system to generate all error messages
- Design inputs that cause input buffers to overflow
- Repeat the same input or series of inputs numerous times
- Force invalid outputs to be generated
- Force computation results to be too large or too small.

Component testing

- Composite components are made up of several interacting objects.
- Access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface or interfaces behave according to its specification.
- Test cases are not applied to the individual components, but rather to the interface in the composite component created by combining these components.
- Interface errors in the composite component may not be detectable by testing the individual objects because these errors result from interactions between the objects in the component.

Component Interface Testing



Component Interface Testing

- There are different types of interface between program components and, consequently, different types of interface error that can occur:
 - **Parameter interfaces:** These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.
 - **Shared memory interfaces:** These are interfaces in which a block of memory is shared between components.
 - Data is placed in the memory by one subsystem and retrieved from there by other subsystems.
 - This type of interface is used in embedded systems, where sensors create data that is retrieved and processed by other system components.

Component Interface Testing

- **Procedural interfaces:** These are interfaces in which one component encapsulates a set of procedures that can be called by other components.
 - Objects and reusable components have this form of interface.
- **Message passing interfaces:** These are interfaces in which one component requests a service from another component by passing a message to it.
 - A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client–server systems.

Interface Errors

- Interface errors are one of the most common forms of error in complex systems. These errors fall into three classes:
 - Interface misuse
 - A calling component calls some other component and makes an error in the use of its interface.
 - This type of error is common in parameter interfaces, where parameters may be of the wrong type or be passed in the wrong order, or the wrong number of parameters may be passed.
 - Interface misunderstanding
 - A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior.
 - The called component does not behave as expected, which then causes unexpected behavior in the calling component. For example, a binary search method may be called with a parameter that is an unordered array. The search would then fail.

Interface Errors

- Timing errors
 - These occur in real-time systems that use a shared memory or a message-passing interface.
 - The producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

Testing for Interface Errors

- Testing for interface defects is difficult because some interface faults may only manifest themselves under unusual conditions.
 - For example, say an object implements a queue as a fixed-length data structure. A calling object may assume that the queue is implemented as an infinite data structure, and so it does not check for queue overflow when an item is entered.
- This condition can only be detected during testing by designing a sequence of test cases that force the queue to overflow. The tests should check how calling objects handle that overflow.
- However, as this is a rare condition, testers may think that this isn't worth checking when writing the test set for the queue object.

Testing for Interface Errors

- A further problem may arise because of interactions between faults in different modules or objects. Faults in one object may only be detected when some other object behaves in an unexpected way. Say an object calls another object to receive some service and the calling object assumes that the response is correct.
- If the called service is faulty in some way, the returned value may be valid but incorrect. The problem is therefore not immediately detectable but only becomes obvious when some later computation, using the returned value, goes wrong.
- Sometimes it is better to use inspections and reviews rather than testing to look for interface errors. Inspections can concentrate on component interfaces and questions about the assumed interface behavior asked during the inspection process.

Interface Testing Guidelines

- Examine the code to be tested and identify each call to an external component.
- Design a set of tests in which the values of the parameters to the external components are at the extreme ends of their ranges.
- Where pointers are passed across an interface, always test the interface with null pointer parameters.
- Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.
- Use stress testing in message passing systems. This means that you should design tests that generate many more messages than are likely to occur in practice. This is an effective way of revealing timing problems.
- Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

What is Stress Testing

- It is a form of **software testing** that is used to determine the stability of a given system.
- A type of **non-functional testing**.
- It involves testing beyond normal operational capacity(breaking point), in order to observe the results.
- It put greater emphasis on robustness, availability, and error handling under a heavy load, rather than correct behavior under normal circumstances.
- The goals of such tests may be to ensure the software does not crash in conditions of insufficient computational resources (such as memory or disk space).

SYSTEM TESTING



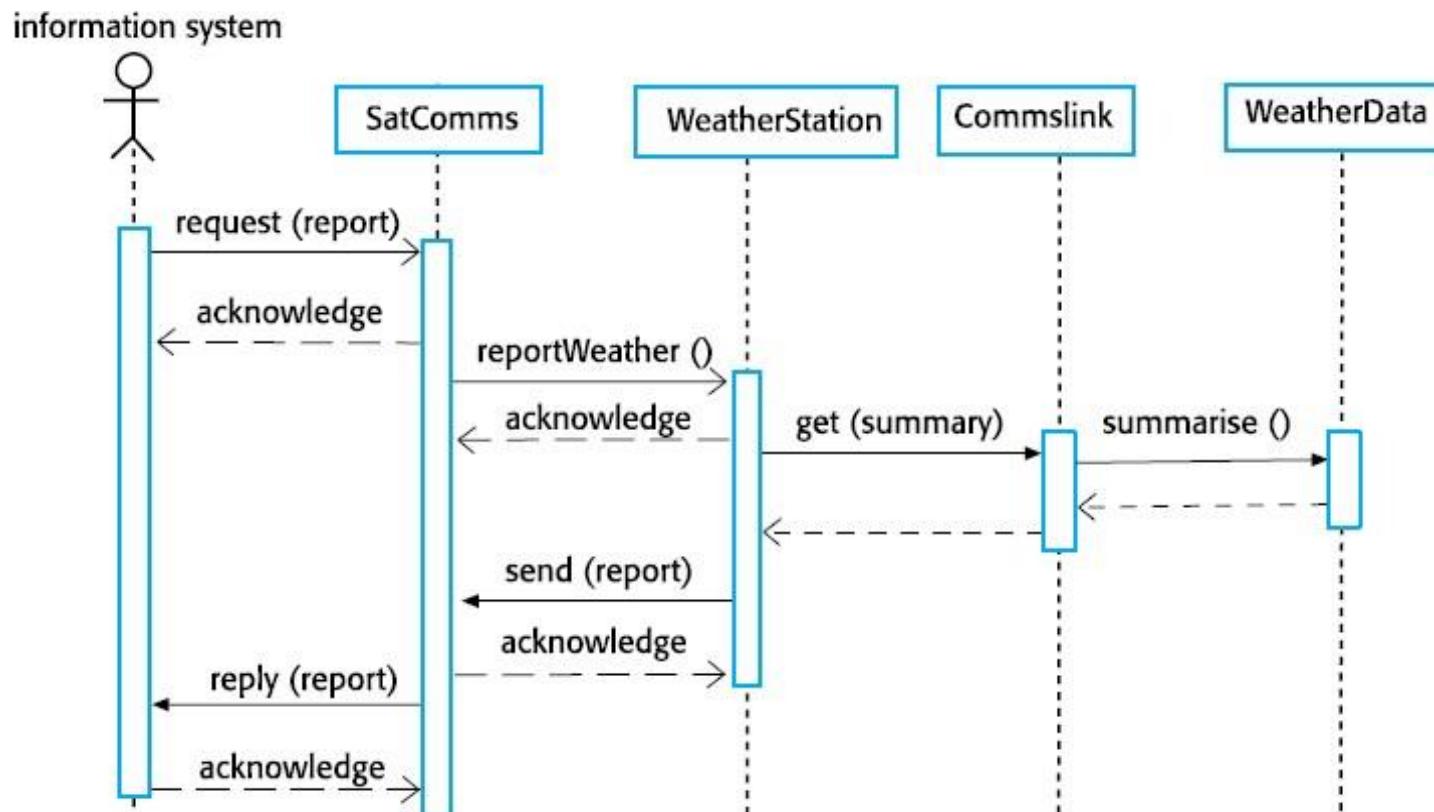
System Testing

- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- System testing checks that components are compatible, interact correctly, and transfer the right data at the right time across their interfaces.
- It obviously overlaps with component testing, but there are two important differences:
 - During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
 - Components developed by different team members or subteams may be integrated at this stage. System testing is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

System Testing

- System testing should focus on testing the interactions between the components and objects that make up a system.
- Because of its focus on interactions, use case-based testing is an effective approach to system testing.
- Several components or objects normally implement each use case in the system.
- Testing the use case forces these interactions to occur.
- A sequence diagram used to model the use case implementation, and help to see the objects or components that are involved in the interaction.

Sequence Diagram: Example



Test cases derived from sequence diagram

- An input of a request for a report should have an associated acknowledgement. A report should ultimately be returned from the request.
 - You should create summarized data that can be used to check that the report is correctly organized.
- An input request for a report to Weather Station results in a summarized report being generated.
 - Can be tested by creating raw data corresponding to the summary that you have prepared for the test of SatComms and checking that the Weather Station object correctly produces this summary.
 - This raw data is also used to test the Weather Data object.

Testing policies

- Difficult to know how much system testing is required.
- Exhaustive system testing is impossible.
- Testing should be based on a subset of possible test cases.
- Software companies should have policies in choosing test cases.

Examples of testing policies:

- All system functions accessed through menus should be tested.
- Combinations of functions accessed through the same menu must be tested.
- Where user input is provided, all functions must be tested with both correct and incorrect input.

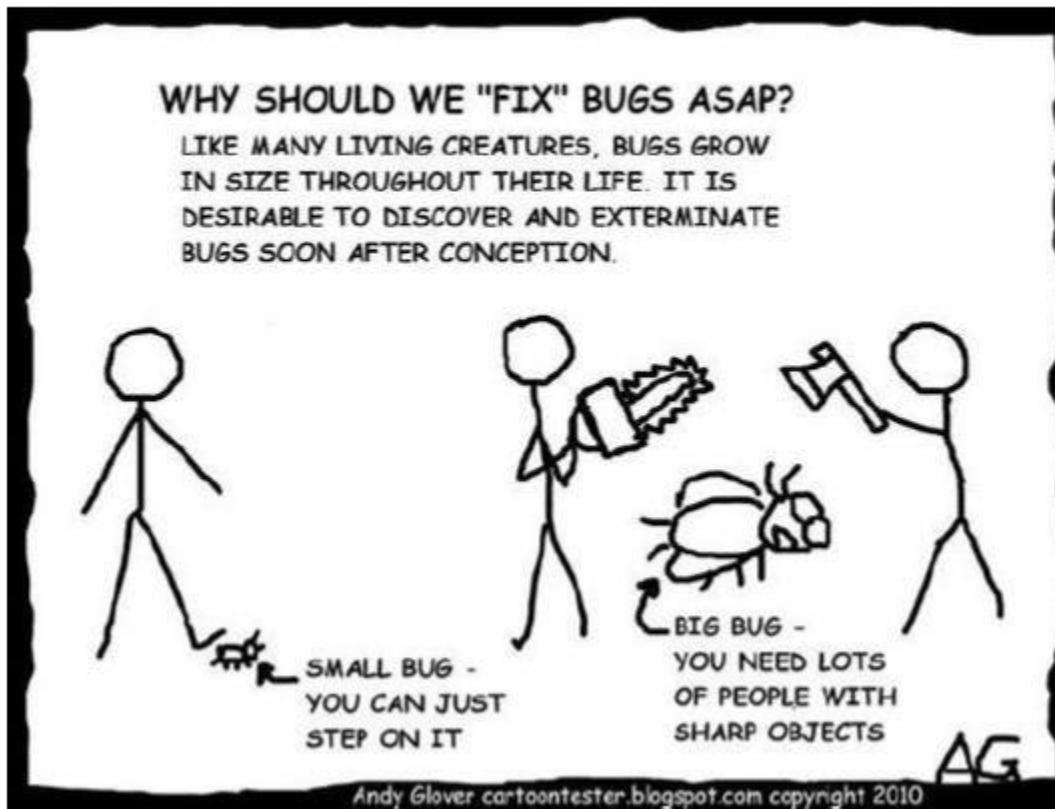
Automated System Testing

- When features of the software are used in isolation, they normally work.
- Problems arise when combinations of less commonly used features have not been tested together.
- Automated system testing is usually more difficult than automated unit or component testing.
- Automated unit testing relies on predicting the outputs and then encoding these predictions in a program. The prediction is then compared with the result.
- However, the point of implementing a system may be to generate outputs that are large or cannot be easily predicted.

TEST FIRST DEVELOPMENT / TEST-DRIVEN DEVELOPMENT(TDD)



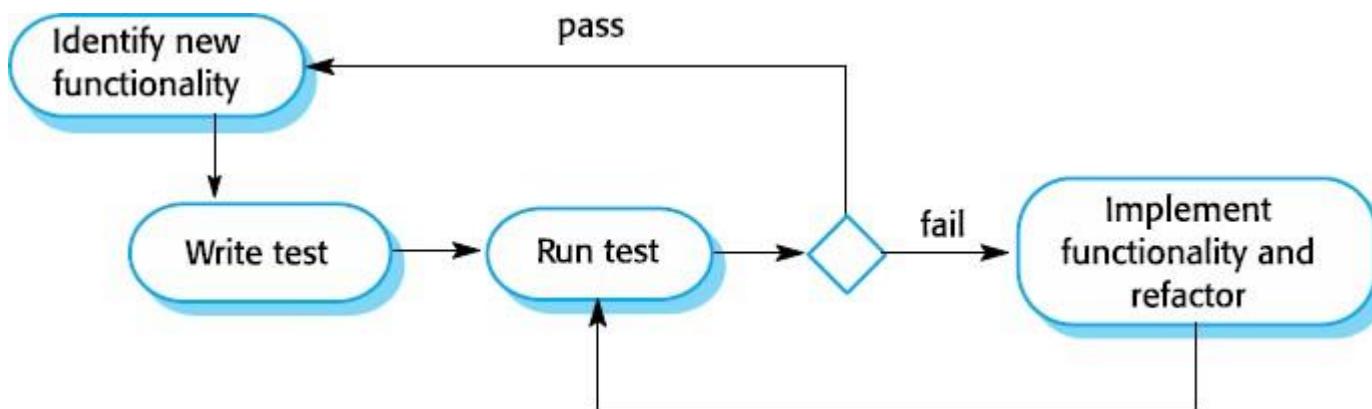
Why should you care about TDD



Test-driven development

- Tests are written before code.
- Tests should pass and it is a critical driver of development.
- You develop code incrementally, along with a test for that increment.
- You don't move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

Test-driven development



TDD process activities

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

Example

- Suppose you were asked to write a simple function to concatenate two words.
 - One
 - Two
 - Output : OneTwo

Start with a test

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyUnitTest {

    @Test
    public void testConcatenate() {
        MyUnit myUnit = new MyUnit();

        String result = myUnit.concatenate("one", "two");

        assertEquals("onetwo", result);
    }
}
```

Implement

```
public class MyUnit {  
    public String concatenate(String one, String two){  
        return one + two;  
    }  
}
```

Benefits of test-driven development

- Code coverage
 - Every code segment that you write has at least one associated test so all code written has at least one test.
- Simplified debugging
 - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- System documentation
 - The tests themselves are a form of documentation that describe what the code should be doing.
- Regression testing
 - A regression test suite is developed incrementally as a program is developed. Regression tests used to check that changes to the program have not introduced new bugs.

What is Regression Testing



Regression testing

- Regression testing is testing the system **to check that changes have not ‘broken’ previously working code.**
- The regression test checks that these changes have **not introduced new bugs** into the system and that the **new code interacts as expected** with the existing code.
- In a **manual** testing process, regression testing is **expensive** but, with **automated** testing, it is **simple** and **straightforward**.
- **All tests are re-run** every time a change is made to the program.
- Tests must **run ‘successfully’ before** the change is **committed**.

Regression testing

- Automated testing dramatically reduces the costs of regression testing. Existing tests may be re-run quickly and cheaply.
- Test-driven development may also be ineffective with multithreaded systems.
- The different threads may be interleaved at different times in different test runs, and so may produce different results.
- System testing also tests performance, reliability, and checks that the system does not do things that it shouldn't do, such as produce unwanted outputs.

Release testing

Release testing

- The process of testing a particular release of a system that is intended for use outside of the development team.
- There are two important distinctions between release testing and system testing during the development process:
 - The system development team should not be responsible for release testing.
 - Release testing is a process of validation checking to ensure that a system meets its requirements and is good enough for use by system customers\
 - Needs to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a **black-box testing** process whereby tests are derived from the system specification.

Requirements based testing

- A general principle of good requirements engineering practice is that requirements should be testable.
- Requirements-based testing, therefore, is a systematic approach to test-case design where you consider each requirement and derive a set of tests for it.
- Requirements-based testing is validation rather than defect testing.
- Mentcare system requirements:
 - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
 - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored

Mentcare system requirements - Requirements tests

- Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
- Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system.
- Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
- Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
- Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

Scenario Testing - A usage scenario for the Mentcare system

George is a nurse who specializes in mental healthcare. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients that he visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so he notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. Jim agrees so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation and the system re-encrypts Jim's record.

After, finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for George of those patients who He has to contact for follow-up information and make clinic appointments.

Scenario Testing - Features tested by scenario

- Authentication by logging on to the system.
- Downloading and uploading of specified patient records to a laptop.
- Home visit scheduling.
- Encryption and decryption of patient records on a mobile device.
- Record retrieval and modification.
- Links with the drugs database that maintains side-effect information.
- The system for call prompting.

Performance testing

- Performance tests have to be designed to ensure that the system can process its intended load.
- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

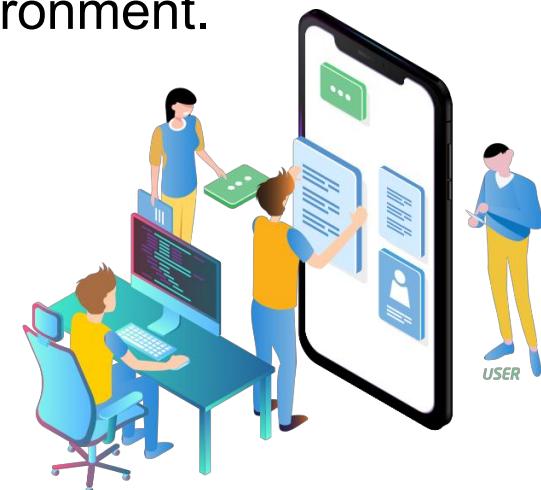
Stress testing

- Stress testing helps to do two things:
 - Test the failure behavior of the system.
 - Reveal defects that only show up when the system is fully loaded.
- Stress testing helps you discover when the degradation begins so that you can add checks to the system to reject transactions beyond this point.

User Testing

User testing

- Users or customers provide input and test the system.
- User testing is essential, even when comprehensive system and release testing have been carried out.
 - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.



Types of user testing

- **Alpha testing**
 - Users of the software work with the development team to test the software at the developer's site.
- **Beta testing**
 - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- **Acceptance testing**
 - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

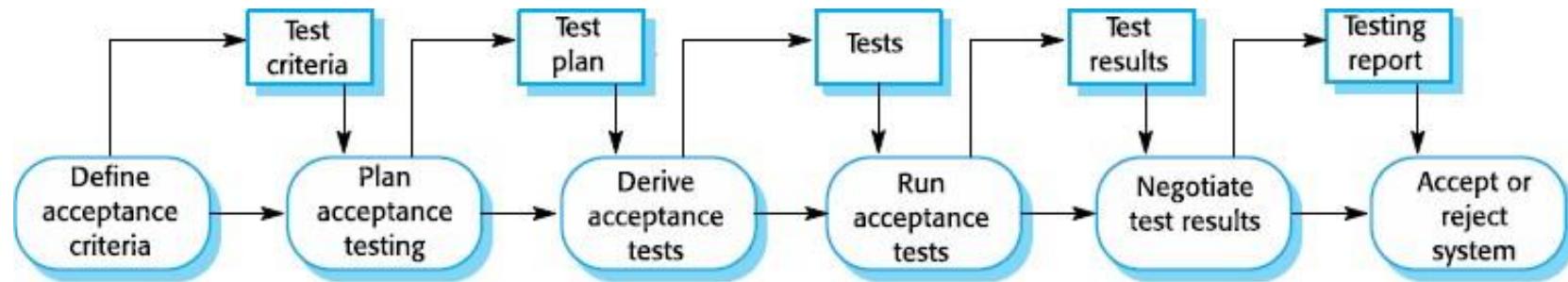
Types of user testing - Alpha Testing

- In alpha testing, users and developers work together to test a system as it is being developed.
- This means that the users can identify problems and issues that are not readily apparent to the development testing team.
- Developers can only really work from the requirements, but these often do not reflect other factors that affect the practical use of the software.
- Users can therefore provide information about practice that helps with the design of more realistic tests.
- It also reduces the risk that unanticipated changes to the software will have disruptive effects on their business.
- Agile development methods advocate user involvement in the development process, and that users should play a key role in designing tests for the system.

Types of user testing - Beta Testing

- Beta testing takes place when an early, sometimes unfinished, release of a software system is made available to a larger group of customers and users for evaluation.
- Beta testers may be a selected group of customers who are early adopters of the system.
- Alternatively, the software may be made publicly available for use by anyone who is interested in experimenting with it.
- This is important as, unlike custom product developers, there is no way for the product developer to limit the software's operating environment. It is impossible for product developers to know and replicate all the settings in which the software product will be used.
- Beta testing is also a form of marketing.

The Acceptance Testing Process



Stages in the acceptance testing process

- **Define acceptance criteria**
 - This stage should ideally take place early in the process before the contract for the system is signed
 - The acceptance criteria should be part of the system contract and be approved by the customer and the developer.
- **Plan acceptance testing**
 - This stage involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule.
 - The acceptance test plan should also discuss the required coverage of the requirements and the order in which system features are tested.
 - It should define risks to the testing process such as system crashes and inadequate performance, and discuss how these risks can be mitigated.

Stages in the acceptance testing process

- **Derive acceptance tests**
 - Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable.
 - Acceptance tests should aim to test both the functional and non-functional characteristics(e.g., performance) of the system.
 - They should ideally provide complete coverage of the system requirements.
- **Run acceptance tests**
 - The agreed acceptance tests are executed on the system.Ideally, this step should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore user environment may have to be set up to run these tests
 - It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system.

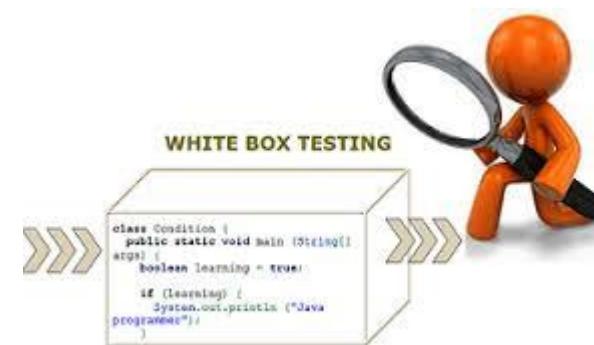
Stages in the acceptance testing process

- **Negotiate test results**
 - It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system.
 - More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be used.
 - They must also agree on how the developer will fix the identified problems.
- **Reject/accept system**
 - This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted.
 - If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.

Agile methods and acceptance testing

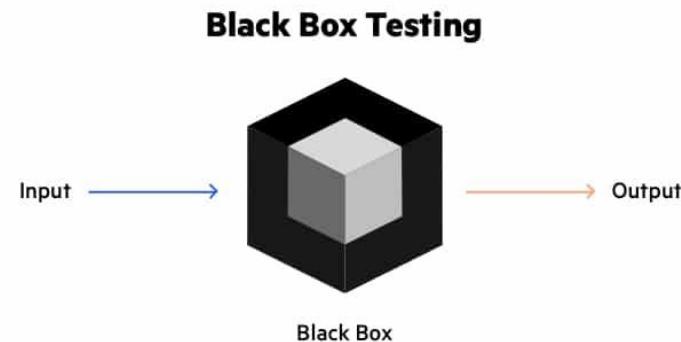
- In agile methods such as Extreme Programming, there may be no separate acceptance testing activity. The end-user is part of the development team and provides the system requirements in terms of user stories.
- He or she is also responsible for defining the tests, which decide whether or not the developed software supports the user stories.
- These tests are therefore equivalent to acceptance tests.
- The tests are automated, and development does not proceed until the story acceptance tests have successfully been executed.

BLACK BOX TESTING AND WHITE BOX TESTING



What is ‘Black Box Testing’

- A method of software testing.
- Examines the functionality of an application **without peering into its internal structures or workings.**
- **Can be applied to every level** of software testing:
- Unit, Integration / Component ,System and acceptance.

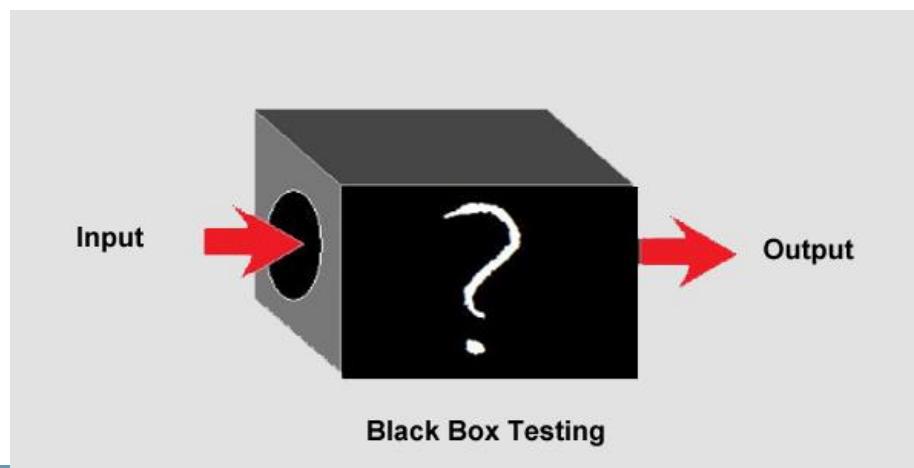


Black Box: Testing Procedure

- Specific knowledge of the application's code/internal structure and programming knowledge in general is not required.
- The tester is aware of **what the software is supposed to do but is not aware of how it does it.**
 - For instance, the tester is aware that a particular input returns a certain, invariable output but is not aware of how the software produces the output in the first place.

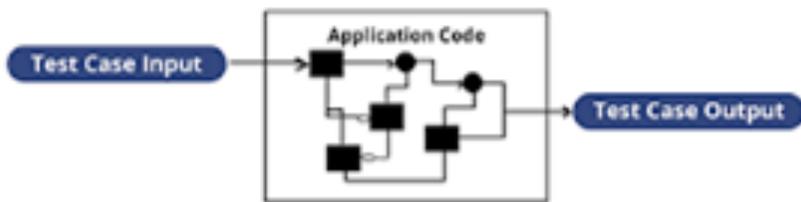
Black Box Testing: Choosing Test Cases

- Test cases are built around specifications and requirements,
- Both functional and non-functional tests can be applied.
- The test designer selects both valid and invalid inputs and determines the correct output, without any knowledge of the test object's internal structure



What is White Box Testing

- A method of testing software that tests internal structures or workings of an application.
- Also known as **clear box testing, glass box testing, transparent box testing, and structural testing**
- It can be applied at the unit, integration and system levels of the software testing process.



White box testing

- Unit testing:
 - to ensure that the code is working as intended, catches any defects early on and prevents errors that can occur later on.
- Integration testing:
 - test the interactions of each interface with each other.

Comparison Base	Black box testing	White box testing
Other terms	Black box testing is also called data-driven testing, box testing, or functional testing.	White box testing is also called structural testing. Some developers call it clear box testing, code-based testing, glass box testing or transparent box testing.
Meaning	It is a testing approach which is used for testing the software without the knowledge of the internal design or structure of program or application.	It is a testing approach in which internal structure or design is known to the software tester who is going to test the software.
Testing Techniques	Equivalence Partitioning Boundary Value Analysis Decision Table State Transition	Statement Coverage Branch Coverage Path Coverage
Implementation Knowledge	Implementation Knowledge is not vital to perform Black Box Testing.	Implementation Knowledge is vital to perform White Box Testing.
Programming Knowledge	It is not required to carry out Black Box Testing.	It is required to carry out White Box Testing.
Prime Focus	Primarily focus on the functionality of the system under test.	Primarily focus on the testing of the program code of the system under test like branches, code structure, loops, conditions, etc.
Time-Period	It is less exhaustive & time-consuming.	Exhaustive & time-taking method.
Target	The main aim is to check on what functionality is performing by the system under test.	The main aim of is to check on how the system is performing.
Types of Testing	A. Functional Testing B. Non-functional testing C. Regression Testing	A. Path Testing B. Loop Testing C. Condition testing

