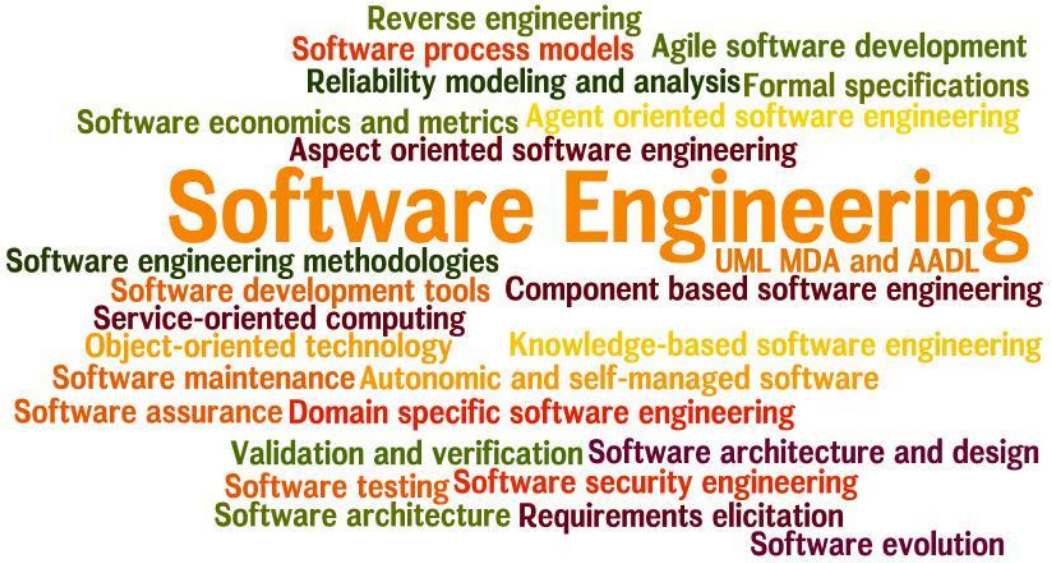
SINGAPORE INSTITUTE OF TECHNOLOGY

INF1001: Introduction to Computing

Part 2

L1: Software Engineering



Topic outline: Software Engineering

Software life cycle

Development, use, maintenance, importance

Software engineering methodologies

Waterfall model, incremental model, reuse-oriented, Agile

Tools of the Trade

UML: Use Case Diagram (UCD)

Data Flow Diagram (DFD)

Defining Software Engineering

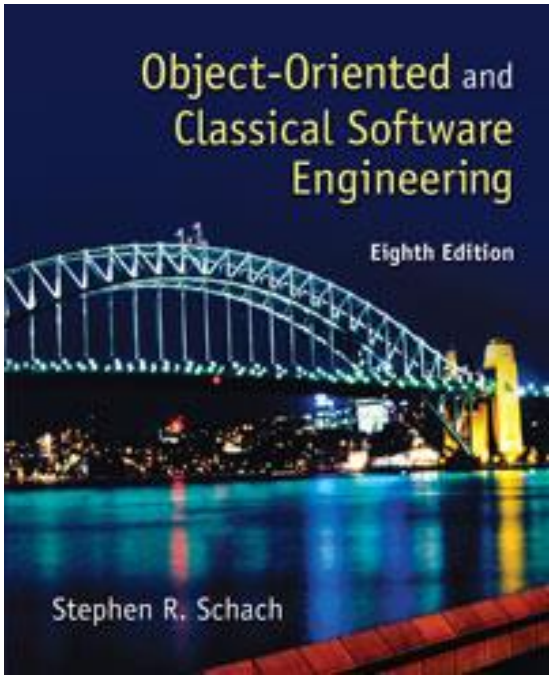
"The application of a ***systematic***, ***disciplined***, ***quantifiable*** approach to the ***development***, ***operation*** and ***maintenance*** of software."

It is not just about coding!

IEEE Advancing Technology for Humanity

Defining Software Engineering

"Software engineering is a discipline whose aim is the production of ***fault-free*** software, delivered ***on time*** and ***within budget***, that ***satisfies the user's needs***"



DEFINING "SOFTWARE ENGINEERING"

Software Engineering Discipline

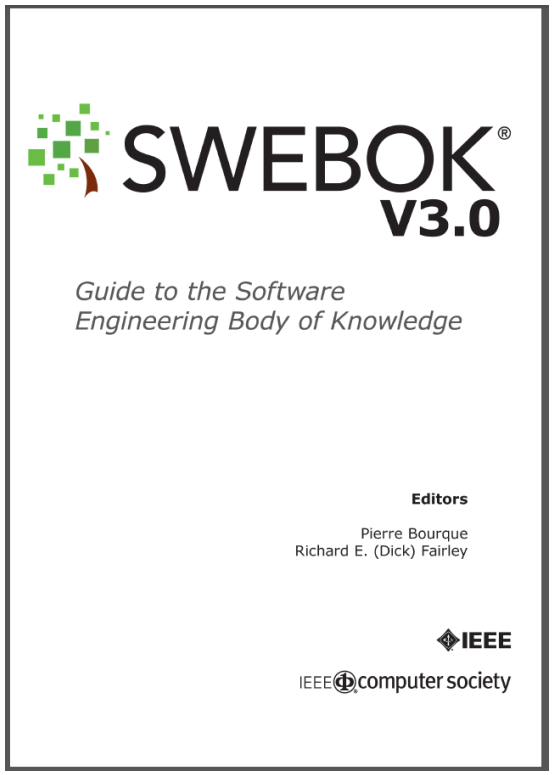


Table I.1. The 15 SWEBOK KAs
Software Requirements
Software Design
Software Construction
Software Testing
Software Maintenance
Software Configuration Management
Software Engineering Management
Software Engineering Process
Software Engineering Models and Methods
Software Quality
Software Engineering Professional Practice
Software Engineering Economics
Computing Foundations
Mathematical Foundations
Engineering Foundations

Refer to https://www.computer.org/web/swebok

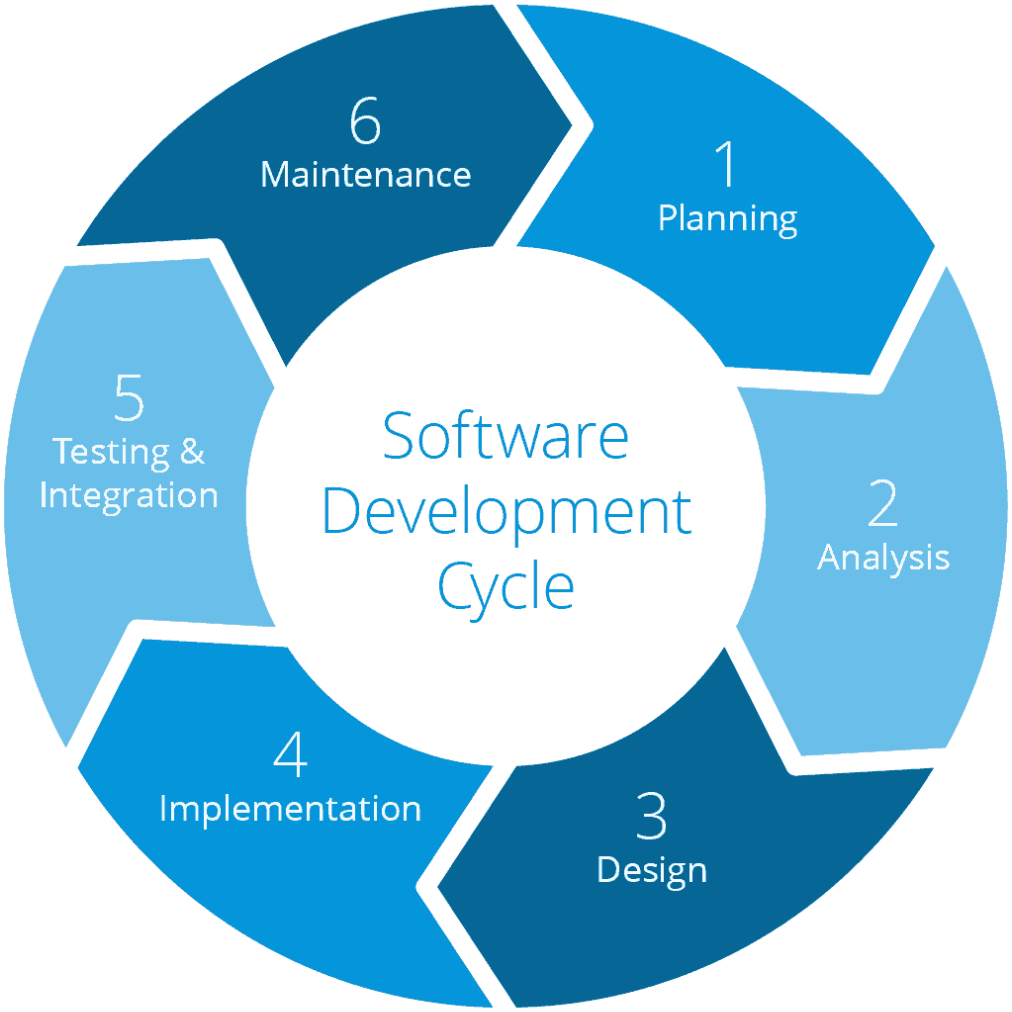
Software Engineering Discipline

- Focus is programming in the **large scale**
  - Many people, many components, complex system

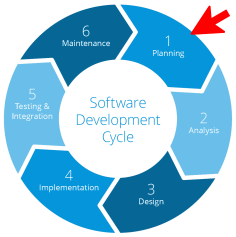
- Emphasis is on:
  - **Product**: what is produced
  - **Process**: how the product is produced
  - **Quality**: how the product is create with quality

- Software Development Life Cycle (SDLC)
  - Overall sequence of steps needed to complete a large-scale software project
  - Implementation occupies only 10-20% of the total time spent by programmers/designers

SDLC: Software Development Life Cycle

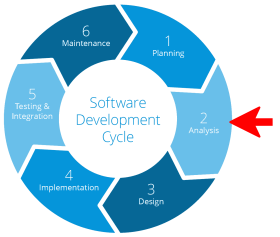


SDLC: 1- Planning



- Start eliciting client's requirements
- Find out scope of the project
- Consider resource, cost, time, benefits

SDLC: 2- Analysis



- Analyze client's requirement
- Draw specification document
- Draw software project management plan

"What the product is supposed to do"

SDLC: 3- Design

- Architectural design
- Detailed design
- E.g. System Design document.

"How the product does it"

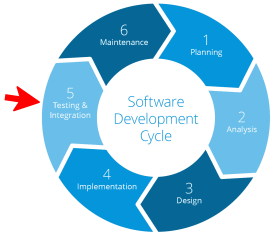
SDLC: 4- Implementation

- Coding

- Involves the actual writing of the programs, creation of data files, and development of databases.

- Create system from design
  - Write programs
  - Create data files
  - Develop databases

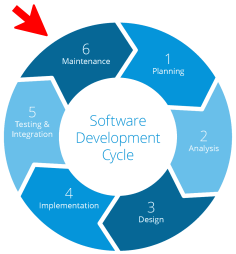
SDLC: 5- Testing and Integration



- Unit testing
- Integration testing
- Acceptance testing

- The process of debugging the programs and confirming that the final software product was compatible with the software requirements specification

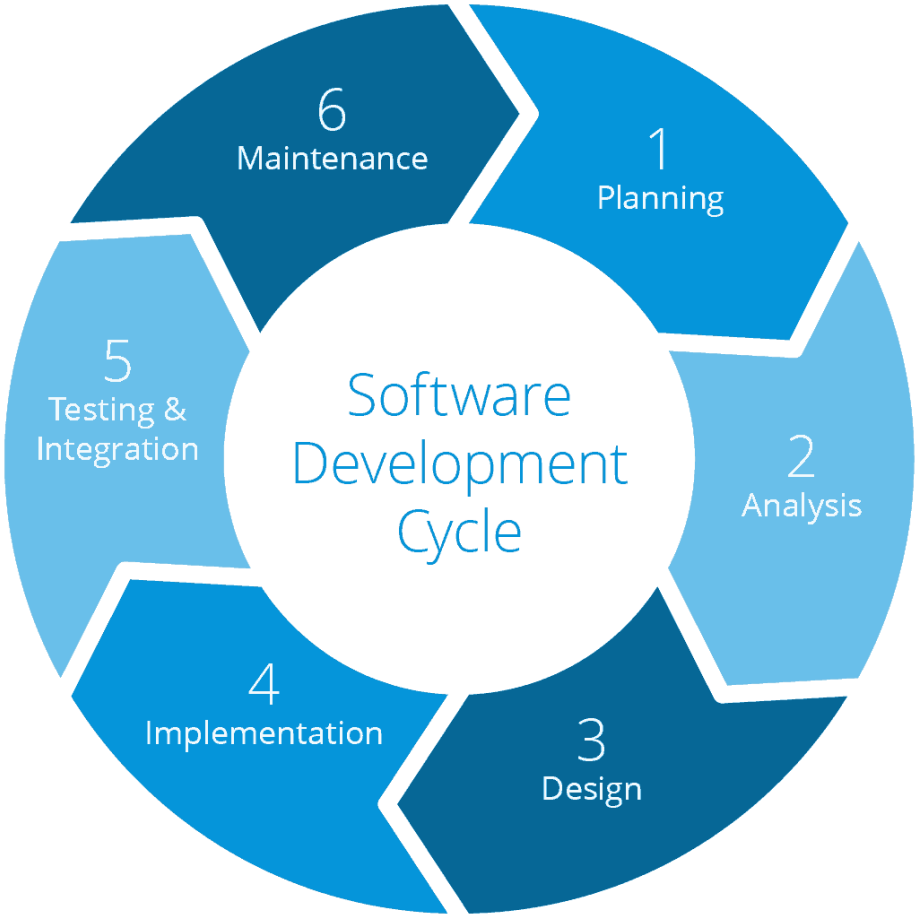
SDLC: 6- Maintenance



- **Corrective**
  - Focus on operational day-to-day errors e.g. Incorrect sequencing of records, invalid function implementation
- **Adaptive**
  - Due to changes in system's environment e.g. New printer, new OS
  - No major changes to basic functionality
- **Preventive**
  - Enhance software reliability, security and robustness
- **Perfective**
  - Enhancement to improve functionality, usability, and performance

Challenges and Common Issues

- Maintenance stage – most time spent
- Challenges and issues can happen at any SLDC stage
- Need for systematic approach to overcome challenges

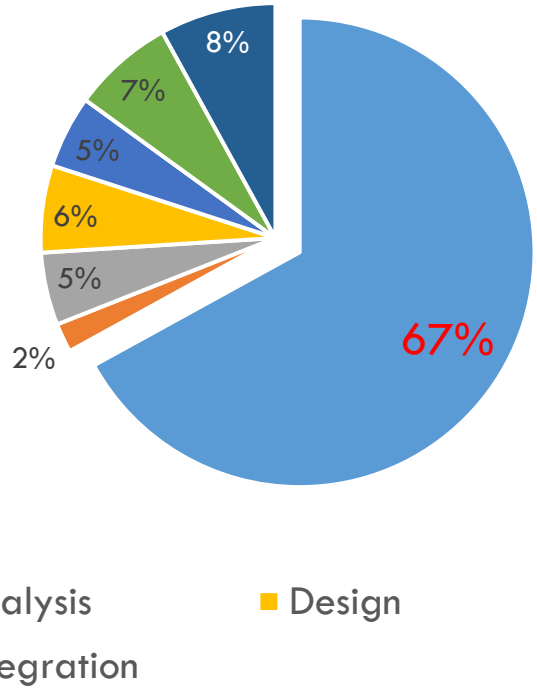


In your opinion, which phase is known to be most problematic?

Phase costing approximation

- Maintenance constitutes 67% of total cost

We need techniques, tools and practices to reduce maintenance costs!



Importance of Software Engineering

- Software engineering and SDLC play crucial role in developing reliable and high-quality software:
  - Provide systematic and structured approach to:
    - Manage complexity
    - Reduce errors
    - Ensure efficiency
  - Emphasise thorough **requirement analysis** through understanding client's needs and expectations
  - Requires **proper planning** so that activities proceed in controlled and manageable manner.
  - Use **systematic testing** approach, software engineers identify and fix defects early ➔ more reliable and robust software.

Software life cycle

Development, use, maintenance, importance

Software engineering methodologies

Waterfall model, incremental model, reuse-oriented, Agile

Tools of the Trade

UML: Use Case Diagram (UCD)

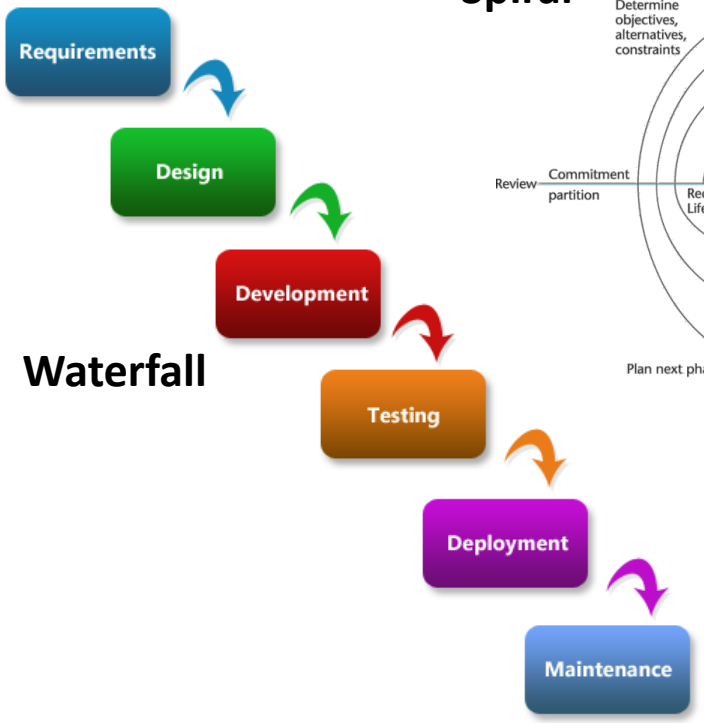
Data Flow Diagram (DFD)

Software Engineering Methodologies

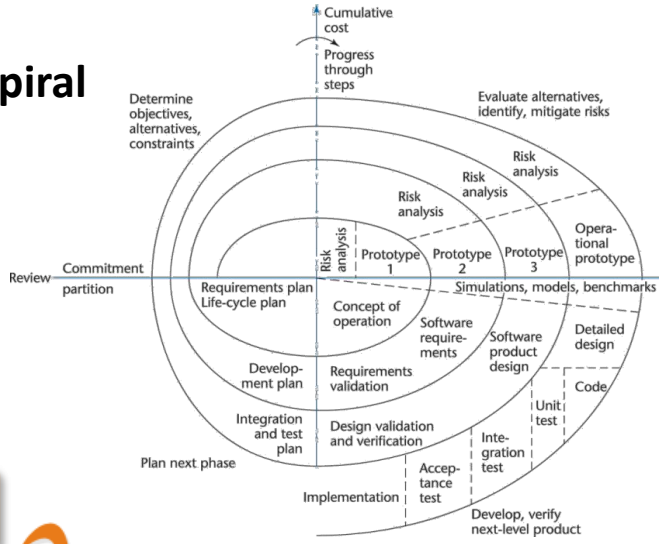
- Provide platform for developers to work together as a team.
- Formalises communication.
- Determines how information is shared within the team.

Software Engineering Methodologies

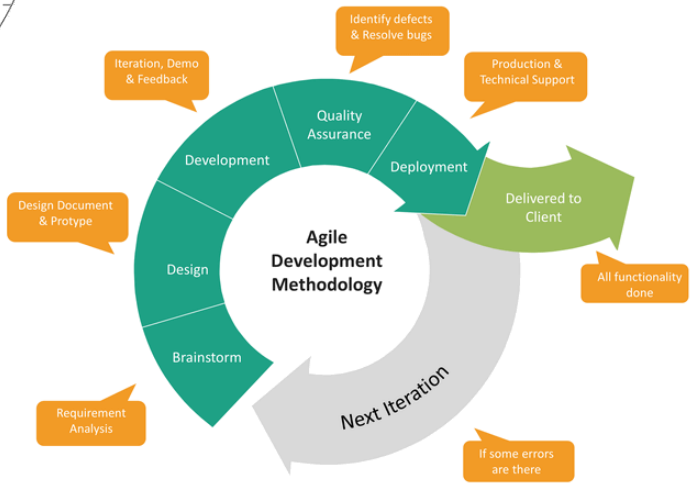
Waterfall



Spiral

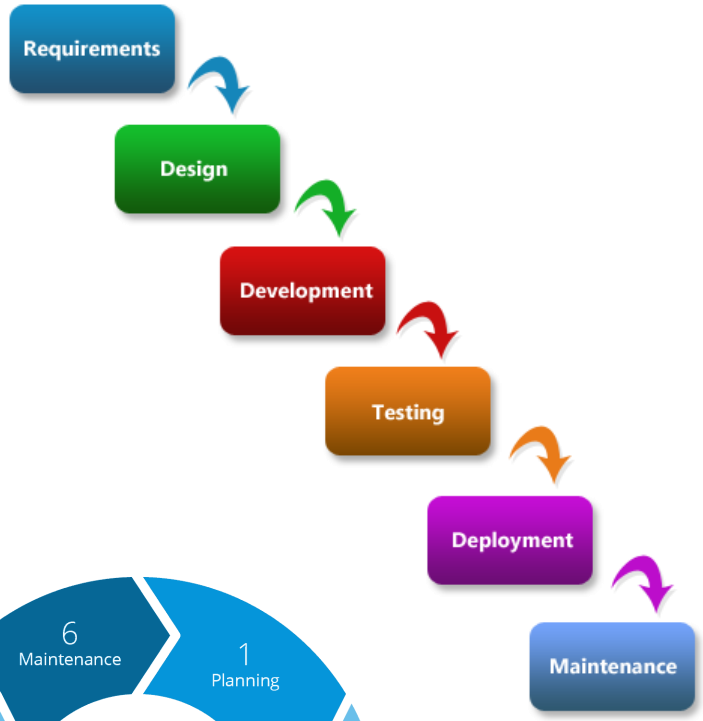


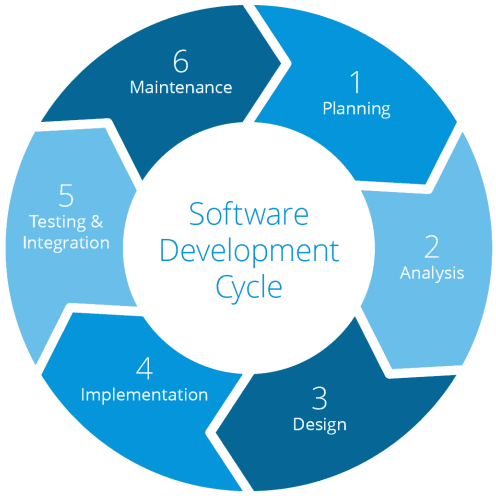
Agile



Waterfall model

- Most classical model
- **Linear** process: one phase is completed and next one begins
- Very **document-driven**



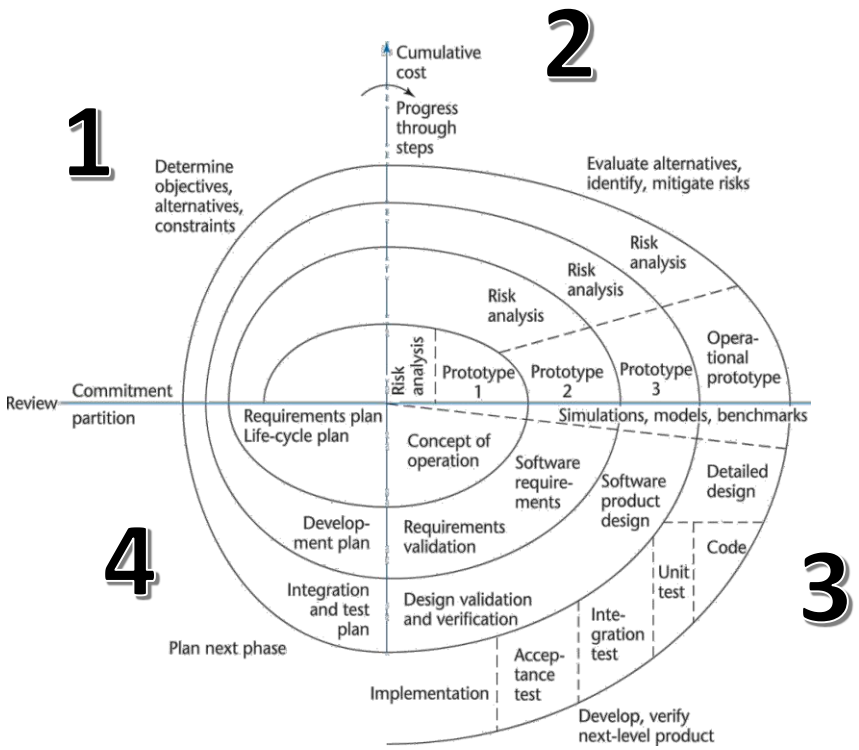


Waterfall model

Strength	Weakness
Clear and well-defined phases	Lack of flexibility
Emphasises documentation	Limited customer involvement
Easy to understand and use	High-risk of late-stage failures
Early identification of issues	Limited progress visibility

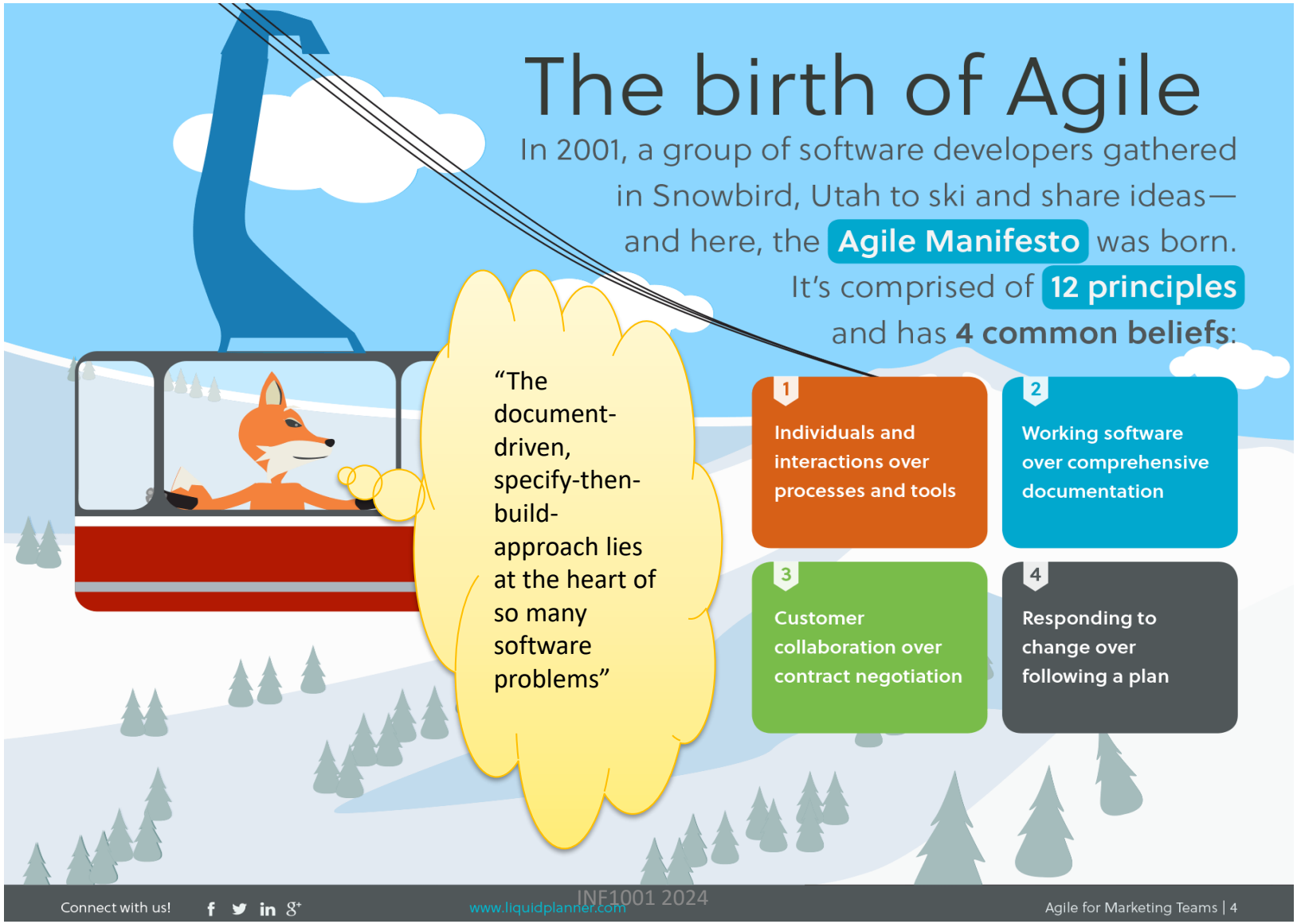
Spiral model

- **4 phases**
- Done in **iterations**
- There is an emphasis on risk analysis
- Combines planning and documentation with prototyping in iterations
- Customer is involved in every iteration
- **Radius** of the iteration reflects the accumulated **cost** involved
- If all risks cannot be mitigated, the project is immediately terminated
- Final spiral is like your waterfall



Spiral model

Strength	Weakness
Customer see the product as it evolves	Iterations are pretty long – they could be 0.5 -2 years
Risk management is part of the life cycle in every iteration	Lots of documentation for every iteration
Project monitoring and scheduling are easy because of the clear phases	You cannot start a phase till the other ends
Features can be added	Need staff who are experts in risk identification and resolution
	Cost of the process is high e.g. Time in prototyping


The birth of Agile
In 2001, a group of software developers gathered in Snowbird, Utah to ski and share ideas—and here, the Agile Manifesto was born. It's comprised of 12 principles and has 4 common beliefs:
"The document-driven, specify-then-build-approach lies at the heart of so many software problems"
1
Individuals and interactions over processes and tools
2
Working software over comprehensive documentation
3
Customer collaboration over contract negotiation
4
Responding to change over following a plan
Connect with us!
www.liquidplanner.com
Agile for Marketing Teams | 4

Agile Principles

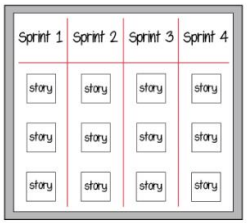
1 Satisfy the **customer**

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

2 Welcome **change**

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

3 Deliver **frequently**



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

4 Work **together**

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

5 Trust and **support**

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

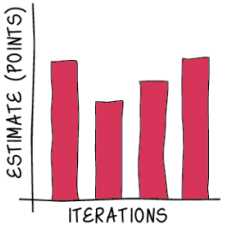
6 Face-to-face **conversation**

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

7 Working **software**

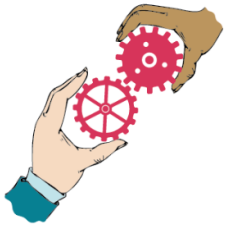
Working software is the primary measure of progress.

8 Sustainable **development**



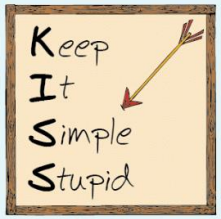
Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9 Continuous **attention**



Continuous attention to technical excellence and good design enhances agility.

10 Maintain **simplicity**



The art of maximizing the amount of work not done - is essential.

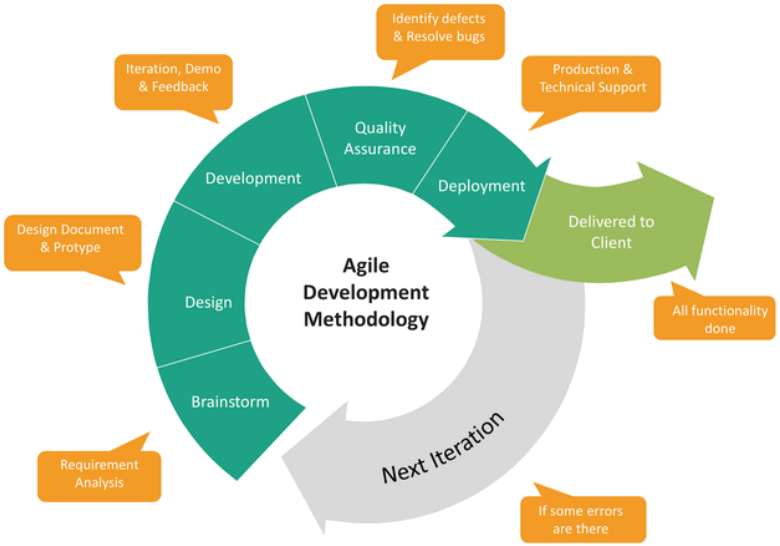
11 Self-organizing **teams**

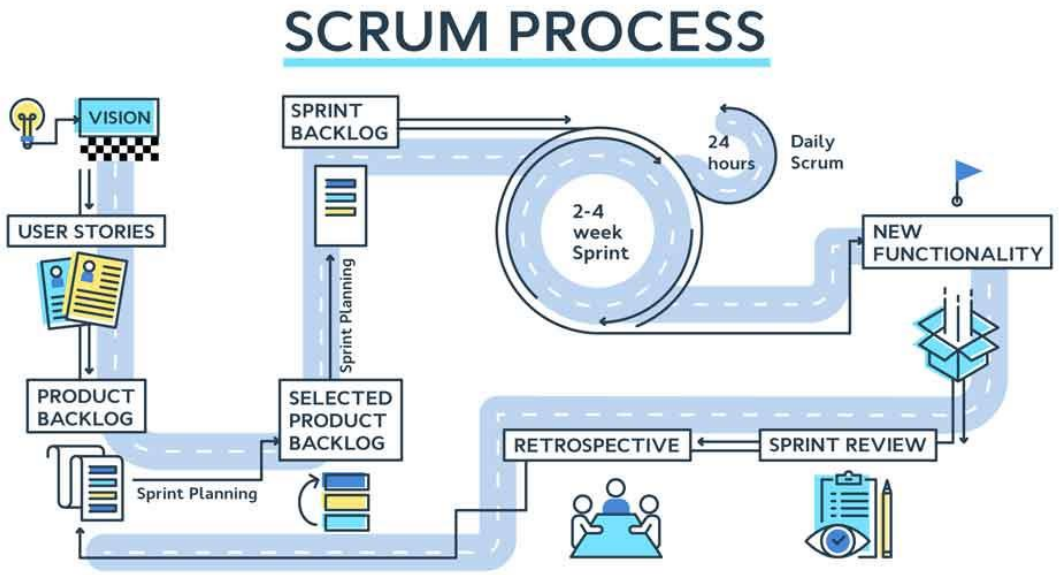
The best architectures, requirements, and designs emerge from self-organizing teams.

12 Reflect and **adjust**

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

Agile model


Identify defects & Resolve bugs
Iteration, Demo & Feedback
Production & Technical Support
Quality Assurance
Development
Deployment
Design Document & Protype
Delivered to Client
Agile Development Methodology
Design
All functionality done
Brainstorm
Next Iteration
Requirement Analysis
If some errors are there


SCRUM PROCESS
VISION
SPRINT BACKLOG
24 hours
Daily Scrum
USER STORIES
2-4 week Sprint
NEW FUNCTIONALITY
Sprint Planning
PRODUCT BACKLOG
SELECTED PRODUCT BACKLOG
RETROSPECTIVE
SPRINT REVIEW
Sprint Planning

Agile model

Strength	Weakness
Shorter iterations about 2-4 weeks sprint	Lack of predictability
Daily sprint of 24 hours	Dependency on customer availability
Emphasis on talking with the team	Resource-intensive
People centric	Lack of upfront planning
Quick response to changes	Change in project requirements without proper control.

Essential Attributes of Good Software

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to **meet the changing needs** of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including **reliability, security and safety**. **Dependable** software should **not** cause physical or economic damage in the event of system failure. Malicious users should **not** be able to access or damage the system.
Efficiency	Software should **not** make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes **responsiveness, processing time, memory utilisation**, etc.
Acceptability	Software must be **acceptable** to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

Software life cycle

Development, use, maintenance

Software engineering methodologies

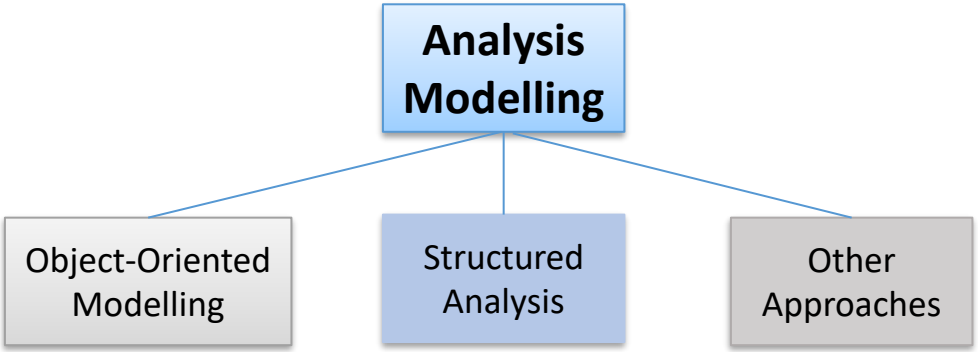
Waterfall model, incremental model, reuse-oriented, Agile

Tools of the Trade

UML: Use Case Diagram (UCD)

Data Flow Diagram (DFD)

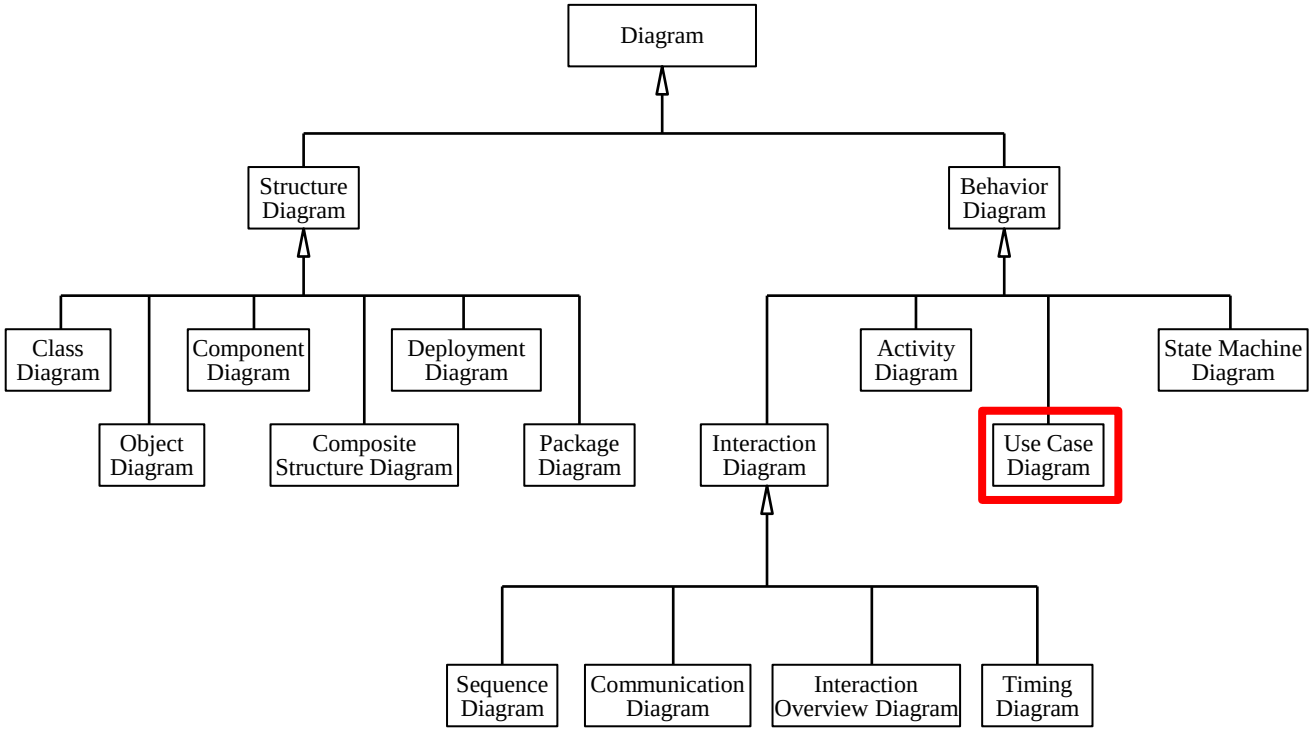
Analysis Modelling



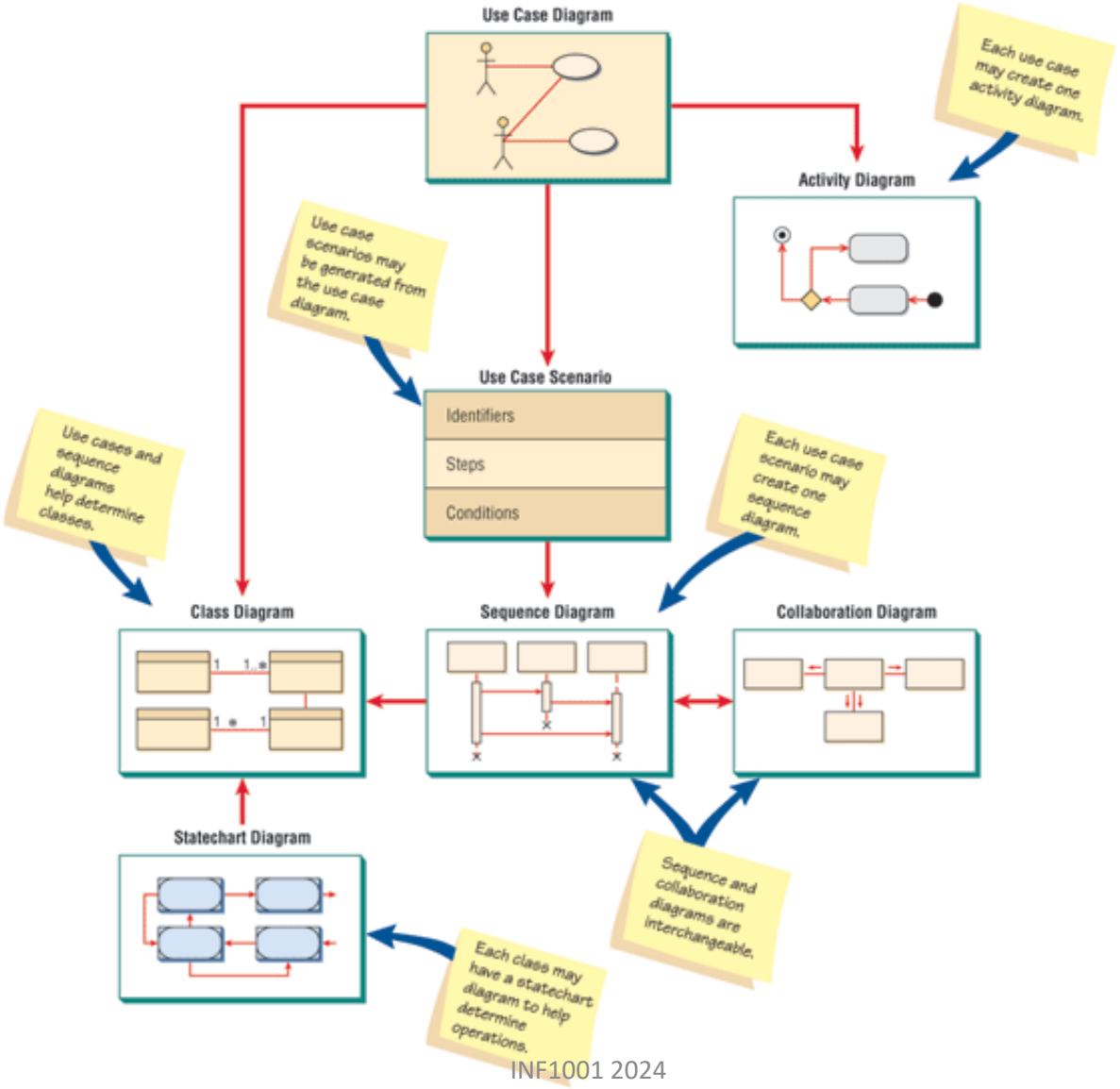
- Analysis Model is constructed to provide information of "**what**" the software should do instead of "**how**" to fulfill the requirements in the software.
- The model emphasizes information such as
  - Functions that software should perform
  - Behaviour it should exhibit
  - Constraints that are applied on the software
  - Relationship of one component with other components
- Object Oriented Modelling specifies functional and behavioural information using objects
- Structured Analysis expresses this information through Data Flow Diagram (DFD)
- Other Approaches include ER Modelling, Problem Statement Language (PSL), etc.

Unified Modelling Language (UML)

- Developed with object-oriented paradigm in mind



Overview of UML Diagrams



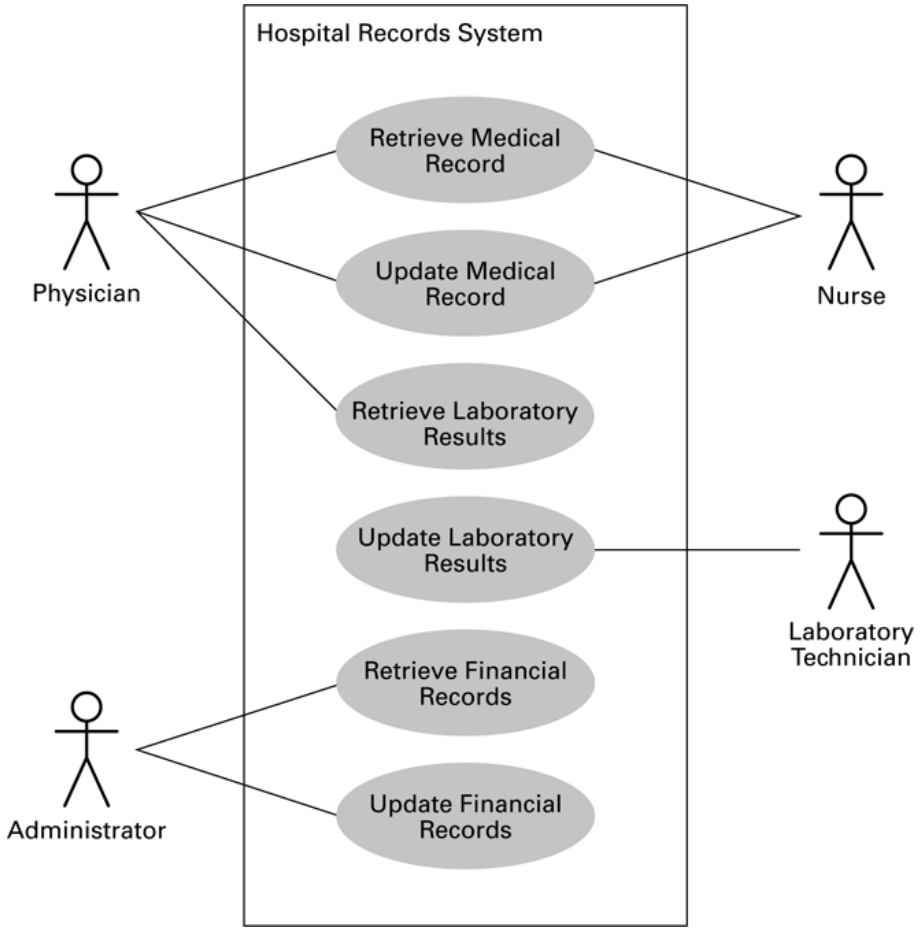
Use Case Diagram

- Attempts to capture the image of the proposed system from a **user's point of view (outside)** – use case diagram

Who can do **what** in the **system**?

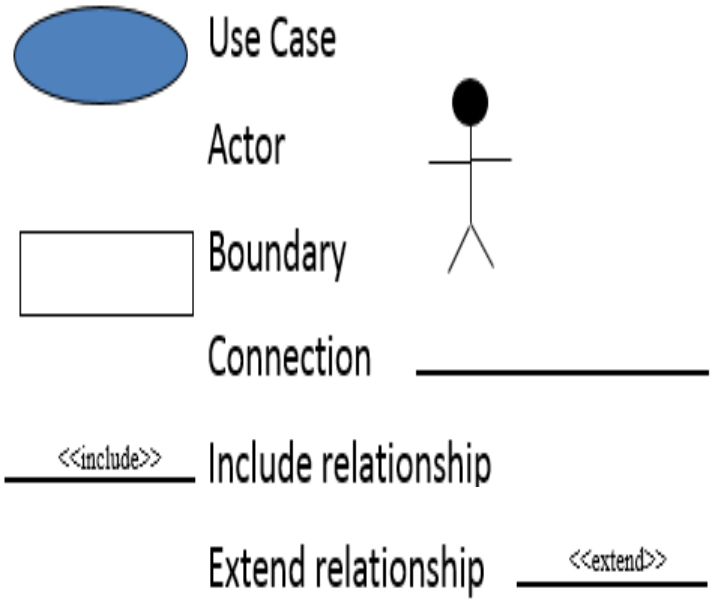
- Indicate the **interaction** between the system and the users.

- E.g.: **Hospital Records System:**

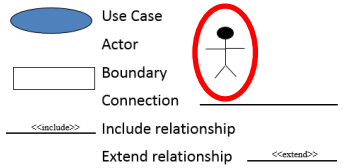


Use Case Diagram: **Symbols/components**

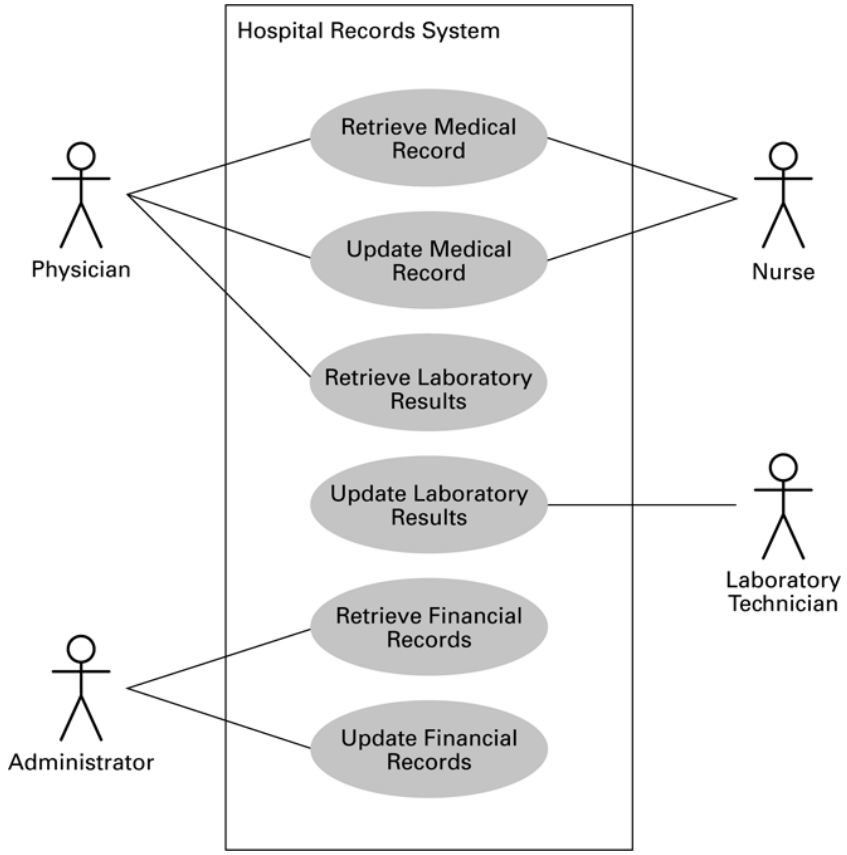
- **Actor**
  - Refers to a particular role of a user of the system
  - Similar to external entities; they exist outside
- **Boundary**
  - which defines the system of interest in relatic
- **Use case symbols**
  - An oval indicating the task of the use case
- **Connecting lines (relationships)**
  - Arrows and lines are used in diagram for beha



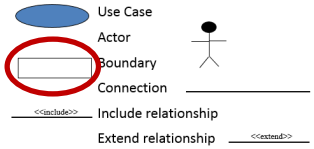
Actors



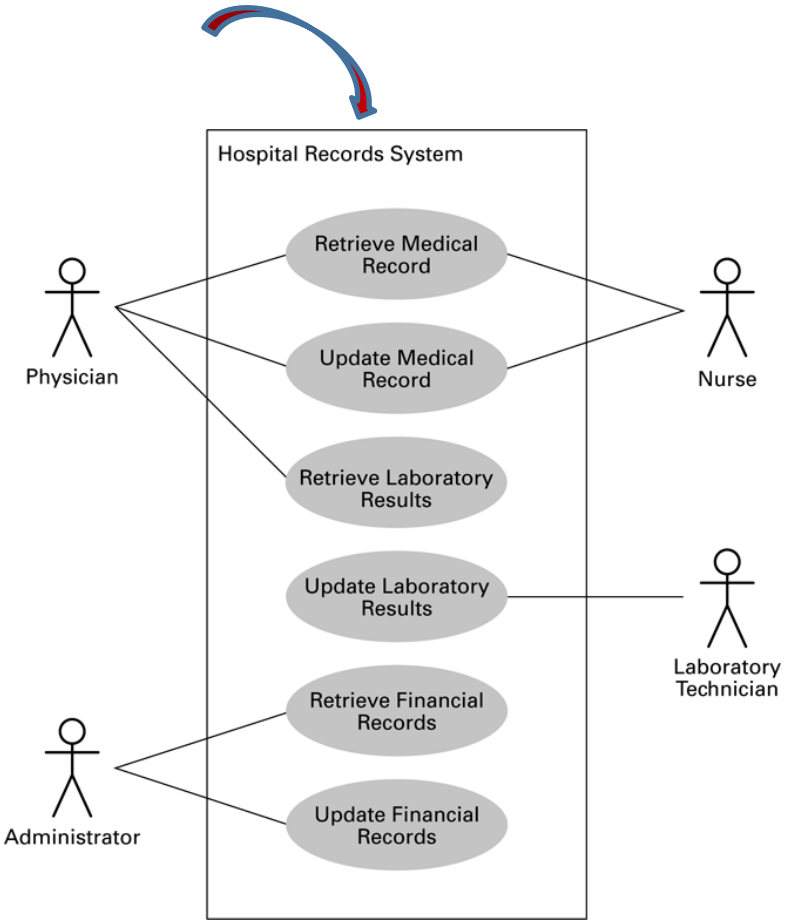
- **Actors** represent role played by one or more users
- Exist outside of the system
- May be a person, another system, a device, such as a keyboard or Web connection
- May interact with one or more use cases and a use case may involve one or more actors
- Naming the actor – based on the function that the actor plays



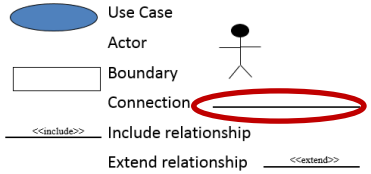
Boundary



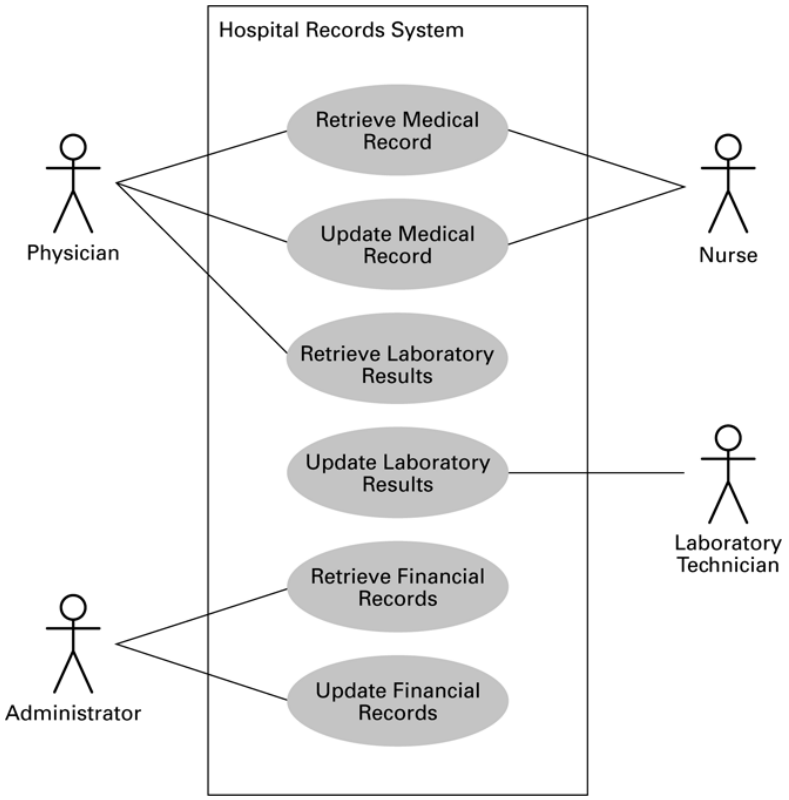
- A **boundary** is the dividing line between the system and its environment.
- System scope defines its boundaries:
  - What is in or outside the system
  - Project has a budget that helps to define scope
  - Project has a start and an end time
- Use cases are within the boundary.
- Actors are outside of the boundary.



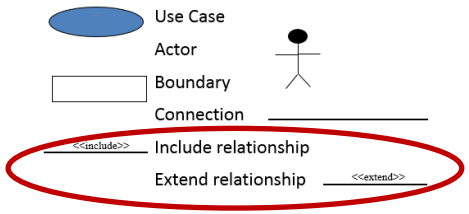
Connection



- A **connection** is an association between an actor and a use case.
- Depicts a usage relationship
- Connection **does not** indicate **data flow**



Use Case Connection

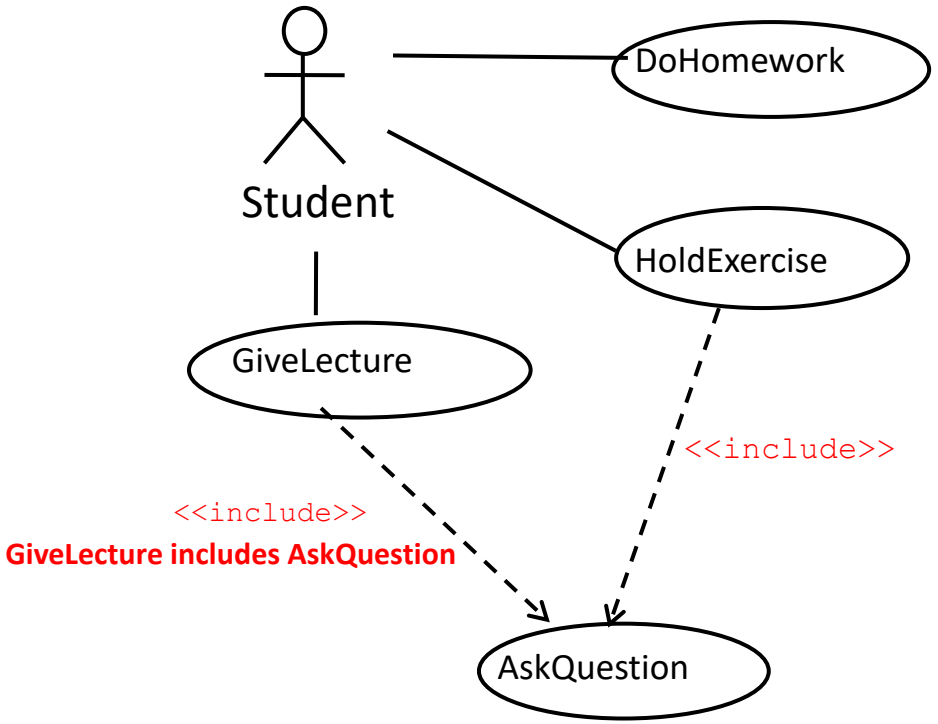


- An actor that initiates an event, which triggers a use case
- The use case that performs the **action** (verb) triggered by the event

Relationship	Symbol	Meaning
Communicates		An actor is **connected** to a use case using a line with no arrowheads.
Include	<<include>>	A use case contains a behavior that is **common** to more than one other use case **or mandatory**. The arrow **points to** the common use case.
Extend	<<extend>>	A different use case handles **additional options** from the basic use case. The arrow **points from** the extended use case to the basic use case.
Generalizes		One UML "thing" is more **general** than another "thing". The arrow points to the general "thing".

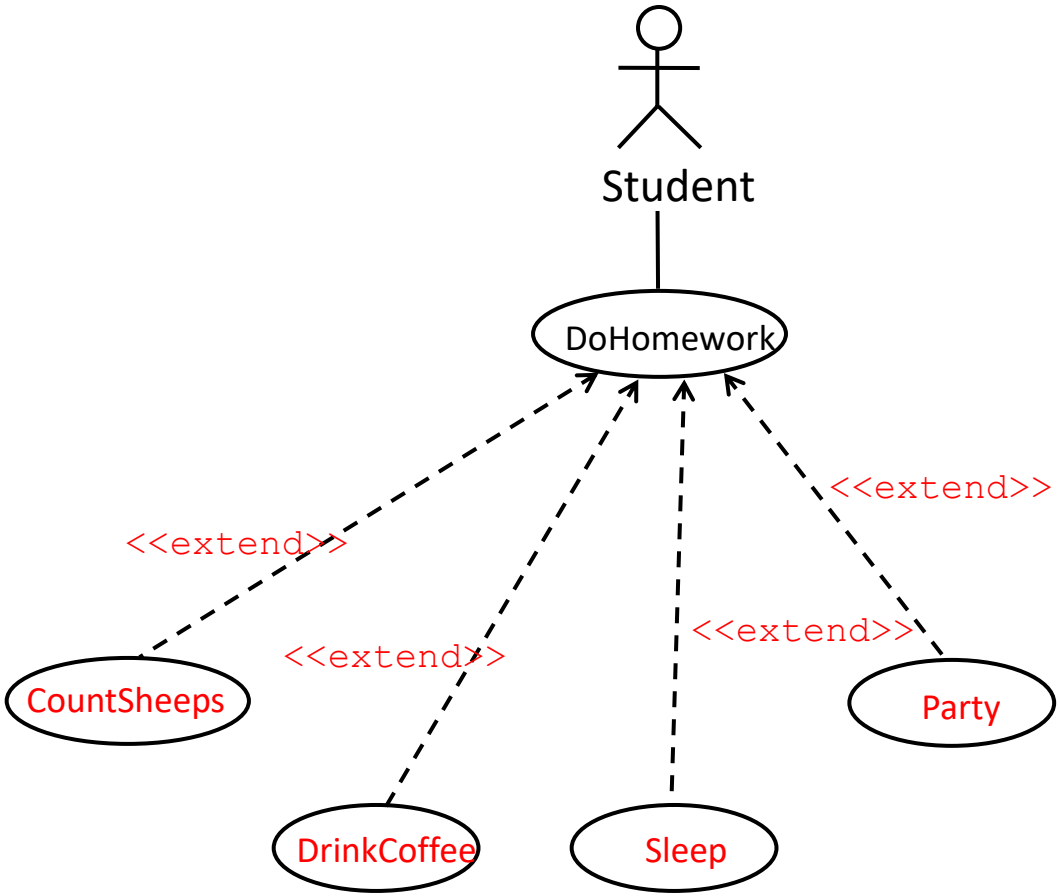
<<include>> Relationship

- Behavior that is **similar** (common) across two or more use cases or **mandatory**
- Break this out as a separate use case and let the other ones “include” it.
- **Include use case required (MUST), NOT optional**

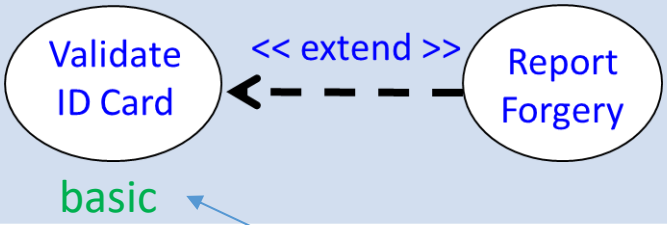
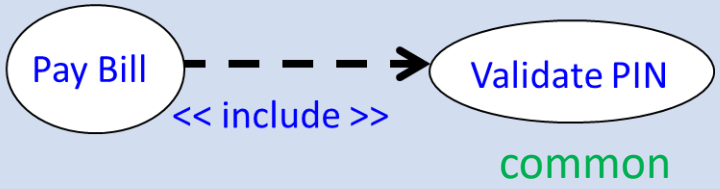


<<extend>> Relationship

- Extends a use case by adding additional options (e.g. behaviors or actions)
- Specialized use case extends the general use case
- **Extending use case is Optional (NOT necessary at all times), supplementary**

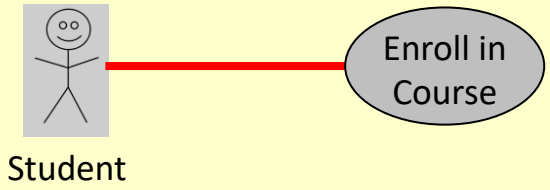


<<include>> vs <<extend>>

<< extend >>	<< include >>
	
Arrow pointing towards basic use case	Arrow pointing towards common use case
Extending use case is **optional**	Included use case is **required**

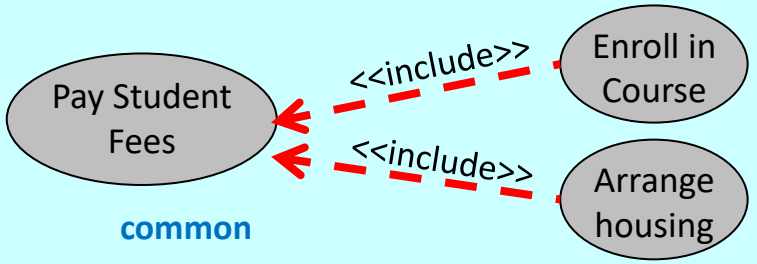
Use Case Relations

- Examples of Use Cases, Behavioral Relationships for Student Enrollment

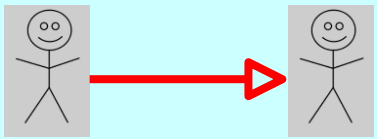


Communicates Relationship

Both **Enroll in Course** and **Arrange housing** include **Pay Student fees**

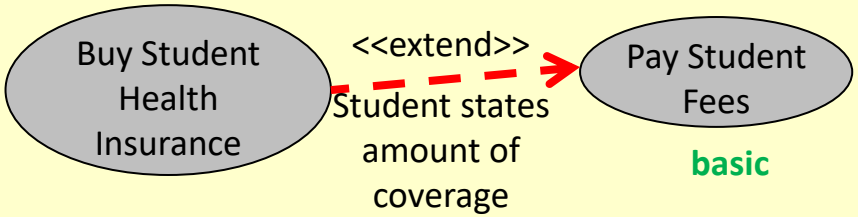


Includes Relationship



Generalizes Relationship

The extended use case **Student Health Insurance** extends the basic use case **Pay Student Fees**.



Extends Relationship

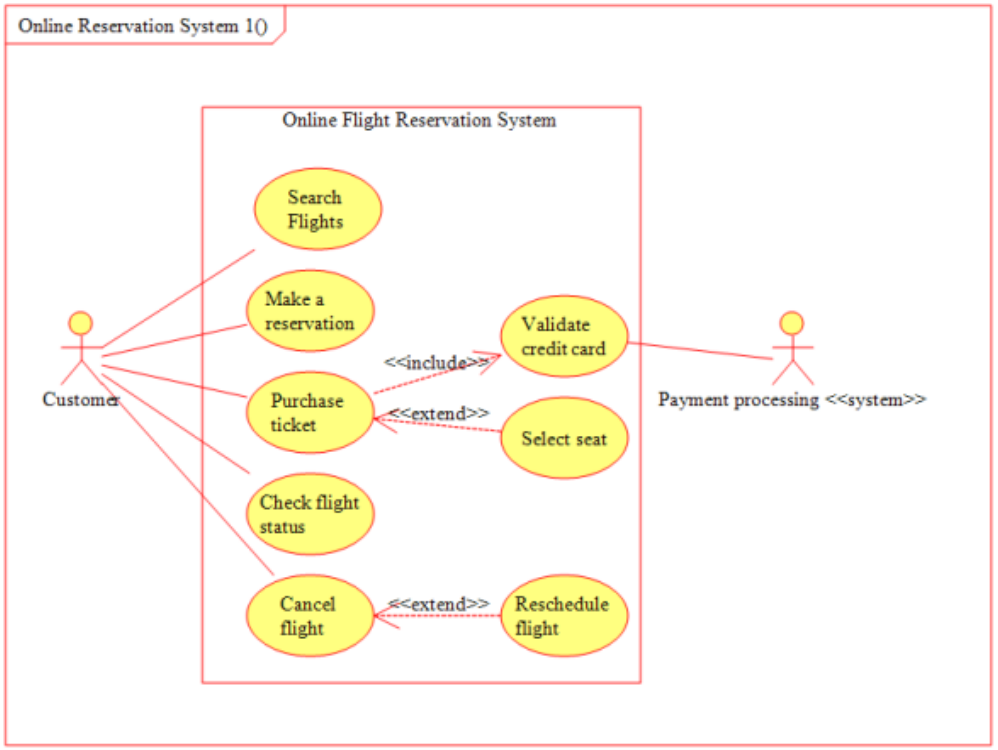
Developing Use Case Diagrams

1. Review the business specifications and identify the actors involved.
2. Identify the **high-level** events and develop the **primary** use cases that describe those events and how the actors initiate them.
3. Review each **primary** use case to determine the possible variations of flow through the use case.

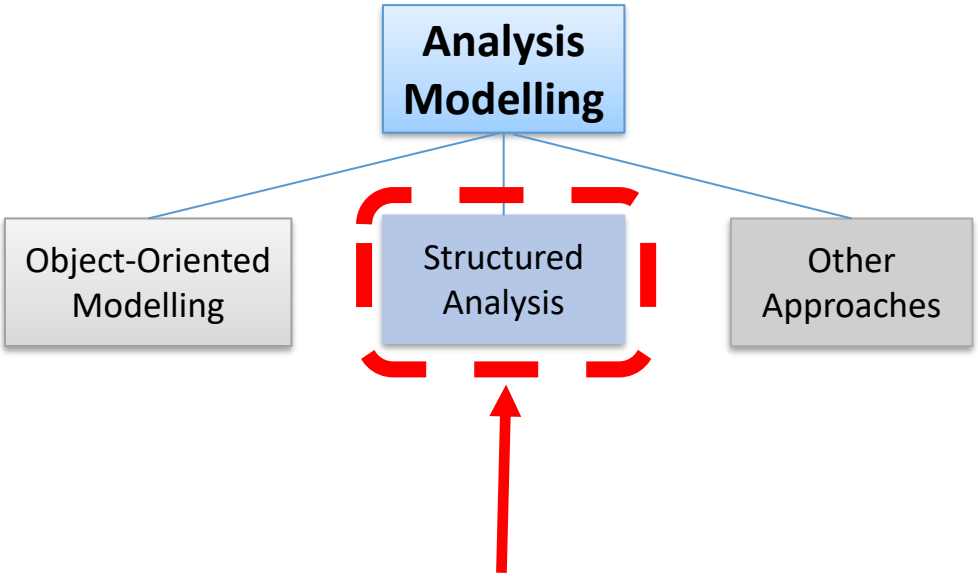
Why are use case diagrams helpful

- Identify all the actors in the problem domain.
- Actions that need to be completed are also clearly shown on the use case diagram.
- The use case scenario is also worthwhile.
- Simplicity and lack of technical detail

Example: Use Case Diagram Representing an Online Flight Reservation System



Structured Analysis



Let's turn our focus now to **Structured Analysis**

Structured Analysis

- Top down approach
- Focuses on **functions** and **data**
- Provide graphical representation to develop new software or enhance existing software

- There are various levels of Data Flow Diagram (DFD)
  - Level of detail process increases with increase in level
  - **Level 0** : Overall view of system – **context diagram**
  - Level 1 : Provides a more detailed breakout of pieces of the context diagram
  - Level 2 : Goes one step deeper into parts of Level 1
  - Level 3 : Goes deeper into parts of Level 2.

Data Flow Diagrams (DFD)

- Focus is on the **data flowing into and out** of the system and the **processing** of the data
- Shows the **scope** of the system:
  - What is to be included in the system.
  - The external entities are outside the scope of the system.

Data Flow Diagrams (DFD)

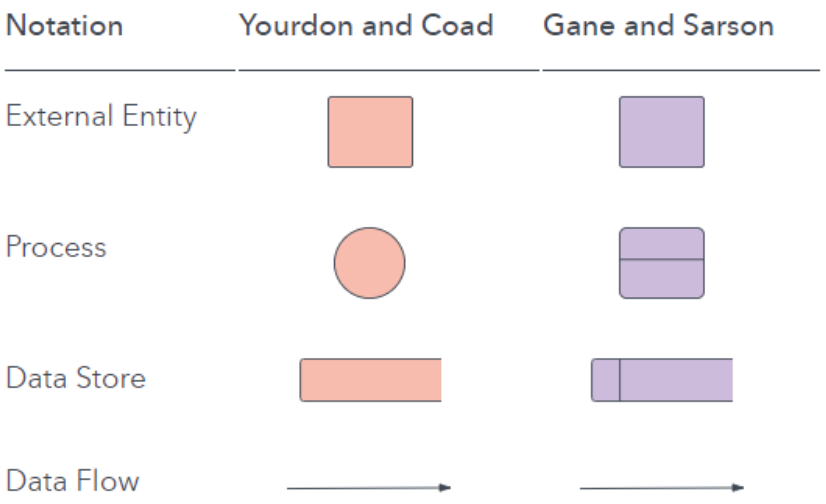
External entity:

- System that sends or receives **data**, communicating with the system being diagrammed
- Sources and destinations of **information** entering or leaving the system.

Process: any process that changes the data, producing an output

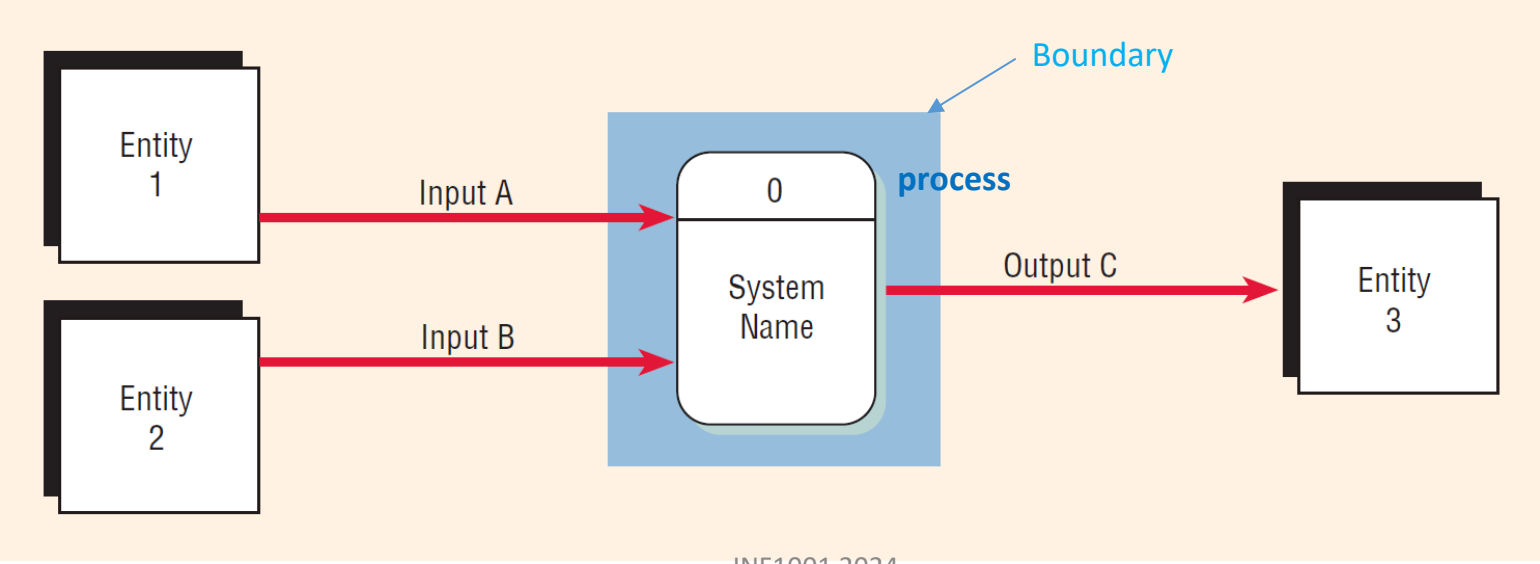
Data Store: files or repositories that hold information for later use, such as a database table or a membership form.

Data Flow: the route that data takes between the external entities, processes and data stores



Context diagram (level 0): Basic rules

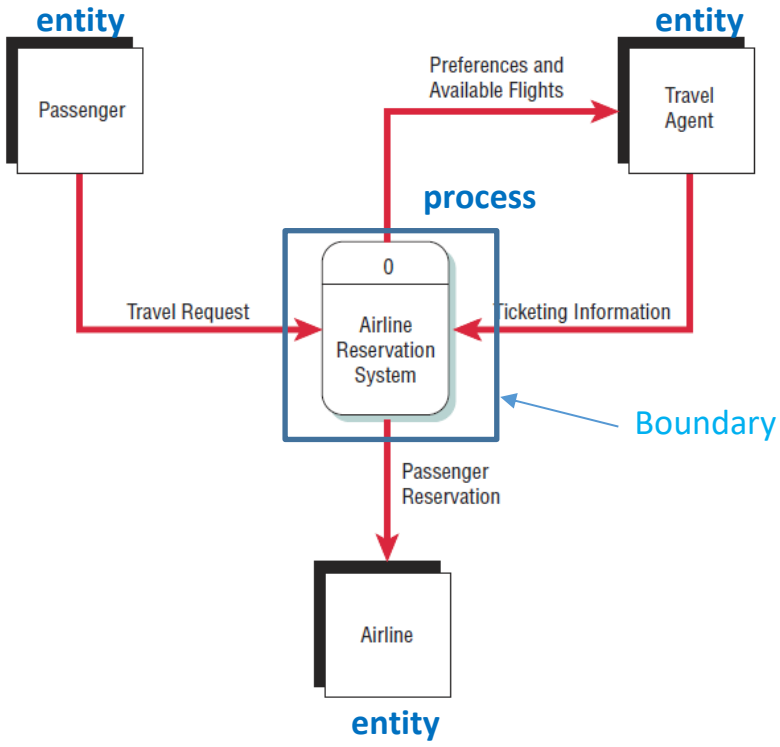
- The data flow diagram must have only **ONE** process
- Must not have any freestanding objects
- A process must have both an **input** and **output** data flow.
- A data store must be connected to at least one process.
- External entities should not be connected to one another.



Context diagram (level 0) – Example 1

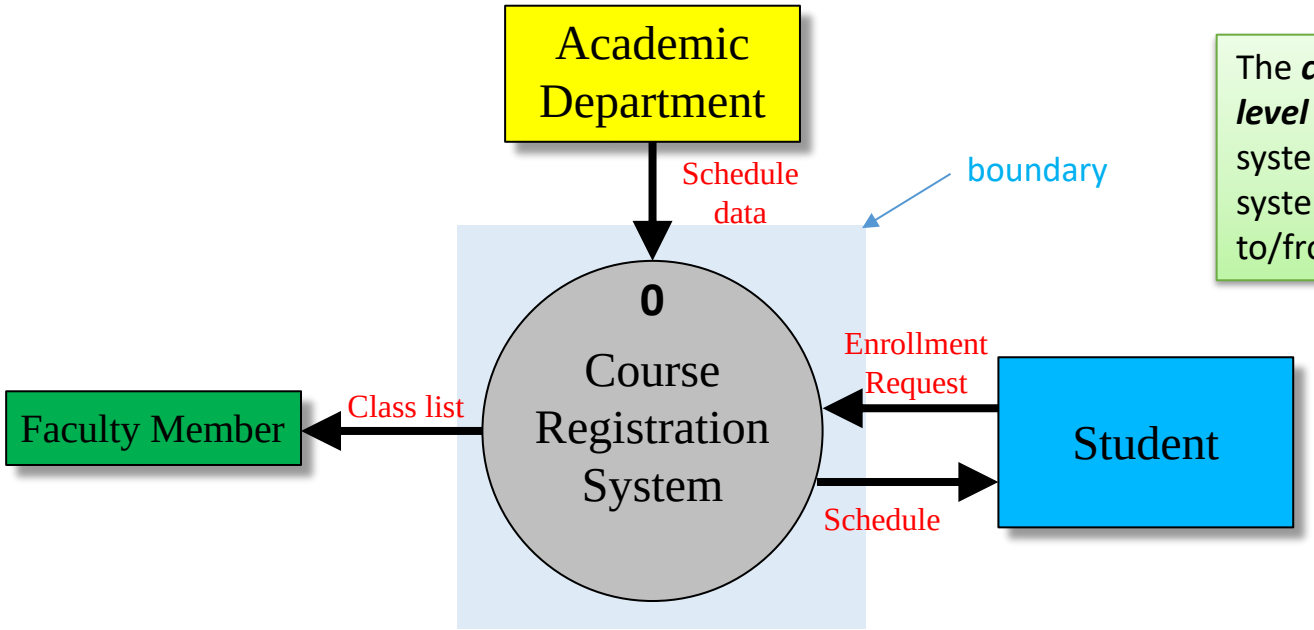
- Shows the highest level of interactions between a system and other actors which the system is designed to face
- Also known as **level 0**
- Contains only one process, representing the entire system
- The process is given the number 0
- All external entities, as well as major data flows are shown

A context-level data flow diagram for an airline reservation system



Context diagram (level 0) – Example 2

Context Diagram (Level 0): Course Registration System

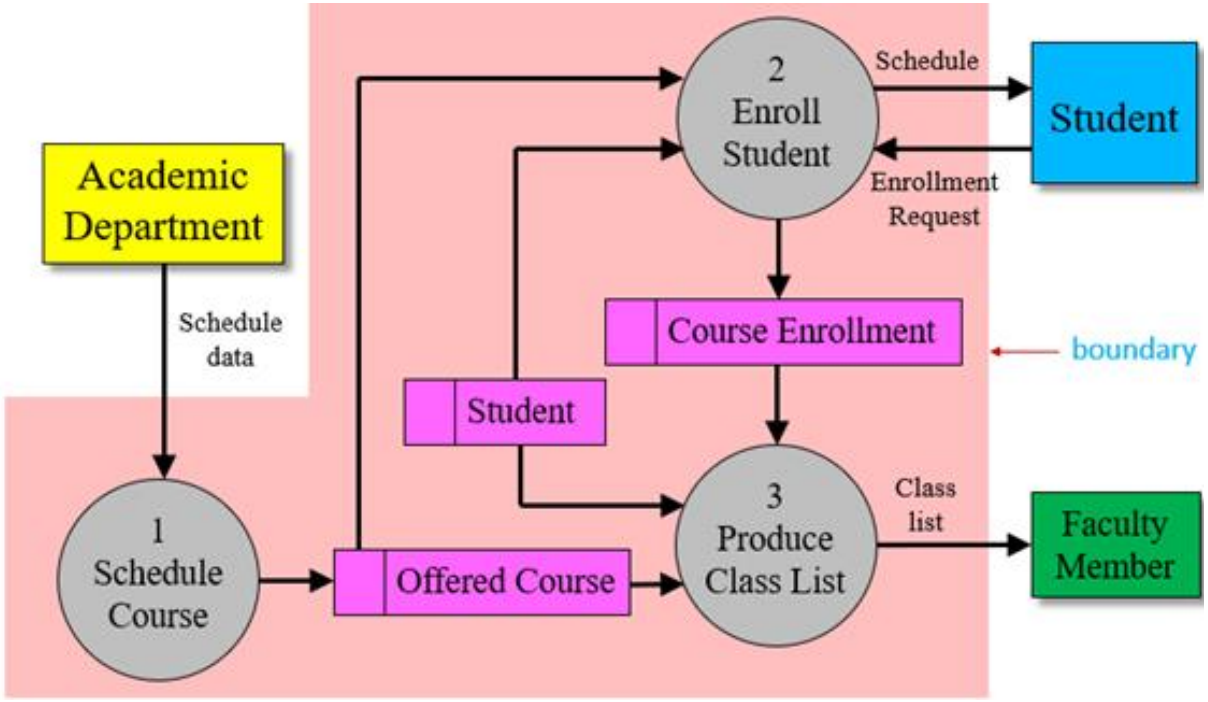


The ***context diagram*** **(also known as *level 0 diagram*)** partitions the entire system. It has only one process (the system), and from it, the data flows to/from the external agents.

Source: "Systems Analysis and Design in a Changing World", Satzinger, Course Technology, 2002

Example 1: Structured Analysis

Level 1 Diagram: Course Registration System



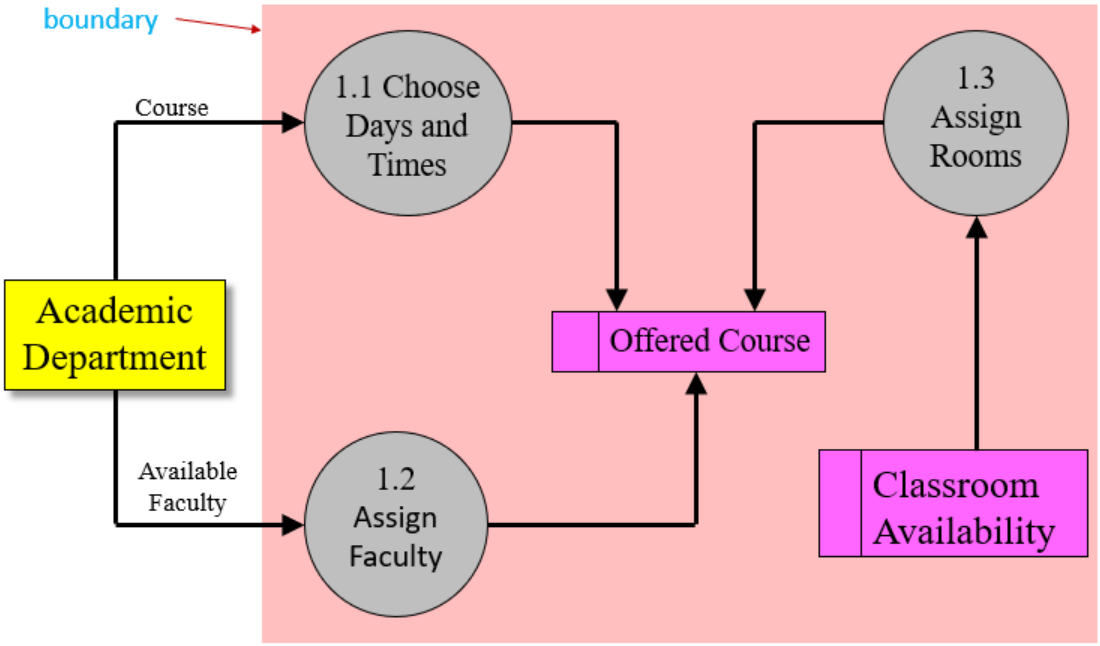
The ***level 1 diagram*** decomposes the system into 3 processes:

- Schedule Courses
- Enroll Student
- Product Class List.

Notice that the 4 data flows represented in the context diagram are preserved in the level 0. This is required.

Example 1: Structured Analysis

Level 2 Diagram: Course Registration System



The ***level 2 diagram*** **decomposes/explodes** the "1 Schedule Course" process into 3 sub-process:

- 1.1 Choose Days and Times
- 1.2 Assign Faculty
- 1.3 Assign Rooms

Note that the data flow "Schedule Data" from level 1 is broken into 2 sub data flows in the level 2: "Course" and "Available Faculty".

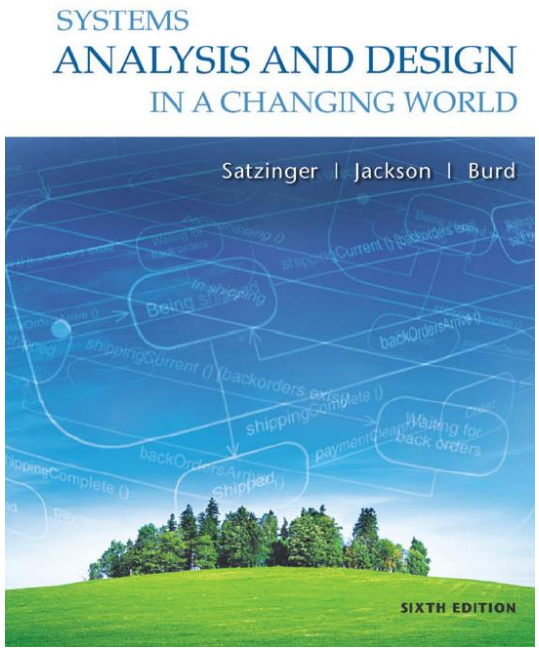
Also note that the "Offered Course" file is still preserved.

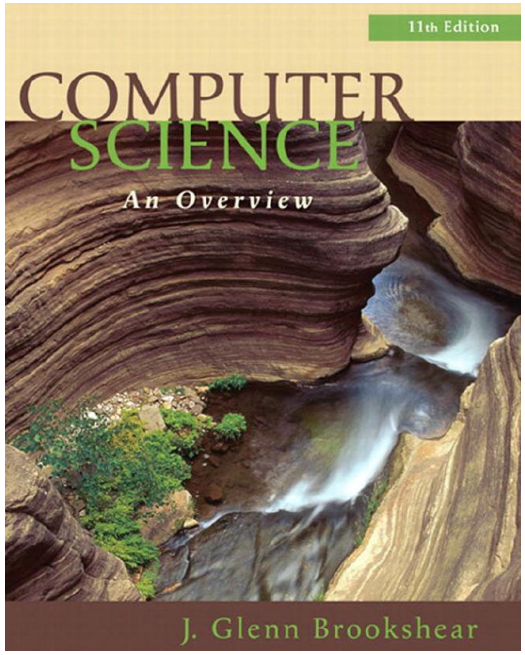
ICT2101 Introduction to Software Engineering
ICT2106 Software Design
ICT2108 Software Modeling and Analysis

CSC2001 Professional Software Development
CSC2002 Team Project

References

- *Chapter 7 Software Engineering*, Computer Science: An Overview, 11th Edition, Addison-Wesley, J. Glenn Brookshear
- *Chapter 4 Domain Modelling*, Systems Analysis Design in a Changing World, 6th Edition, Course Technology, John W. Satzinger
- http://ecomputernotes.com/software-engineering/requirementsanalysis
- https://www.computer.org/web/swebok





Summary

Software life cycle

Development, use, maintenance, importance

Software engineering methodologies

Waterfall model, incremental model, reuse-oriented, Agile

Tools of the Trade

UML: Use Case Diagram (UCD)

Data Flow Diagram (DFD)