"Owning a hammer doesn't make one an architect"

Knowing an object-oriented language (such as Java) is a necessary but insufficient first step to create object systems.

Knowing how to "think in objects" is also critical.

Knowing UML helps you to communicate with others in creating software, but the real work in this course is learning Object-Oriented Analysis and Design, not only how to draw diagrams.

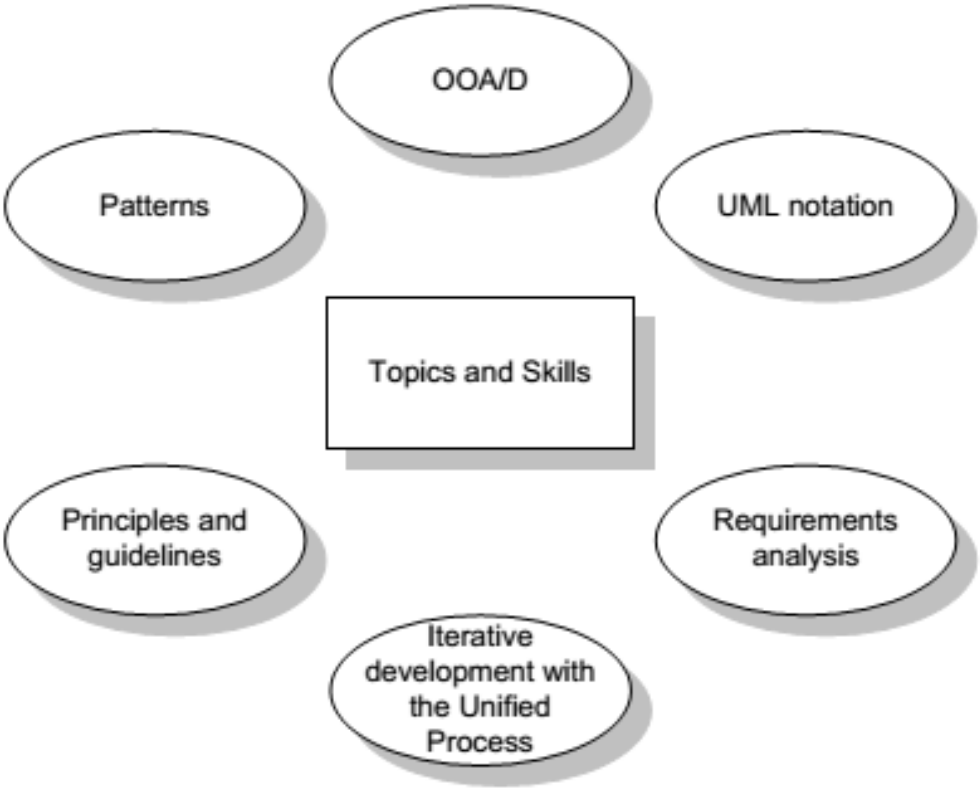


Figure 1.1 Topics and skills covered

Analysis and Design

Analysis emphasizes an investigation of the problem and requirements, rather than a solution. For example, if a new online trading system is desired, Analysis answers the following questions:

How will it be used?

What are its functions?

"Analysis" is a broad term, and it is referred as requirements analysis (an investigation of the requirements) or object-oriented analysis (an investigation of the domain objects).

Design emphasizes a conceptual solution (in software and hardware) that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Design ideas often exclude low-level or "obvious" details obvious to the intended consumers. Ultimately, designs can be implemented, and the implementation (such as code) expresses the true and complete realized design. As with analysis, the term is best qualified, as in object-oriented design or database design.

Useful analysis and design have been summarized in the phrase **do the right thing (analysis)**, and **do the things right (design).**

Structured analysis and design technique (SADT) is a systems engineering and software engineering methodology for describing systems as a hierarchy of functions.

Key Differences Between Structured and Object-Oriented Analysis and Design

	Structured	Object-Oriented
Methodology	SDLC	Iterative/Incremental
Focus	Processs	Objects
Risk	High	Low
Reuse	Low	High
Maturity	Mature and widespread	Emerging (1997)
Suitable for	Well-defined projects with stable user requirements	Risky large projects with changing user requirements

5

Object Oriented Concepts

In case of an Object Oriented System, the overall architecture of the system is perceived as a collection of objects that are assigned specific responsibilities. These objects exhibit their behavior based on the type of responsibility assigned to them. Thus the main goal of the system is achieved thru the collaboration of objects that exist in the system.

The steps in designing an Object Oriented System comprises of mainly

i. **Identification of objects** that exist in a domain

ii. **Assigning responsibilities** to these objects so that each object can exhibit the desired behavior

iii. **Seek collaboration** between these objects for fulfilling the goal of the system (computation as a simulation)

Responsibility Driven Design

Just like in our daily lives we get to see different entities fulfilling their goals independently, in case of an object oriented system, objects exhibit behavior based on the kind of responsibility that has been assigned to it. We expect a lamp to glow whenever we pass it a message to glow by switching the lamp on and we do not expect a lamp to rotate or make sound.

Decomposition

It is the process of partitioning the problem domain into smaller parts so that the overall complexity of the problem can be comprehended and tackled easily (divide and conquer approach)

- Algorithmic/Functional decomposition

  Dividing the problem domain into smaller parts by focusing on the functionalities that the system provides. For example functionalities of a Library Management System may include issue_book, return_book, search_book

- Object Oriented Decomposition

  Dividing the problem domain into smaller parts by focusing on the objects with specific responsibilities that form the system. For example, objects in a Library Management System may include IssueManager, Book, Librarian

In an object oriented system, the overall goal of the system is decomposed into sub goals and then responsibilities are assigned to specific objects.

So the overall functionality of the system is achieved thru the collaboration of these objects.

Abstraction

Abstractions are created by hiding the details and representing only the necessary features.

Forming abstractions help in communicating ideas with others and in representing complexity in a much simplified form.

Abstractions can be formed of objects, ideas or functionalities.

- moving towards the door
- grabbing its handle
- pulling it towards you

So if you want to ask someone to open the door you do not need to relate to him the details regarding how it is done every time

A person who is a human being , who has a name and id, who works in a library, who can issue, return and stock book is represented by an abstraction called Librarian.

It means to filter out an objects property and operations until just the one needed are left

Object-oriented analysis

It is method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain

A thorough investigation of the problem domain is done during this phase by asking the **WHAT** questions (rather than how a solution is achieved)

During OO analysis, there is an **emphasis on finding and describing the objects** (or concepts) in the problem domain.

For example, concepts in a Library Information System include Book, and Catalog.

Object-oriented design

It is method of design encompassing the process of the object oriented decomposition and notation for depicting both logical and physical as well as static and dynamic models of the system under design

There are two important parts of this definition

It leads to object oriented decomposition

Uses different notations to express different models of the logical (class and object) structure and physical (module and cross architecture) design of a system in the addition to the static and dynamic aspects of a system

Emphasizes a conceptual solution that fulfils the requirements.

There is a need to define software objects and how they collaborate to fulfill the requirements.

For example, in the Library Information System, a Book software object may have a title attribute and a display() method. The issueManager may be asked to issue a book, in response this object may ask the BookRecord object to update the status of the book.

Designs are implemented in a programming language.

OOA	OOD
Elaborate a problem	Provide conceptual solution
WHAT type of questions asked	HOW type of questions asked
During Analysis phase questions asked include Q. What is required in the Library Information System? A. Authentication!!	Later during Design phase questions asked include Q. How is Authentication in the Library Information System achieved? A. Thru Smart Card!! Or Fingerprint!!

Object Oriented Programming

It is a method of implementation in which programs are organized as co operative collection of objects each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united via inheritance relationship.

Three important aspects of definition

- Uses objects not algorithms as fundamental building blocks
- Each object is an instance of some class
- Classes are related to one another via inheritance

Characteristics of OOD

Objects are abstractions of real-world or system entities and manage themselves

Objects are independent and encapsulate state and representation information.

System functionality is expressed in terms of object services

Shared data areas are eliminated. Objects communicate by message passing

Objects may be distributed and may execute sequentially or in parallel

Advantages of OOD

Easier maintenance. Objects may be understood as stand-alone entities

Objects are appropriate reusable components

For some systems, there may be an obvious mapping from real world entities to system objects

Object-oriented development

Object-oriented analysis, design and programming are related but distinct

OOA is concerned with developing an object model of the application domain

OOD is concerned with developing object-oriented system model to implement requirements

OOP is concerned with realizing an OOD using an OO programming language such as C++

Object-oriented design methods

Some methods which were originally based on functions (such as the Yourdon method) have been adapted to object-oriented design.

Other methods such as the Booch method have been developed specifically for OOD

OOD is an object-oriented design method developed to support Ada programming.

OO Design method commonality

The identification of objects, their attributes and services

The organization of objects into an aggregation hierarchy

The construction of dynamic object-use descriptions which show how services are used

The specification of object interfaces

Objects

An object is an entity which has a state and a defined set of operations which operate on that state.

The state is represented as a set of object attributes.

The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.

Objects are created according to some object **class definition**. An object class definition **serves as a template** for objects. It includes **declarations of all the attributes and services** which should be associated with an object of that class.

Objects are entities in a software system, that represent instances of real-world and system entities

Object classes are templates for objects. They may be used to create objects

Object classes may inherit attributes and services from other object classes

Assigning Responsibilities

A critical, fundamental ability in OOA/D is to skillfully **assign responsibilities** to software components

It strongly influences the robustness, maintainability, and reusability of software components.

In an object oriented system every object is capable of accomplishing a specific goal and it does so by fulfilling the responsibilities assigned to it.

e.g. a dice object can come up with a random face value by fulfilling the responsibility it has been assigned that is to roll()

e.g. a book object can reveal its name by fulfilling the responsibility it has been assigned that is to getBookInfo().

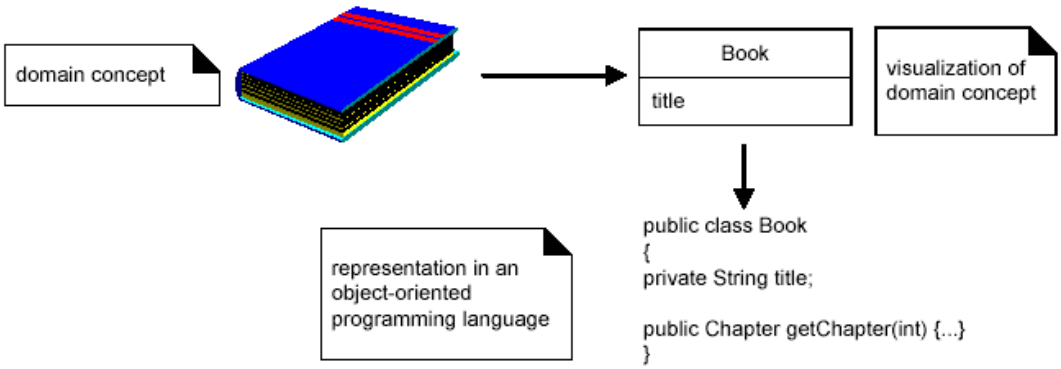
Thus in short,

During **object-oriented analysis,** there is an emphasis on finding and describing the objects—or concepts—in the problem domain. For example, in the case of the library information system, some of the concepts include *Book, Library,* and *Patron.*

During **object-oriented design,** there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a *Book* software object may have a *title* attribute and a *getChapter* method

Finally, during implementation or object-oriented programming, design objects are implemented, such as a *Book* class in Java.

Object-orientation emphasizes representation of objects.



An Example

Before diving into the details of iterative development, requirement analysis, UML, and OOA/D here we see the birds-eye view of steps and diagrams involved in a "dice game" in which a player rolls two die. If the total is seven, they win; otherwise, they lose.

i. Define Use Cases

Requirements analysis may include a description of related domain processes; these can be written as **use cases.**

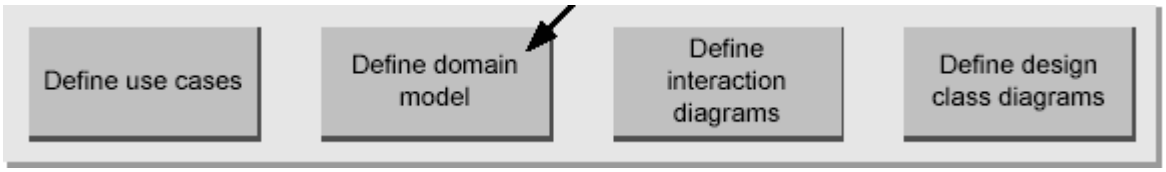
Use cases are not an object-oriented artifact—they are simply written stories.

However, they are a popular tool in requirements analysis and are an important part of the Unified Process. For example, here is a brief version of the *Play aDice Game* use case:

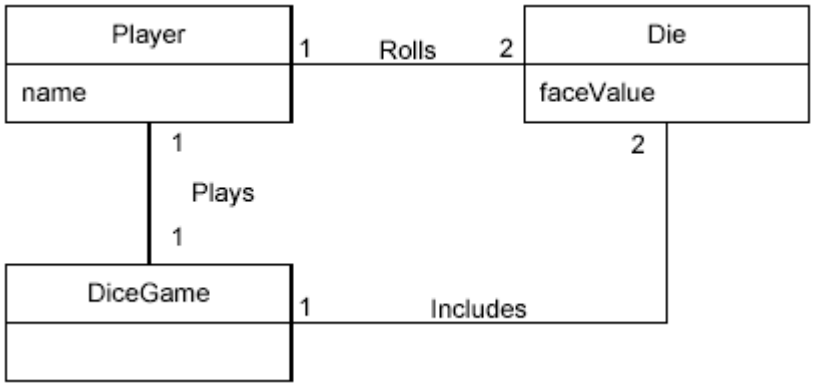
Play a Dice Game: A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose.

ii. Define a Domain Model

Object-oriented analysis is concerned with creating a description of the domain from the perspective of classification by objects. A decomposition of the domain involves an identification of the concepts, attributes, and associations that are considered noteworthy. The result can be expressed in a **domain model,** which is illustrated in a set of diagrams that show domain concepts or objects.



Note that a domain model is not a description of software objects; it is a visualization of concepts in the real-world domain.

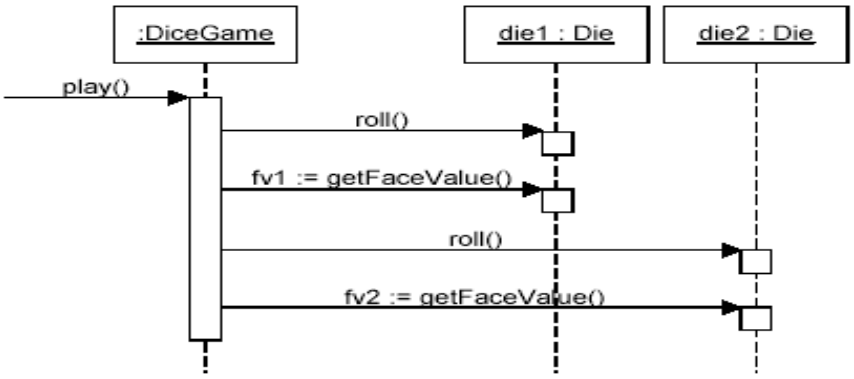


iii. Define Interaction Diagrams

Object-oriented design is concerned with defining software objects and their collaborations.

A common notation to illustrate these collaborations is the **interaction diagram.** It shows the flow of messages between software objects, and thus the invocation of methods.

For example, assume that a software implementation of the dice game is desired. The interaction diagram in figure illustrates the essential step of playing, by sending messages to instances of the *DiceGame* and *Die* classes.

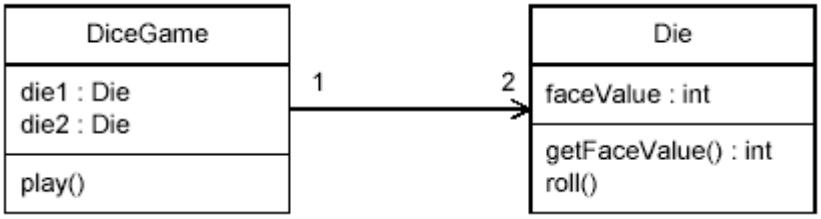


Although in the real world a *player* rolls the dice, **in the software design the *DiceGame* object "rolls" the dice** (that is, sends messages to *Die* objects). Software object designs and programs do take some inspiration from real-world domains, but they are *not* direct models or simulations of the real world.

iv. Define Design Class Diagrams

In addition to a *dynamic* view of collaborating objects shown in interaction diagrams, it is useful to create a *static* view of the class definitions with a **design class diagram.** This illustrates the attributes and methods of the classes.

For example, in the dice game, an inspection of the interaction diagram leads to the partial design class diagram . Since a *play* message is sent to a *DiceGame* object, the *DiceGame* class requires a *play* method, while class *Die* requires a *roll* and *getFaceValue* method.



In contrast to the domain model, this diagram does not illustrate real-world concepts; rather, it shows software classes.

Requirements Process

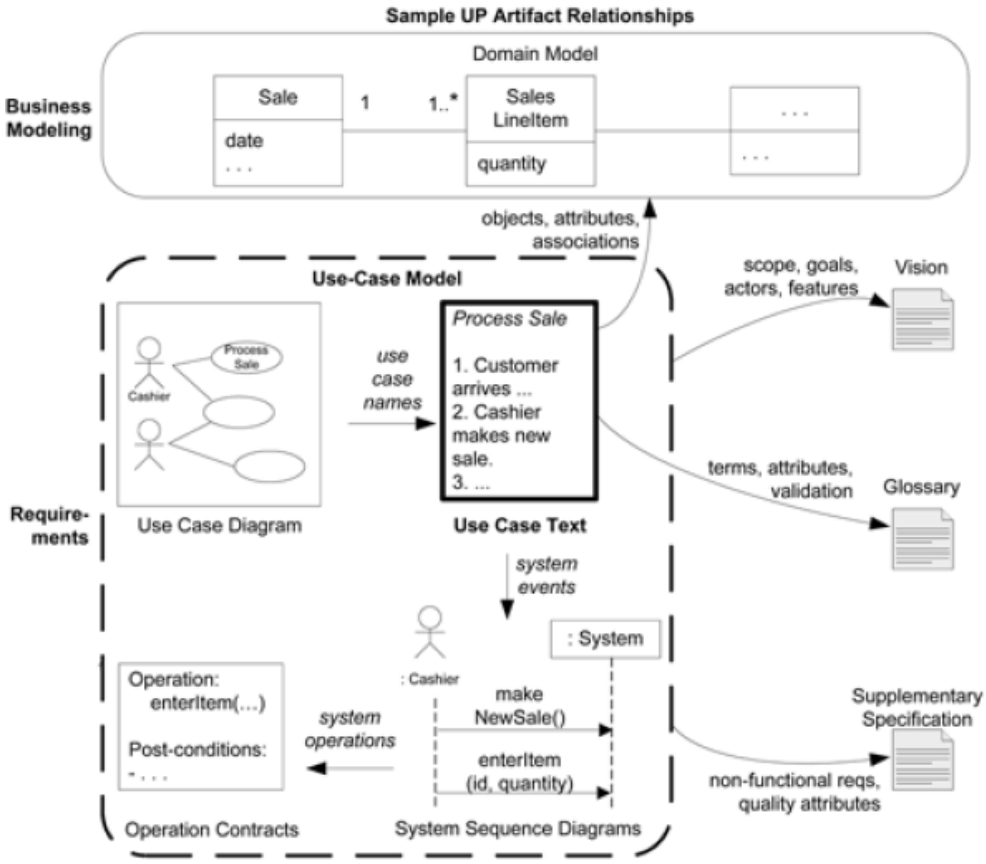
Use Cases: Describing Processes

Writing use cases, **stories of using a system**, is an excellent technique to **understand and describe requirements**. Informally, they are **stories of using a system** to meet goals.

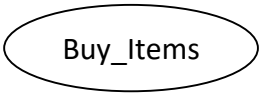
The essence is **discovering and recording functional requirements by writing stories** of using a system to help fulfill various stakeholder goals

"Use cases are **narrative description** of domain processes"

Use cases are dependent on having at least partial understanding of the system, ideally **expressed in a requirements specification document.**



Definition: - It is a **narrative document** that describes the sequence of events of an actor (an external agent) using a system to complete a process.



THREE USE CASE FORMATS

- Brief (High Level)
- Casual
- Fully Dressed

1. **Brief**—terse one-paragraph summary, usually of the main success scenario.

It describes a process **very briefly**, usually in two or three sentences.

It is useful to create this type of use case **during the initial requirements and project scoping** in order quickly understand the degree of complexity and functionally in a system.

They are very **terse and vague** on design decisions.

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items

It is useful to start with high level use cases to quickly obtain some understanding of overall major processes.

2. **Casual**—informal paragraph format. Multiple paragraphs that cover various scenarios.

Handle Returns

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...

Alternate Scenarios:

If the credit authorization is reject, inform the customer and ask for an alternate payment method.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external tax calculator system, ...

3. **Fully dressed**—the most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

These detailed use cases are written after many uses cases have been identified and written in a brief format

Template:

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Use Case UC1: Process Sale
Scope: NextGen POS Application
Level: User goal
Primary Actor: Cashier
Stakeholders and Interests:
Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer short ages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.

Preconditions: Cashier is identified and authenticated.
Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated.
Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.
Main Success Scenario (or Basic Flow):
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.

Extensions (or Alternative Flows):

*a. At any time, System fails:

To support recovery and correct accounting, ensure all transaction sensitive

state and events can be recovered from any step of the scenario.

1. Cashier restarts System, logs in, and requests recovery of prior state.

2. System reconstructs prior state.

2a. System detects anomalies preventing recovery:

1. System signals error to the Cashier, records the error, and enters a clean state.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

- What are the tax law variations?

- Explore the remote service recovery issue.

- What customization is needed for different businesses?

- Must a cashier take their cash drawer when they log out?

- Can the customer directly use the card reader, or does the cashier have to do it?

Preconditions state what must always be true before beginning a scenario in the use case.

A precondition implies a scenario of another use case that has successfully completed, such as logging in, or "cashier is identified and authenticated".

Success guarantees (or postconditions) state what must be true on successful completion of the use case. either the main success scenario or some alternate path.

The guarantee should meet the needs of all stakeholders.

Sale is saved. Tax is correctly calculated.

Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Extensions (or Alternate Flows) indicate all the other scenarios or branches, both success and failure. They are also known as "Alternative Flows."

Special Requirements

If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case. These include qualities such as performance, reliability, and usability, and design constraints (often in I/O devices) that have been mandated or considered likely.

A Two Column Format : Typical Course of Events

Actor Action	**System Response**
Numbered action of the actor	Numbered description of system response

Alternative Course of events describes important alternatives or exceptions that may arise with respect to typical course.

Actor Action	**System Response**
1. The use case begins when a Customer arrives at a POST checkout with Items to purchase	
2. The Cashier records the identifier from each item	3. Determines the item price and adds the item information to the running sales transaction.
If there are more than one of the same type of item, the cashier can enter the quantity as well.	The description and price of the current item are presented.
4. On completion of item entry the Cashier indicates to the POST that item entry is complete.	5. Calculates and presents the sales total.

6. The Cashier tells the Customer the Total 7. The customer gives cash payment- the "cash tendered "- possibly greater than the sale total. 8. The Cashier records the cash received amount	9. Shows the balance due back to the Customer. Generates a receipt…
10. The Cashier deposits the cash received and extract the balance owing 12. The customer leaves with the items purchased	11. Logs the completed sale.

Alternative Course

Line 2: Invalid identifier entered. Indicate error

Line 7: Customer didn't have enough cash. Cancel sales transaction

Actor

It is an **entity that is external** to the system, who in some way participates in the story of Use Case.

An Actor typically stimulates the system with input events, or receives something form it. Actors are represented by the role they play in the Use Case.

Actors can include

- Roles that people play
- Computer System
- Electrical or Mechanical Devices

Types of Actors

Primary actors have user goals fulfilled thru using services of system under discussion. e.g. cashier . These are identified to find user goals that drive use cases

Supporting actors provide service or information to the system. e.g. automated payment authorization. These are identified to clarify external interfaces and protocols

Offstage actor has interest in the behavior of the use case . e.g. government tax agency. These are identified to ensure all interests are identified and satisfied

Scenario

A scenario is a specific sequence of actions and interactions between actors and the system under discussion; it is also called a use case instance.

A use case is a collection of related success and failure scenarios that describes an actor using a system to support a goal

Use cases are functional or behavioral requirements that indicate what the system will do

Common Mistakes with the Use Cases

Remember: A **use case is a relatively large end to end process description** that typically **includes many steps** or transactions; it is not normally an individual step or activity in a process. e.g printing a receipt; is not a use case but a step in a use case called Buy_Item

Finding Primary Actors, Goals, and Use Cases

1. Choose the **system boundary**.

   Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?

2. Identify the **primary actors**.

   Those that have user goals fulfilled through using services of the system.

3. For each, identify their **user goals**.

   Raise them to the highest user goal level that satisfies the EBP guideline.

4. Define **use cases** that satisfy user goals; **name them** according to their goal.

   Usually, user goal-level use cases will be one-to-one with user goals.

System and their boundaries:

A use case describes interaction with the system. Typical boundaries includes

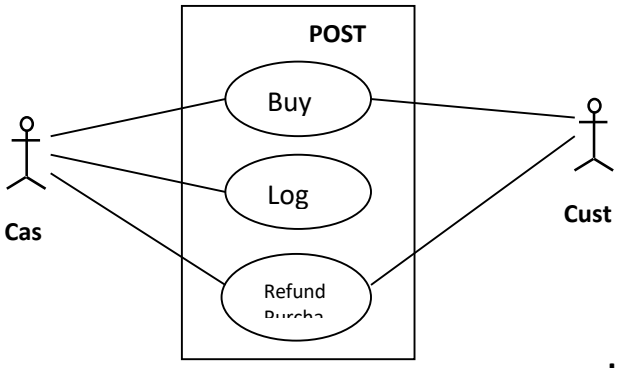
- Hardware/software boundary of a device or computer system.
- Department of an organization.
- Entire organization.

The system boundary is important as it identifies:

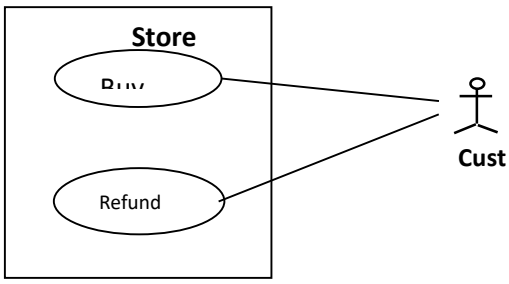
- what is external versus internal.
- the responsibilities of the system

Scenario one:

If we choose the entire store or business as the system then the only customer is an actor not the cashier because the cashier acts as a resource within the business system.



Use case and actors when the POST



Use case and actors when the Store is the

Scenario two:

If we choose the point of sale hardware and software as the system both cashier as well as customer may be treated as actors.

Identifying the Use Cases

Method 1 (Actor based)

Identifying the actor related to a system or organization and then for each actor, identifying the processes they initiate or participate in.

Using actor goal list

A recommended procedure:

1. Find the user goals.
2. Define a use case for each.

Actor	Goal
Cashier	process sales, process rentals, handle return, cash in , cash out
Manager	start up , shut down
System Admin	add users, modify users, delete users, manage security, manage system tables
Sales Activity Sys.	analyze sales and performance data
Customer	Buy Items, Refund Items

Method 2 (External events based)

Identifying the external events that a system must respond to and then relating the events to actor and use cases.

External Event	Actor	Goal/use case
Enter sale line item	Cashier	process a sale
Enter payment	Cashier or customer	process a sale

Naming Use Case and Domain Processes

A use case describes a process (Business process) , it is a complete run of a scenario e.g. all of the events that occur from the time a customer enters the ATM station till the time he comes out with cash with one goal achieved that includes inserting card, validation etc.

A process describes, from start to finish, a sequence of events, actions and transactions required to produce or complete something of value to an organization or actor

Process e.g. Withdraw Cash form ATM, Order a Product

A use case name starts with a verb. So naming a uses case based on goal:

Goal: Process a sale Use case: ProcessSale

Use Cases, System Functions and Traceability

Identified system functions in requirements specification should be allocated to Use Cases

Using Cross reference of the Use Cases, all functions that have been allocated should be verified, this helps in traceability between the artifacts

Essential versus Real use cases:

Essential use cases

- created during **early stage** of eliciting the requirements
- independent of the design decisions
- very abstract

High level use cases are always essential in nature due to their brevity (concise) and abstractions.

It is desirable to create essential use cases during only requirements, elicitations in order to more fully understand the scope of the problem and the functions required.

Essential use case

Actor action	**System response**
1. customer identifies themselves	2. Provide Customer with options
3. ……	

Real use cases

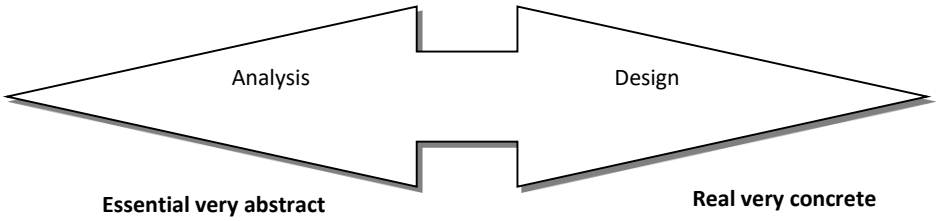
- Describe the functionality of the system in terms of its **actual current design** committed to specific input output technologies
- Developed only after the design decisions have been made
- Design artifacts
- very concrete

This concretely describes the process in terms in its rare current design committed to specific input and output technologies and so on.

Real use case

Actor action	**System response**
1. customer inserts their cards	2. Prompts for PIN
3. enters PIN on key pad	4. Displays option menu

Use Case Degree of Design Commitment



Buy Items use case

Essential

Actor action	**System response**
1. Cashier **records identifier** for each item	2. Determine the item price and add the item info
If there is more than one of the same	to the running sales transaction description
The cashier enters quantity	price of the current item in item presented

Real

Actor action	**System response**
1. For each item cashier types **UPC**	2. Display item price and add the item information
in the UPC input field of window then	to the running sales transactions the description
They **press "enter item" button**	and price of current item are displayed
with the mouse or keyboard	

Notational Points: Naming Use Cases

- Name a use case starting with a verb to emphasize that it is a verb

  e.g. Buy_Items, Enter_Order

- Starting off an Expanded Use Case, use the Following Schema

This use case begins when an <Actor> <initiates an event>

This use case begins when a customer arrives at a POST with items to purchase

Notating Decision Points and Branching

A Use case may contain decision points. For e.g. A customer may choose to pay via cash, credit, or check.

If all alternatives are relatively equal, the notational structure is as follows

Section: Main

Typical course of Events

Actor Action	System Response
1.This use case begins when a Customer arrives at POST to check out with items to purchase	
2.Intermediate steps	
3.Customer chooses payment type a) If cash payment, Pay by Cash b) If credit payment, Pay by Credit c) If check payment, Pay by Check	
	4. Logs the completes sales
	5. Prints a receipt
6. The cashier gives the receipt to the customer.	
7. The customer leaves …	

Section: Pay by Cash

Typical Course of Events

Actor Action	System Response
1. The Customer gives cash payment possibly greater than the sale total to the cashier.	
2. The Cashier records the cash amount	
	3. Shows the balance due back to the Customer
4. The Cashier deposits the cash received and extracts the balance owing.	
The Cashier gives balance owing to the customer.	

Alternative Courses:

Line 4: Insufficient balance in the drawer to pay balance, ask for cash from supervisor, or ask amount closer to sale total from the Customer

Use Cases within a Development Process

Plan and Elaborate Phase steps

After the System Functions have been listed, define the System boundary and then Identify Actors and the Use Cases

Write down all the Use Cases in High Level Format and categorize them as Primary and Secondary or optional

Draw a Use Case diagram

- Relate Use Cases
- Write the most critical, influential and risky use cases in the expanded format to better understand and estimate the size and nature of the problem.
- Real use case should be deferred (delayed) as they involve design decisions
- Rank Use cases

Process Steps for the Point of Sale System

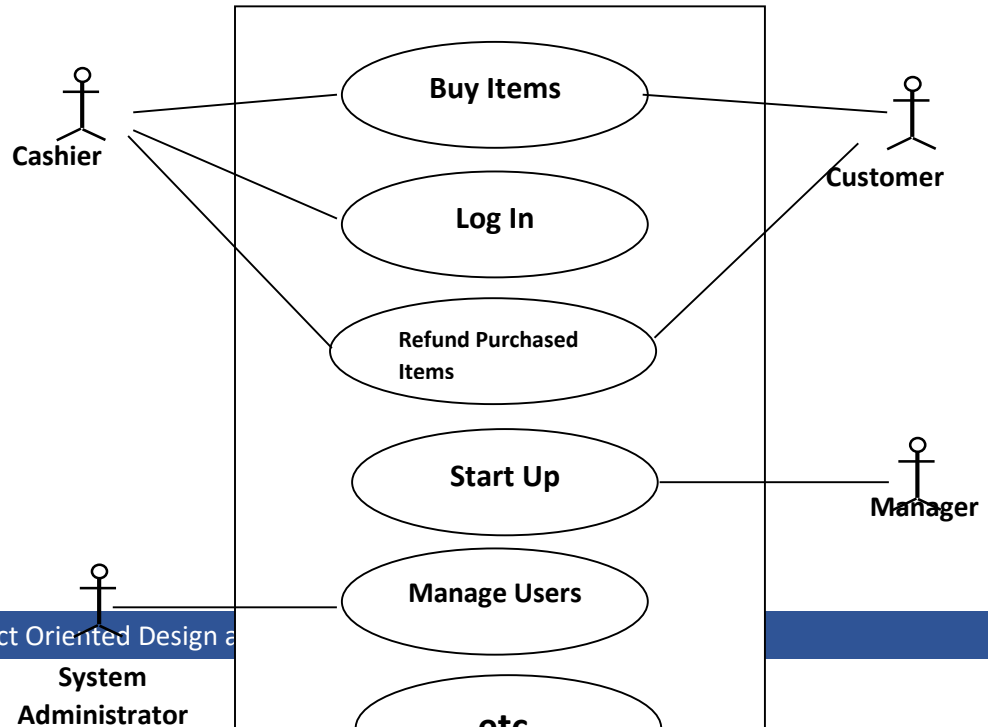
Identifying Actors and Use Cases

The system boundary is defined as Software and Hardware system---the Usual Case

A list of relevant Actors include

Cashier	Log In/Cash Out
Customer	Buy Items/Refund Items
Manager	Start Up/Shut Down
System Administrator	Add New User

Draw a Use Case Diagram



Partial Use Case Diagram for the POST Application

Relate Use Case

Write Some Expanded Essential Use Cases

These Include Buy Items and Refund Purchased Items

Writing these in expanded form gives clear and better understanding

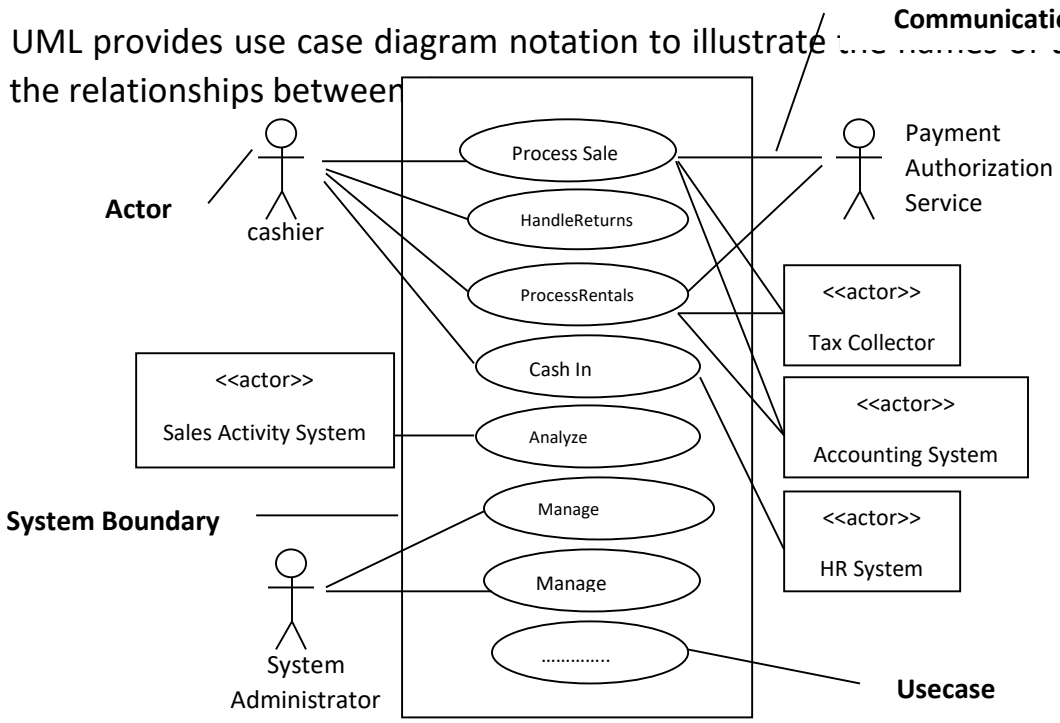
Use Case: Buy Items

Section Main

Use Case:	**Buy Items**
Actors:	**Customer (initiator), Cashier**
Purpose	**Capture a sale and its payment**
Overview	**A Customer arrives at a check out with Items to purchase. The cashier records the purchased items and collects a payment. On completion the customer leaves with the items**
Type	**primary**
Cross Reference	**R1.1, R1.2, R1.3, R1.7, R1.9, R2.1, R2.2, R2.3, R2.4**

Use Case Diagrams

The UML provides use case diagram notation to illustrate the names of use cases can actors, and the relationships between



Object Oriented Development Cycle

Waterfall Model:

The Waterfall Model was first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed fully before the next phase can begin. This type of software development model is basically used for the for the project which is small and there are no uncertain requirements. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. In this model software testing starts only after the development is complete. In waterfall model phases do not overlap.

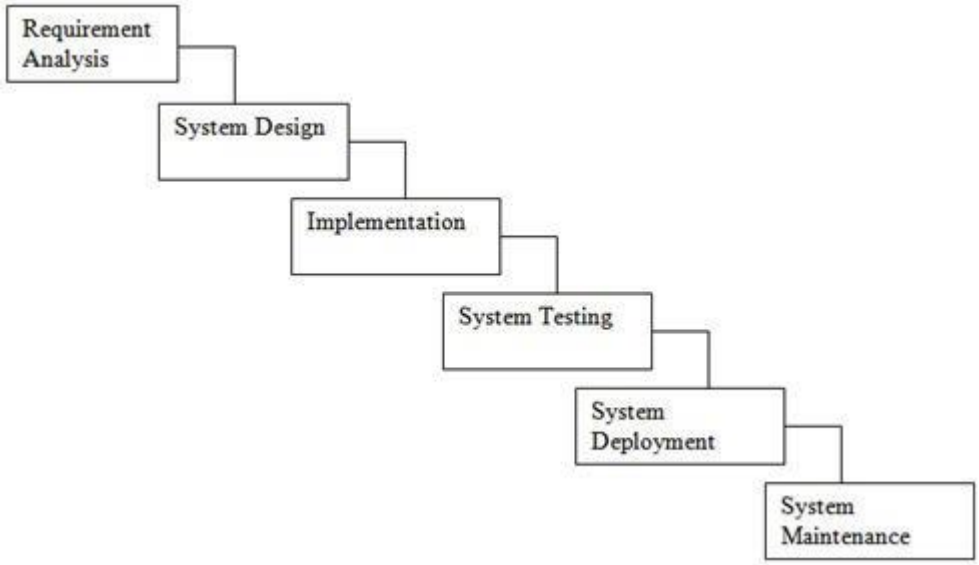


Figure : Waterfall Model

Disadvantages of waterfall model:

- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.

Why Waterfall is so failure prone?

There isn't one simple answer to why the waterfall is so failure-prone, but it is strongly related to a key false assumption underlying many failed software projects that the specifications are predictable and stable and can be correctly defined at the start, with low change rates. This turns out to be far from accurate and a costly misunderstanding. A study by Boehm and Papaccio showed that a typical software project experienced a 25% change in requirements. And this trend was corroborated in another major study of thousands of software projects, with change rates that go even higher35% to 50% for large projects as illustrated in Figure below.

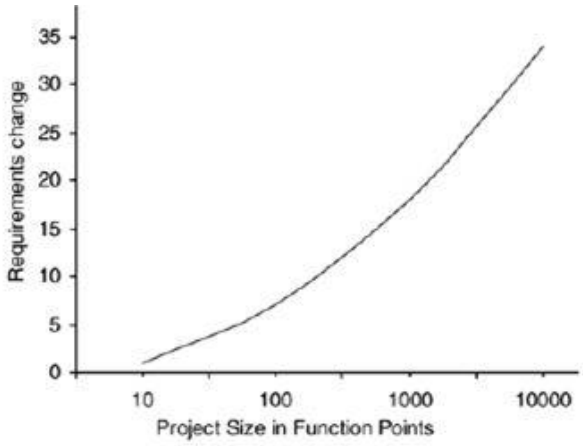


Figure: Percentage of change on software projects of varying sizes.

Iterative and Evolutionary (Incremental) Development

In this lifecycle approach, development is organized into a series of short, fixed-length (for example, three-week) mini-projects called iterations; the outcome of each is a tested, integrated, and executable partial system. Each iteration includes its own requirements analysis, design, implementation, and testing activities.

The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation as core drivers to converge upon a suitable system. The system grows incrementally over time, iteration by iteration, and thus this approach is also known as iterative and incremental development (see Figure below). Because feedback and adaptation evolve the specifications and design, it is also known as iterative and evolutionary development.

Early iterative process ideas were known as spiral development and evolutionary Development [Boehm]

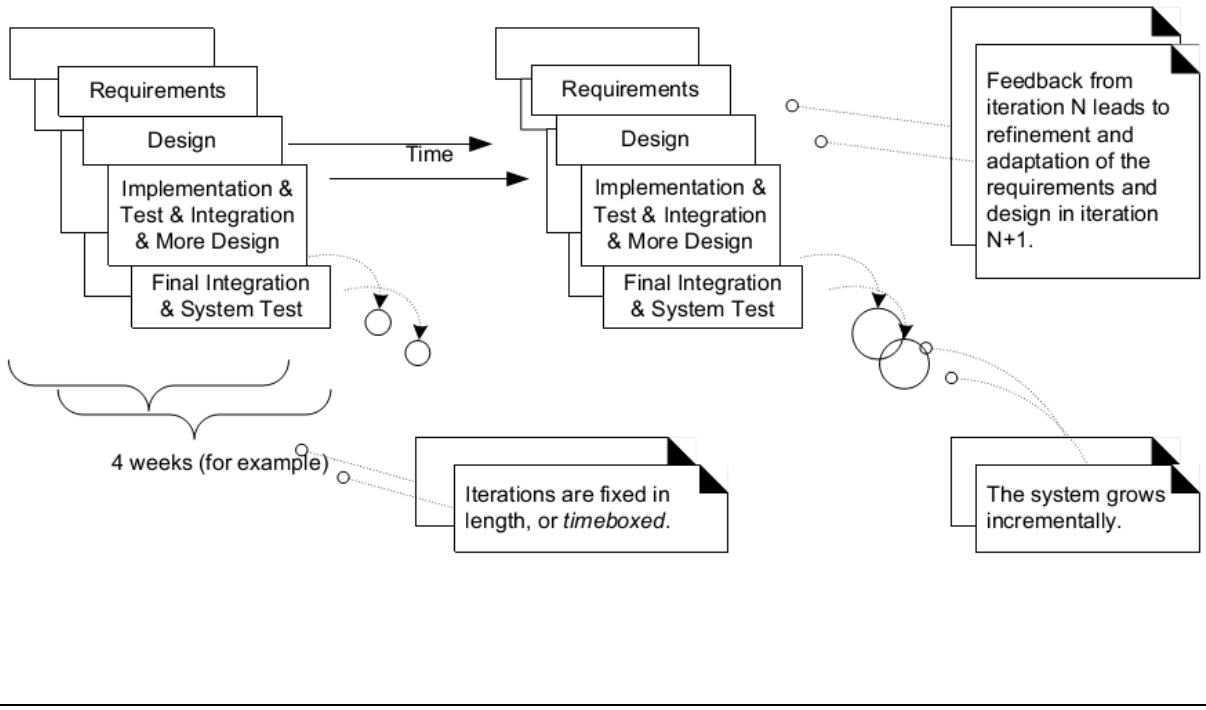


Figure : Iterative and evolutionary development.

In complex, changing systems (such as most software projects) feedback and adaptation are key ingredients for success.

- Feedback from early development, programmers trying to read specifications, and client demos to refine the requirements.
- Feedback from tests and developers to refine the design or models.
- Feedback from the progress of the team tackling early features to refine the schedule and estimates.
- Feedback from the client and marketplace to re-prioritize the features to tackle in the next iteration

Benefits include:

- less project failure, better productivity, and lower defect rates; shown by research into iterative and evolutionary methods
- early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth) early visible progress
- early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
- managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps
- the learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

The Unified Process

Until recently, three of the most successful object-oriented methodologies were

– Booch's method

– Jacobson's Objectory

– Rumbaugh's OMT (Object Modeling Technique)

• Today, the unified process is usually the primary choice for object-oriented software production. That is, the unified process is the primary object-oriented methodology

In 1999, Booch, Jacobson, and Rumbaugh published a complete object-oriented analysis and design methodology, which is called **Unified Process**. It unified their three separate methodologies

– **Original name**: Rational Unified Process (RUP)

– Next name: Unified Software Development Process (USDP)

– **Name used today**: Unified Process (for brevity)

- The Unified Process is **not a specific series of steps** for constructing a software product since
  – There is a wide variety of different types of software products
  – No such single "one size fits all" methodology could exist
- The Unified Process is an **adaptable methodology**
  – It has to **be modified for the specific software product** to be developed
- The Unified Process is a **modeling technique**
  – A model is a set of UML diagrams that represent various aspects of the software product to be developed
- UML stands for unified **modeling language**
  – UML is the tool that we use to represent (model) the target software product
  – UML is graphical since a picture is worth a thousand words
- UML diagrams enable software engineers to **communicate more quickly and accurately** than if only verbal descriptions were used.
- The object-oriented paradigm is iterative and incremental in nature

  – Each workflow consists of a number of steps, and to carry out that workflow, the steps of the workflow are repeatedly performed until the members of the development team are satisfied that they have an accurate UML model of the software product they want to develop.

  – There is no alternative to repeated iteration and incrementation until the UML diagrams are satisfactory

Core Workflows of the Unified Process

i. The Requirements Workflow

- The aim of the requirements workflow is to **determine the client's needs**
  - **Gain an understanding** of the application domain, that is, the specific business environment in which the target software product is to operate
    - Build a **business model**
- Use UML to **describe the client's business processes**
- If at any time the client does not feel that the cost is justified, development terminates immediately
- The preliminary investigation of the client's needs is called **concept exploration**
- It is vital to determine exactly what the client needs and to find out the client's constraints
  - **Deadline**
  - **Reliability**
  - **Cost**
- The client will rarely inform the developer how much money is available
- A bidding procedure is used instead
  - Parallel running, Portability, and Rapid response time

ii. The Analysis Workflow

- The aim of the analysis workflow is to **analyze and refine the requirements** to achieve the detailed understanding of the requirements essential for developing a software product correctly and maintaining it easily.
- Why not do this during the requirements workflow?
  - The requirements artifacts must be totally comprehensible by the client
  - The artifacts of the requirements workflow must therefore be expressed in a natural (human) language
- All **natural languages are imprecise**! An example from a manufacturing information system:
  - "A part record and a plant record are read from the database. If it contains the letter A directly followed by the letter Q, then calculate the cost of transporting that part to that plant"
  - To what does it refer?
- Two separate workflows are needed
- The requirements artifacts must be expressed **in the language of the client**
- The analysis artifacts must be precise and complete enough **for the designers**

iii. The Design Workflow

- The aim of the design workflow is to refine the analysis workflow until the **material is in a form that can be implemented by the programmers**
  – Many **nonfunctional requirements need to be finalized** at this time, including
- **Choice** of programming language
- **Reuse** issues
- **Portability** issues
- Retain design decisions

– To backtrack and redesign certain pieces when a dead-end is reached

– To prevent the maintenance team reinventing the wheel

iv. The Implementation Workflow

- The aim of the implementation workflow is to **implement the target software product** in the selected implementation language

– A large software product is partitioned into smaller subsystems, which are implemented in parallel by coding teams

– The subsystems consist of components or code artifacts implemented by an individual programmer

v. The Test Workflow

- The test workflow is the **responsibility** of

– Every **developer** and **maintainer**, and

– The **quality assurance** group

- Traceability of artifacts is an important requirement for successful testing

And finally**, Postdelivery Maintenance**

Postdelivery maintenance is an essential component of software development

– More money is spent on postdelivery maintenance than on all other activities combined

• Problems can be caused by

– Lack of documentation of all kinds

• Two types of testing are needed

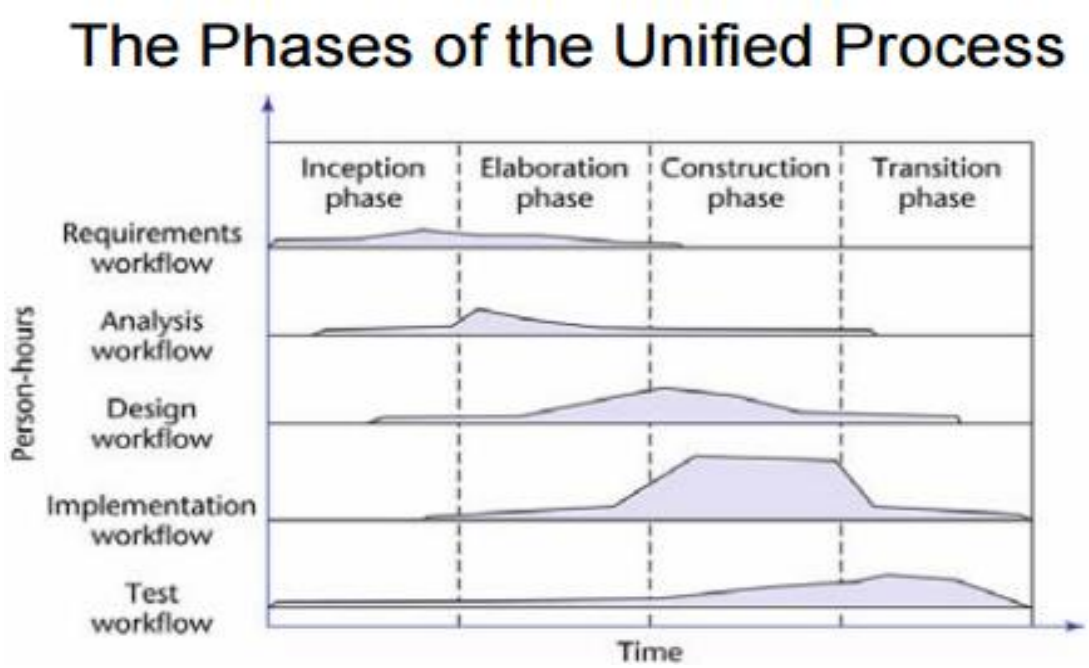
– Testing the changes made during postdelivery

maintenance

– Regression testing:

- Make sure that the functionality of the rest of the product has not been compromised.

• All previous test cases (and their expected outcomes) need to be retained



The four increments are labeled as:

– Inception phase

– Elaboration phase

– Construction phase

– Transition phase

• The phases of the Unified Process correspond to **increments**

• In theory, there could be any number of increments

– In practice, development seems to consist of four increments

• Every step performed in the Unified Process falls into

– One of the five core workflows and also

– One of the four phases

• Why does each step have to be considered twice?

• Workflow

– Technical context of a step

• Phase

– Business (i.e., economic) context of a step

• Example: One step to determine the client's needs is to build a business model.

• Building a business model should be presented within both technical and economic contexts.

i. The Inception Phase

The aim of the inception phase is to determine **whether the proposed software product is economically viable.**

Steps for this phase:

1. Gain an **understanding of the domain**
2. Build the **business model**
   - Understand precisely how the client organization

operates in the domain

3. **Delimit the scope** of the proposed project
   - Focus on the subset of the business model that is covered by the proposed software product
4. Begin to make the **initial business case**

The Inception Phase: Requirements Workflow -- The Initial Business Case (Sample Questions)

- Is the proposed software product cost effective?
  - Have the necessary marketing studies been performed?
  - How long will it take to obtain a return on investment?
  - What will be the cost not to develop the product?
- Can the proposed software product be delivered in time?
  - What will be the impact if the product is delivered late?
- What are the risks involved in developing the software product, and how can these risks be mitigated?
  - Does the team have the necessary experience?
  - Is new hardware needed for this software product?
    - If so, is there a risk that it will not be delivered in time?
    - If so, is there a way to mitigate that risk, perhaps by ordering back-up hardware from another supplier?
  - Are software tools needed? Are they currently available? Do they have all the necessary functionality?

The Inception Phase: Requirements Workflow – Risks

- There are three major risk categories:
  - Technical risks
  - Not getting the requirements right
    - Mitigated by performing the requirements workflow correctly
  - Not getting the architecture right
    - The architecture may not be sufficiently robust
- To mitigate all three classes of risks
  - The **risks need to be ranked** so that the critical risks are mitigated first

The Inception Phase: Analysis, Design, Implementation, Testing Workflows

- A small amount of the analysis workflow may be performed
  - Information needed for the design of the architecture is extracted
- A small amount of the design workflow may be performed
- Coding is generally **not** performed. However, a proof of concept prototype is sometimes built to test the feasibility of constructing part of the software product
- The test workflow commences almost at the start of the inception phase
  - The aim is to **ensure that the requirements have been accurately determined**

The Inception Phase: Planning

- There is insufficient information at the beginning of the inception phase to plan the entire development
  - The only planning that is done at the start of the project is the planning for the inception phase itself
- Similarly, the only planning that can be done at the end of the inception phase is the plan for the elaboration phase

The Inception Phase: Documentation

- The deliverables of the inception phase include:
  - The initial version of the **domain model**
  - The initial version of the **business model**
  - The initial version of the **requirements artifacts**
  - A preliminary version of the **analysis artifacts**
  - A preliminary version of the **architecture**
  - The initial **list of risks**
  - The initial **ordering of the use cases**
  - The plan for the elaboration phase – The initial version of the business case

The Inception Phase: The Initial Business Case

- Obtaining the initial version of the business case is the **overall aim of the inception phase**

- This initial version incorporates
  - A **description of the scope** of the software product
  - **Financial details**
  - For marketed product, the business case will include
    - Revenue projections, market estimates, initial cost estimates
  - For in-house product, the business case will include
    - The initial cost–benefit analysis TVM, NPV

ii. Elaboration Phase

- The aim of the elaboration phase is to
  - **Refine** the initial requirements
  - Refine the architecture
  - **Monitor** the risks and refine their priorities
  - Refine the business case
  - **Produce** the project management plan
- The major activities of the elaboration phase are **refinements or elaborations** of the previous phase

The Tasks of the Elaboration Phase

- The tasks of the elaboration phase correspond to:
  - All but completing the requirements workflow
  - Performing virtually the entire analysis workflow
  - Starting the design of the architecture

The Elaboration Phase: Documentation

- The deliverables of the elaboration phase include:
  - The completed **domain model**
  - The completed **business model**
  - The completed **requirements artifacts**
  - The completed **analysis artifacts**
  - An **updated version** of the architecture
  - An **updated** list of risks

– The project management plan (for the rest of the project)
– **The completed business case**

iii. Construction Phase

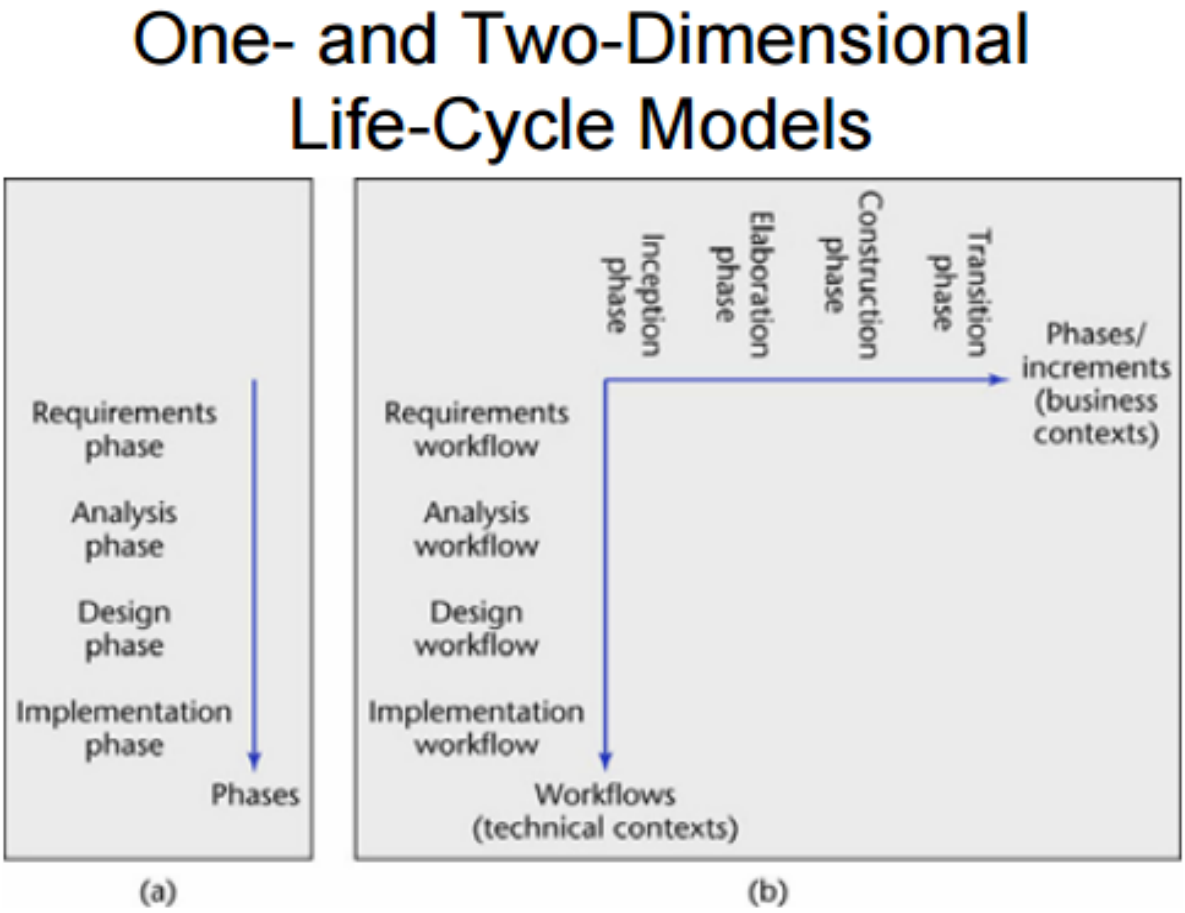
• The aim of the construction phase is to **produce the first operational-quality version** of the software product
– This is sometimes called the **beta release**
• The emphasis in this phase is on
– Implementation
– Testing
• Unit testing of modules
• Integration testing of subsystems
• Product testing of the overall system

iv. The Transition Phase

• The aim of the transition phase is to **ensure that the client's requirements have indeed been met**
– **Faults** in the software product are **corrected**
– All the **manuals** are **completed**
– Attempts are made to **discover any previously unidentified risks**
• This phase is **driven by feedback** from the site(s) at which the beta release has been installed

The Transition Phase: Documentation

• The deliverables of the transition phase include:
– All the artifacts (final versions)
– The completed manuals



Why a Two-Dimensional Model?

- A traditional life cycle is a **1D model**
  - Represented by the single axis e.g. Waterfall model
- The Unified Process is a **2D model**
  - Represented by the two axes e.g. Evolution Tree Model
- The two-dimensional figure shows
  - The **workflows** (technical contexts), and
  - The **phases** (business contexts)
- Are all the additional complications of the 2D model necessary?
  - In an ideal world, each workflow would be completed before the next workflow is started

– In reality, the development task is too big for this

– As a consequence of Miller's Law

• The development task has to be **divided into increments (phases)**

• Within each increment, **iteration is performed until the task is complete**

• At the beginning of the process, there is not enough information about the software product to carry out the requirements workflow and other core workflows

• A software product has to be broken into subsystems

• Even subsystems can be too large at times

– Components may be all that can be handled until a fuller understanding of all the parts of the product as a whole has been obtained

• The Unified Process handles the inevitable changes well

– The moving target problem

– The inevitable mistakes

• The Unified Process is the best solution to date for treating a large problem as a set of smaller, largely independent sub problems

– It provides a framework for incrementation and iteration

– In the future, it will inevitably be superseded by some

better methodology

As a summary,

Q) What is the Unified Process (UP)?

A software development process describes an approach to building, deploying, and possibly maintaining software.

The Unified Process has emerged as a popular iterative software development process for building object-oriented systems. In particular, the Rational Unified Process or RUP, a detailed refinement of the Unified Process, has been widely adopted.

The UP is very flexible and open, and encourages including skillful practices from other iterative methods, such as from Extreme Programming (XP), Scrum, and so forth. For example, XP's test-driven development, refactoring and continuous integration practices can fit within a UP project. So can Scrum's common project room ("war room") and daily Scrum meeting practice.

The UP combines commonly accepted best practices, such as an iterative lifecycle and risk-driven development, into a cohesive and well-documented process description.

Q) What is the importance of the Unified Process (UP)?

a. The UP is an *iterative* process. Iterative development influences how to Introduce OOA/D, and to understand how it is best practiced

b. UP practices provide an example structure for how to do and thus how to explain OOA/D.

c. The UP is flexible, and can be applied in a lightweight and agile approach that includes practices from other agile methods (such as XP or Scrum).

Agile Unified Process (AUP)

Agile Unified Process (AUP) is a simplified version of the Rational Unified Process (RUP) developed by Scott Ambler. It describes a simple, easy to understand approach to developing business application software using agile techniques and concepts yet still remaining true to the RUP. The AUP applies agile techniques including test-driven development (TDD), agile modeling (AM), agile change management, and database refactoring to improve productivity.

Manifesto for Agile Software Development

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

Common Values from the Agile Manifesto

Scrum is an Agile framework and, as such, is consistent with the values of the Agile Manifesto.

Individuals and interactions over processes and tools
Scrum is a team-based approach to delivering value to the business. Team members work together to achieve a shared business goal. The Scrum framework promotes effective interaction between team members so the team delivers value to the business.

Once the team gets a business goal, it:

- Figures out how to do the work
- Does the work
- Identifies what's getting in its way
- Takes responsibility to resolve all the difficulties within its scope
- Works with other parts of the organization to resolve concerns outside their control

This focus on team responsibility in Scrum is critical.

Working software over comprehensive documentation
Scrum requires a working, finished product increment as the primary result of every sprint.

Whatever activities take place during the sprint, the focus is on the creation of the product increment. A Scrum team's goal is to produce a product increment every sprint. The increment may not yet include enough functionality for the business to decide to ship it, but the team's job is to ensure the functionality present is of shippable quality.

Customer collaboration over contract negotiation

Scrum is a framework designed to promote and facilitate collaboration. Team members collaborate with each other to find the best way to build and deliver the software, or other deliverables, to the business. The team, especially the product owner, collaborates with stakeholders to inspect and adapt the product vision so the product will be as valuable as possible.

Responding to change over following a plan

Scrum teams make frequent plans. For starters, they plan the current sprint. In addition, many teams create longer-term plans, such as release plans and product roadmaps. These plans help the team and the business make decisions. However, the team's goal is not to blindly follow the plan; the goal is to create value and embrace change. In essence, the thought process and ideas necessary for planning are more important than the plan itself.

A plan created early is based on less information than will be available in the future so, naturally, it may not be the best plan. As new information is discovered, the team updates the product backlog. That means the direction of the product likely shifts. This continuous planning improves the team's chances of success as it incorporates new knowledge into the experience.

Scrum teams constantly respond to change so that the best possible outcome can be achieved. Scrum can be described as a framework of feedback loops, allowing the team to constantly inspect and adapt so the product delivers maximum value.

The twelve principles of agile development include:

Customer satisfaction through early and continuous software delivery– Customers are happier when they receive working software at regular intervals, rather than waiting extended periods of time between releases.

Accommodate changing requirements throughout the development process – The ability to avoid delays when a requirement or feature request changes.

Frequent delivery of working software – Scrum accommodates this principle since the team operates in software sprints or iterations that ensure regular delivery of working software.

Collaboration between the business stakeholders and developers throughout the project – Better decisions are made when the business and technical team are aligned.

Support, trust, and motivate the people involved – Motivated teams are more likely to deliver their best work than unhappy teams.

Enable face-to-face interactions – Communication is more successful when development teams are co-located.

Working software is the primary measure of progress – Delivering functional software to the customer is the ultimate factor that measures progress.

Agile processes to support a consistent development pace – Teams establish a repeatable and maintainable speed at which they can deliver working software, and they repeat it with each release.

Attention to technical detail and design enhances agility – The right skills and good design ensures the team can maintain the pace, constantly improve the product, and sustain change.

Simplicity – Develop just enough to get the job done for right now.

Self-organizing teams encourage great architectures, requirements, and designs – Skilled and motivated team members who have decision-making power, take ownership, communicate regularly with other team members, and share ideas that deliver quality products.

Regular reflections on how to become more effective – Self-improvement, process improvement, advancing skills, and techniques help team members work more efficiently.

The intention of Agile is to align development with business needs, and the success of Agile is apparent. Agile projects are customer focused and encourage customer guidance and participation. As a result, Agile has grown to be an overarching view of software development throughout the software industry and an industry all by itself.

The UML

The Unified Modeling Language (UML) is a language for **specifying, visualizing, constructing, and documenting the artifacts** of software systems, as well as for business modeling and other non-software systems .

The UML has emerged as the de facto and de jure standard diagramming notation for object-oriented modeling.

It started as an effort by Grady Booch and Jim Rumbaugh in 1994 to combine the diagramming notations from their two popular methods—the Booch and OMT (Object Modeling Technique) methods.

They were later joined by Ivar Jacobson, the creator of the Objectory method, and as a group came to be known as the *three amigos.*

Many others contributed to the UML, perhaps most notably Cris Kobryn, a leader in its ongoing refinement.

The UML is a language for

-visualizing -constructing

-specifying -documenting

UML is a language which combines vocabulary and rules that tells us how to create and read well-framed models

1. UML is a language for visualizing

Modeling of any system requires certain degree of visualization of the system.

If the developer directly codes the requirement of a system, he can have the following problems.

Communicating these conceptual models to others. As all speak different language, there are chances of error and understanding the system.

Something in a software cannot be understood unless they build modules that transcend the textual programming language.

If a developer never wrote down the models, that are in his head such information is lost forever from the implementation once the developer moved on.

In a system development some things are best modeled textually, while some are best modeled graphically.

Thus UML is such a language that is more than just a bunch of graphical symbols. Each symbol in contrast has a well defined semantics which is best for visualization

2. It is a language for specifying.

Specifying means building models that are **precise, unambiguous and complete**.

In particular UML addresses the specification of all the important analysis, design and implementation decisions that must be:

-Scientific -Distributed web-based services.

3. UML is a language for constructing.

The models of UML can directly be **connected to a variety of programming languages** i.e . It is possible to map a UML model to a programming language such as Java, C++, etc.

4. UML is a language for documenting.

A healthy software organization produces **all sorts of artifacts** in addition to executable code.

Their artifacts include:

Requirements	Source code	Prototypes
Architecture	Project plans	Release
Design	Tests	

The UML addresses the **documentation of a system's architecture** and all of its details. The UML also provides the language for **expressing requirements** and for test. Finally, the UML provides a language for **modeling the activities of a project planning and release management**

The word UML stands for Unified Modeling Language

The UML was adopted in 1997 as a standard by the OMG (Object Management Group, an industry standards body), and has continued to be refined in new OMG UML versions.

Unified because it …– **Combines** main preceding OO methods (**Booch by Grady Booch, OMT by Jim Rumbaugh and OOSE by Ivar Jacobson**)

Modeling because it is …– Primarily used for **visually modeling** systems. Many system views are supported by appropriate models

Language because …– It **offers a syntax** through which to express modeled knowledge

UML is not:

- A **visual programming language** or environment
- A **database specification** tool
- A **development process** (i.e. an SDLC)
- A **panacea** (The ultimate solution)

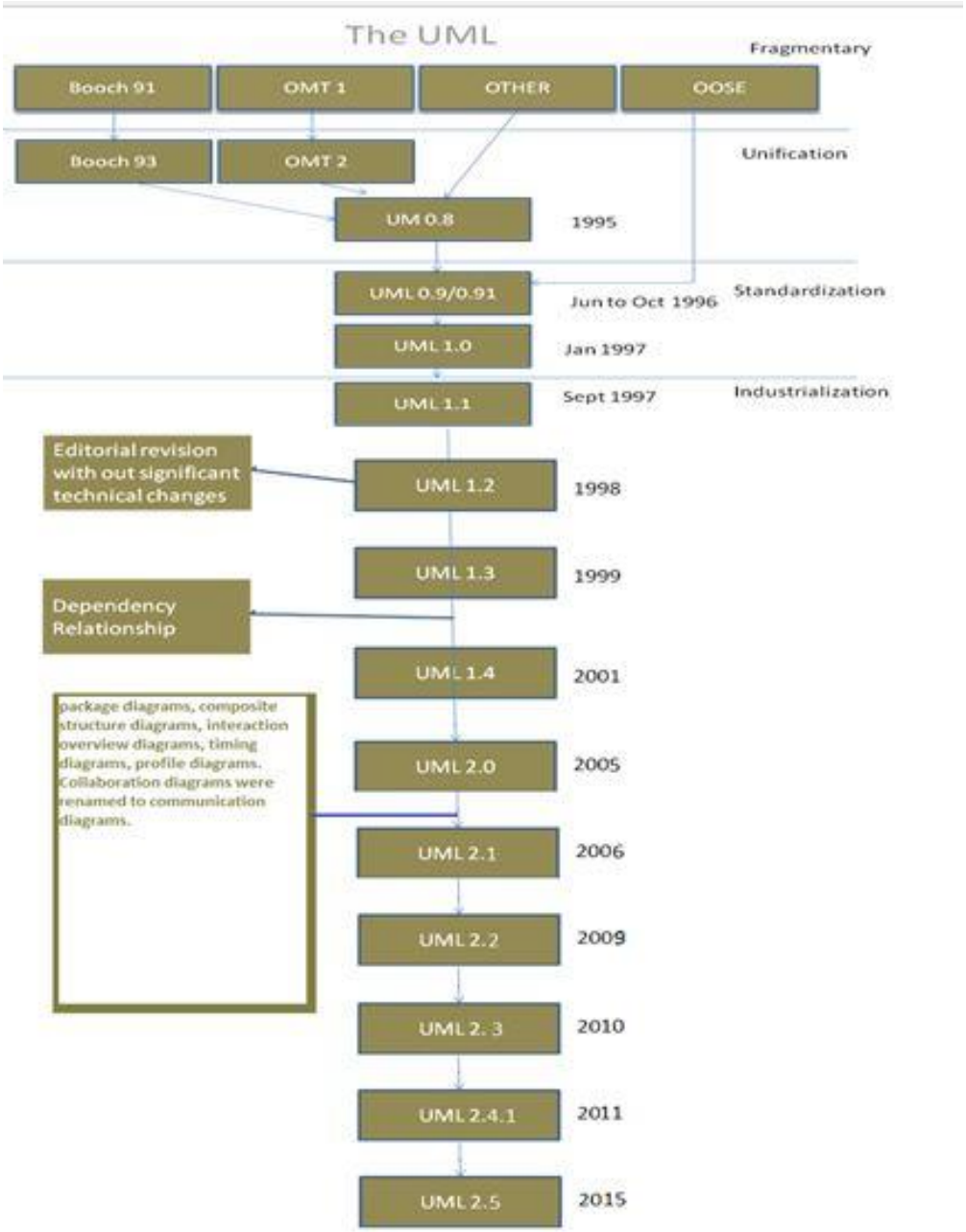
• A **quality guarantee**

UML

Helps to reduce cost and time-to-market.

Helps managing a complex project architecture.

Helps to convey ideas between developers\designers\etc.



Q) What are the three ways to apply UML?

a. **UML as sketch** Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.
b. **UML as blueprint** Relatively detailed design diagrams used either for reverse engineering to visualize and better understanding existing code in UML diagrams, or for 2) code generation (forward engineering).

   If reverse engineering, a UML tool reads the source or binaries and generates (typically) UML package, class, and sequence diagrams. These "blueprints" can help the reader understand the bigpicture elements, structure, and collaborations.

   Before programming, some detailed diagrams can provide guidance for code generation (e.g., in Java), either manually or automatically with a tool. It's common that the diagrams are used for some code, and other code is filled in by a developer while coding (perhaps also applying UML sketching).

c. **UML as programming language** - Complete executable specification of a software system in UML. Executable code will be automatically generated, but is not normally seen or modified by developers; one works only in the UML "programming language." This use of UML requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams), and is still under development in terms of theory, tool robustness and usability.

Q) What are the three perspectives to apply UML?

Three Perspectives to Apply UML

a. **Conceptual perspective** the diagrams are interpreted as describing things in a situation of the real world or domain of interest.
b. **Specification (software) perspective** the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).
c. **Implementation (software) perspective** the diagrams describe software implementations in a particular technology (such as Java).

The UML: Terms and Concepts

A **system** is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints.

A **subsystem** is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained **elements**.

A **model** is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system.

A **view** is a projection into the organization and structure of a system's model, focused on one aspect of that system.

A **diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

To understand UML we require three major elements learning; the UML's building blocks, the rules that dictate how those building blocks may be put together and some common mechanism that apply throughout the UML.

The vocabulary of UML encompasses three kinds of building blocks:

- Things.
- Diagrams
- Relationship.

Things: Those are abstractions that are first class citizens in UML modeling.

There are four kinds of things in UML.

- Structural things.
- Grouping things
- Behavioral things.
- Annotational things.

Structural things:-

These are the nouns of the UML models. They are the static part of the model representing elements that are either conceptual or physical.

- Active Class
- Component
- Node etc.

Behavioral things

- Interaction
- State machine etc.

Grouping things

- Package

Annotational things

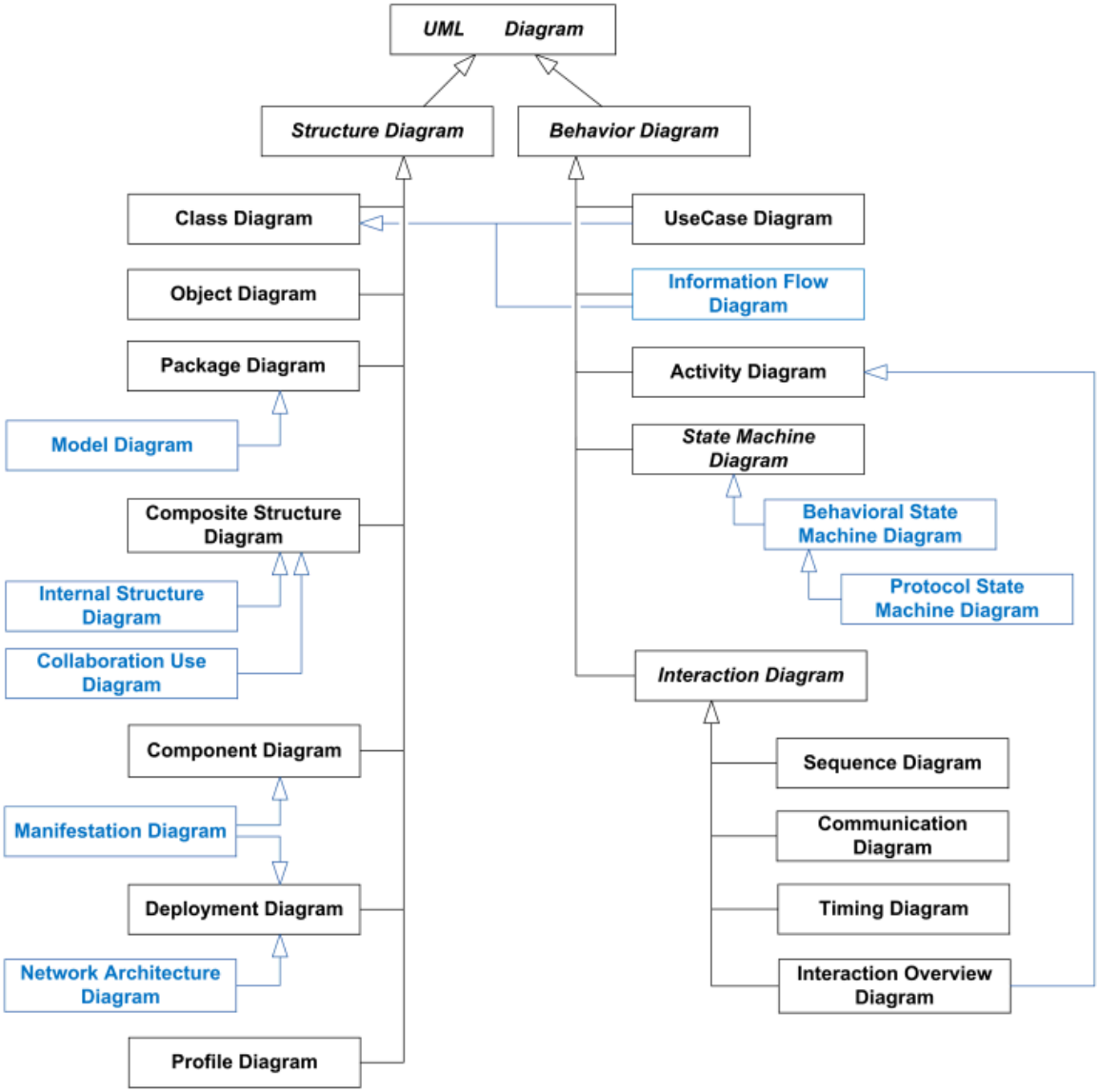
- Note

Relationships in UML

- Dependency
- Association
- Generalization
- Realization

Classification of UML Diagrams

UML specification defines two major kinds of UML diagram: **structure diagrams** and **behavior diagrams** UML diagrams could be categorized hierarchically as shown below:



A. Structural Diagrams

Static aspects of a house: walls, doors, windows, pipes, wires, and vents,

Static aspects of a software system : classes, interfaces, components, and nodes.

Used to describe the **building blocks** of the system

– features **that do not change** with time.

These diagrams answer the question – **What's there?**

Structure diagram shows static structure of the system and its parts on different abstraction and implementation levels and how those parts are related to each other.

The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

Structure diagrams are not utilizing time related concepts, do not show the details of dynamic behavior. However, they may show relationships to the behaviors of the classifiers exhibited in the structure diagrams.

1. Class diagram

It is a static structure diagram which describes structure of a system at the level of classifiers (classes, interfaces, etc.).

It shows some classifiers of the system, subsystem or component, different relationships between classifiers, their attributes and operations, constraints

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

Class diagrams are static – display **what interacts** but **not what happens when interaction occurs.**

May contain notes and constraints.

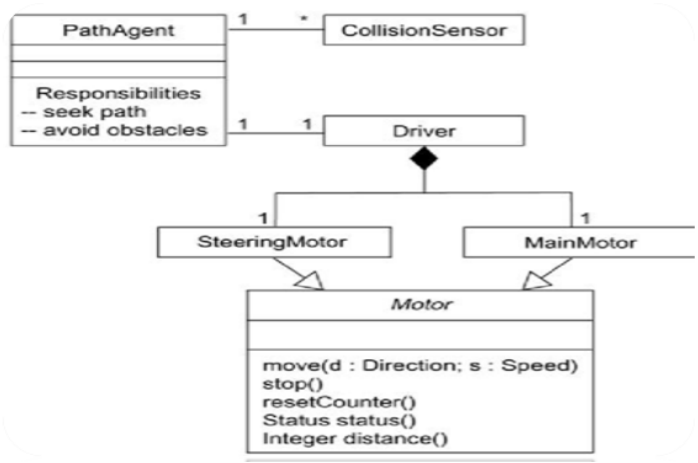
They are used :

1. To model the **vocabulary of a system**

Making a decision about which abstractions are apart of the system under consideration and which fall outside its boundaries. Used to specify these abstractions and their responsibilities.

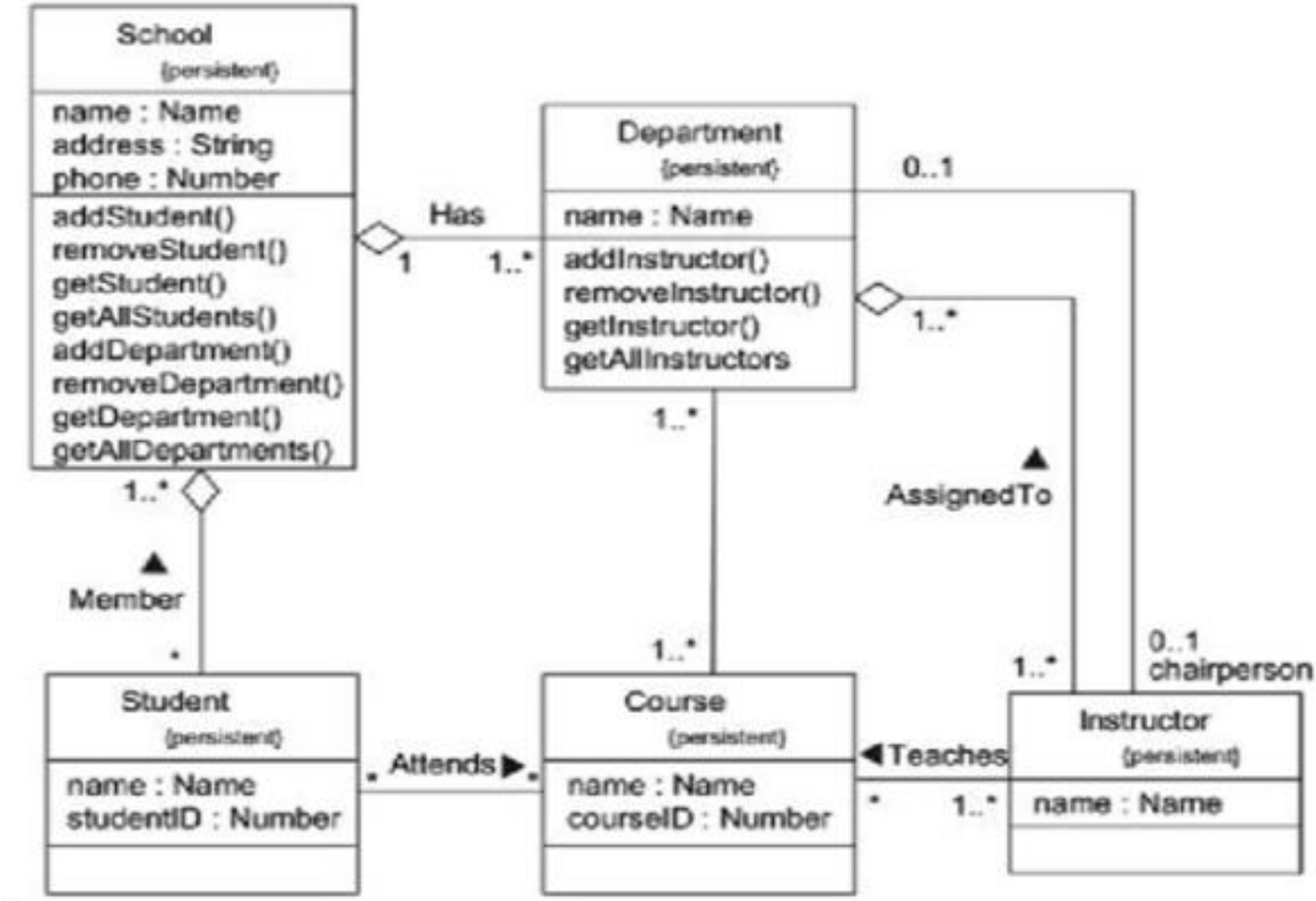
2. To model simple **collaborations**

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior



3. To model a **logical database schema**

The blueprint for the conceptual design of a database.



Classes are represented by a rectangle divided to three parts: class name, attributes and operations.

Attributes are written as:

visibility name [multiplicity] :type-expression =initial-value

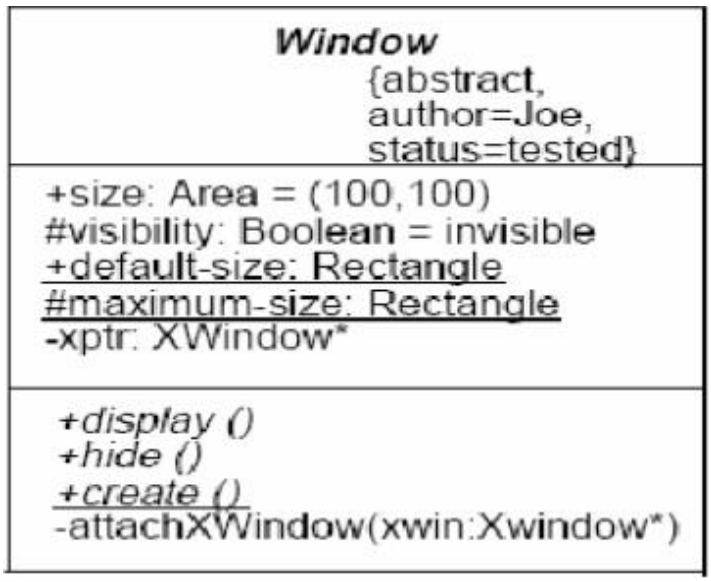
Operations are written as:

visibility name (parameter-list) :return type-expression

Visibility is written as:

+public -private

#protected



Class Diagram Relationships

Association –

Two classes are associated if one class has to know about the other. (uses)

Aggregation –

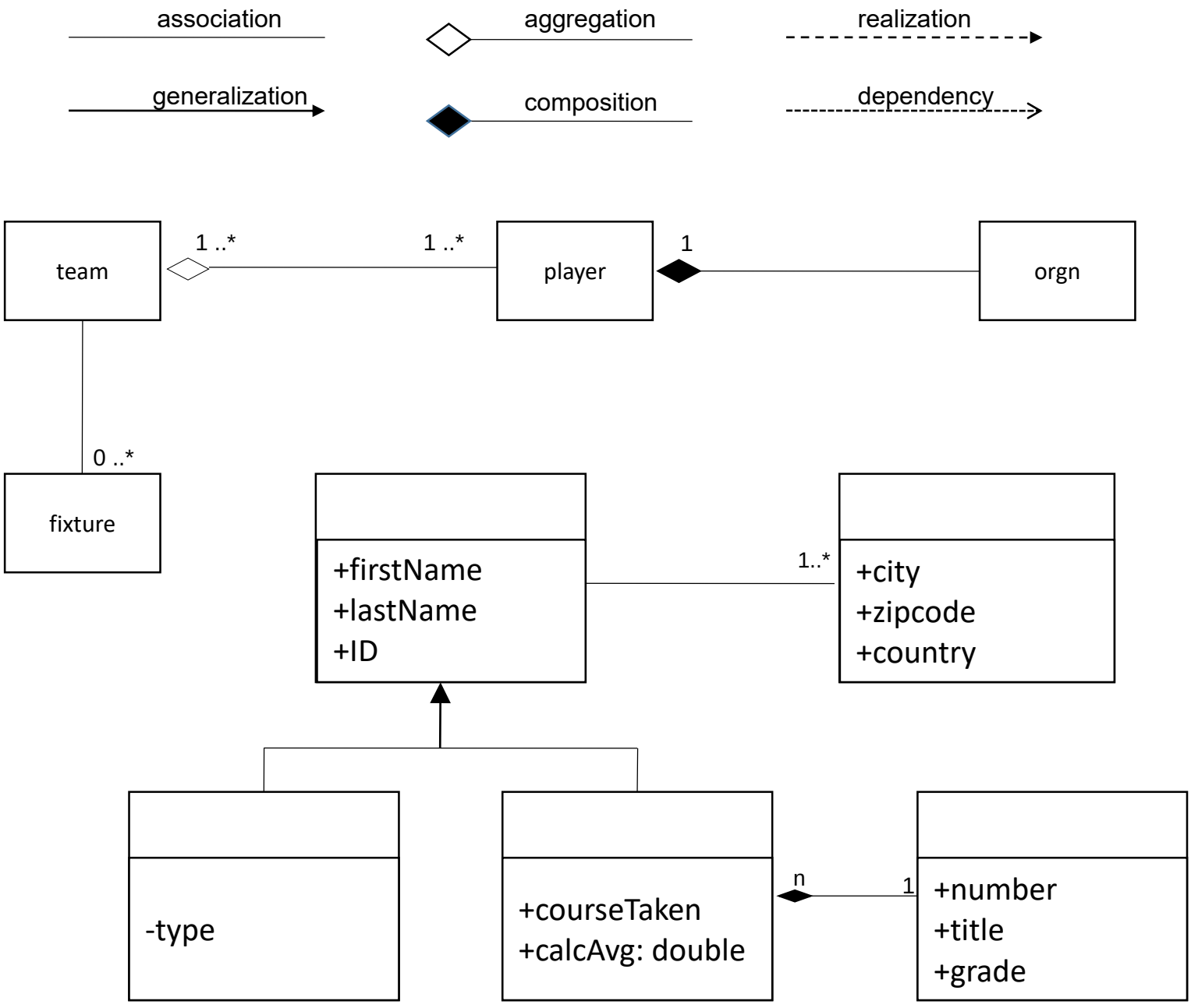
An association in which one class belongs to a collection in the other.(whole part)

Generalization –

An inheritance link indicating one class is a base class of the other.

Dependency –

A labeled dependency between classes (such as friend, classes)


association
aggregation
realization
generalization
composition
dependency
team
1 ..*
1 ..*
player
1
orgn
0 ..*
fixture
+firstName
+lastName
+ID
1..*
+city
+zipcode
+country
-type
+courseTaken
+calcAvg: double
n
1
+number
+title
+grade

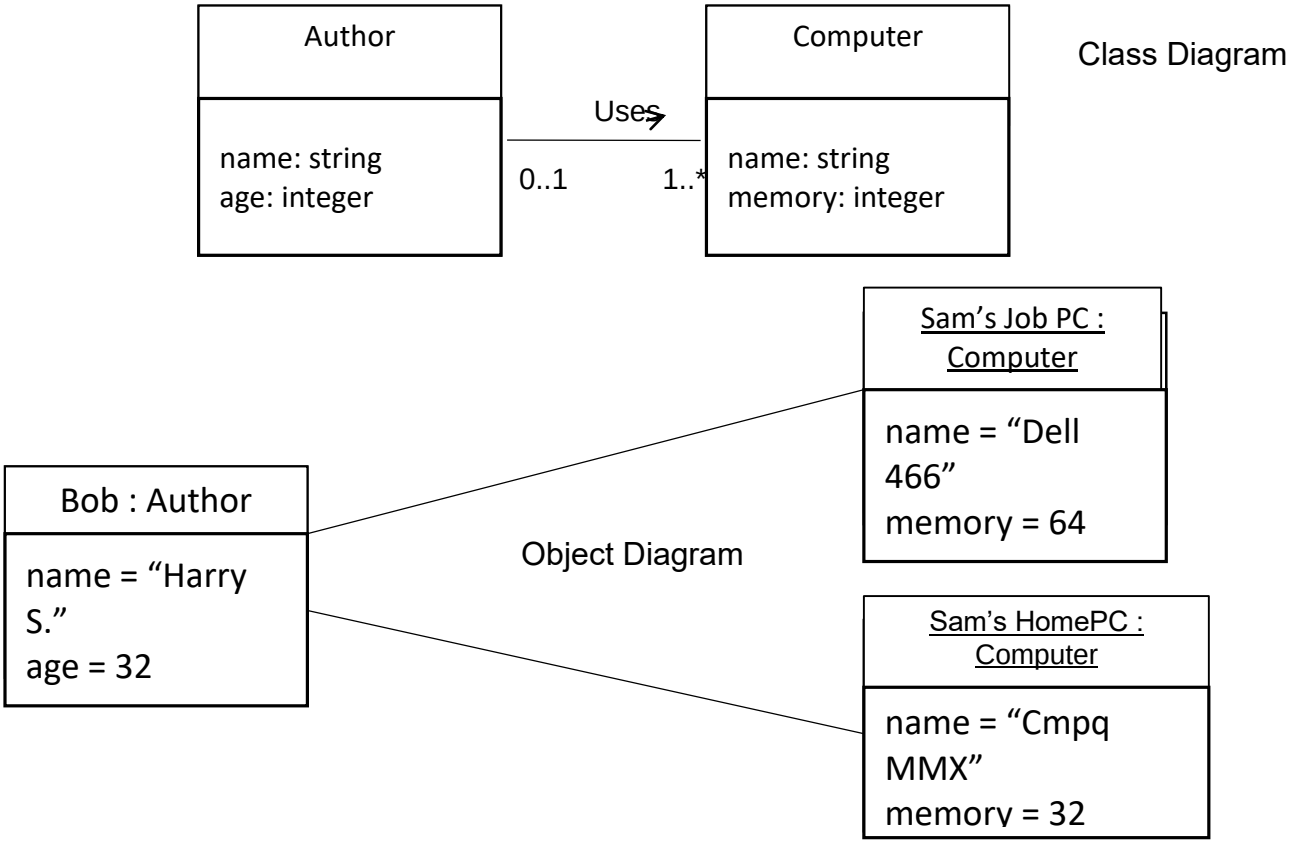
2. Object Diagram

Now obsolete, is defined as "a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time."

An *object diagram* shows a set of **objects and their relationships**.

Used to illustrate **data structures, the static snapshots of instances** of the things found in class diagrams.

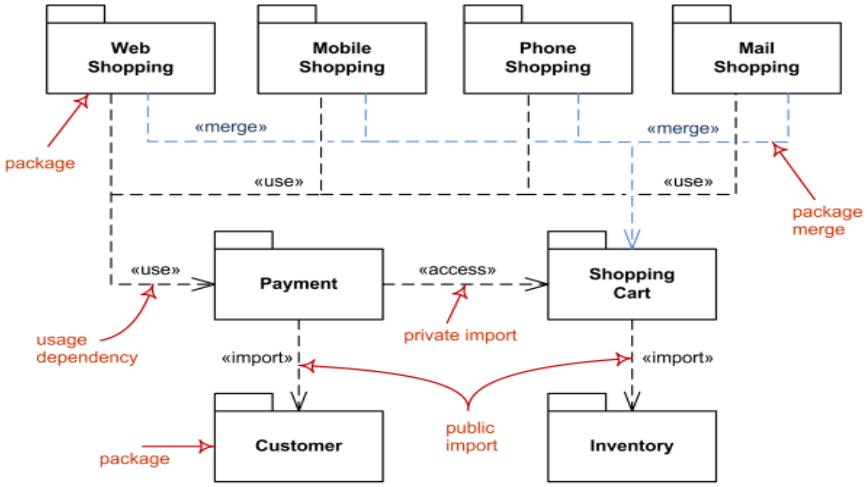
Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the **perspective of real** or **prototypical cases.**



3. Package diagram

Shows packages and relationships between the packages.

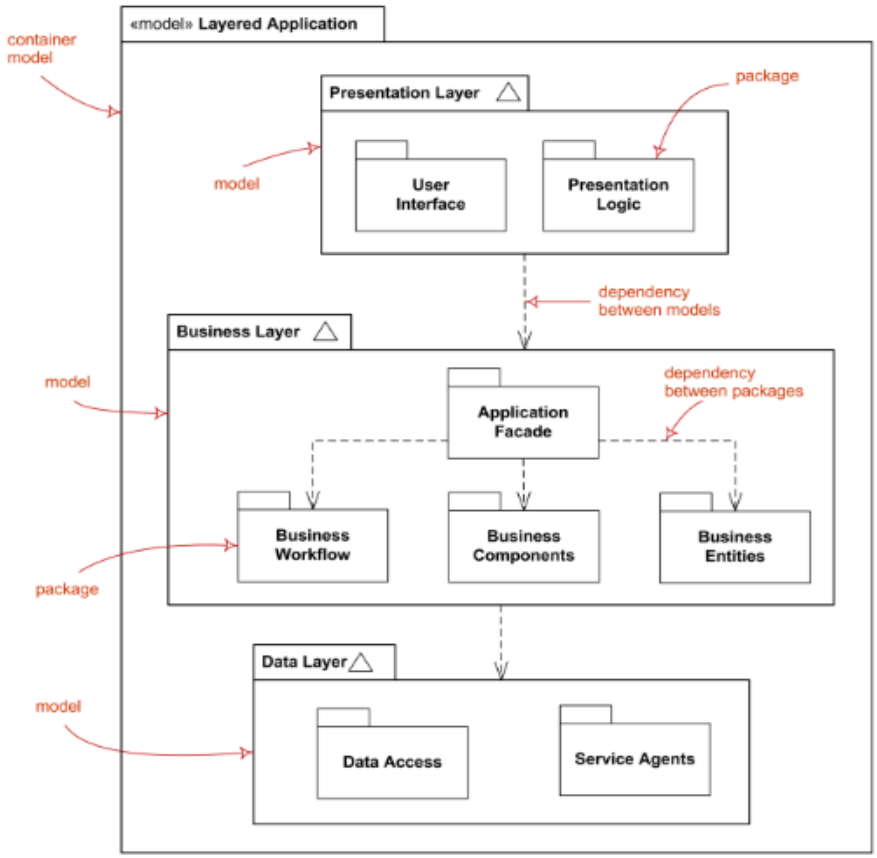
Package diagram includes the following elements: package, packageable element, dependency, element import, package import, package merge.



Model diagram

It is UML auxiliary structure diagram which shows some abstraction or specific view of a system, to describe architectural, logical or behavioral aspects of the system. It could show, for example, architecture of a multi-layered (aka multi-tiered) application - multi-layered application model.

In the diagram below, Layered Application is a "container" model which contains three other models - Presentation Layer, Business Layer, and Data Layer. There are dependencies defined between these contained models.



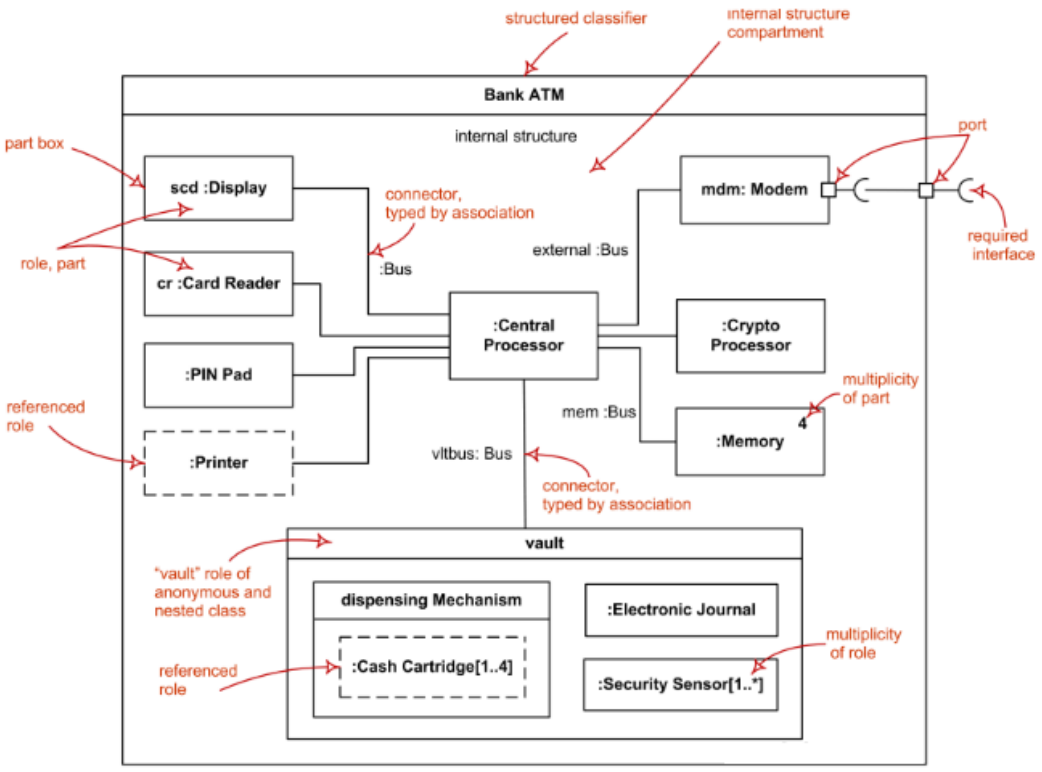
4. Composite structure diagram

It could be used to show:

- Internal structure of a classifier
- A behavior of a collaboration

a. Internal Structure diagrams show internal structure of a classifier - a decomposition of the classifier into its properties, parts and relationships.

A **composite structure diagram** that shows internal structure of a classifier includes the following elements: class, part, port,connector, usage.



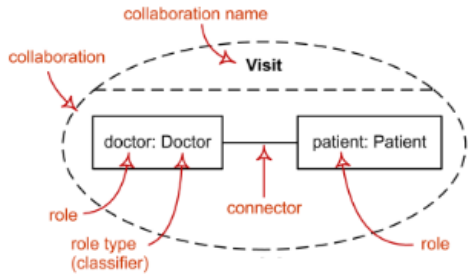
b. Collaboration use diagram shows objects in a system cooperating with each other to produce some behavior of the system.

A behavior of a **collaboration** will eventually be exhibited by a set of cooperating instances (specified by classifiers) that communicate with each other by sending signals or invoking operations. However, to understand the mechanisms used in a design, it may be important to describe only those aspects of these classifiers and their interactions that are involved in accomplishing a **task** or a related set of tasks, projected from these classifiers.

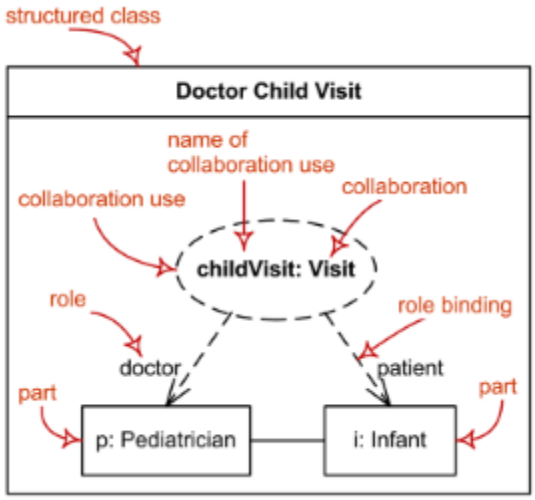
Collaborations allow us to describe only the relevant aspects of the cooperation of a set of instances by identifying the specific roles that the instances will play.

Interfaces allow the externally observable properties of an instance to be specified without determining the classifier that will eventually be used to specify this instance. Consequentially, the **roles** in a collaboration will often be typed by interfaces and will then prescribe properties that the participating instances must exhibit, but will not determine what class will specify the participating instances.

A composite structure diagram that shows behavior of a collaboration includes the following elements: collaboration,connector, part, collaboration specialization, dependency.



Collaboration use represents one particular use (**occurrence**) or application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration. A collaboration use shows how the pattern described by a collaboration is applied in a given context, by binding specific entities from that context to the roles of the collaboration.



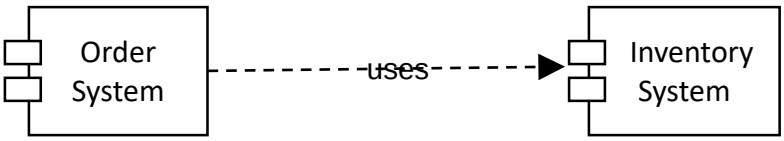
Classifier (in internal structures and collaborations) is extended with the capability to own collaboration uses. These collaboration uses link acollaboration with the classifier to give a description of the behavior of the classifier.

5. Component Diagram

Shows a set of **components and their relationships.**

Used to illustrate the **static implementation view** of a system.

Component diagrams are related to class diagrams in that a component **typically maps to one or more classes, interfaces, or collaborations**.



6. Deployment Diagram

A *deployment diagram* shows a **set of nodes and their relationships**.

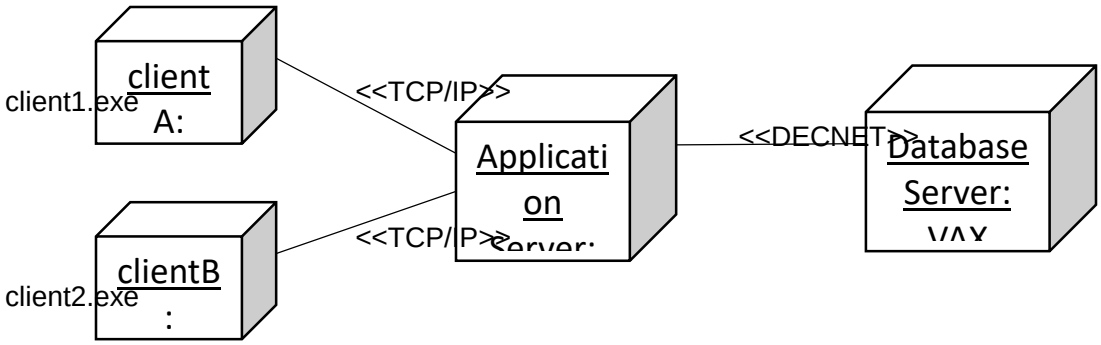
Used to illustrate the static deployment **view of an architecture**.

Deployment diagram shows architecture of the system as deployment (distribution) of software artifacts to deployment targets.

Components were directly deployed to nodes in UML 1.x deployment diagrams. In UML 2.x artifacts are deployed to nodes, and artifacts could manifest (implement) components. Components are deployed to nodes indirectly through artifacts.

Specification level deployment diagram (also called type level) shows some overview of deployment of artifacts to deployment targets, without referencing specific instances of artifacts or nodes.

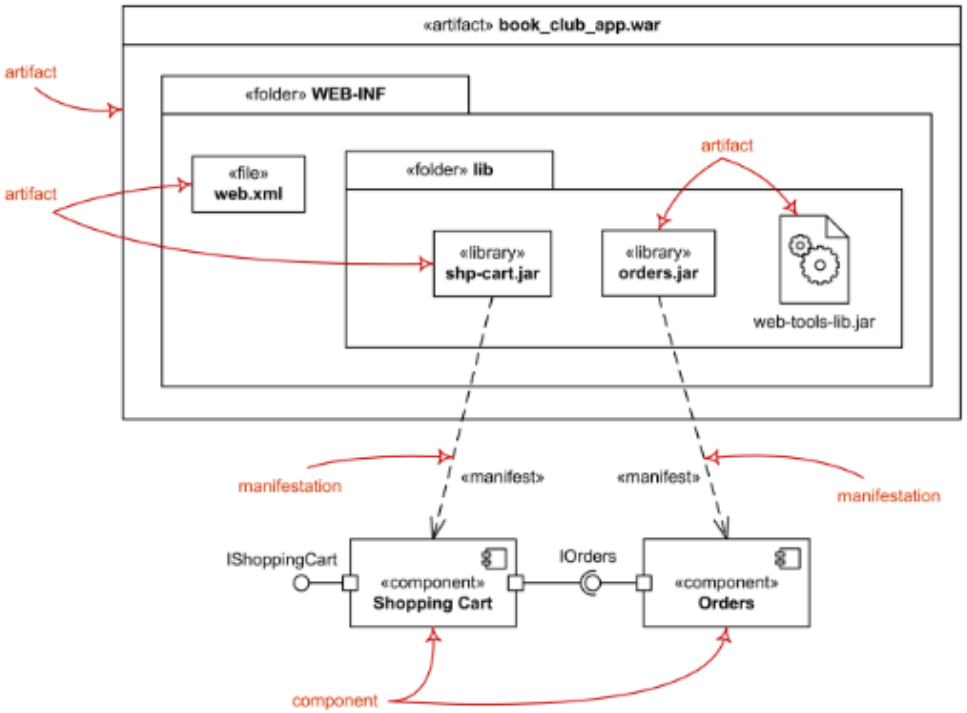
Instance level deployment diagram shows deployment of instances of artifacts to specific instances of deployment targets. It could be used for example to show differences in deployments to development, staging or production environments with the names/ids of specific build or deployment servers or devices.
While component diagrams show components and relationships between components and classifiers, and deployment diagrams - deployments of artifacts to deployment targets



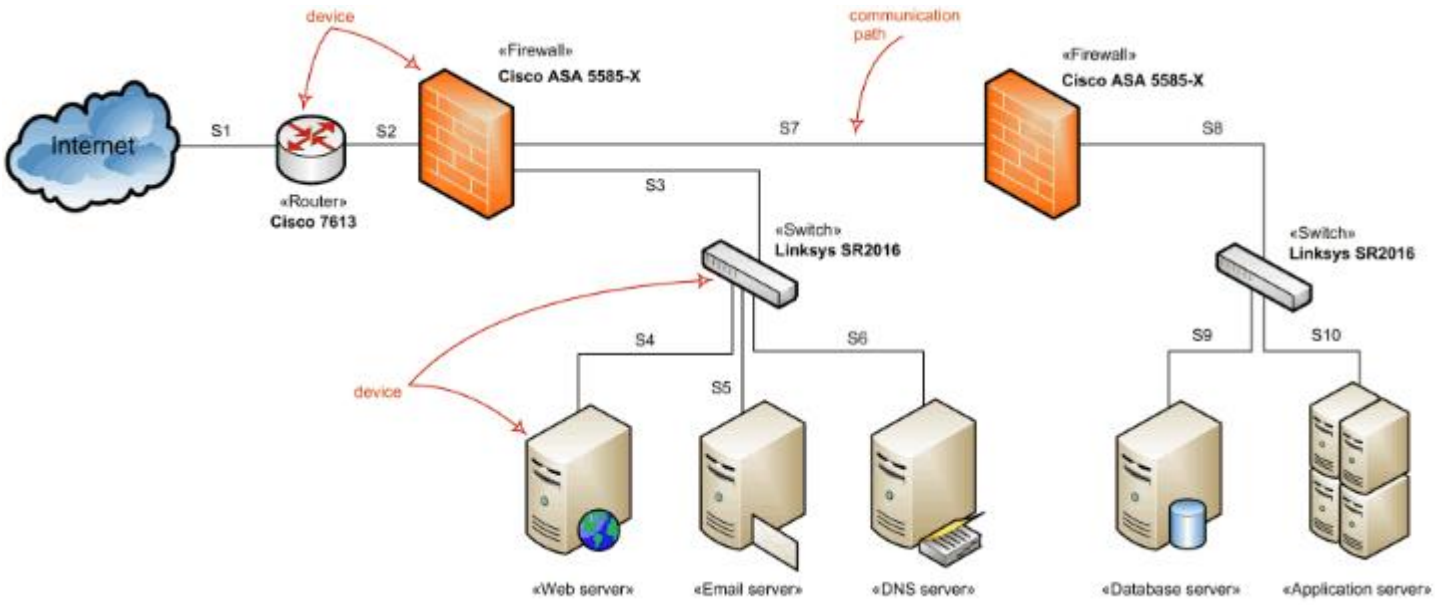
Manifestation diagram

It is the missing intermediate diagram used to show manifestation (implementation) of components by artifacts and internal structure of artifacts.

Manifestation of components by artifacts could be shown using either component diagrams or deployment diagrams.



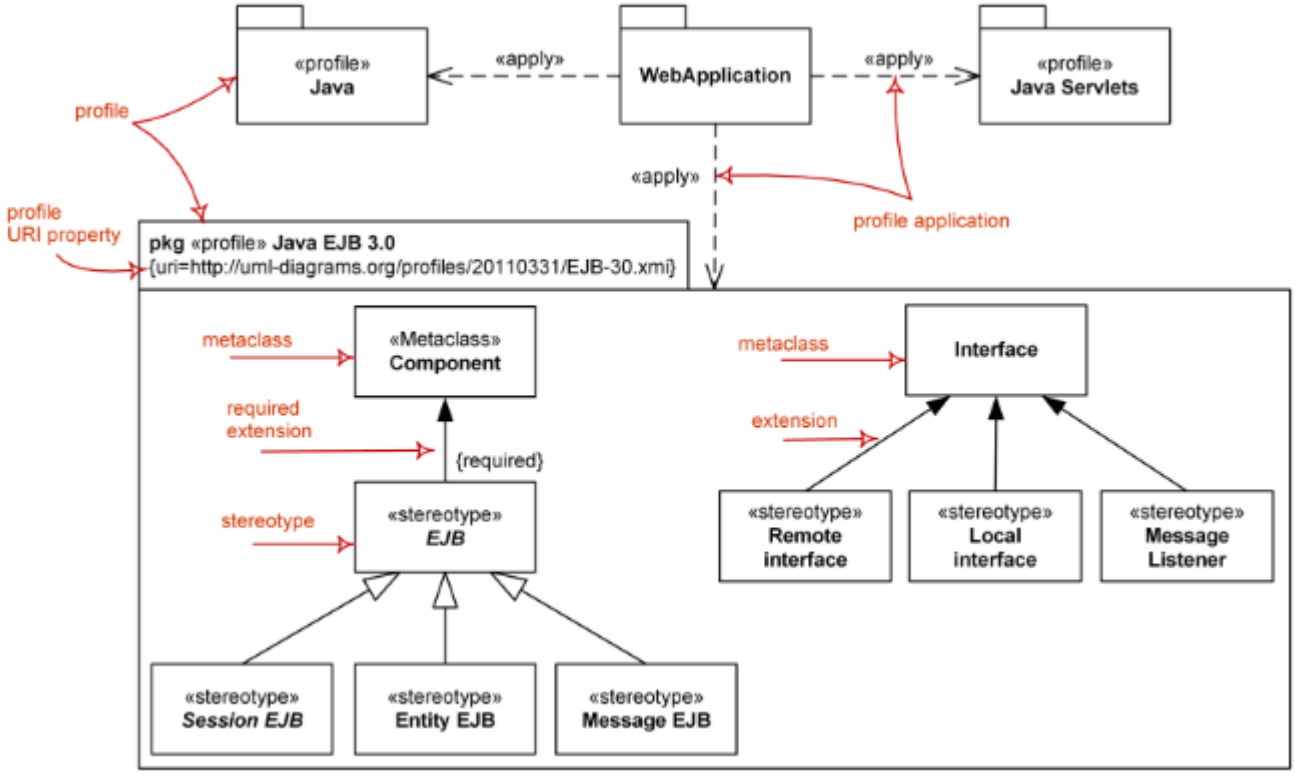
Network architecture diagram is a kind of deployment diagram that is used to show logical or physical network architecture of the system.


device
communication path
«Firewall»
Cisco ASA 5585-X
«Firewall»
Cisco ASA 5585-X
Internet
S1
S2
S7
S8
S3
«Router»
Cisco 7613
«Switch»
Linksys SR2016
«Switch»
Linksys SR2016
S4
S6
S9
S10
device
S5
«Web server»
«Email server»
«DNS server»
«Database server»
«Application server»

7. Profile diagram

It is auxiliary UML diagram which allows defining custom stereotypes, tagged values, and constraints. The Profile mechanism has been defined in UML for providing a lightweight extension mechanism to the UML standard. Profiles allow to adapt the UML metamodel for different

- platforms (such as J2EE or .NET), or
- domains (such as real-time or business process modeling).



B. Behavioral Diagrams

The UML's behavioral diagrams are used to visualize, specify, construct, and document **the dynamic aspects of a system**.

Dynamic aspects of a system represent **its changing parts.**

They show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

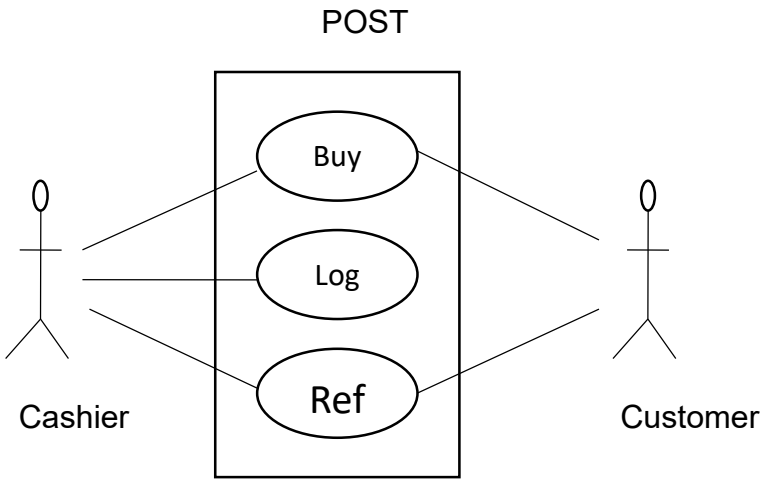
Used to show **how** the system evolves over time (responds to requests , events etc)
The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

1. Use Case Diagram

A *use case diagram* shows **a set of use cases and actors** (a special kind of class) and their relationships.

Use case diagrams are especially important in organizing and modeling the behaviors of a system.

Describes **what a system does** from the standpoint of an external observer.



Emphasis on what *a system does* **rather than how.**

Scenario – Shows what happens **when someone interacts with system.**

Actor – A user or another system that interacts with the modeled system.

A use case diagram describes **relationships between actors and scenarios.**

Provides system requirements **from the user's point of View.**

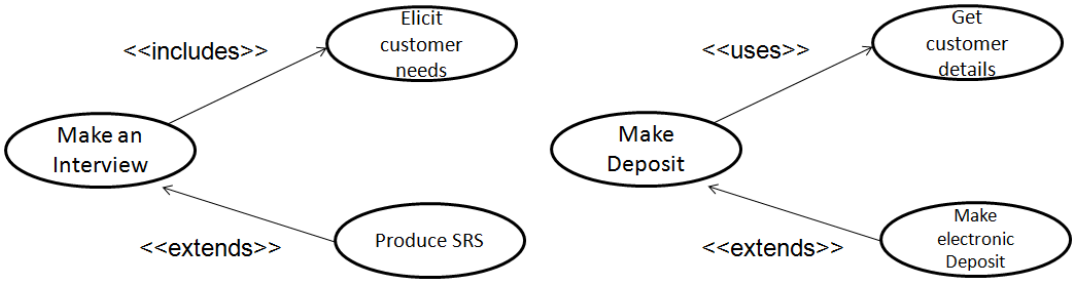
Use case Relationships:

Association – defines a relationship between an actor and a use case.

Extend - defines that instances of a use case may be augmented with some additional behavior defined in an extending *use case*.

Use/Include - drawn as a **dependency relationship** that points from the base use case to the used use case.

- defines that a use case uses a behavior defined in another use case.



Use Case Diagram

- Attention **focused on the part of the business process** that is going to be supported by the IS.
- It is the **end-user perspective** model.
- It is **goal driven**
- It helps to **identify system services.**
- It is **not used as DFDs**.
- **Sequences, branching, loops, rules**, etc. cannot (and should not) be directly expressed.

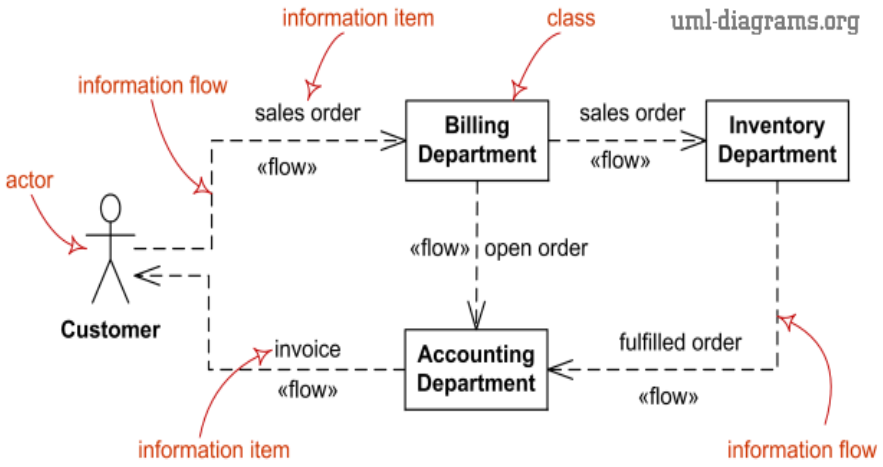
They are used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of

the system (actors) to provide some observable and valuable results to actors or other stakeholders of the system(s).

UML specification also defines use case diagrams as specialization of class diagrams (which are structure diagrams). Use case diagrams could be considered as a special case of class diagrams where classifiers are restricted to be either actors or use cases and the most used relationship is association.

Information flow diagram is UML behavior diagram which shows exchange of information between system entities at some high levels of abstraction. Information flows may be useful to describe circulation of information through a system by representing aspects of models not yet fully specified or with less details.

Information flows do not specify the nature of the information, mechanisms by which it is conveyed, sequences of exchange, or any control conditions. Information items can be used to represent the information that flows through a system along the information flows before details of their realization have been designed.



Limitations of Information Flow Diagrams

Information flow diagrams in UML provide fairly meager expressive capabilities.

Information items provide no details about the information they transfer. They do not have features, generalizations, or associations.

2. Activity Diagram

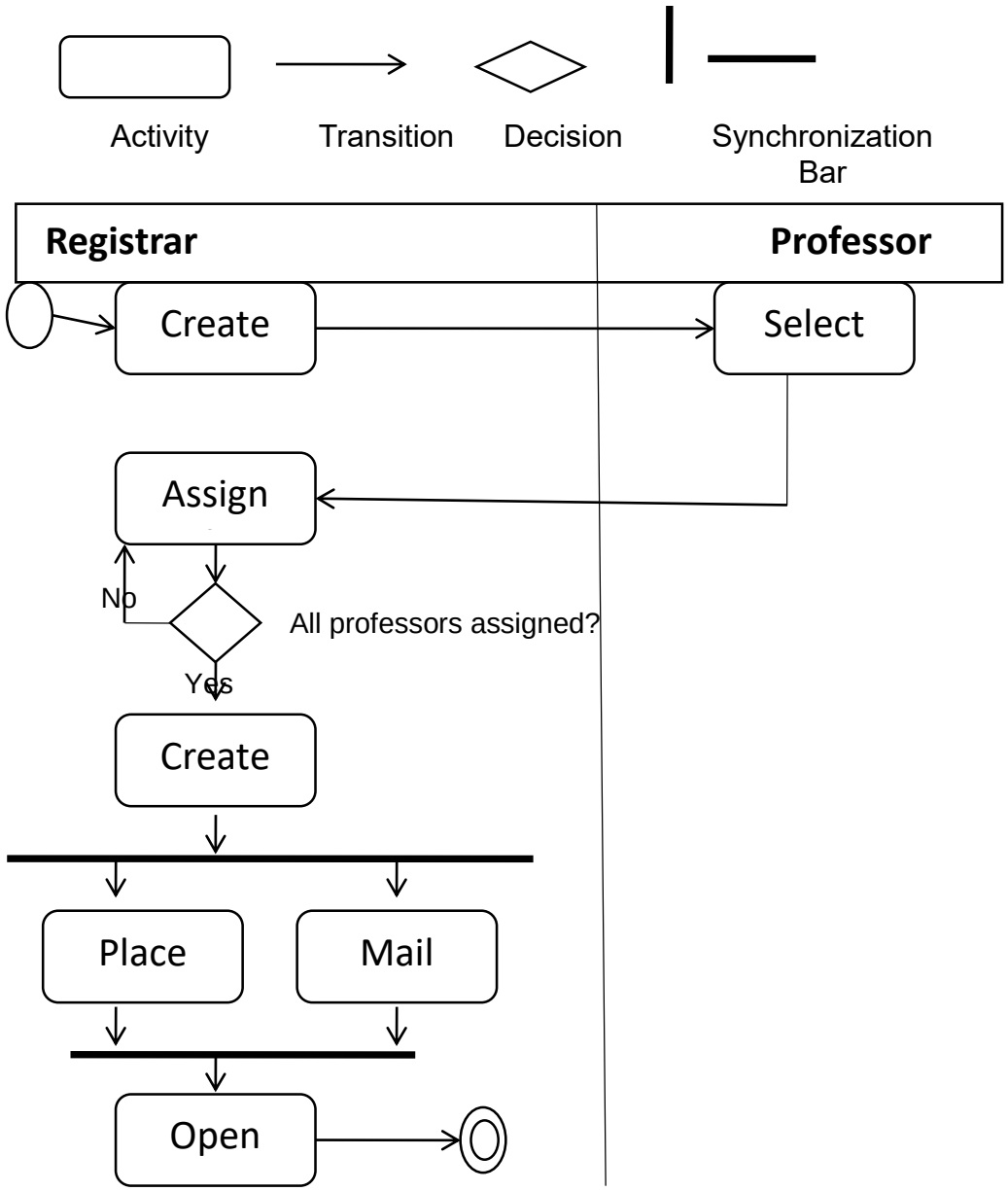
An *activity diagram* shows the flow from **activity to activity** within a system.

An activity shows a **set of activities, the sequential or branching flow** from activity to activity, and objects that act and are acted upon.

Shows what activities can be done in parallel, and any alternate paths through the flow

Activity diagrams contain **activities, transitions between the activities, decision points, and synchronization bars**

Activity diagrams emphasize the **flow of control** among objects.


Activity
Transition
Decision
Synchronization Bar
Registrar
Professor
Create
Select
Assign
No
All professors assigned?
Yes
Create
Place
Mail
Open

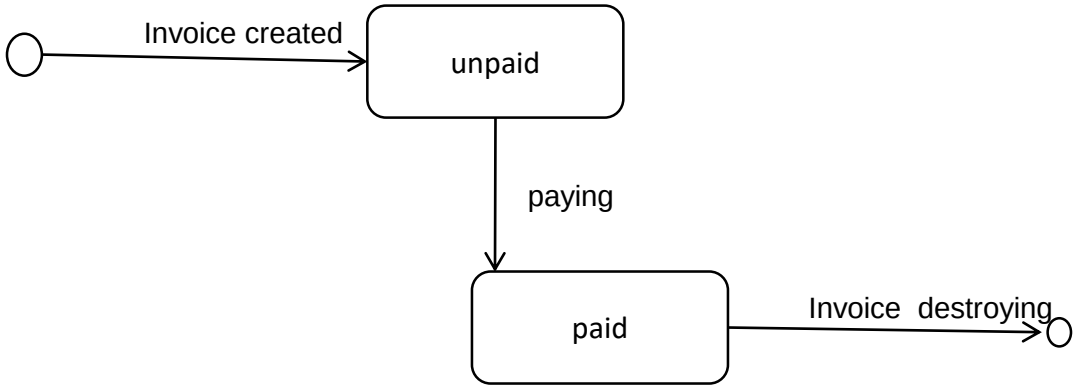
3. State machine diagram

It is used for modeling discrete behavior through finite state transitions.

In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage protocol of part of a system. These two kinds of state machines are referred to as behavioral state machines and protocol state machines.

A *statechart diagram* shows a **state machine**, consisting of states, transitions, events, and activities.

They are especially important in **modeling the behavior** of an interface, class, or collaboration.



Statechart diagrams emphasize the **event-ordered behavior** of an object, which is especially useful in modeling reactive systems.

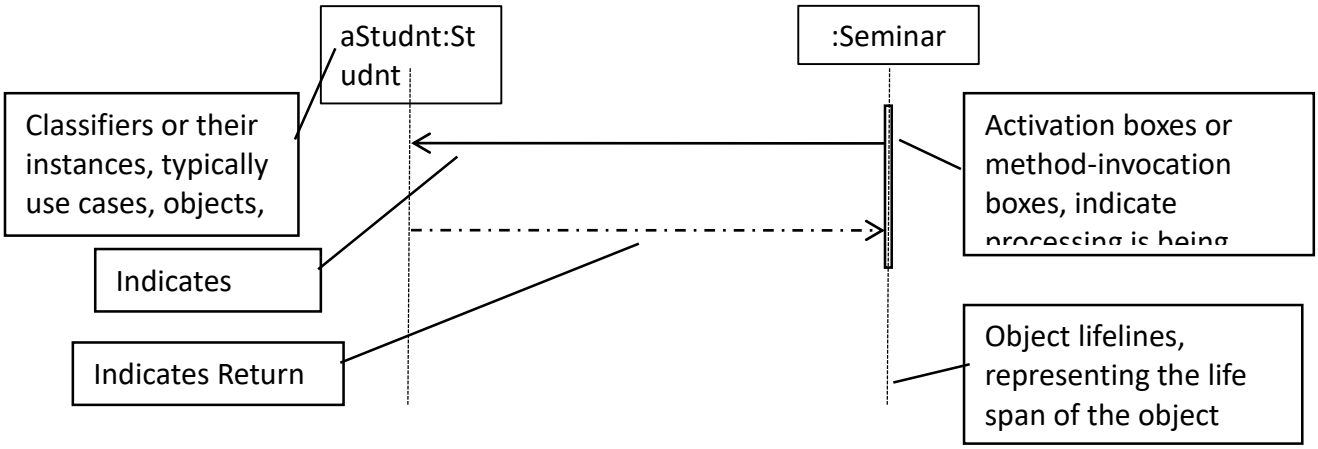
Interaction diagrams include several different types of diagrams:

- sequence diagrams,
- interaction overview diagrams,
- communication diagrams, (known as collaboration diagrams in UML 1.x)
- timing diagrams.

4. Sequence Diagram

A *sequence diagram* is an **interaction diagram** that **emphasizes the time ordering of messages** and shows a set of objects and the messages sent and received by those objects.

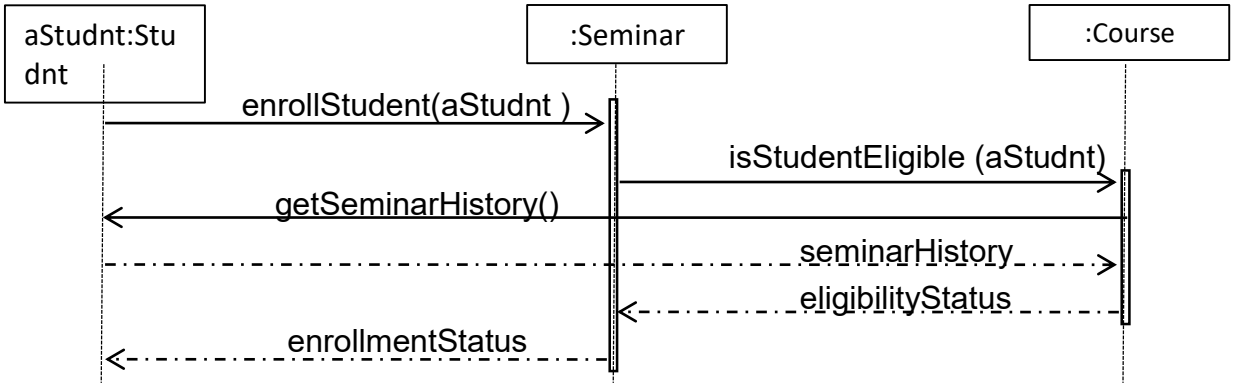
The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.



A *sequence diagram* is an **interaction diagram** that **emphasizes the time ordering of messages.**

A sequence diagram shows a set of objects and the messages sent and received by those objects.

The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.



5. Communication diagram (previously known as Collaboration Diagram)

It is a kind of interaction diagram, which focuses on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central.

The sequencing of messages is given through a sequence numbering scheme.

It is an **interaction diagram** that emphasizes the structural organization of the objects that send and receive messages.

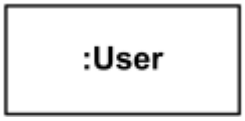
A collaboration diagram **shows a set of objects, links among those objects, and messages** sent and received by those objects.

The objects are typically **named or anonymous instances of classes,** but may also represent instances of other things, such as collaborations, components, and nodes.

Lifeline is a specialization of named element which represents an **individual participant** in the interaction. While parts and structural features may have multiplicity greater than 1, lifelines represent **only one** interacting entity.

If the referenced connectable element is multivalued (i.e, has a multiplicity > 1), then the lifeline may have an expression (**selector**) that specifies which particular

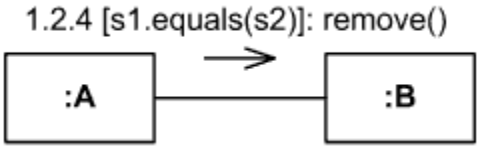
part is represented by this lifeline. If the selector is omitted, this means that an **arbitrary representative** of the multivalued connectable element is chosen.
A **Lifeline** is shown as a rectangle (corresponding to the "head" in sequence diagrams). Lifeline in sequence diagrams does have "tail" representing the **line of life** whereas "lifeline" in **communication diagram** has no line, just "head".
The lifeline head has a shape that is based on the classifier for the part that this lifeline represents. Usually the head is a white rectangle containing name of the class after colon.



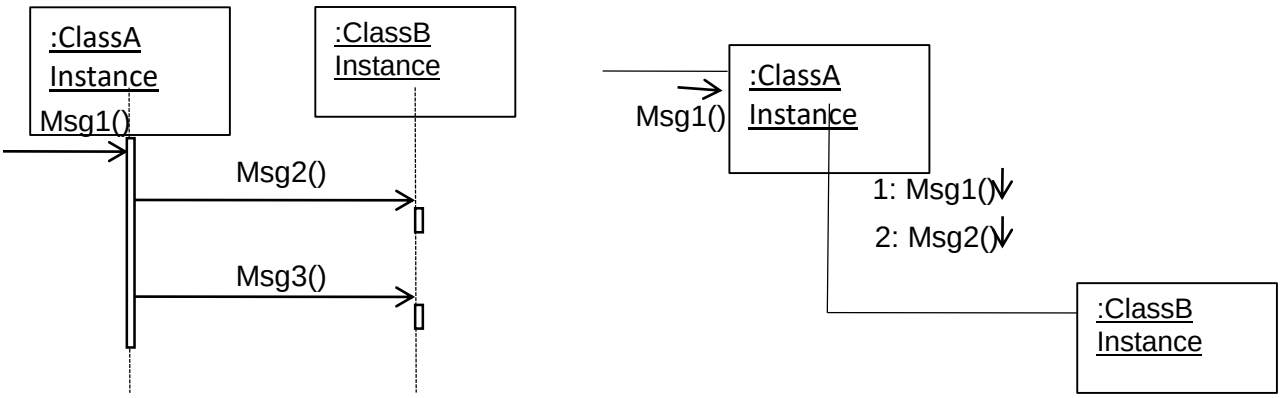
Anonymous lifeline of class User.

Lifeline "data" of class Stock

Message in communication diagram is shown as a line with sequence expression and arrow above the line. The arrow indicates direction of the communication.



Sequence and collaboration diagrams are **isomorphic**, meaning that you can convert from one to the other without loss of information.



Difference between Communication Diagram and Sequence Diagram

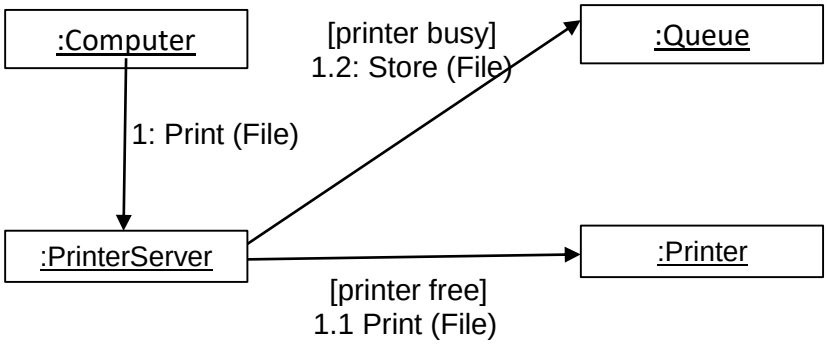
A sequence diagram includes **chronological sequence** of messages

A sequence diagram **does not include object relationships**

Important to **visualize timing order** of messages

Communication diagram is used when you **are interested in the structural relationships** among the instances in an interaction

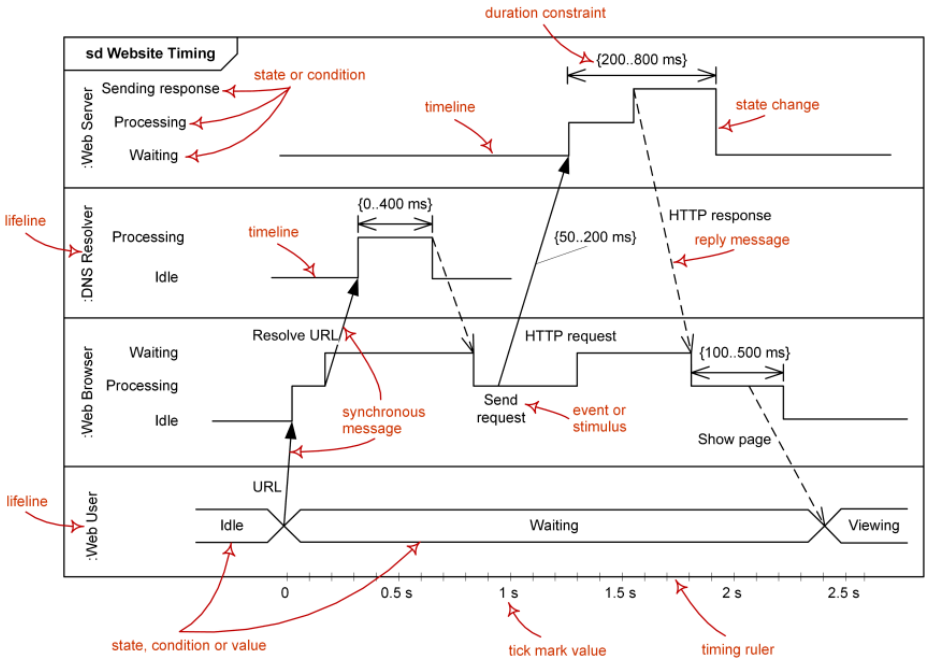
Communication diagram emphasizes on the **organization structure**



6. Timing diagrams

They are used to show interactions when a primary purpose of the diagram is to reason about time and focus on conditions changing within and among Lifelines along a linear time axis.

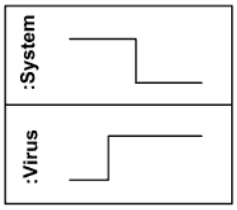
Elements used are lifeline, state or condition timeline, destruction event,duration constraint, time constraint.



Lifeline

Lifeline is a named element which represents an individual participant in the interaction. While parts and structural features may have multiplicity greater than 1, lifelines represent only one interacting entity. See lifeline from sequence diagrams for details.

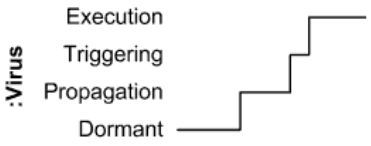
Lifeline on the timing diagrams is represented by the name of classifier or the instance it represents. It could be placed inside diagram frame or a "swimlane".



Lifelines representing instances of System and Virus

State or Condition Timeline

Timing diagram could show states of the participating classifier or attribute, or some testable conditions, such as a discrete or enumerable value of an attribute.



Timeline shows Virus changing its state between Dormant, Propagation, Triggering and Execution state

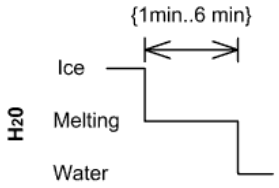
UML also allows the state/condition dimension be continuous. It could be used in scenarios where entities undergo continuous state changes, such as temperature or density.

Duration Constraint

Duration constraint is an interval constraint that refers to a duration interval. The duration interval is duration used to determine whether the constraint is satisfied.

The semantics of a duration constraint is inherited from constraints. If constraints are violated, traces become negative which means that system is considered as failed.

Duration constraint is shown as some graphical association between a duration interval and the constructs that it constrains.



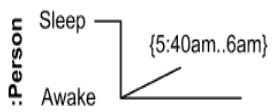
Ice should melt into water in 1 to 6 minutes

Time Constraint

Time constraint is an interval constraint that refers to a time interval. The time interval is time expression used to determine whether the constraint is satisfied.

The semantics of a time constraint is inherited from constraints. All traces where the constraints are violated are negative traces, i.e., if they occur, the system is considered as failed.

Time constraint is shown as graphical association between a time interval and the construct that it constrains. Typically this graphical association is a small line, e.g., between an occurrence specification and a time interval.



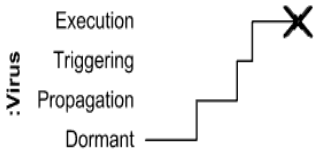
Person should wake up between 5:40 am and 6 am

Destruction Occurrence

Destruction occurrence is a message occurrence which represents the destruction of the instance described by the lifeline. It may result in the subsequent destruction of other objects that this object owns by composition. No other occurrence may appear after the destruction event on a given lifeline.

Notation

The destruction event is depicted by a cross in the form of an X at the end of a timeline.



Virus lifeline is terminated

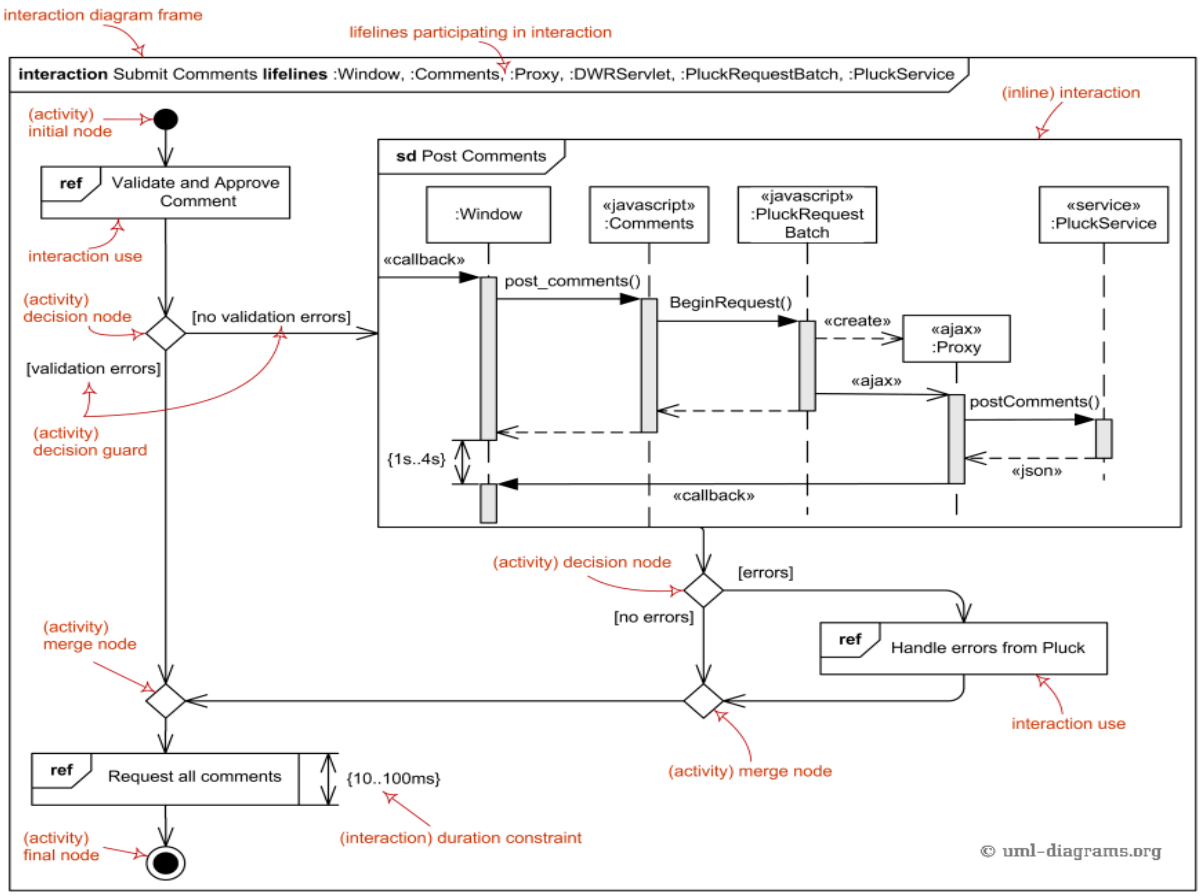
7. Interaction overview diagram

It defines interactions through a variant of activity diagrams in a way that promotes overview of the control flow.

Interaction overview diagrams focus on the overview of the flow of control where the nodes are interactions or interaction uses.

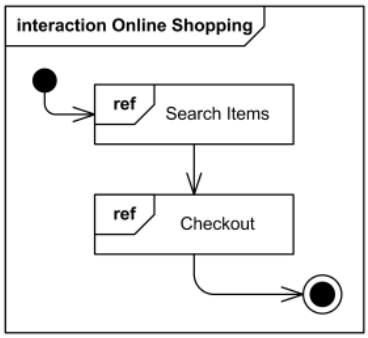
The lifelines and the messages do not appear at this overview level.

UML interaction overview diagram combines elements from activity and interaction diagrams.



Frame

Interaction overview diagrams are framed by the same kind of frame that encloses other forms of interaction diagrams - a rectangular frame around the diagram with a name in a compartment in the upper left corner. Interaction kind is interaction or sd (abbreviated form). Note, that UML has no io or iod abbreviation as some would expect.



Interaction overview diagram Online Shopping

The heading text may also include a list of the contained lifelines (that do not appear graphically).

Elements of Activity Diagram

Interaction overview diagrams are defined as specialization of activity diagrams and as such they inherit number of graphical elements.

Interaction overview diagrams can only have inline interactions or interaction uses instead of actions, and activity diagram actions could not be used.

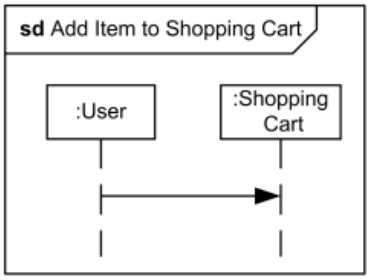
The elements of the activity diagrams used on interaction overview diagrams are initial node, flow final node, activity final node, decision node, merge node, fork node, join node

Elements of Interaction Diagram

The elements of the interaction diagrams used on interaction overview diagrams are interaction, interaction use, duration constraint, time constraint

Interaction

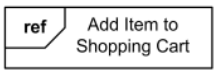
An interaction diagram of any kind may appear inline as an invocation action. The inline interaction diagrams may be either anonymous or named.



Interaction Add Item to Shopping Cart may appear inline on some interaction overview diagram

Interaction Use

An interaction use may appear as an invocation action.



Interaction use Add Item to Shopping Cart may appear on some interaction overview diagram