Requirements Analysis

Lecture 1

Welcome

Fundamentals of Software Development (FSD) is a comprehensive introduction to software development process. FSD teaches problem solving techniques used in the software development lifecycle to develop a solution for an industry-like case-study. The result solution is developed using popular programming languages (Java or Python).

The Case Study

Your team is hired to develop an in-house desktop banking application **"DeskBankApp"** to provide CLI and GUI interactive banking functions for users.

The Problem:

Develop a Desktop Banking application

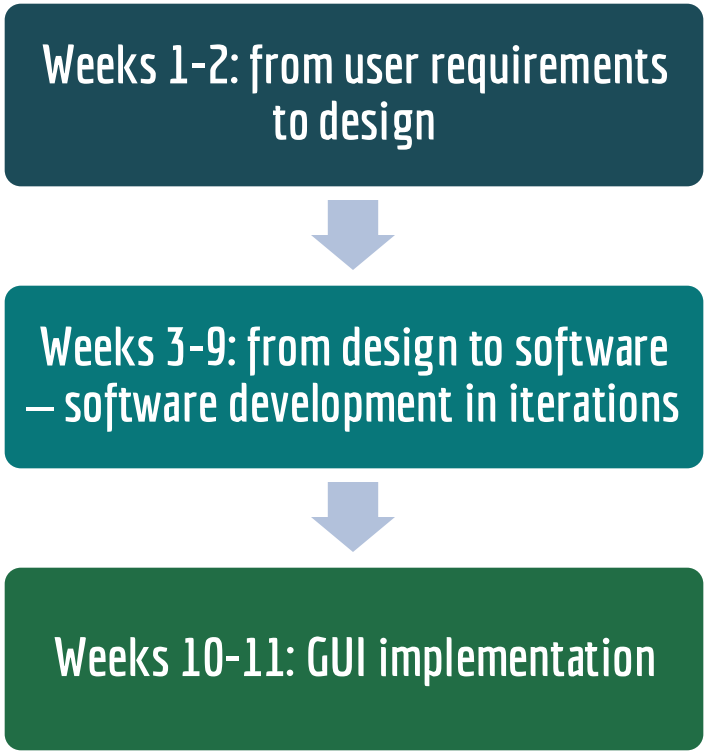
The Process:

Use Software Development Process to develop the application

The Result:

Interactive CLI and GUI Desktop Banking application

Subject Structure



Assessment Overview

Quiz 1 (15)

- Week 7; in-class; content of Weeks 1-2

Quiz 2 (15)

- Week 12; in-class; content of Weeks 3-11

Project (70):

- Part 1 (end of Week 7): requirement analysis (group, Sparkplus peer rating)
- Part 2 (end of Week 13): software development (individual)
- Part 3 (end of Week 13): Showcase

Contents

What is Software Engineering?

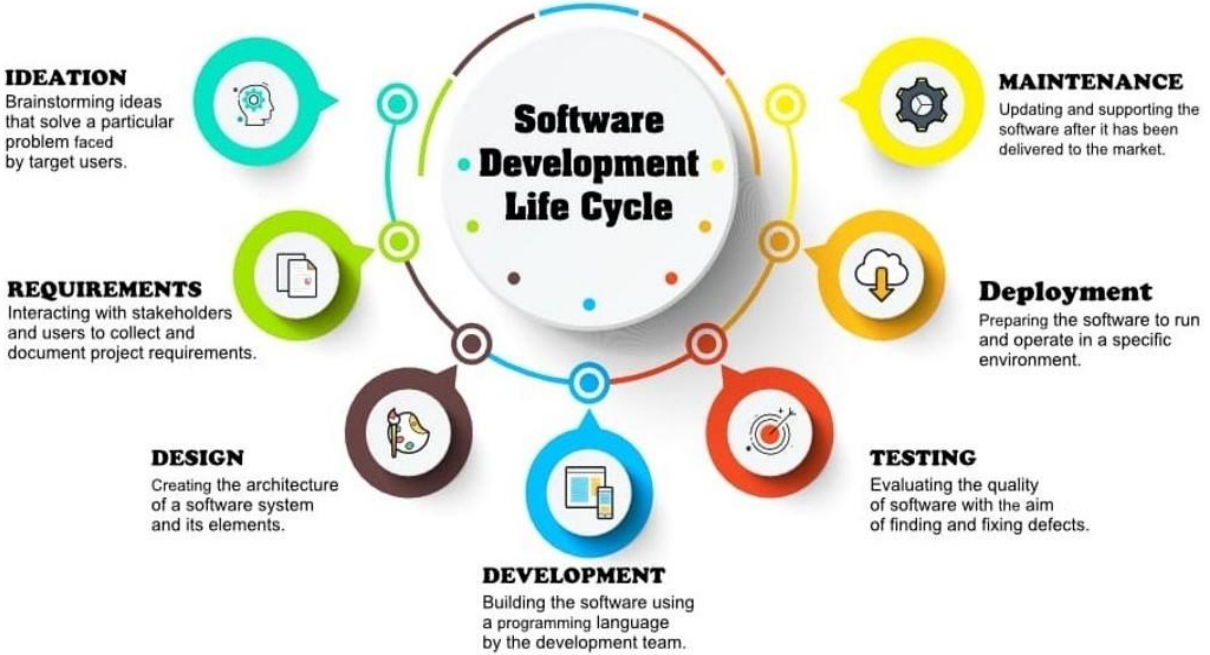
Software engineering (IEEE Standard): The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

- **Systematic**: software development lifecycle
- **Disciplined**: established methodologies
- **Quantifiable**: measurable outcomes

Software Development Lifecycle

SDLC is a set of activities and associated results that produce a software.

SDLC is the methodology used by developers to implement the software development steps.



Software Development Methodologies

This lesson discusses two commonly used software development methodologies:

- The **Waterfall** model (Reference: https://en.wikipedia.org/wiki/Waterfall_model)
- The **Agile** model (Reference: https://en.wikipedia.org/wiki/Agile_software_development)

The Waterfall model is a sequential development process, where each phase begins at the end of the previous. It is called a Linear model.

The Agile model is a repetitive iterative approach where tasks are completed in cyclical progression known as sprints of 2 to 4 weeks each.

Software Development Methodologies

The Waterfall model is a linear progression of the SDLC

Software Development Methodologies

The Waterfall model has its own strengths and weaknesses:

Strengths

- **Simple and easy to manage – each phase has specific deliverables**
- **Clear and set milestones**
- **Fixed project requirements**
- **Works well for smaller projects with specific set of requirements**
- **Determine the schedule and end goals early**
- **Uses clear structure**

Weaknesses

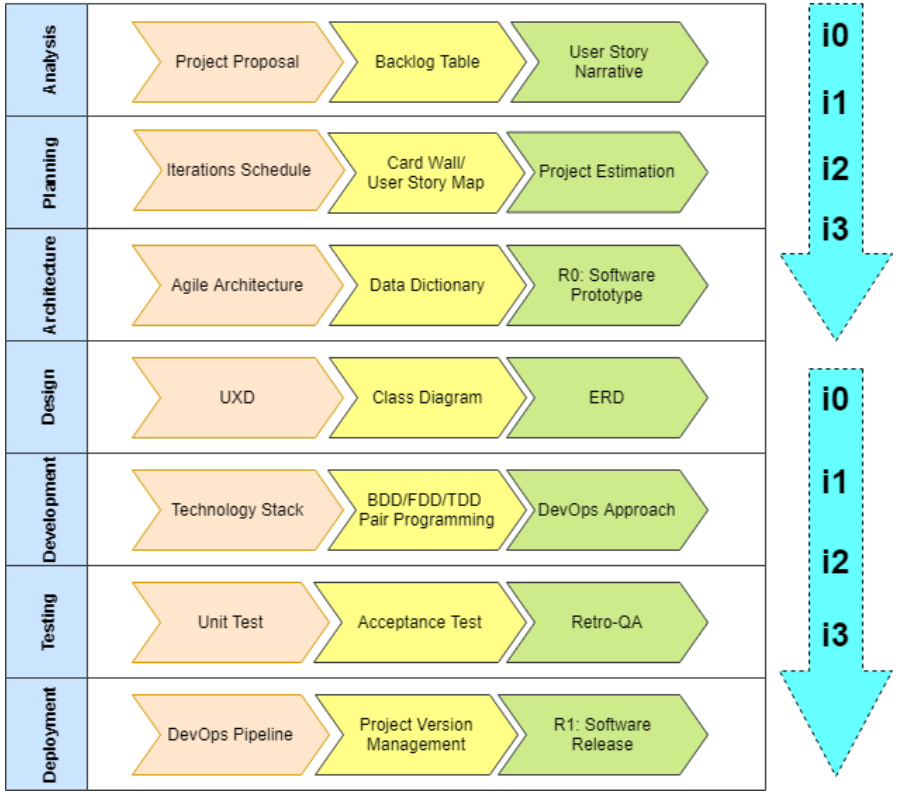
- **Working software produced at the end**
- **High uncertainty of software quality and functionality**
- **Delayed Testing until the end which delays software bugs discovery**
- **After the requirements phase is completed, there is no formal way to change the requirements**
- **Difficult to implement for complex projects since it has a fix working model.**

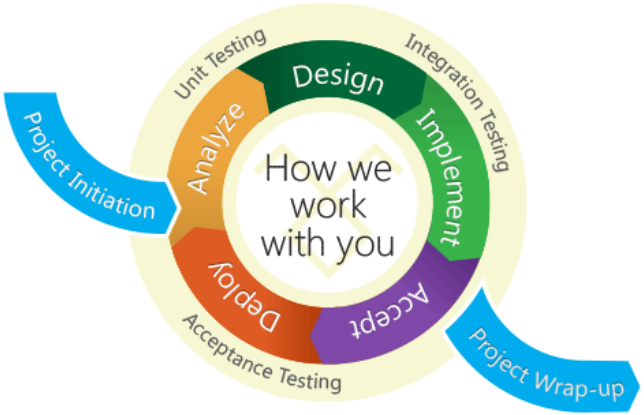
Software Development Methodologies

The Agile model is an iterative cyclical progression of the SDLC

- The agile model follows a repetitive structure based on iterations (sprints).
- Each iteration is 2-4 weeks, and the phases of SDLC repeats in each iteration.
- Each release is composed of 3 or more iterations.
- At the end of every release (cycle) a working prototype software is produced.
- The prototype is tested for QA and used as input in the next release (cycle).
- A project contains multiple releases.

Software Development Methodologies





Software Development Methodologies

The Agile model has its own strengths and weaknesses:

Strengths

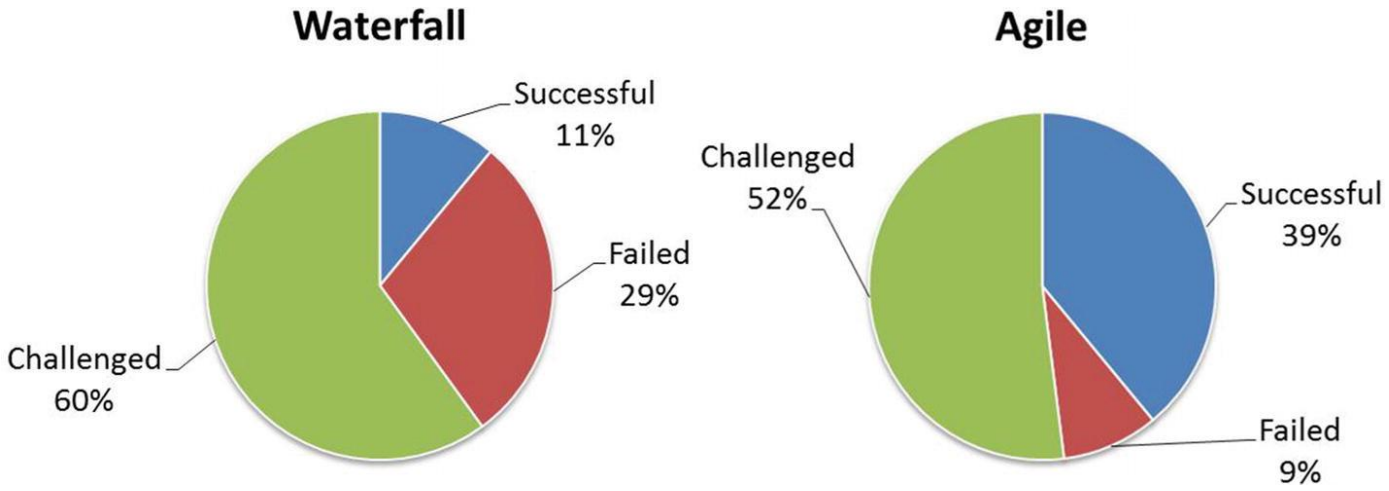
- **Allows innovation through team collaboration**
- **Reduce product time to market**
- **Improved software quality through continuous testing**
- **Risk reduction and chances of finding software bugs early**
- **Customer is kept in the loop to ensure customer satisfaction**
- **Flexibility to implement requirements changes at anytime**
- **Automates most of the SDLC by integrating with DevOps approach**

Weaknesses

- **Lack of long-term planning**
- **Cost estimation is not easy to determine**
- **Difficult to coordinate daily workflow**
- **Limited documentation**
- **No finite end to the project**
- **Difficult to see the end result due to the cyclical nature of agile**

Software Development Methodologies

Agile vs Waterfall

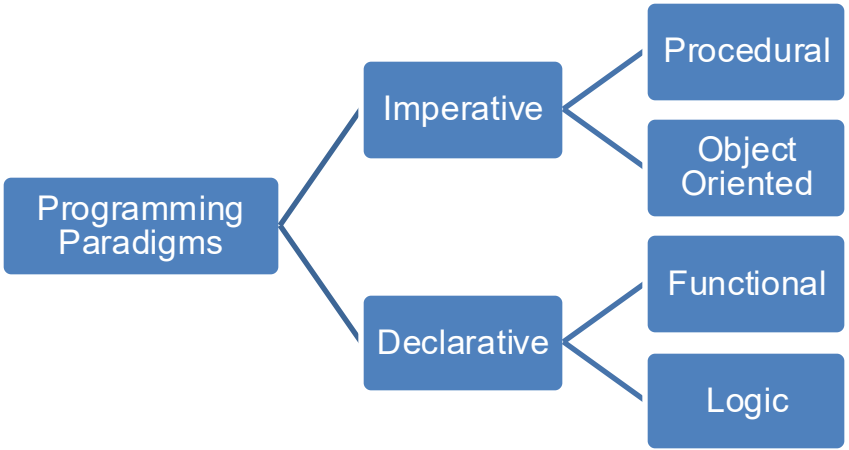


Reference: https://www.atlassian.com/agile/project-management/project-management-intro

Software Development Paradigms

A programming paradigm conceptualizes and structures the implementation of a computer program. This lesson discusses :

- Procedural paradigm
- Object-oriented paradigm

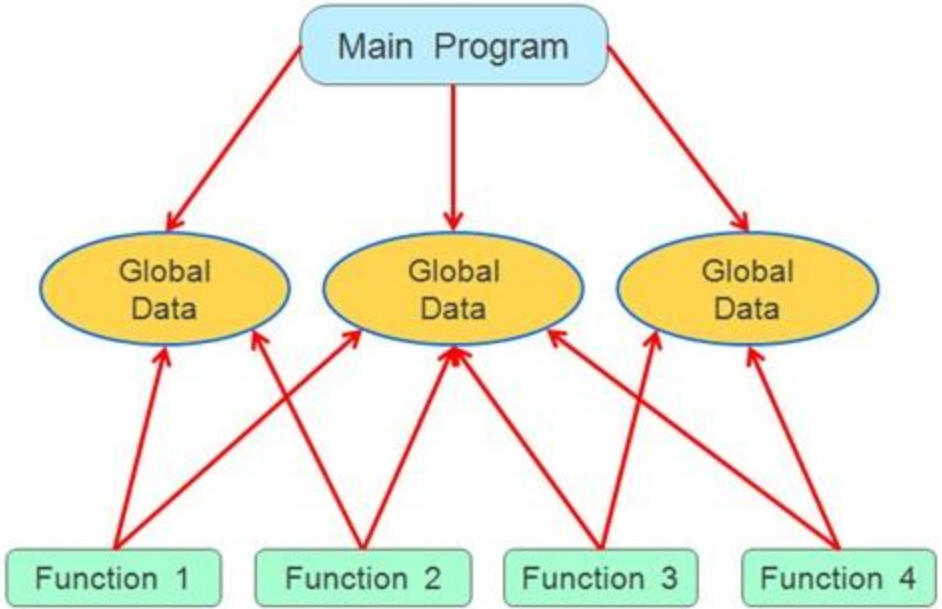


Procedural Programming Hierarchy

- Procedural programming organizes a program as a sequence of steps (i.e., procedures). These steps are often decomposed into smaller sub-problems — this is called **top-down decomposition**. Each sub-problem is typically implemented as a function or procedure. Programming languages such as C, Pascal are designed for procedural programming.
- Procedural programming decomposition is a hierarchal decomposition of functions, into a tree. The top or root of the tree represents the main problem or the entry point (e.g., main()), the leaves at the bottom denote individual procedures or functions, and the intermediate nodes are functions called by the functions above them and that call functions below.

Procedural Programming Data Access

- Procedural program functions interact within the same workspace.
- Procedural program functions interact and modify the global program data. Functions may also maintain local state information, but only for the duration of their execution, by having their own data stored into local variables.



Object-Oriented Programming Paradigm

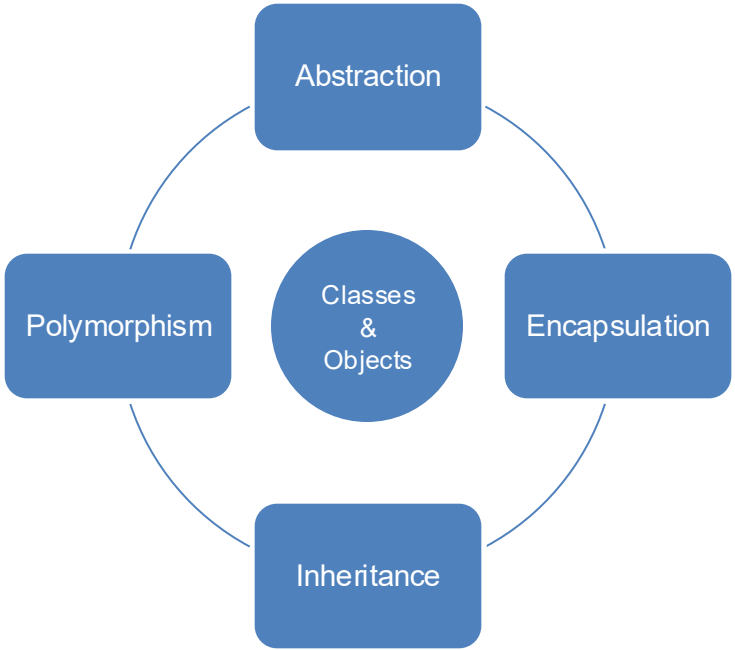
Object-oriented programming paradigm organizes a software into multiple classes. A class is a template containing code using appropriate programming language (Java, C++, Python, C#, etc...).

- Object-oriented programming evolves around the concepts of:
    - Better code reusability
    - Better code security
    - Better design
    - Better data security
- Object-oriented program executes multiple classes and creates many objects at runtime. The objects interact to complete the functions of the software.

Object-Oriented Programming Paradigms

Object-Oriented Paradigm is built upon the 4 key principles:

- **A**bstraction: The process of hiding unnecessary details about an object while exposing its essential properties and behaviours.
- **E**ncapsulation: Restricts the direct access to components of an object, while using methods such as getters and setters instead.
- **I**nheritance: The process of creating sub-classes (children) that inherits the properties (functions and data) from the super-classes (parents).
- **P**olymorphism: Allows for the creation, use, and storage of multiple objects that inherit from the same parent class.



Requirements Analysis

Software requirements are instructions provided by the stakeholders that describe a target system for development.

- Requirements are a description of the system properties or attributes, and how a system should behave.
- Requirements are analysed by development teams and decomposed into “functional requirements” and “non-functional requirements”.
    - Functional requirements refer to the functionality or services that the system is expected to provide, describing interactions between the system and its environment, without focusing on implementation details.
    - Non-functional requirements is a collective term adopted to describe the quality attributes and constraints of the system that are not directly related to the functional behaviours.

Requirements Analysis Overview

Steps to follow:

- Gathering/Eliciting requirements:
  - Interviews and meetings with stakeholders
  - Questionnaires and feedbacks
  - Observations
- Analyse the documented requirements
  - Reading and understanding the documentation
  - Perform **requirements validation**
- Identify use cases or user stories
  - A use case is a sequence of actions that an actor performs to complete a given task.
  - In agile, use cases are expressed as user stories that describe software features from a user's perspective.
- Use case modelling
  - Organize the high-level system functions into UML use case diagram

Requirements Validation

Requirements Validation is a documentation-based testing used to evaluate if the system [specified in the user story backlog] meets the stakeholders' requirements. The following steps are used to conduct document-based testing:

- Define the testing criteria: Check for compliance, correctness, completeness, consistency, usability.
- Perform the testing: Verify the requirements against the above criteria
- Record the results: Record the test results in a properly formatted document
- Discuss the results: Discuss the results with stakeholders (customers, management, etc ...)
- Implement changes: Implement changes to the requirements based on the discussions

Requirements Validation Template

Criteria	Description	Satisfactory Score (out of 5)	Recommendations
Compliance	Degree to which the requirement complies with industry standards.		
Correctness	Degree to which the requirement is correct in terms of spelling, accuracy, grammatically, etc.		
Consistency	Degree to which the requirement can be mapped to use cases.		
Completeness	Degree to which the functional requirements match the intended software.		
Expandability	Degree to which the requirement can be modified and improved to meet the project objectives.		

Requirements Analysis - Use Case

What are use cases?

- A use case is something an actor wants the system to do, and it primarily captures functional requirements (occasionally it captures non-functional requirements). Use cases are always started by an actor or are always written from the perspective of actors.
- A use case consists of a series of actions that a user must initiate to carry out some useful work and to achieve his/her goal (preconditions, main flow, alternative flows, postconditions).
- Use cases reflect all the possible events in the system in the process of achieving an actor's goal.
- A complete set of use cases specifies all the possible ways the system will behave and defines all the requirements of the system.

Requirements Analysis - Identifying Use Cases

How can we identify a use case ?

To simplify the process of identifying and selecting the correct use cases for modelling, ask the following questions:

- What functions will a specific actor want from the system?
- Does the system store or retrieve information?
- What happens when the system changes state ?
- Does the system interact with any external system?
- Does the system generate any reports ?

Use Case Backlog Template

To track the status.

ID	Use Case Title	Actor(s)	Goal/ Description	Preconditions	Postconditions	Priority	Status	Notes
UC-01	Search for books	Customer	User searches books by keyword, title, or author	User is on the homepage	Books matching the search criteria are shown	(High, Medium, Low)	(Backlog, In-progress, To do, Completed, Q&A, Planned, etc)	(When to complete, any related use cases, dependencies, etc)

The Agile Approach to Requirements - User Story

- A user story is a requirement (linked to a process, product, service or system).
- A user story is usually planned to be delivered in a single iteration or sprint. Though, some user stories may span over multiple iterations or sprints and subject to decomposition.
- A user story is a single requirement expressed from developers' perspective and it describes a functional or non-functional "goal" in the system. We can use the template "As …, I want to …, so that …" to write user-stories.

As a student

I want to launch the online gym registration page

So that I can start the registration process.

As a student

I want to be able to view different gym registration options or types

So that I can select the appropriate registration type.

User Story Backlog Template

Estimate the relative effort, complexity, and risk
- Low values for small, easy stories
- High values for big, hard stories, which should potentially be split

- No user story name.
- Both functional and non-functional stories.

Urgency.

ID	Use Story	Acceptance Criteria	Priority	Status	Story points	Notes
US-101 (Indicate stories related to the same “Search for books” feature by assigning them IDs with the same hundreds digit.)	As a customer, I want to search for a book by title, author, or theme, so that I can find a book I want to buy.	- Search results match keywords - Partial matches are supported - No results message shown	(High, Medium, Low)	(Backlog, In-progress, To do, Completed, Q&A, Planned, etc)	Fibonacci sequence (1, 2, 3, 5, 8, 13...)	Dependencies

UML Use Case Modelling

What is UML (Unified Modelling Language) ?

- UML is a standard set of diagrams that provide developers with visual representation of a software (functions, actions, objects, components, activities, machine states, time sequence, etc)
- UML helps the developments teams communicate visually, explore potential software designs, and validate the architectural design of the software.
- There are two main UML types:
    - **Structure Diagrams**: Class Diagram, Object Diagram, Component Diagram, Deployment Diagram, etc
    - **Behaviour Diagrams**: Use Case Diagram, Sequence Diagram, State Machine Diagram, Activity Diagram, etc

This subject discusses: Use Case Diagram and Class Diagram

Steps for Use Case Modelling

1. **Choose the system boundary**. Draw the system boundary as a box and decide what is inside and what is outside of the system.
2. **Identify actors**. Identify primary/secondary actors and represent them using a labelled figure (a person) or box (a system).
3. **For each actor identify their goals.** Define use cases that satisfy an actor's goals or the tasks an actor needs to do with the system. A use case is a complete interaction leading to a goal, including one or more subtasks.
4. Names of the use cases should be **goal-oriented**, typically **verb-noun phrases**, e.g., Place Order, Update Shopping Cart
5. **Connect the actors and use cases**. Identify the relationships and represent them in the use case diagram.

Basic Elements in Use Case Modelling

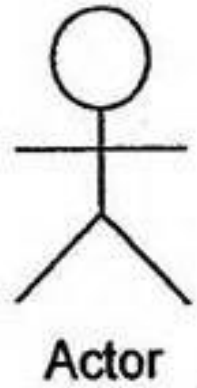
- Actors → An entity (a person or a system) that performs a role in the system
- Use cases → Oval representation inside the system boundary of a functional requirement
- System Boundary → Square representation of the system scope
- Relationships:
  - *Association* → Between an **actor** and a **use case.**
  - *Include* → The included use case is always necessary for the completion of the activating use case.
  - *Extend* → The extension use case is activated occasionally at specific extension point.
  - *Generalization/Inheritance* → You can generalize use cases when they achieve same goal by different means.

Between Use Cases

UML Use Case Modelling

What is an actor ?

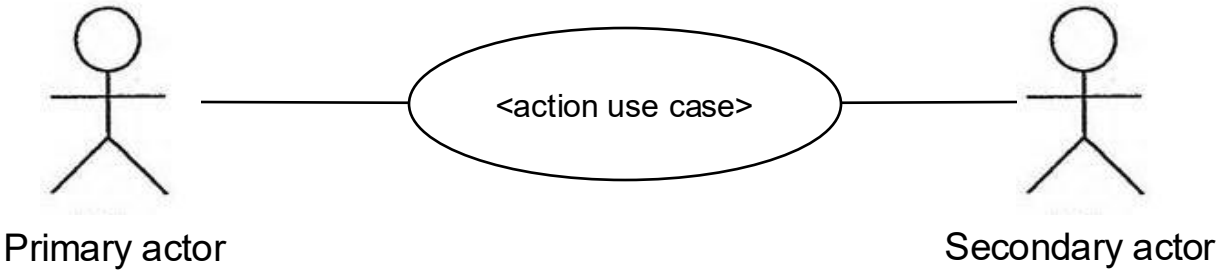
- An actor is an entity (person, software, third-party organization, etc...) that has access to the system and plays a role. e.g., a cashier in a convenient store. An actor is a labelled figure (a person) or box (a system) outside the box or system boundary. There are two categories of actors:
    - Primary actor: is the entity whose goal identifies and drives the use case.
    - Secondary actor: is the entity that the system needs assistance from to achieve the primary actor's goal.



UML Use Case Modelling

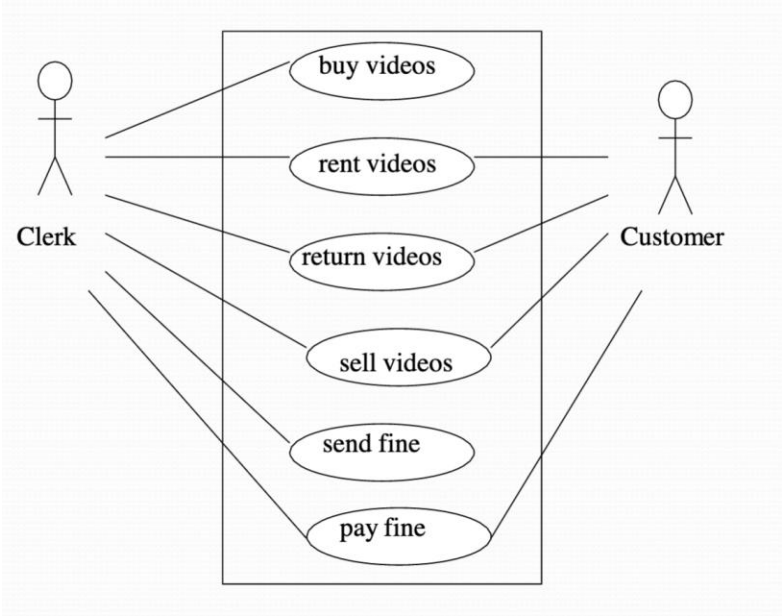
What is a use case ?

- A use case is an oval visual representation of a distinct business function.
- A use case represents a goal that an actor intend to achieve in the system (or a task an actor needs to do with the system).
- The name of a use case has the form ‹verb object›, e.g., ‹read a book›, ‹play music›, ‹sell a pen›, etc
- A solid link or line between an actor and a use case means the actor participates in the use case.

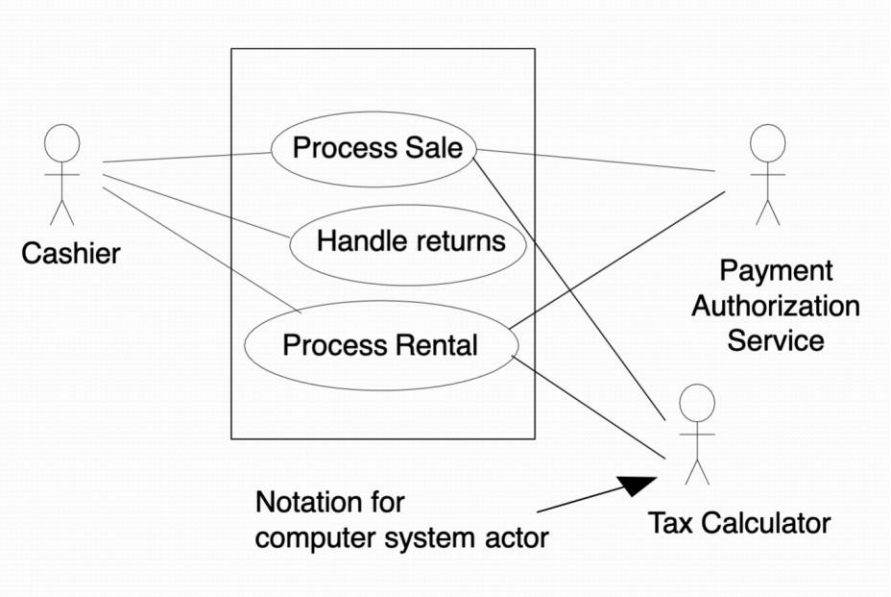


Use Case Model Examples

Video Stream



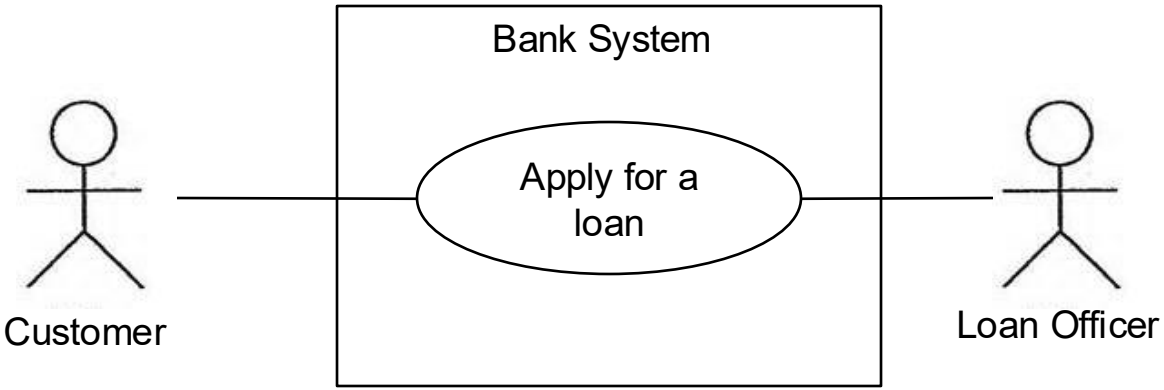
Process Sale



UML Use Case Modelling

What is a boundary ?

- A system boundary defines the scope of a system. The boundary is represented by a rectangle.
- A system boundary of a use case diagram defines the limits of the system and shows the use cases (functionalities included in the system).
- Actors invoke (interact) with the use cases of the system existing within the boundary scope.



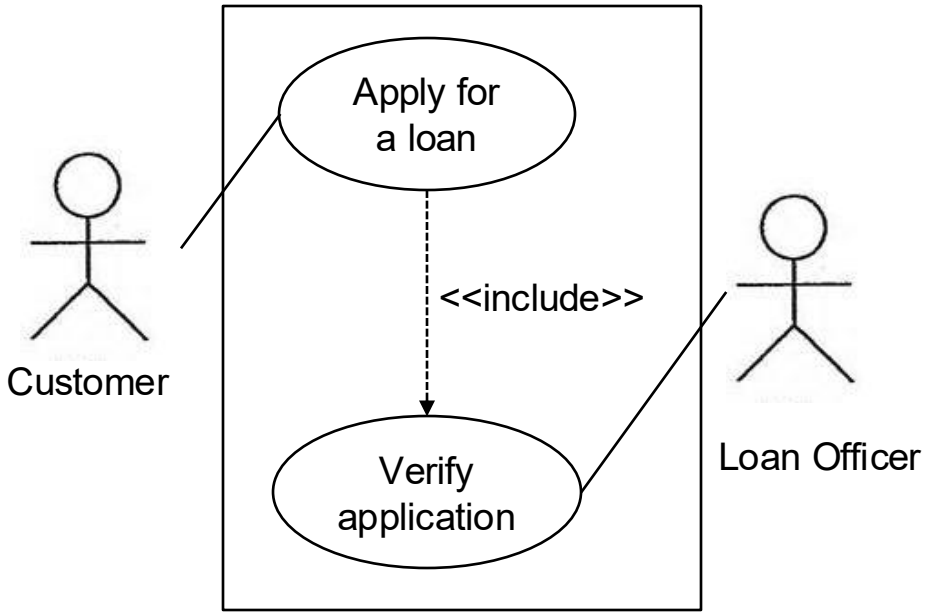
UML Use Case Modelling

Relationships: Include

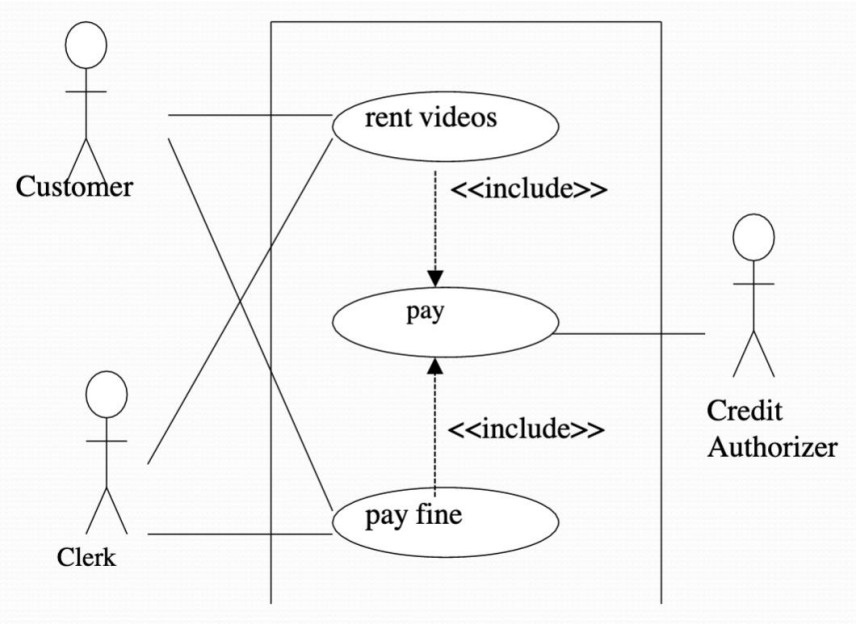
- The include relation is used to show that the same set of actions are included in several use cases.
- The included actions are shown as a use case and linked to relevant actors.
- A dotted arrow labelled ‹‹include›› is drawn from the primary case to the included case.

Include Relation Examples

Loan Application



Video Stream



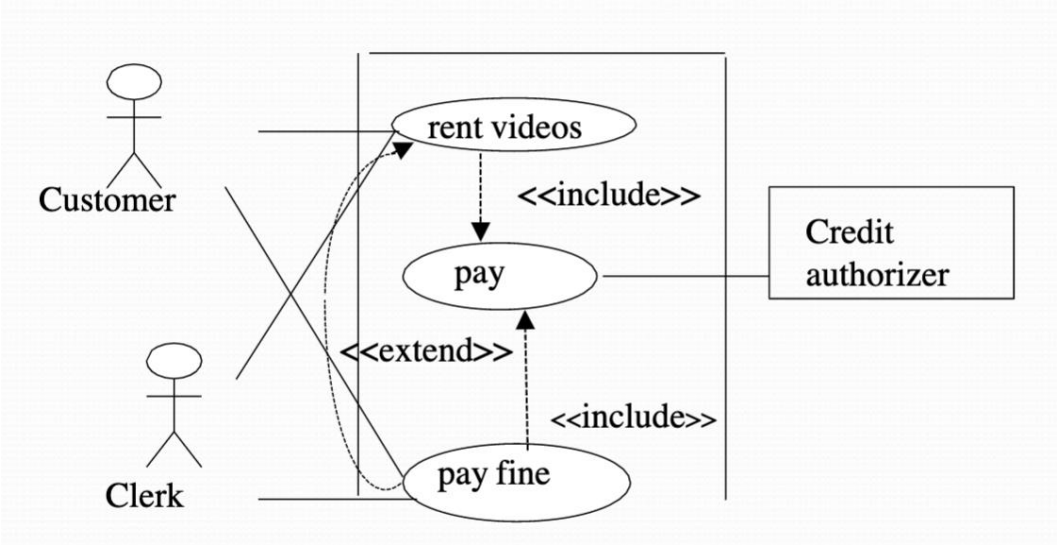
UML Use Case Modelling

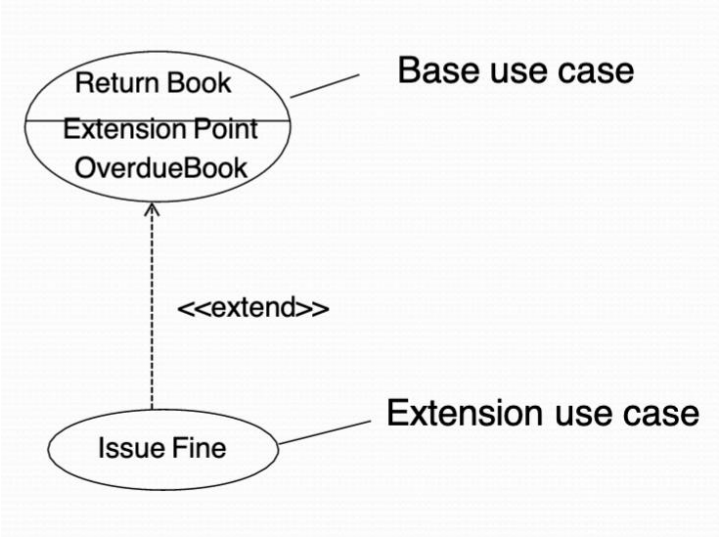
Relationships: Extend

- The extend relation is used to show that a set of actions sometimes occur in a base use case; the actions do not always have to be executed as part of that base use case.
- These extension actions are shown as separate use cases, with a dotted arrow pointing towards the primary use case labelled ‹‹extend››.
- Base use case does not do anything about extension use cases – it just provides hooks for them. In fact, the base use case is complete without extensions.

Extend Relation Examples

Video Stream



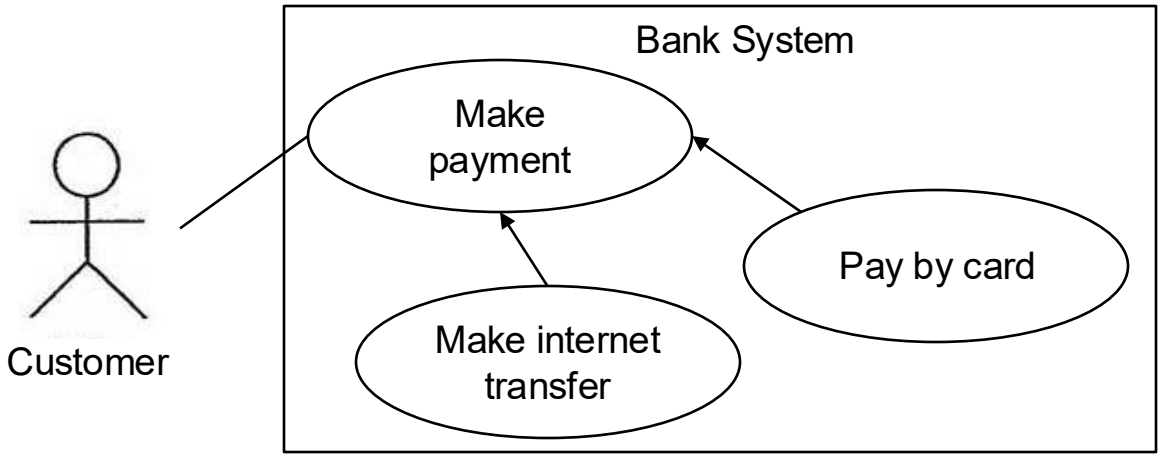


UML Use Case Modelling

Relationships: Generalization/Inheritance

- Generalization is used when two or more use cases have commonalities in behaviour, structure, and purpose.
- Describes the shared parts in a parent use case, that is then specialized by child use cases.
- The child use case may
    - Inherit features from their parent use case
    - Add new features
    - Change inherited features
- A solid arrow from the child use case pointing to the parent use case.

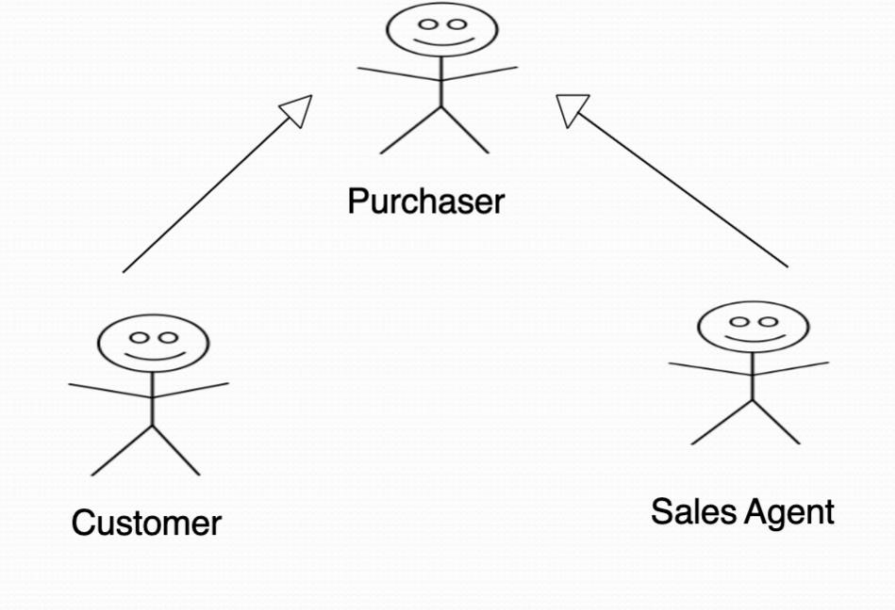
Generalization/Inheritance Relation Example



UML Use Case Modelling

Relationships: Actor Generalization

- Actor generalization factors out behaviour common to two or more actors into a parent actor.



Incremental Development of Use Case Model

- Not all requirements are shown in one iteration or model.
- It is common to work on varying scenarios or features of the same use case over several iterations and gradually extend the system to ultimately handle all the functionalities required.
- On the other hand, short, simple use cases on the core functionality may be completed within one iteration.

Tips for Modelling Use Cases

- Keep use cases short and simple
    - A good rule of thumb is to ensure that the main f low of a use case fits on a single side of paper
- Focus on what, not the how
    - Remember that you are writing use cases to work out what the actors need the system to do, not how the system should do it.
- Avoid functional decomposition
    - One common error in use case analysis is to create a set of high-level use cases and then break these into lower-level use cases.

UML Use Case Modelling

CRUD Operations:

A business system is built on four basic operations performed by a database:

- **C**reate a record, such as a customer record
- **R**etrieve the record given a (usually unique) key
- **U**pdate the record with new data and store it
- **D**elete a record

The operations are known as the CRUD operations. The business system offers the actors the CRUD capabilities. As a result of executing the operations data is stored or retrieved from storages (e.g., collections, files, databases, etc ...)

UML Use Case Modelling

CRUD Use Cases:

- Uses cases of the sort - Create an Item, Retrieve an Item, Update an Item, Delete an Item.
- In principle, they are separate because each is separate goal, possibly carried by a different person with a different security level.
- However, they clutter up the use case set and can triple the number of items to track.
- Trend is to just start with one use case, Manage an Item. If description becomes more complex break out the use case.

Summary

- Software development process and lifecycle (SDLC); Waterfall and Agile methodologies.
- Requirements analysis & validation; use cases and user stories.
- UML use case modelling.

References

- Introduction to Software Engineering, By Elvis C. Foster
- Chapter 3, Object-Oriented Systems Analysis and Design. Noushin Ashrafi and Hessam Ashrafi
- Object-Oriented Design with UML and Java, By Kenneth A. Barclay, and J. Savage.