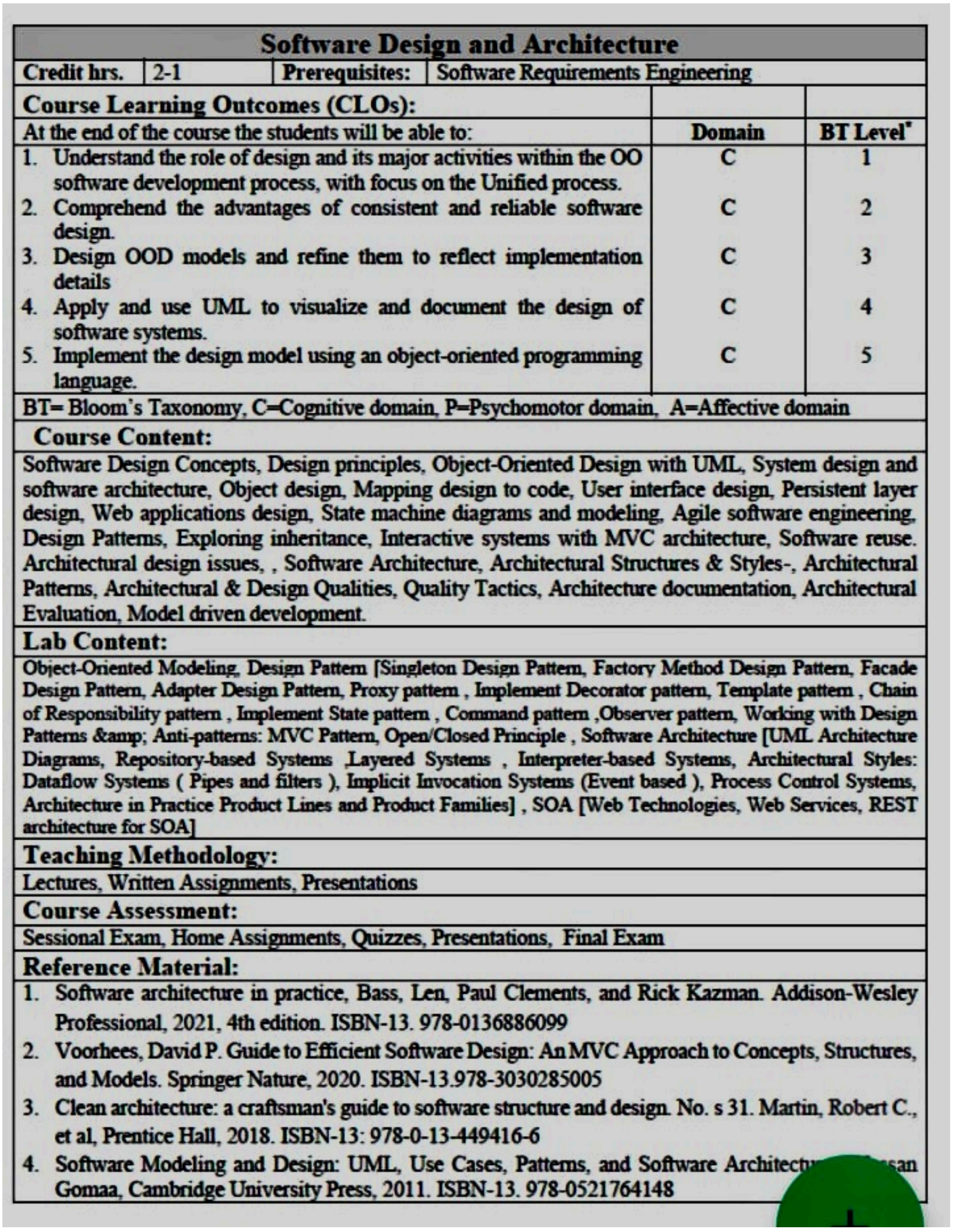
Software Design and Architecture

Spring 2024



Software Design and Architecture

Credit hrs.	2-1	**Prerequisites:**	Software Requirements Engineering

Course Learning Outcomes (CLOs):

At the end of the course the students will be able to:	**Domain**	**BT Level***
1. Understand the role of design and its major activities within the OO software development process, with focus on the Unified process.	C	1
2. Comprehend the advantages of consistent and reliable software design.	C	2
3. Design OOD models and refine them to reflect implementation details	C	3
4. Apply and use UML to visualize and document the design of software systems.	C	4
5. Implement the design model using an object-oriented programming language.	C	5

BT= Bloom's Taxonomy, C=Cognitive domain, P=Psychomotor domain, A=Affective domain

Course Content:

Software Design Concepts, Design principles, Object-Oriented Design with UML, System design and software architecture, Object design, Mapping design to code, User interface design, Persistent layer design, Web applications design, State machine diagrams and modeling, Agile software engineering, Design Patterns, Exploring inheritance, Interactive systems with MVC architecture, Software reuse. Architectural design issues, , Software Architecture, Architectural Structures & Styles-, Architectural Patterns, Architectural & Design Qualities, Quality Tactics, Architecture documentation, Architectural Evaluation, Model driven development.

Lab Content:

Object-Oriented Modeling, Design Pattern [Singleton Design Pattern, Factory Method Design Pattern, Facade Design Pattern, Adapter Design Pattern, Proxy pattern , Implement Decorator pattern, Template pattern , Chain of Responsibility pattern , Implement State pattern , Command pattern ,Observer pattern, Working with Design Patterns &amp; Anti-patterns: MVC Pattern, Open/Closed Principle , Software Architecture [UML Architecture Diagrams, Repository-based Systems ,Layered Systems , Interpreter-based Systems, Architectural Styles: Dataflow Systems (Pipes and filters), Implicit Invocation Systems (Event based), Process Control Systems, Architecture in Practice Product Lines and Product Families] , SOA [Web Technologies, Web Services, REST architecture for SOA]

Teaching Methodology:

Lectures, Written Assignments, Presentations

Course Assessment:

Sessional Exam, Home Assignments, Quizzes, Presentations, Final Exam

Reference Material:

1. Software architecture in practice, Bass, Len, Paul Clements, and Rick Kazman. Addison-Wesley Professional, 2021, 4th edition. ISBN-13. 978-0136886099
2. Voorhees, David P. Guide to Efficient Software Design: An MVC Approach to Concepts, Structures, and Models. Springer Nature, 2020. ISBN-13.978-3030285005
3. Clean architecture: a craftsman's guide to software structure and design. No. s 31. Martin, Robert C., et al, Prentice Hall, 2018. ISBN-13: 978-0-13-449416-6
4. Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectu[illegible]san Gomaa, Cambridge University Press, 2011. ISBN-13. 978-0521764148

1: Introduction to Software Design and Architecture

Define Software Design and Architecture

Software design and architecture refers to the process of defining a structured solution that meets all of the technical and operational requirements, while optimising common quality attributes such as performance, security, and manageability. The process entails planning the structure of software components, their interfaces, and the data flow between them.

Software design focuses on how the system will be constructed at the module level, while architecture provides a higher level overview, defining the overall structure and relationship between different parts of the system.

Importance in Software Development

Understanding software design and architecture is crucial in software development as it lays the groundwork for creating effective and scalable systems. Good design and architectural practices ensure that software is reliable, maintainable, and adaptable to change. They help in managing complexity by dividing the system into manageable components, thereby enabling developers to focus on specific areas without being overwhelmed by the entire system. Furthermore, well-defined architectures facilitate easier communication among team members and stakeholders, streamlining the development process.

2: Software Design Concepts

Definition and Objectives of Software Design

Software design involves creating a blueprint for the development of a software system, detailing the structure, components, interfaces, and data for the system to meet specified requirements. The primary objectives are to provide a solution that meets users' needs, is easy to maintain, and scalable with minimal cost. Effective design minimises complexity, making the software easier to understand and modify.

Role of Abstraction, Encapsulation, Modularisation

- **Abstraction** simplifies complexity by hiding unnecessary details, allowing designers to focus on high-level concepts.
- **Encapsulation** bundles data and methods that operate on the data within one unit, hiding internal details from outside interference and misuse.
- **Modularisation** divides the software into smaller, manageable pieces (modules) with specific responsibilities, improving maintainability and reusability.

High-Level vs. Low-Level Design

- **High-Level Design (HLD)** focuses on the system architecture and the relationships between the major components.
- **Low-Level Design (LLD)** details the individual components and their functions, including algorithms and data structures.

3: Key Software Design Models

Top-Down vs. Bottom-Up Approaches

- **Top-Down Design** starts with the general system architecture and decomposes it into more detailed parts.
- **Bottom-Up Design** begins with detailed components or subsystems, which are then combined into a larger system.

Software Design- Building Methods

- **Structured Design** focuses on process and data structures, using a top-down approach.
- **Object-Oriented Design (OOD)** centres around objects that are instances of classes, emphasising reusability and encapsulation.
- **Component-Based Design** involves assembling pre-existing components to create new applications.

4: Software Design Principles - SOLID (Part 1)

Single Responsibility Principle (SRP)

A class should have only one reason to change, meaning it should have only one job or responsibility. This simplifies maintenance and enhances cohesion within the class.

Open/Closed Principle (OCP)

Software entities (classes, modules, functions) should be open for extension but closed for modification. This allows systems to grow without modifying existing code, reducing the risk of bugs.

Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of subclasses without affecting the correctness of the program. This ensures that a subclass can stand in for its superclass.

5: Software Design Principles - SOLID (Part 2)

Interface Segregation Principle (ISP)

Clients should not be forced to depend upon interfaces they do not use. This principle leads to the creation of specific interfaces rather than one general-purpose interface, improving system modularity.

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules, but both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions. This reduces the coupling between high-level and low-level modules.

6: Additional Design Principles

DRY (Don't Repeat Yourself)

This principle emphasizes the importance of avoiding repetition of software patterns, stating that every piece of knowledge must have a single, unambiguous, authoritative representation within a system. Applying DRY helps reduce redundancy, making code easier to maintain, understand, and reduce errors.

KISS (Keep It Simple, Stupid)

KISS advocates for simplicity in design, suggesting that systems work best if they are kept simple rather than made complex. Therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.

YAGNI (You Aren't Gonna Need It)

YAGNI is a principle of extreme programming that states you should not add functionality until it is necessary. It's a reminder to developers not to add functionality based on speculation of future use, preventing over-engineering.

7: Introduction to Software Architecture

Definition and Significance

Software architecture involves making fundamental structural choices that are costly to change once implemented. It's about setting the high-level structure of software and the discipline of creating such structures. It plays a critical role in determining a system's quality, performance, scalability, and maintainability.

Architecture vs. Design

While both architecture and design are about structuring software applications, architecture focuses on the high-level structure of the major components and their interactions, whereas design concentrates on the lower-level aspects, such as the detailed organization of the system and its components.

Key Concerns in Software Architecture

Key architectural concerns include scalability, reliability, security, and performance. These aspects need to be addressed early in the design process to meet the system's requirements and stakeholders' expectations effectively.

8: Architectural Styles and Patterns

Layered Architecture

A common architectural style that organizes software into layers, each with a specific responsibility. The most typical is the three-layered architecture consisting of presentation, business logic, and data access layers.

Client-Server and Peer-to-Peer

- **Client-Server**: A model where multiple clients request and receive services from a centralized server.
- **Peer-to-Peer (P2P)**: A decentralized model where each node in the network acts both as a client and a server.

Service-Oriented Architecture (SOA) and Microservices

- **SOA**: An architectural pattern in which application components provide services to other components via a communications protocol, typically over a network.
- **Micro-services**: An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms.

9: Object-Oriented Design (OOD) Basics

Principles of OOD: Encapsulation, Inheritance, Polymorphism

- **Encapsulation**: The bundling of data with the methods that operate on that data. It restricts direct access to some of an object's components, which can prevent the accidental modification of data.
- **Inheritance**: A mechanism wherein a new class is derived from an existing class. The new class inherits attributes and behavior (methods) from the parent class, allowing for reuse and extension of code.

- **Polymorphism**: The ability of different classes to respond to the same message (method call) in different ways. This allows for implementing abstract interfaces that can perform different actions based on the object's actual class.

10: Object-Oriented Design Process

Identifying Objects and Classes

The process begins with identifying the objects necessary for the system, based on the requirements. Objects represent entities in the real world, and classes define the structure and behavior of these objects.

Defining Relationships and Hierarchies

After identifying objects and classes, the next step is to define the relationships (such as associations, aggregations, and compositions) between them and organize the classes into hierarchies using inheritance, where appropriate.

Designing Class Interfaces and Interactions

The final step involves designing the interfaces for the classes (the methods through which objects of the class will communicate with the outside world) and the interactions between objects to achieve the functionality required by the system.

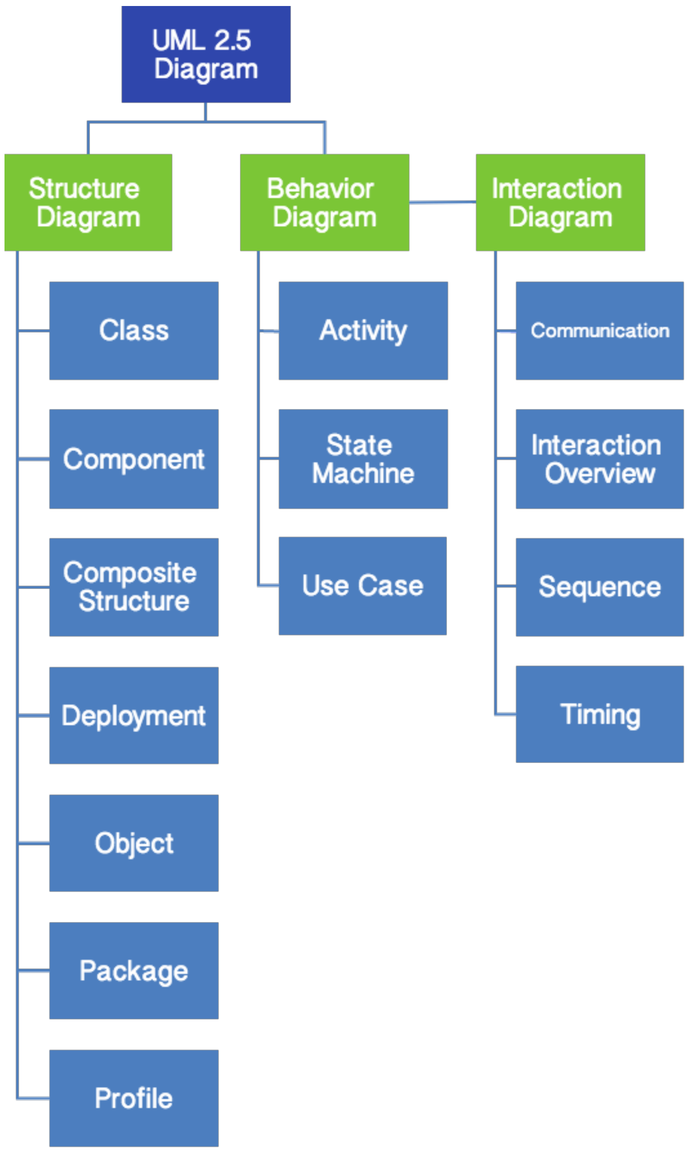
11: Unified Modelling Language (UML) Overview

Importance of UML in OOD

UML is a standardised modelling language that provides a way to visualise a system's architectural blueprints, including aspects like activities, processes, logical structures, and system interactions. It's crucial for OOD because it offers a universal language that architects, developers, and stakeholders can understand.

Types of UML Diagrams

UML includes several types of diagrams, divided into **structure diagrams** (like class diagrams, package diagrams) that show the static aspects of the system, and **behaviour diagrams** (like use case diagrams, activity diagrams) that show the dynamic aspects.



12: UML: Structure Diagrams

There are seven structure diagrams in UML 2.5:

1. **Class diagrams** show the structure of a system as related classes and interfaces with their features, constraints, and relationships.
2. **Component diagram**s show components and the dependencies between them.
3. **Composite structure diagrams** show the internal structure of a classifier and the behavior of collaborations the structure makes possible.
4. **Deployment diagrams** show a system's various hardware and the software deployed on it.
5. **Object diagrams** show a real-world example of a structure at a specific time.
6. **Package diagrams** show packages and the dependencies between those packages.
7. **Profile diagrams** show custom stereotypes, tagged values, and constraints.

13: UML: Behaviour Diagrams

There are seven behaviour diagrams, the last four of which fall under the interaction diagram subset:

1. **Activity diagrams** show business or operational workflows of components in a system.
2. **Use Case diagrams** show how functionalities relate under particular actors.
3. **State machine diagrams** show the states and state transitions of a system.

Interaction Diagrams

4. **Communication diagrams** show the interactions between objects in terms of sequenced messages.
5. **Interaction overview diagrams** show an overview of the flow of control with nodes that represent interactions or interactions uses.
6. **Sequence diagrams** show how objects communicate and the sequence of their messages.
7. **Timing diagrams** show timing constraints of a system in a given time frame.

14: UML: Commonly Used Diagrams

Activity diagrams

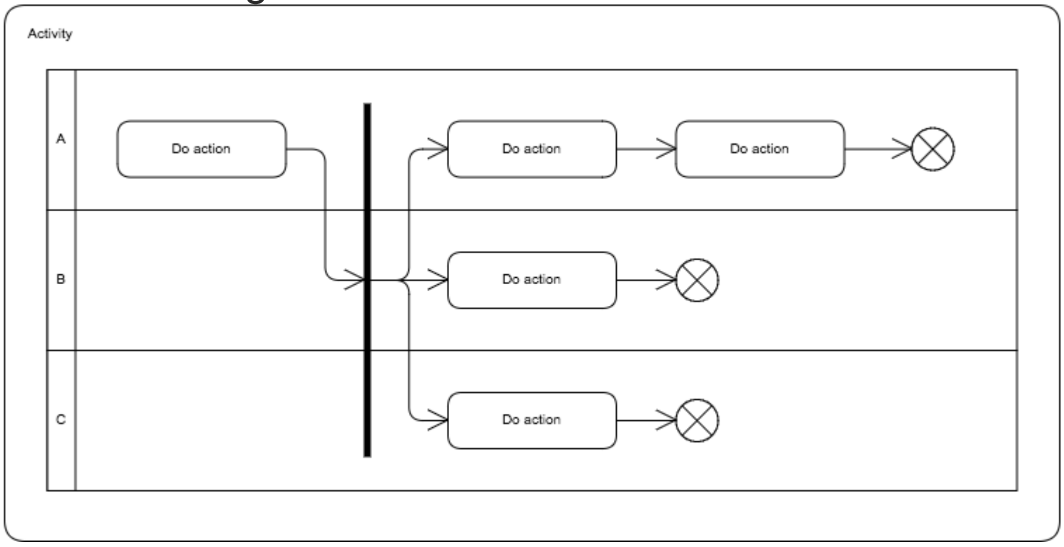
An activity diagram is exactly what it sounds like — a diagram that creates a visual depiction of an activity. The activity diagram itself can hold any amount of information detailing a wide variety of actions. If you can think of a workflow, you can diagram it out.

Using words and symbols, you can map out workflows to include the order in which tasks and operations need to be done, who needs to do them, what tasks can only be done once others are completed, which tasks are free-standing, and more.

Actions are tasks performed by a user, the system, or both in collaboration.

Connectors link the actions in sequence.

Nodes indicate the start or end of an activity. They can also indicate a fork or merge.



Class diagrams

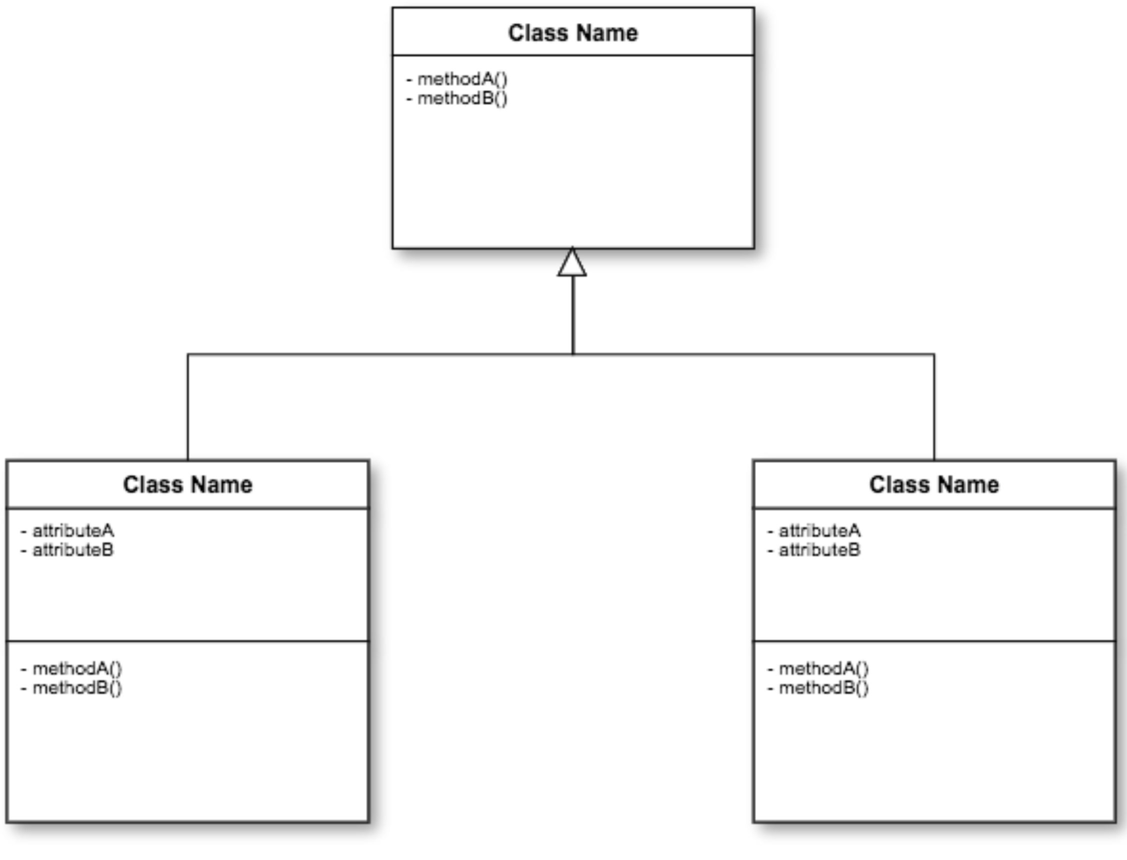
Class diagrams are a subsection of Structural UML diagrams and function as the most basic building tool to create applications.

It is most widely used to depict OOPs content, more efficient app design and analysis, and as the base for the deployment and component diagram.

Classes represent data or object types. They are visualised using a rectangular shape with the class name as the top section.

Attributes are the named values that every instance of a type can have. They are listed under the class name.

Methods are the functions that instances of a type can perform. They are listed below attributes.



Sequence diagrams

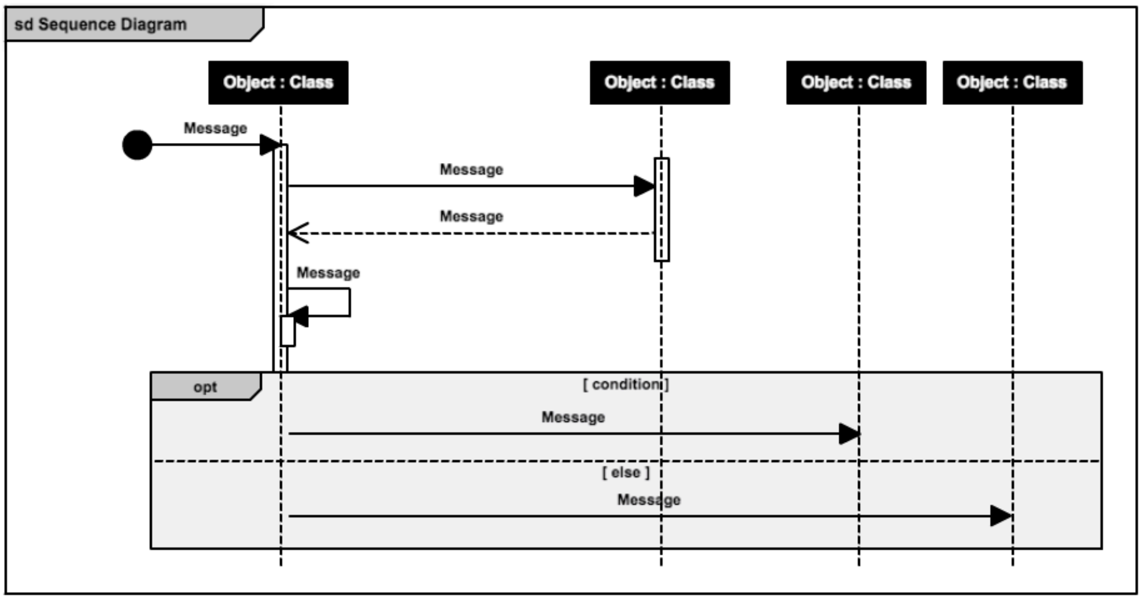
The difference between a sequence diagram and other types is that the sequence diagram depicts the actions in more detail. You can easily see how they are implemented, by whom, in what order, what needs to be completed beforehand, and what can be done after.

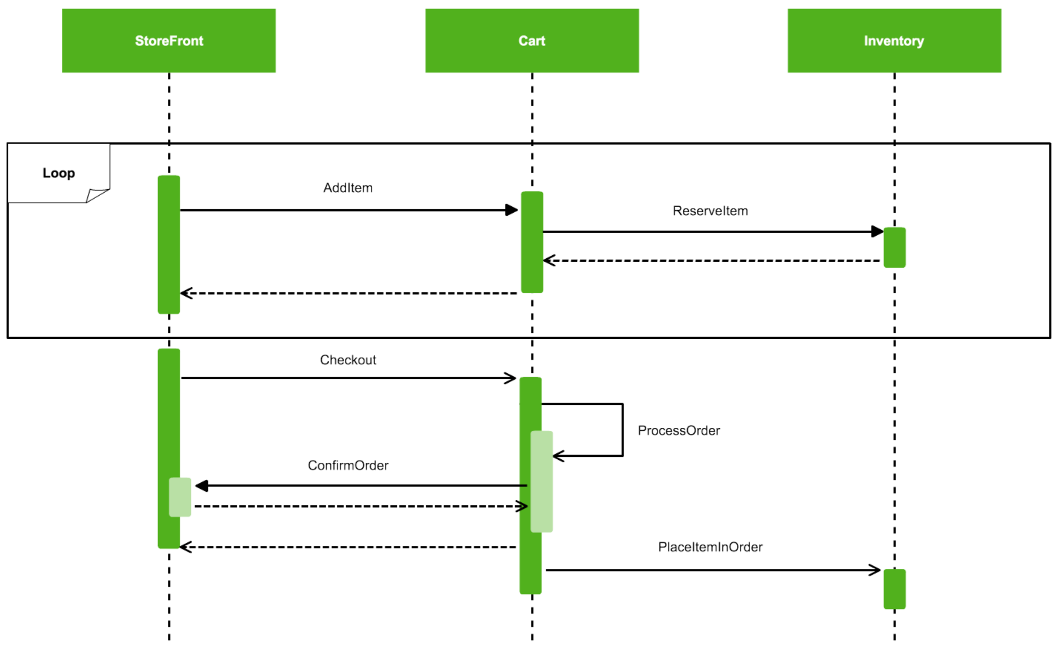
At a higher level, a sequence diagram can be thought of how the process moves forward over time, including the order of actions. It therefore also shows the interaction between multiple actions and the passage of time and completion of past tasks moves the process forward.

Classes represent data or object types. They are visualised using a rectangular shape.

Lifelines are vertical lines that represent the sequence of events that occur to a participant as time progresses. This participant can be an instance of a class, component, or actor.

Messages are represented by lines between objects.





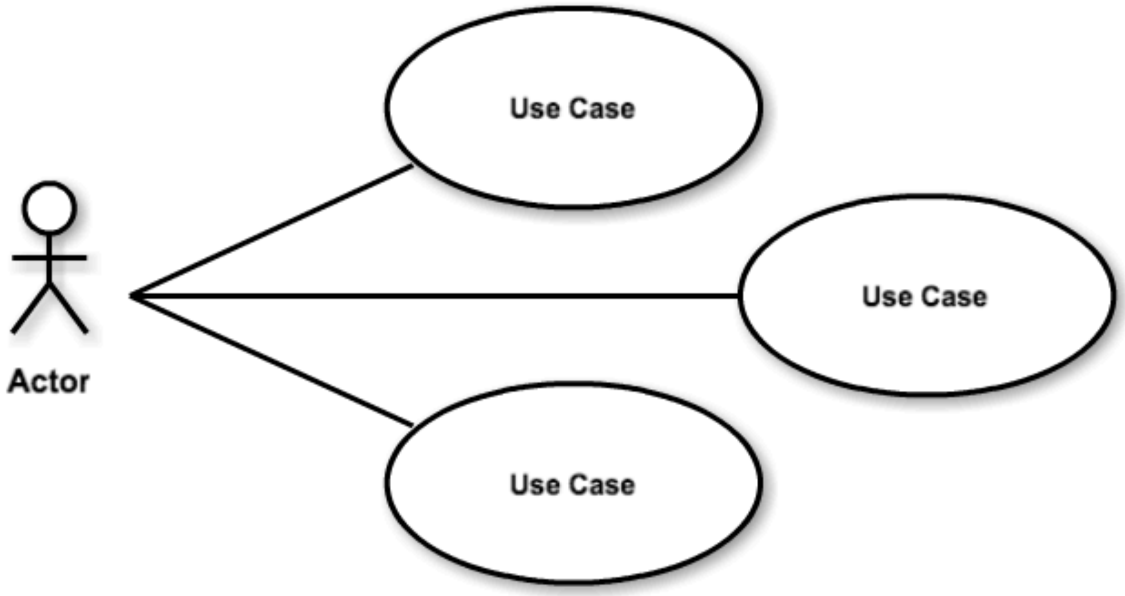
Use case diagrams

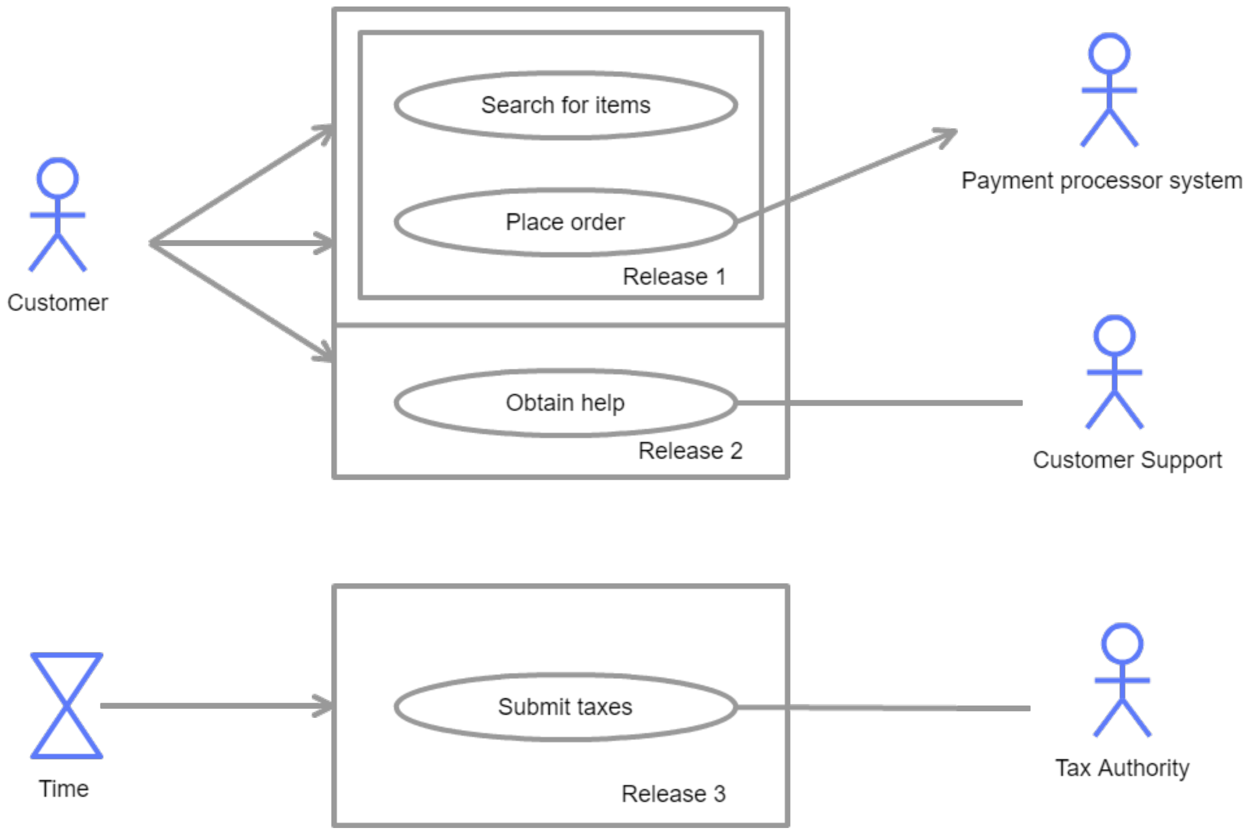
Use case diagrams help to communicate what the end result of an underdeveloped application WILL be. It’s extremely useful in meeting with a client and creating the idea of the functions, letting the developers work backward from there. Being as this diagram focuses mainly on functionality and end results, it shows much more of WHAT the app will do without very much explanation of HOW the app will perform these individual functions.

Actors represent users, organizations, or external systems that interact with your application or system. An actor is a kind of type.

Use Cases represent the actions performed by one or more actors in the pursuit of a particular goal. A use case is a kind of type.

Associations indicate where an actor takes part in a use case.




Customer
Search for items
Place order
Release 1
Payment processor system
Obtain help
Release 2
Customer Support
Time
Submit taxes
Release 3
Tax Authority

Version	Date	Description
1.1	11-1997	UML 1.1 proposal is adopted by the OMG.
1.3	03-2000	Contains a number of changes to the UML metamodel, semantics, and notation, but should be considered a minor upgrade to the original proposal.
1.4	09-2001	Mostly "tuning" release but not completely upward compatible with the UML 1.3. Addition of **profiles** as UML extensions grouped together. Updated **visibility** of features. Stick arrowhead in **interaction diagrams** now denotes **asynchronous call**. Model element may now have multiple **stereotypes**. Clarified collaborations. Refined definitions of components and related concepts. **Artifact** was added to represent physical representations of components.
1.5	03-2003	Added **actions** (see Part 5 of spec) - executable actions and procedures, including their run-time semantics, defined the concept of a data flow to carry data between actions, etc.
1.4.2	01-2005	This version was accepted as ISO specification (standard) ISO/IEC 19501. UML 1.5 was released 2 years before.

2.0	08-2005	New diagrams: object diagrams, **package diagrams**, **composite structure diagrams**, interaction overview diagrams, timing diagrams, **profile diagrams**. **Collaboration diagrams** were renamed to **communication diagrams**. **Activity diagrams** and **sequence diagrams** were enhanced. Activities were redesigned to use a Petri-like semantics. Edges can now be contained in partitions. Partitions can be hierarchical and multidimensional. Explicitly modeled **object flows** are new. Classes have been extended with internal structures and **ports** (composite structures). Information flows were added. A collaboration now is a kind of classifier, and can have any kind of behavioral descriptions associated. Interactions are now contained within classifiers and not only within collaborations. It is now possible for **use cases** to be owned by **classifiers** in general and not just packages. New notation for concurrency and branching using combined fragments. Notation and/or semantics were updated for components, realization, deployments of artifacts. Components can no longer be directly deployed to **nodes**. **Artifacts** should be deployed instead. Implementation has been replaced by «**manifest**». Artifacts can now manifest any packageable element (not just components, as before). It is now possible to deploy to nodes with an internal structure. New metaclasses were added: connector, collaboration use, connector end, **device**, deployment specification, **execution environment**, accept event action, send object action, structural feature action, value pin, activity final, central buffer node, data stores, flow final, interruptible regions, loop nodes, parameter, **port**, behavior, behaviored classifier, duration, interval, time constraint, combined fragment, creation event, destruction event, execution event, interaction fragment, interaction use, receive signal event, send signal event, extension, etc. Many stereotypes were eliminated from the Standard UML Profile, e.g. «destroy», «facade», «friend», «profile», «requirement», «table», «thread». Integration between structural and behavioral models was improved with better support for executable models.
2.1	04-2006	Minor revision to UML 2.0 - corrections and consistency improvements.
2.1.1	02-2007	Minor revision to the UML 2.1
2.1.2	11-2007	Minor revision to the UML 2.1.1
2.2	02-2009	Fixed numerous minor consistency problems and added clarifications to UML 2.1.2
2.3	05-2010	Minor revision to the UML 2.2, clarified **associations** and association classes, added **final classifier**, updated **component diagrams**, composite structures, actions, etc.

2.4.1	08-2011	UML revision with few fixes and updates to classes, packages - added **URI package attribute**; updated actions; removed creation event, execution event, send and receive operation events, send and receive signal events, renamed destruction event to **destruction occurrence specification**; **profiles** - changed stereotypes and applied stereotypes to have upper-case first letter - **«Metaclass»** and **stereotype application**.
2.5	06-2015	UML 2.5 is called a "minor revision" to the UML 2.4.1, while they spent a lot of efforts to simplify and reorganize UML specification document. The UML specification was re-written "*to make it easier to read*". For example, they tried to "*reduce forward references as much as possible*". There are no longer two separate infrastructure and superstructure documents, the UML 2.5 specification is a single document. **Package merge** is no longer used within the specification. Four UML compliance levels (L0, L1, L2, and L3) are eliminated, as they were not useful in practice. UML 2.5 tools will have to support complete UML specification. **Information flows**, **models**, and **templates** are no longer auxiliary UML constructs. At the same time, **use cases**, **deployments**, and the **information flows** became "**supplementary concepts**" in UML 2.5. UML 2.5 has a number of fixes, clarifications, and explanations added. They updated description for multiplicity and multiplicity element, clarified definitions of aggregation and composition, and finally fixed wrong «instantiate» dependency example for Car Factory. New notation for **inherited members** with a caret '^' symbol was introduced. UML 2.5 clarified **feature redefinition** and overloading. They also moved and rephrased definition of qualifiers. Default for **generalization sets** was changed from **{incomplete, disjoint}** to **{incomplete, overlapping}**. There are few clarifications and fixes for stereotypes, state machines, and activities. Protocol state machines are now denoted using «protocol» instead of {protocol}. **Use cases** are no longer required to express some needs of **actors** and to be initiated by an actor.