UNIT-3

SAQ'S & LAQ'S

1 Explain why design is important in design engineering.

The design allows a software engineer to model the system or product that Is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end–users become involved in large numbers. Design is the place where software quality is established.

2 Discuss analysis and design model.

The analysis model describes the structure of the system or application that you are modelling. It consists of class diagrams and sequence diagrams that describe the logical implementation of the functional requirements that you identified in the use case model.

The design model builds on the analysis model by describing, in greater detail, the structure of the system and how the system will be implemented. Classes that were identified in the analysis model are refined to include the implementation constructs.

3 Describe quality attributes.

The FURPS quality attributes represent a target for all software design:

➢ *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
➢ *Usability* is assessed by considering human factors, overall aesthetics, consistency and documentation.
➢ *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, and the mean – time –to- failure (MTTF), the ability to recover from failure, and the predictability of the program.
➢ *Performance* is measured by processing speed, response time, resource consumption, throughput, and efficiency.
➢ *Supportability c*ombines the ability to extend the program (extensibility), adaptability, and serviceability- these three attributes represent a more common term for maintainability.

4 List the design concepts.

Fundamental software design concepts provide the necessary framework for "getting it right." There are:

- Abstraction
- Architecture
- Patterns
- Modularity
- Information Hiding
- Functional Independence
- Refinement
- Refactoring
- Design classes.

5 Justify the importance of refactoring.

Refactoring is a crucial part of software development because it improves the design of existing code without changing its functionality. It can make code easier to read, maintain, and scale.

6 Discuss low coupling.

Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of classes in other subsystems. This restriction, called the law of Demeter, suggests that a method should only sent messages to methods in neighbouring classes.

7 Define software architecture with its importance.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. three key reasons that software architecture is important:

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together."

8 Explain the taxonomy of architectural styles?

Taxonomy of Architectural Styles

- Data centred architectures:
- Data flow architectures:
- Call and Return architectures.
- Object Oriented architecture.
- Layered architecture.

9 Write short notes on architecture patterns?

An architectural pattern is a general, reusable resolution to a commonly occurring problem in software architecture within a given context. The architectural patterns address various issues in software engineering, such as computer hardware performance limitations, high availability, and minimization of a business risk.

10 Compare function-oriented and object-oriented design.

COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions, which are given to the user, are real world functions.	The basic abstractions are not the real-world functions but are the data abstraction where the real world entities are represented.
Function	Functions are grouped together by which a higher-level function is obtained.	Function are grouped together on the basis of the data they operate since the classes are associated with their methods.
execute	carried out using structured analysis and structured design i.e, data flow diagram	Carried out using UML
State information	In this approach the state information is often represented in a centralized shared memory.	In this approach the state information is not represented in a centralized memory but is implemented or distributed among the objects of the system.
Approach	It is a top-down approach.	It is a bottom-up approach.
Begins basis	Begins by considering the use case diagrams and the scenarios.	Begins by identifying objects and classes.
Decompose	In function-oriented design we decompose in function/procedure level.	We decompose in class level.

12 Write short notes on coupling.
Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates throughout a system.

13 Write short notes on cohesion.

   Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.

14. Explain Design Process and goals of Design.

Design Process:

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.

Goals of design:

McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design.

•The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

•The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

•The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

15. what is a Design Engineering?

   Design engineering encompasses the **set of principals, concepts, and practices** that lead to the development of a high- quality system or product.

   ✓ Design principles establish an overriding philosophy that guides the designer in the work that is performed.

   ✓ Design concepts must be understood before the mechanics of design practice are applied and

   ✓ Design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

16. What is design:

Design is what almost every engineer wants to do. It is the place where creativity rules –customer's requirements, business needs, and technical considerations all come together in the formulation of a product or a system. Design creates a representation or model of the software, but unlike the analysis model, the design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system.

17. what is UML?

UML (Unified Modeling Language) is a general-purpose, graphical modeling language in the field of Software Engineering. UML is used to specify, visualize, construct, and document the artifacts (major elements) of the software system. It helps in designing and characterizing, especially those software systems that incorporate the concept of Object orientation. It describes the working of both the software and hardware systems. It was initially developed by Grady Booch, Ivar Jacobson, and James Rumbaugh in 1994-95 at Rational software, and its further development was carried out through 1996. In 1997, it got adopted as a standard by the Object Management Group.

18.What are the goals of UML?

Goals of UML

- o Since it is a general-purpose modeling language, it can be utilized by all the modelers.
- o UML came into existence after the introduction of object-oriented concepts to systemize and consolidate the object-oriented development, due to the absence of standard methods at that time.
- o The UML diagrams are made for business users, developers, ordinary people, or anyone who is looking forward to understand the system, such that the system can be software or non-software.
- o Thus it can be concluded that the UML is a simple modeling approach that is used to model all the practical systems.

19. What are the characteristics of UML
Characteristics of UML
The UML has the following features:
- o It is a generalized modeling language.
- o It is distinct from other programming languages like C++, Python, etc.
- o It is interrelated to object-oriented analysis and design.
- o It is used to visualize the workflow of the system.
- o It is a pictorial language, used to generate powerful modeling artifacts.

LAQ'S

1 Discuss briefly the following fundamental concepts of software design:
i) Abstraction
ii) Modularity
iii) Information hiding

Abstraction

Many levels of abstraction are there.
- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction, a more detailed description of the solution is provided.

As different levels of abstraction are developed, you work to create **both procedural and data abstractions.**

A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of procedural abstraction implies these functions, but specific details are suppressed.

A *data abstraction* is a named collection of data that describes a data object.
In the context of the procedural abstraction *open*, we can define a data abstraction called **door.**
Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing operation, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door.**

Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

It has been stated that ***"modularity is the single attribute of software that allows a program to be intellectually manageable".***

Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

The "divide and conquer" strategy- it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to modularity and software. If we subdivide software indefinitely, the effort requiredto develop it will become negligibly small. The effort to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort associated with integrating the modules also grow.

Information Hiding:

The principle of *information hiding* suggests that modules be "characterized by design decision that hides from all others. "Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and the local data structure used by the module. The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

2 Briefly explain the following:
i) Coupling between the modules,
ii) The internal Cohesion of a module

**i)   Coupling between the modules,**

Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other

modules. Low coupling means that modules are independent, and changes in one module have little impact on other modules.

Coupling

Module 1 — Module 2

Module 3

Coupling refers to the degree
of interdependence between modules
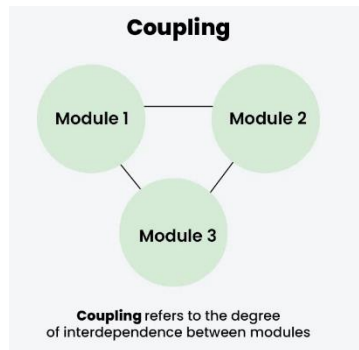
ii) The internal Cohesion of a module

Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.

Cohesion

E1 — E2

E3

Cohesion refers to the degree to which elements within a
module work together to fulfill a single, well-defined purpose
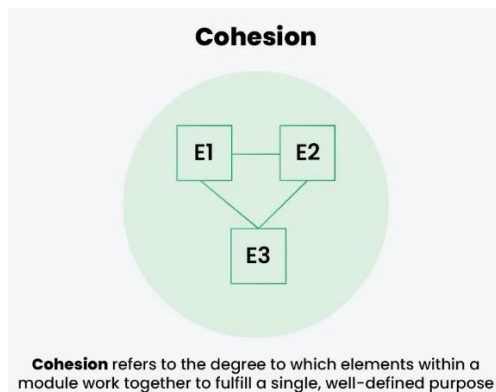
Both coupling and cohesion are important factors in determining the maintainability, scalability, and reliability of a software system. High coupling and low cohesion can make a system difficult to change and test, while low coupling and high cohesion make a system easier to maintain and improve.

3. **Describe quality attributes and its guidelines?**

Design Quality:

In software engineering, design quality is a measure of how well a software system is designed and conforms to that design. It is often described as the "fitness for purpose" of a piece of software. Design quality attributes are the characteristics of a software system that affect its performance, reliability, usability, security, and maintainability. They are often defined by the stakeholders of the system, such as users, customers, developers, and managers.

Quality guidelines:

To evaluate the quality of a design representation we must establish technical criteria for good design. These are the following guidelines:

1. A design should exhibit an architecture that
   - has been created using recognizable architectural styles or patterns,
   - is composed of components that exhibit good design characteristics

- can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5. A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communicates its meaning.

Quality attributes:

The FURPS quality attributes represent a target for all software design:

- ➢ *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ➢ *Usability* is assessed by considering human factors, overall aesthetics, consistency and documentation.
- ➢ *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, and the mean – time –to- failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ➢ *Performance* is measured by processing speed, response time, resource consumption, throughput, and efficiency.
- ➢ *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability- these three attributes represent a more common term maintainability.

4. **Explain software architecture in a detail manner?**

Software Architecture:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not operational software. Rather, it is a representation that enables you to

- analyze the effectiveness of the design in meeting its stated requirements,
- consider architectural alternatives at a stage when making design changes is still relatively easy, and
- reduce the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components.

Why Is Architecture Important?
Bass and his colleagues [Bas03] identify three key reasons that software architecture is important:
- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together."

5. **Discuss analysis and design model? Or Explain Design Concepts**

The analysis model describes the structure of the system or application that you are modelling. It consists of class diagrams and sequence diagrams that describe the logical implementation of the functional requirements that you identified in the use case model.

The design model builds on the analysis model by describing, in greater detail, the structure of the system and how the system will be implemented. Classes that were identified in the analysis model are refined to include the implementation construct**s.**

The design model can be viewed in two different dimensions.
The process dimension that indicates the evolution of the design model as design
tasks are executed as part of the software process.
The abstraction dimension that represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

Referring to Figure 8.4, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

FIGURE 8.4   Dimensions of the design model

FIGURE 8.4 Dimensions of the design model
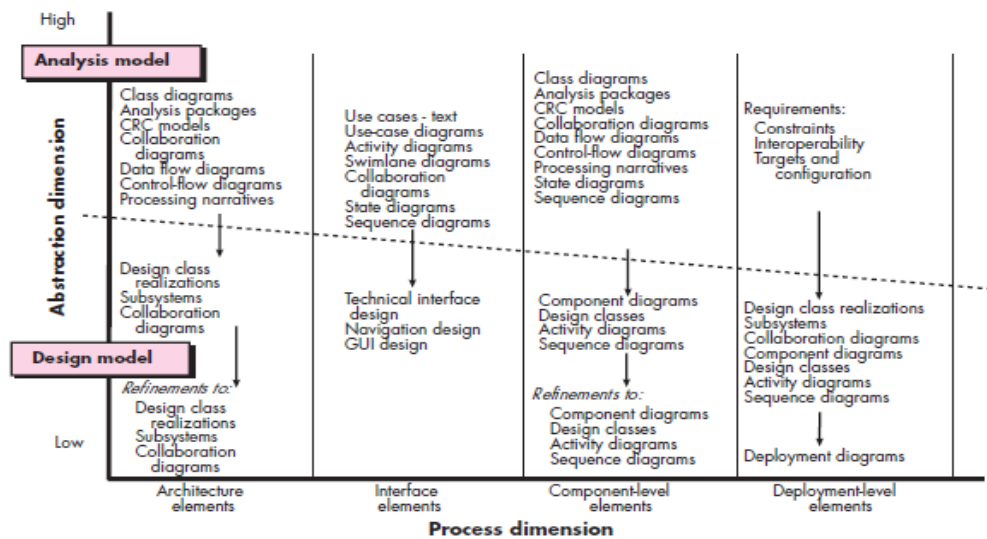
Data Design Elements

Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it. The structure of data has always been an important part of software design.

- At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.
- At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Architectural Design Elements

The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software. The architectural model is derived from three sources:

- information about the application domain for the software to be built;
- specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and
- the availability of architectural styles and patterns.

Interface Design Elements

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they
operate, the way in which utility connections (e.g., water, electrical, gas, telephone)
come into the house and are distributed among the rooms depicted in the floor plan.
They tell us where the doorbell is located, whether an intercom is to be used to announce a visitor's presence, and how a security system is to be installed. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design:
- the user interface (UI);
- external interfaces to other systems, devices, networks, or other producers or consumers of information; and
- internal interfaces between various design components.

Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room. The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all
component operations (behaviors).

Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 8.6. In this figure, a component named SensorManagement (part of the SafeHome security function) is
represented. A dashed arrow connects the component to a class named Sensor that
is assigned to it. The SensorManagement component performs all functions associated with SafeHome sensors including monitoring and configuring them.

FIGURE 8.6
A UML
component
diagram

Deployment-Level Design Elements
Deployment-level design elements indicate how software functionality and subsystems
will be allocated within the physical computing environment that will support the software.
During design, a UML deployment diagram is developed and then refined as shown in Figure 8.7.

FIGURE 8.7
A UML deployment diagram
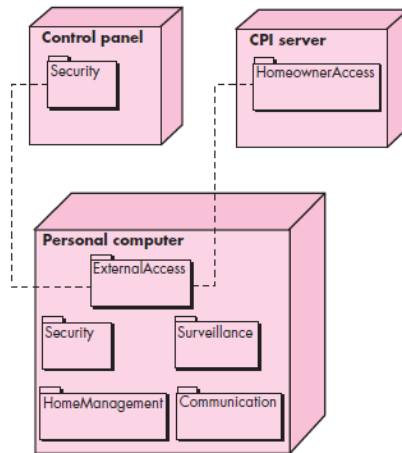
6. **List the design concepts.**

   Design Concepts:
   M.A Jackson once said:" The beginning of wisdom for a software engineer is to recognize the
   difference between getting a program to work, and getting it right." Fundamental software design
   concepts provide the necessary framework for "getting it right."

   Abstraction:
   Many levels of abstraction are there.
   - At the highest level of abstraction, a solution is stated in broad terms using the language of the
     problem environment.
   - At lower levels of abstraction, a more detailed description of the solution is provided.

   As different levels of abstraction are developed, you work to create **both procedural and data
   abstractions.**

   A *procedural abstraction* refers to a sequence of instructions that have a specific and limited
   function. The name of procedural abstraction implies these functions, but specific details are
   suppressed.

   A *data abstraction* is a named collection of data that describes a data object.
   In the context of the procedural abstraction *open*, we can define a data abstraction called **door.**
   Like any data object, the data abstraction for **door** would encompass a set of attributes that describe
   the door (e.g., door type, swing operation, opening mechanism, weight, dimensions). It follows
   that the procedural abstraction *open* would make use of information contained in the attributes of

the data abstraction **door.**

Architecture:
Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

One **goal** of software design is to derive an architectural rendering of a system. The rendering serves as a framework from which more detailed design activities are conducted.

The architectural design can be represented using one or more of a number of different models.
- *Structured models* represent architecture as an organized collection of program components.
- *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
- *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function external event.
- *Process models* focus on the design of the business or technical process that the system must accommodate.
- *Functional models* can be used to represent the functional hierarchy of a system.

Patterns

A pattern is a named suggest of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns. in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine
(1) whether the pattern is applicable to the current work,
(2) whether the pattern can be reused (hence, saving design time), and
(3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Modularity
Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

It has been stated that *"modularity is the single attribute of software that allows a program to be intellectually manageable"*

Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you

should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

The "divide and conquer" strategy- it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to modularity and software. If we subdivide software indefinitely, the effort required to develop it will become negligibly small. The effort to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort associated with integrating the modules also grow.

Information Hiding:

The principle of *information hiding* suggests that modules be "characterized by design decision that hides from all others."Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and local data structure used by module. The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within software.

Functional Independence:

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding. *Functional independence* is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure. Software with effective modularity, that is, independent modules, is easier to develop because functions may be compartmentalized and interfaces are simplified. Independent sign or code modifications are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

 Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information hiding. A cohesion module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should do just one thing. Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling.

Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates throughout a system.

Refinement:

Stepwise refinement is a top- down design strategy originally proposed by Niklaus wirth. A program is development by successively refining levels of procedural detail. A hierarchy is development by decomposing a macroscopic statement of function in a step wise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. We begin with a statement of function that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement occurs.Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

Refactoring:

Refactoring is a reorganization technique that simplifies the design of a component without changing its function or behavior. Fowler defines refactoring in the following manner: "refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. The designer may decide that the component should be refactored into 3 separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Object-oriented Design Concepts-Design classes:

The software team must define a set of design classes that Refine the analysis classes by providing design detail that will enable the classes to be implemented, and Create a new set of design classes that implement a software infrastructure to support the design solution.

Five different types of design classes, each representing a different layer of the design architecture are suggested.

User interface classes: define all abstractions that are necessary for human computer interaction. In many cases, HCL occurs within the context of a *metaphor* and the design classes for the interface may be visual representations of the elements of the metaphor.

Business domain classes: are often refinements of the analysis classes defined earlier. The classes identify the attributes and services that are required to implement some element of the business domain.

Process classes implement lower–level business abstractions required to fully manage the business domain classes.

Persistent classes represent data stores that will persist beyond the execution of the software.

System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the design model evolves, the software team must develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation. Each design class be reviewed to ensure that it is "well-formed."

7. **Explain about architectural styles and patterns?**

Architectutal Styles:
The software that is built for computer-based systems also exhibits one of many architectural styles.
Each style describes a system category that encompasses.
(1) a set of components (e.g., a database, computational modules) that perform a function required by a system.
(2) a set of connectors that enable "communication, coordination and cooperation" among components.
(3) constraints that define how components can be integrated to form the system; and
(4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts [Bas03].
An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

An architectural pattern, like an architectural style, imposes a transformation on
the design of an architecture. However, a pattern differs from a style in a number of
fundamental ways:
(1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;
(2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency) [Bos00];
(3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

A Brief Taxonomy of Styles and Patterns

Data-centered architectures: A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 9.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

Data-centered architectures promote integrability. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

FIGURE 9.1
Data-centered architecture
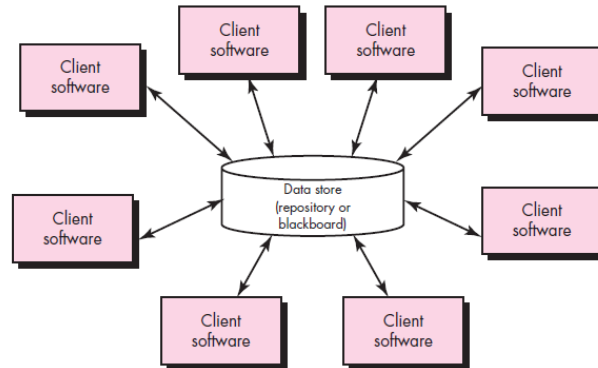
Data-flow architectures: This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 9.2) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters. If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

Call and return architectures: This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles [Bas03] exist within this category:
• Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components.
• Remote procedure call architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.

Figure 9.2

Data-flow
architecture

Pipes

Filter Filter

Filter Filter Filter Filter

Filter Filter Filter

Filter

Pipes and filters
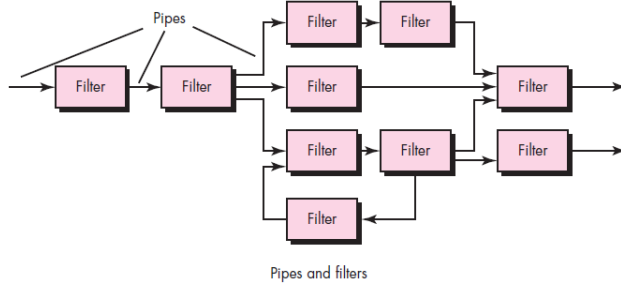
Figure 9.3 Main program/subprogram architecture

Main program

Controller subprogram

Controller subprogram

Controller subprogram

Application subprogram

Application subprogram

Application subprogram

Application subprogram

Application subprogram

Application subprogram

Application subprogram
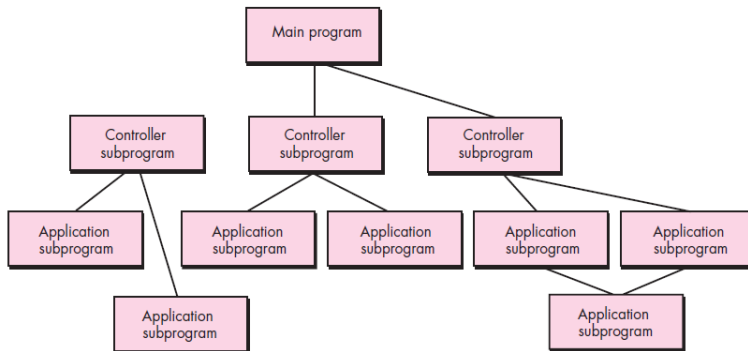
Object-oriented architectures: The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 9.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

Figure 9.4

Layered
architecture

Components

User interface layer

Application layer

Utility layer

Core layer
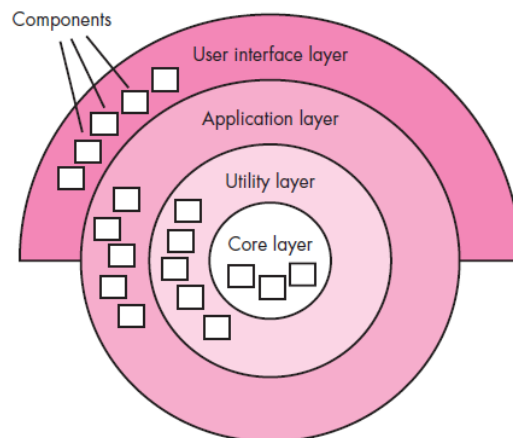
8. Explain about refining the architecture into components

Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? To answer this question, you begin with the classes that were described as part of the requirements model. These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed. Continuing the Safe Home home security function example, you might define the set of top-level components that address the following functionality:

• External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
• Control panel processing—manages all control panel functionality.
• Detector management—coordinates access to all detectors attached to the system.
• Alarm processing—verifies and acts on all alarm conditions.

FIGURE 9.8 Overall architectural structure for *SafeHome* with top-level components
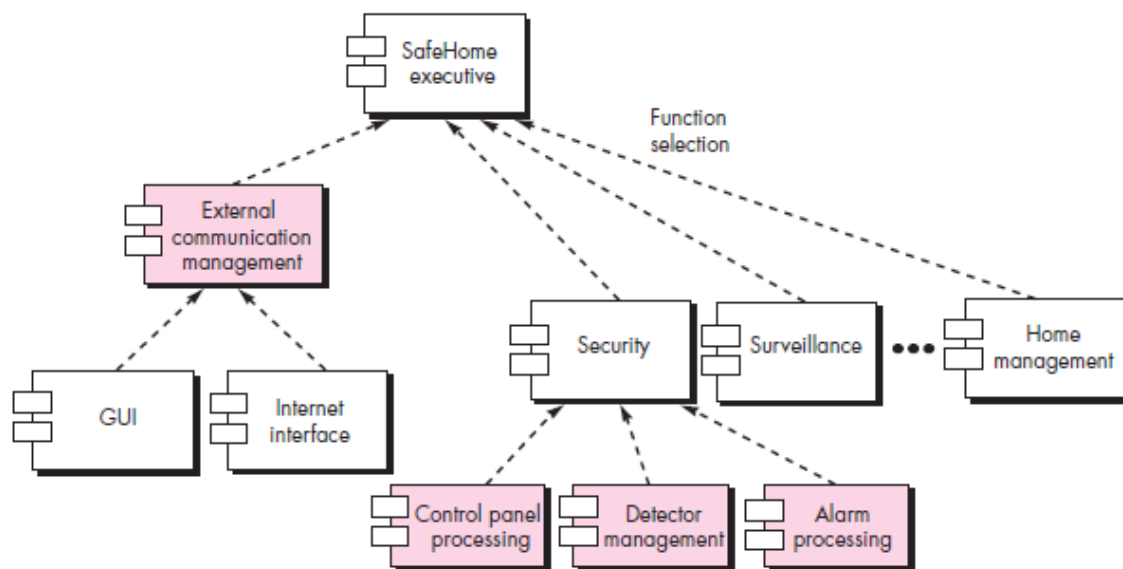
The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 9.8. Transactions are acquired by external communication management as they move in from components that process the SafeHome GUI and the Internet interface. This information is managed by a SafeHome executive component that selects the appropriate product function (in this case security). The control panel processing component interacts with the homeowner to arm/disarm the security function. The detector management component polls sensors to detect an alarm condition, and the alarm processing component produces output when an alarm is detected.

9. Explain about Archetypes

Archetypes

An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the Safe Home home security function, you might define the following archetypes:

• **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of
(1) various sensors and
(2) a variety of alarm (output) indicators.

• **Detector**. An abstraction that encompasses all sensing equipment that feeds information into the target system.

• **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

• **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

Each of these archetypes is depicted using UML notation as shown in Figure 9.7.
Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. For example, Detector might be refined into a class hierarchy of sensors.

FIGURE 9.7

UML relation-
ships for
SafeHome
security
function
archetypes
Source: Adapted from
[Bos00].
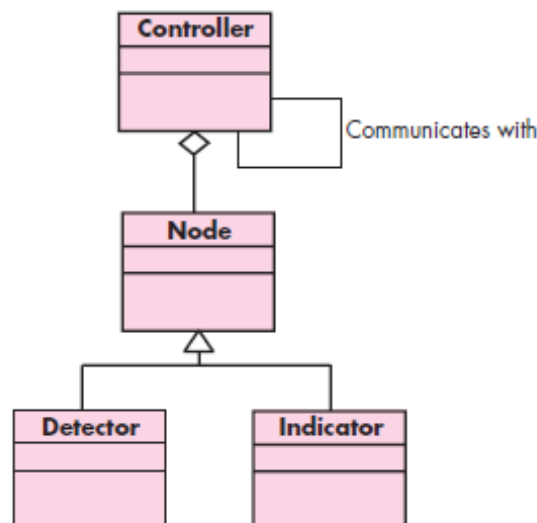
10.Explain about Architectural Design

Architectural Design

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once the context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. An archetype is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

Representing the System in Context

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 9.5. Referring to the figure, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as

• **Superordinate systems**—those systems that use the target system as part of some higher-level processing scheme.

• **Subordinate systems**—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

• **Peer-level systems**—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.

• **Actors**—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

Figure 9.5
Architectural context diagram
Source: Adapted from [Bos00].
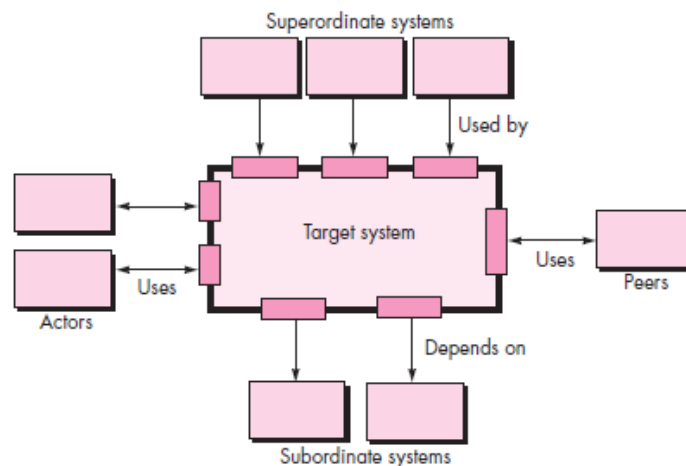
11.Describe Instantiations of the System in context to architectural design

Describing Instantiations of the System
The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this It mean that the architecture is applied to a specific problem with the intent of demonstrating
that the structure and components are appropriate.

FIGURE 9.9 An instantiation of the security function with component elaboration
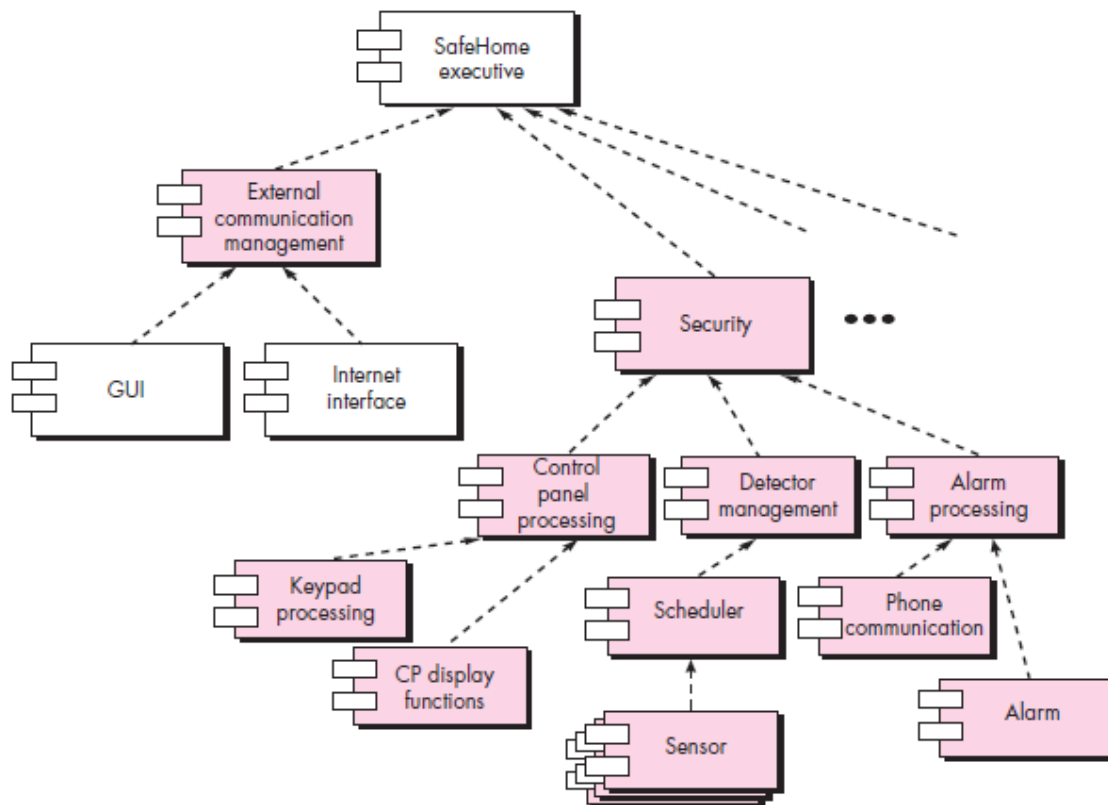
Figure 9.9 illustrates an instantiation of the SafeHome architecture for the security system. Components shown in Figure 9.8 are elaborated to show additional detail. For example, the detector management component interacts with a scheduler infrastructure component that implements polling of each sensor object used by the security system. Similar elaboration is performed for each of the components represented in Figure 9.8.

12.Explain conceptual model of UML.
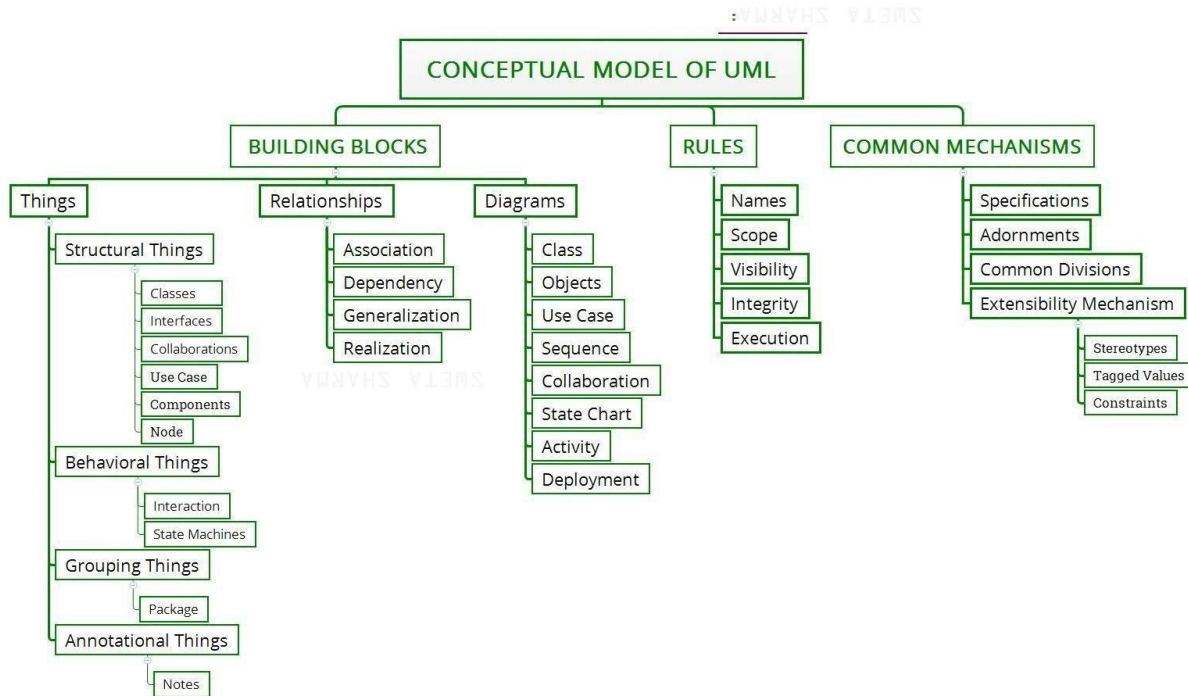
UML-Building Blocks

UML is composed of three main building blocks, i.e., things, relationships, and diagrams. Building blocks generate one complete UML model diagram by rotating around several different blocks. It plays an essential role in developing UML diagrams. The basic UML building blocks are enlisted below:

1. Things
2. Relationships
3. Diagrams

THINGS

Anything that is a real world entity or object is termed as things. It can be divided into several different categories:

o Structural things
o Behavioral things
o Grouping things
o Annotational things

Structural things

Nouns that depicts the static behavior of a model is termed as structural things. They display the physical and conceptual components. They include class, object, interface, node, collaboration, component, and a use case.

Class: A Class is a set of identical things that outlines the functionality and properties of an object. It also represents the abstract class whose functionalities are not defined. Its notation is as follows;

Class-name
+class-attributes
+class-functions()
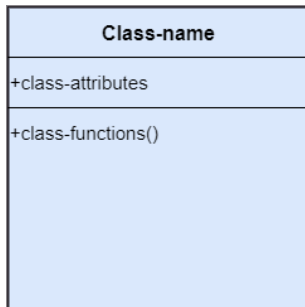
Object:: An individual that describes the behavior and the functions of a system. The notation of the object is similar to that of the class; the only difference is that the object name is always underlined and its notation is given below;

<u>Object-name</u>
+object-attributes
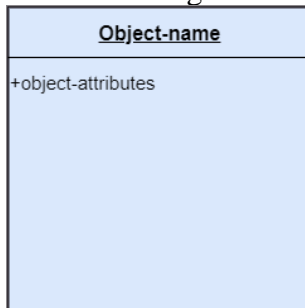
Interface: A set of operations that describes the functionality of a class, which is implemented whenever an interface is implemented.
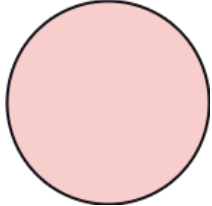
Collaboration: It represents the interaction between things that is done to meet the goal. It is symbolized as a dotted ellipse with its name written inside it.

collaboration-name
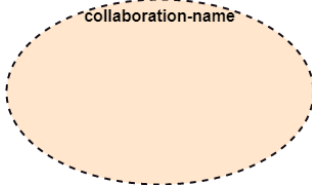
Use case: Use case is the core concept of object-oriented modeling. It portrays a set of actions executed by a system to achieve the goal.

UseCase-name
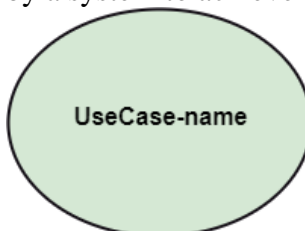
Actor: It comes under the use case diagrams. It is an object that interacts with the system, for example, a user.

Actor
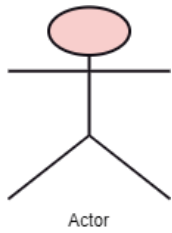
Component: It represents the physical part of the system.

Component-name
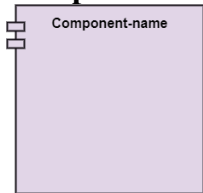
Node: A physical element that exists at run time.

Node-name
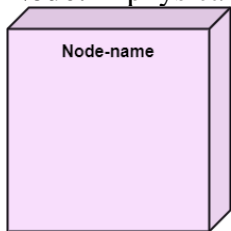
Behavioral Things

They are the verbs that encompass the dynamic parts of a model. It depicts the behavior of a system. They involve state machine, activity diagram, interaction diagram, grouping things, annotation things

State Machine: It defines a sequence of states that an entity goes through in the software development lifecycle. It keeps a record of several distinct states of a system component.

Initial state

State1   State-box

Decision-box
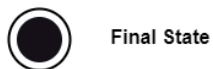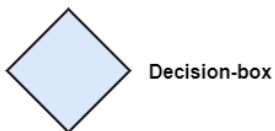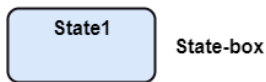
Final State

Activity Diagram: It portrays all the activities accomplished by different entities of a system. It is represented the same as that of a state machine diagram. It consists of an initial state, final state, a decision box, and an action notation.

Initial state

Action box

Decision-box

Final State
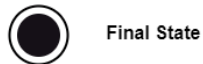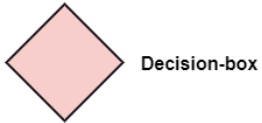
Interaction Diagram: It is used to envision the flow of messages between several components in a system.
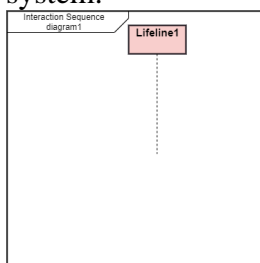
Grouping Things

It is a method that together binds the elements of the UML model. In UML, the package is the only thing, which is used for grouping.

Package: Package is the only thing that is available for grouping behavioral and structural things.
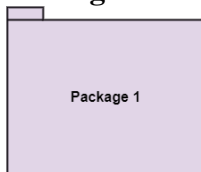
Annotation Things

It is a mechanism that captures the remarks, descriptions, and comments of UML model elements. In UML, a note is the only Annotational thing.

Note: It is used to attach the constraints, comments, and rules to the elements of the model. It is a kind of yellow sticky note.
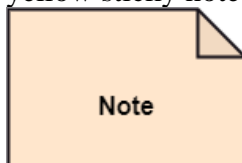
RELATIONSHIPS

1. Association

An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class. Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.
Let's understand association using an example:
Let's consider a simple system for managing a library. In this system, we have two main entities: Book and Library. Each Library contains multiple Books, and each Book belongs to a specific Library. This relationship between Library and Book represents an association.
The "Library" class can be considered the source class because it contains a reference to multiple instances of the "Book" class. The "Book" class would be considered the target class because it belongs to a specific library.

Each Library contains multiple Books, and each Book belongs to a specific Library

2.Dependency Relationship

A dependency exists between two classes when one class relies on another, but the relationship is not as strong as association or inheritance. It represents a more loosely coupled connection between classes. Dependencies are often depicted as a dashed arrow.
Let's consider a scenario where a Person depends on a Book.

- **Person Class:** Represents an individual who reads a book. The Person class depends on the Book class to access and read the content.
- **Book Class:** Represents a book that contains content to be read by a person. The Book class is independent and can exist without the Person class.

The Person class depends on the Book class because it requires access to a book to read its content. However, the Book class does not depend on the Person class; it can exist independently and does not rely on the Person class for its functionality.

The Person class depends on the Book class because it requires access to a book to read its content

3.Generalization(Inheritance)

Inheritance represents an "is-a" relationship between classes, where one class (the subclass or child) inherits the properties and behaviors of another class (the superclass or parent). Inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.

In the example of bank accounts, we can use generalization to represent different types of accounts such as current accounts, savings accounts, and credit accounts.

The Bank Account class serves as the generalized representation of all types of bank accounts, while the subclasses (Current Account, Savings Account, Credit Account) represent specialized versions that inherit and extend the functionality of the base class.

4.Realization (Interface Implementation)

Realization indicates that a class implements the features of an interface. It is often used in cases where a class realizes the operations defined by an interface. Realization is depicted by a dashed line with an open arrowhead pointing from the implementing class to the interface.

Let's consider the scenario where a "Person" and a "Corporation" both realizing an "Owner" interface.

- **Owner Interface:** This interface now includes methods such as "acquire(property)" and "dispose(property)" to represent actions related to acquiring and disposing of property.
- **Person Class (Realization):** The Person class implements the Owner interface, providing concrete implementations for the "acquire(property)" and "dispose(property)" methods. For instance, a person can acquire ownership of a house or dispose of a car.
- **Corporation Class (Realization):** Similarly, the Corporation class also implements the Owner interface, offering specific implementations for the "acquire(property)" and

"dispose(property)" methods. For example, a corporation can acquire ownership of real estate properties or dispose of company vehicles.

Both the Person and Corporation classes realize the Owner interface, meaning they provide concrete implementations for the "acquire(property)" and "dispose(property)" methods defined in the interface.

In this design, any instance of a Person or Corporation can be treated as an Owner, meaning they can acquire and dispose of property. This allows for polymorphic behavior where functions or methods can operate on an Owner type, regardless of whether it is a Person or a Corporation.

DIAGRAMS

The diagrams are the graphical implementation of the models that incorporate symbols and text. Each symbol has a different meaning in the context of the UML diagram. There are thirteen different types of UML diagrams that are available in UML 2.0, such that each diagram has its own set of a symbol. And each diagram manifests a different dimension, perspective, and view of the system.

UML diagrams are classified into three categories that are given below:
1. Structural Diagram
2. Behavioral Diagram
3. Interaction Diagram

Structural Diagram: It represents the static view of a system by portraying the structure of a system. It shows several objects residing in the system. Following are the structural diagrams given below:
- Class diagram
- Object diagram
- Package diagram
- Component diagram
- Deployment diagram

Behavioral Diagram: It depicts the behavioral features of a system. It deals with dynamic parts of the system. It encompasses the following diagrams:
- Activity diagram
- State machine diagram
- Use case diagram

Interaction diagram: It is a subset of behavioral diagrams. It depicts the interaction between two objects and the data flow between them. Following are the several interaction diagrams in UML:
- Timing diagram

- o Sequence diagram
- o Collaboration diagram

RULES

The UML has several rules that specify what a well-formed model should look like. A well-formed model is semantically self-consistent and in harmony with all its related models. The UML has semantic rules for:

1. **Names** – What you can call things, relationships, and diagrams.
2. **Scope** – The context that gives specific meaning to a name.
3. **Visibility** – How those names can be seen and used by others.
4. **Integrity** – How things properly and consistently relate to one another.
5. **Execution** – What it means to run or simulate a dynamic model.

Common Mechanisms

UML has four common mechanisms −
- Specifications
- Adornments
- Common Divisions
- Extensibility Mechanisms

Specifications

Behind every graphical notation in UML, there is a precise specification of the details that element represents. For example, a class icon is a rectangle and it specifies the name, attributes and operations of the class

Adornments

The mechanism in UML that allows the users to specify extra information with the basic notation of an element is the adornments.

- Textual/graphical items added to the basic notation of an element
- They are used for explicit visual representation of those aspects of an element that are beyond the most important

Examples: The basic notation of association is line, but this could be adorned with additional details, such as the role names and multiplicity of each end

Similarly, a class notation may highlight most important aspects of a class, i.e., name, attributes and operations. To show access specifies for the attributes and methods of a class adornments such as +, -, # are used.

In the above example, the access specifiers: + (public), # (protected) and – (private) represent the visibility of the attributes which is extra information over the basic attribute representation.

Common Divisions

In UML there is clear division between semantically related elements like: separation between a class and an object and the separation between an interface and its implementation.

- In modeling, object-oriented systems get divided in multiple ways.
- For example, class vs. object, interface vs. implementation
- An object uses the same symbol as its class with its name underlined

Object-oriented systems can be divided in many ways. The two common ways of division are −
- **Division of classes and objects** − A class is an abstraction of a group of similar objects. An object is a concrete instance that has actual existence in the system.
- **Division of Interface and Implementation** − An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.

Extensibility Mechanisms

Extensibility mechanisms allow extending the language in controlled ways. It includes **Stereotypes, Tagged Values and Constraints**

Stereotypes

- Stereotypes are used to create new building blocks from existing blocks
- New building blocks are domain-specific
- Stereotypes are used to extend the vocabulary of a system
- Graphically represented as a name enclosed by guillemets (« »)

Tagged Values

- Tagged values are used to add to the information of the element (not of its instances)
- Stereotypes help to create new building blocks, whereas tagged values help to create new attributes
- These are commonly used to specify information relevant to code generation, configuration management and so on

Constraints

- Constraints are used to create rules for the model
- Rules that impact the behavior of the model, and specify conditions that must be met
- Can apply to any element in the model, i.e., attributes of a class, relationship
- Graphically represented as a string enclosed by braces {....} and placed near the associated elements or connected to that elements by dependency relationships

1. **Explain the Basic Structural Modelling in UML**

Basic Structural Modeling
1. **Classes**
2. **Relationships**
3. **Common Mechanisms**
4. **Diagrams**

1. Classes:

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle.
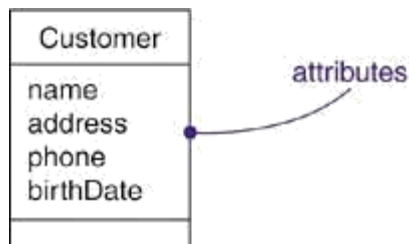
Every class must have a name that distinguishes it from other classes. A *name* is a textual string. That name alone is known as a simple name; a qualified-name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name.
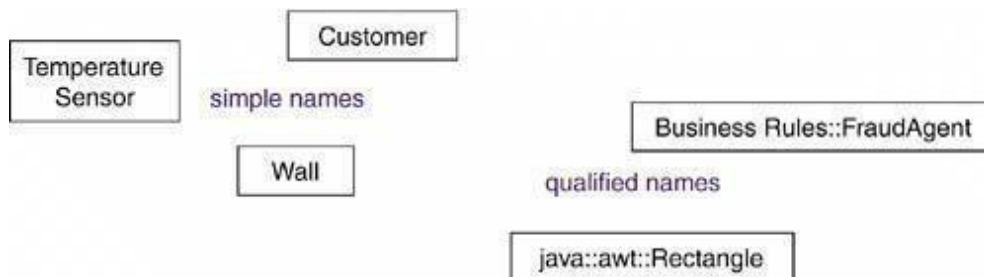
Attributes

An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. For example, every wall has a height, width, and thickness;you might model your customers in such a way that each has a name, address, phone number, and date of birth

Operations

An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object that is shared by all objects of that class. A class may have any number of operations or no operations at all. For example, in a

windowing library such as the one found in Java's awt package, all objects of the class Rectangle can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names
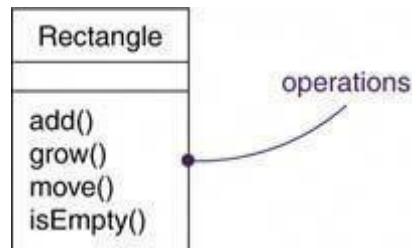

You can specify an operation by stating its signature, which includes the name, type, and default value of all parameters and (in the case of functions) a return type
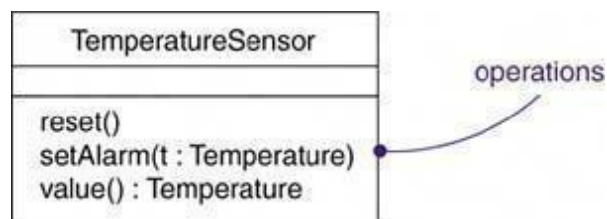
Organizing Attributes and Operations

When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations. You can indicate that there are more attributes or properties than shown by ending each list with an ellipsis ("...").
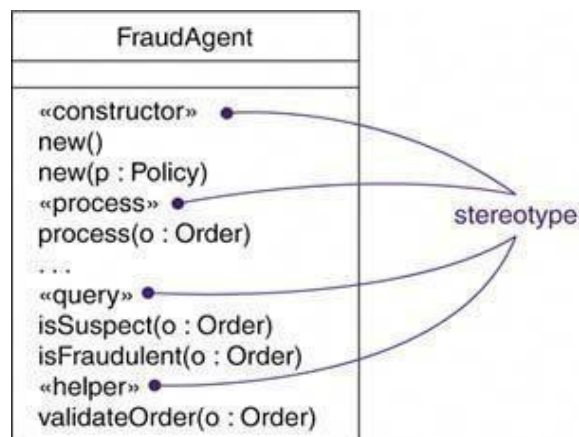
Responsibilities

  A *responsibility* is a contract or an obligation of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior. At a more abstract level, these corresponding attributes and operations are just the features by which the class's responsibilities are carried out. A Wall class is responsible for knowing about height, width, and thickness; a FraudAgent class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.
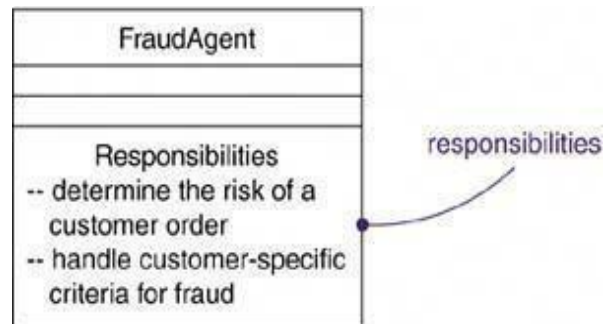
Relationships:

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below. If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

Generalization (Inheritance)
Inheritance represents an "is-a" relationship between classes, where one class (the subclass or child) inherits the properties and behaviors of another class (the superclass or parent). Inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.
In the example of bank accounts, we can use generalization to represent different types of accounts such as current accounts, savings accounts, and credit accounts.
The Bank Account class serves as the generalized representation of all types of bank accounts, while the subclasses (Current Account, Savings Account, Credit Account) represent specialized versions that inherit and extend the functionality of the base class.

1. **Association**

An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class. Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.
Let's understand association using an example:
Let's consider a simple system for managing a library. In this system, we have two main entities: Book and Library. Each Library contains multiple Books, and each Book belongs to a specific Library. This relationship between Library and Book represents an association.
The "Library" class can be considered the source class because it contains a reference to multiple instances of the "Book" class. The "Book" class would be considered the target class because it belongs to a specific library.

Each Library contains multiple Books, and
each Book belongs to a specific Library

Cardinality

Cardinality is expressed in terms of:
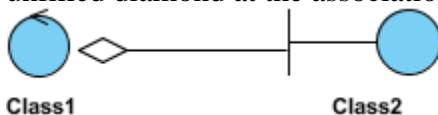- one to one
- one to many
- many to many

Aggregation

A special type of association.
- It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
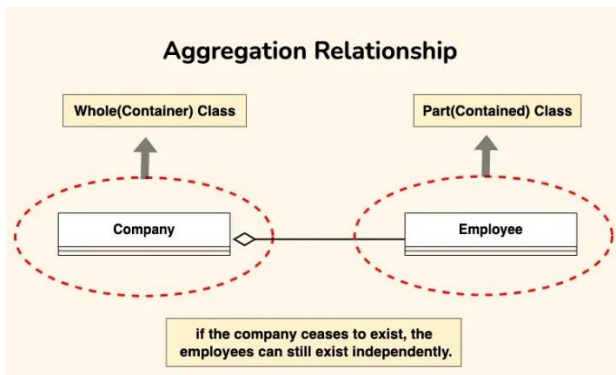- Objects of Class1 and Class2 have separate lifetimes.

The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.

Class1   Class2

Aggregation is a specialized form of association that represents a "whole-part" relationship. It denotes a stronger relationship where one class (the whole) contains or is composed of another class (the part). Aggregation is represented by a diamond shape on the side of the whole class. In this kind of relationship, the child class can exist independently of its parent class.

Let's understand aggregation using an example:

The company can be considered as the whole, while the employees are the parts. Employees belong to the company, and the company can have multiple employees. However, if the company ceases to exist, the employees can still exist independently.

Aggregation Relationship

Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.
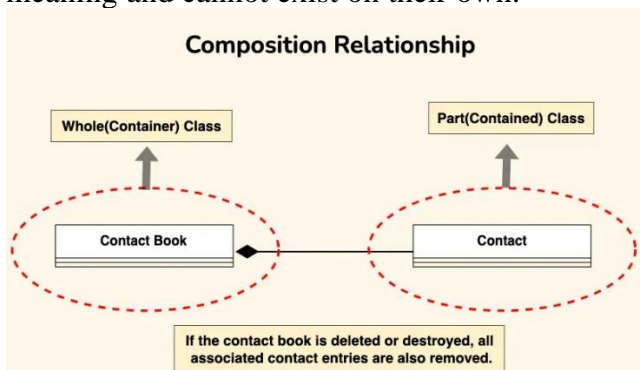

Composition is a stronger form of aggregation, indicating a more significant ownership or dependency relationship. In composition, the part class cannot exist independently of the whole class. Composition is represented by a filled diamond shape on the side of the whole class.

Let's understand Composition using an example:

Imagine a digital contact book application. The contact book is the whole, and each contact entry is a part. Each contact entry is fully owned and managed by the contact book. If the contact book is deleted or destroyed, all associated contact entries are also removed.

This illustrates composition because the existence of the contact entries depends entirely on the presence of the contact book. Without the contact book, the individual contact entries lose their meaning and cannot exist on their own.
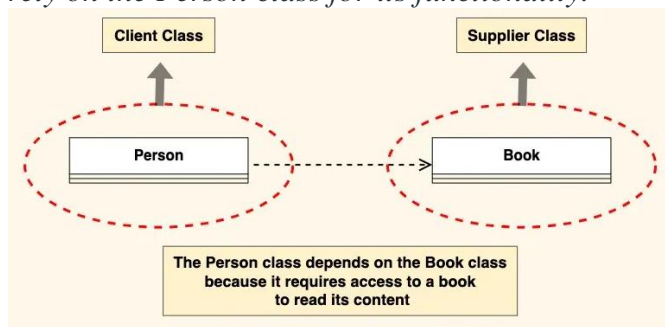

Composition Relationship

Dependency Relationship

A dependency exists between two classes when one class relies on another, but the relationship is not as strong as association or inheritance. It represents a more loosely coupled connection between classes. Dependencies are often depicted as a dashed arrow.

Let's consider a scenario where a Person depends on a Book.

- **Person Class:** Represents an individual who reads a book. The Person class depends on the Book class to access and read the content.
- **Book Class:** Represents a book that contains content to be read by a person. The Book class is independent and can exist without the Person class.

The Person class depends on the Book class because it requires access to a book to read its content. However, the Book class does not depend on the Person class; it can exist independently and does not rely on the Person class for its functionality.

Realization (Interface Implementation)

Realization indicates that a class implements the features of an interface. It is often used in cases where a class realizes the operations defined by an interface. Realization is depicted by a dashed line with an open arrowhead pointing from the implementing class to the interface.

Let's consider the scenario where a "Person" and a "Corporation" both realizing an "Owner" interface.

- **Owner Interface:** This interface now includes methods such as "acquire(property)" and "dispose(property)" to represent actions related to acquiring and disposing of property.
- **Person Class (Realization):** The Person class implements the Owner interface, providing concrete implementations for the "acquire(property)" and "dispose(property)" methods. For instance, a person can acquire ownership of a house or dispose of a car.
- **Corporation Class (Realization):** Similarly, the Corporation class also implements the Owner interface, offering specific implementations for the "acquire(property)" and "dispose(property)" methods. For example, a corporation can acquire ownership of real estate properties or dispose of company vehicles.

Both the Person and Corporation classes realize the Owner interface, meaning they provide concrete implementations for the "acquire(property)" and "dispose(property)" methods defined in the interface.

In this design, any instance of a Person or Corporation can be treated as an Owner, meaning they can acquire and dispose of property. This allows for polymorphic behavior where functions or methods can operate on an Owner type, regardless of whether it is a Person or a Corporation.
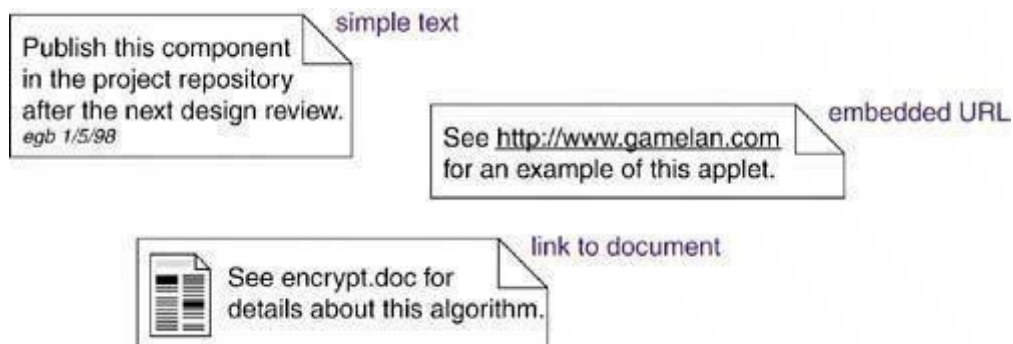
Common Mechanisms:

Note

A *note* is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.
A note may contain any combination of text or graphics

A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. This is why notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.

Specifications

Behind every graphical notation in UML, there is a precise specification of the details that element represents. For example, a class icon is a rectangle and it specifies the name, attributes and operations of the class

Adornments

The mechanism in UML that allows the users to specify extra information with the basic notation of an element is the adornments.

- Textual/graphical items added to the basic notation of an element

- They are used for explicit visual representation of those aspects of an element that are beyond the most important

Examples: The basic notation of association is line, but this could be adorned with additional details, such as the role names and multiplicity of each end

Similarly, a class notation may highlight most important aspects of a class, i.e., name, attributes and operations. To show access specifies for the attributes and methods of a class adornments such as +, -, # are used.
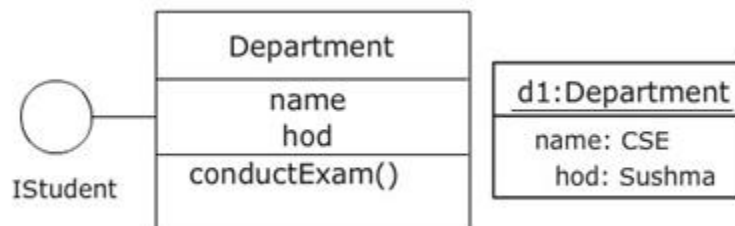

In the above example, the access specifiers: + (public), # (protected) and – (private) represent the visibility of the attributes which is extra information over the basic attribute representation.

Common Divisions

In UML there is clear division between semantically related elements like: separation between a class and an object and the separation between an interface and its implementation.

- In modeling, object-oriented systems get divided in multiple ways.
- For example, class vs. object, interface vs. implementation
- An object uses the same symbol as its class with its name underlined
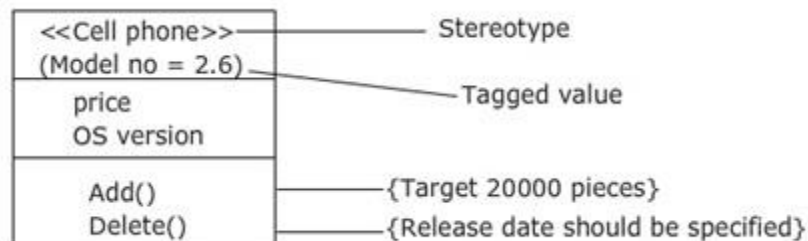

Object-oriented systems can be divided in many ways. The two common ways of division are −
- **Division of classes and objects** − A class is an abstraction of a group of similar objects. An object is a concrete instance that has actual existence in the system.

- **Division of Interface and Implementation** − An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.
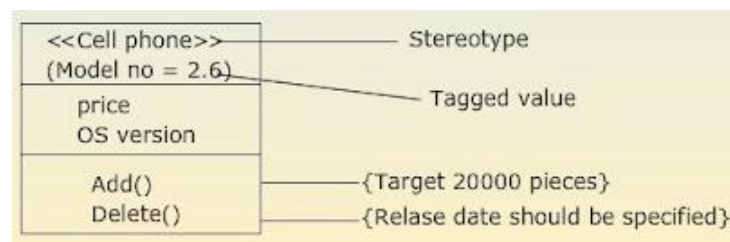
Extensibility Mechanisms

Extensibility mechanisms allow extending the language in controlled ways. It includes **Stereotypes, Tagged Values and Constraints**
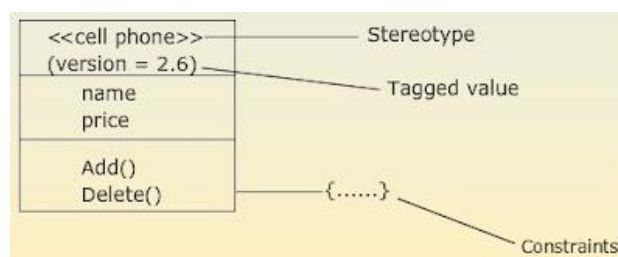

Stereotypes

- Stereotypes are used to create new building blocks from existing blocks
- New building blocks are domain-specific
- Stereotypes are used to extend the vocabulary of a system
- Graphically represented as a name enclosed by guillemets (« »)

Tagged Values

- Tagged values are used to add to the information of the element (not of its instances)
- Stereotypes help to create new building blocks, whereas tagged values help to create new attributes
- These are commonly used to specify information relevant to code generation, configuration management and so on

Constraints

- Constraints are used to create rules for the model
- Rules that impact the behavior of the model, and specify conditions that must be met
- Can apply to any element in the model, i.e., attributes of a class, relationship
- Graphically represented as a string enclosed by braces {....} and placed near the associated elements or connected to that elements by dependency relationships

Diagrams:

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, some of which 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. In the context of architecture,

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).

In modeling real systems, no matter what the problem domain, you'll find yourself creating the same kinds of diagrams, because they represent common views into common models. Typically, you'll view the static parts of a system using one of the following diagrams.

1. Class diagram
2. Component diagram
3. Composite structure diagram
4. Object diagram
5. Deployment diagram
6. Artifact diagram

You'll often use five additional diagrams to view the dynamic parts of a system.

1. Use case diagram
2. Sequence diagram
3. Communication diagram

4. State diagram
5. Activity diagram

2. Discuss the basic structural modelling diagrams in UML, such as class diagrams, and their significance in representing the design of a software system.

Class Diagram

A Class is a blueprint for an object. Objects and classes go hand in hand. We can't talk about one without talking about the other. And the entire point of Object-Oriented Design is not about objects, it's about classes, because we use classes to create objects. So a class describes what an object will be, but it isn't the object itself.

In fact, classes describe the type of objects, while objects are usable instances of classes. Each Object was built from the same set of blueprints and therefore contains the same components (properties and methods). The standard meaning is that an object is an instance of a class and object - Objects have states and behaviors.

Example

A dog has states - color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
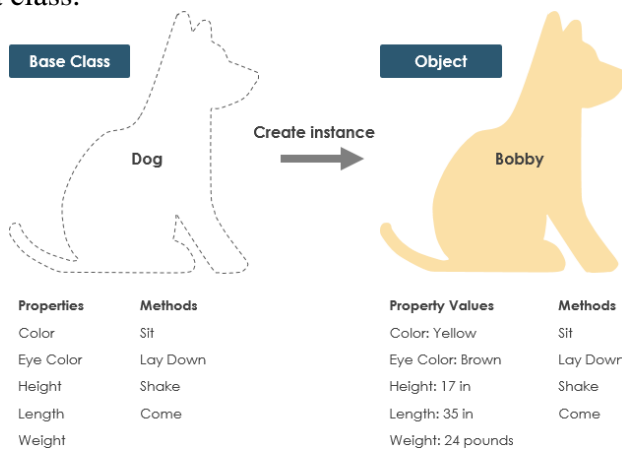
UML Class Notation

A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the **only mandatory** information*.


Class Name:
- The name of the class appears in the first partition.

Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

Class Operations (Methods):
- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code
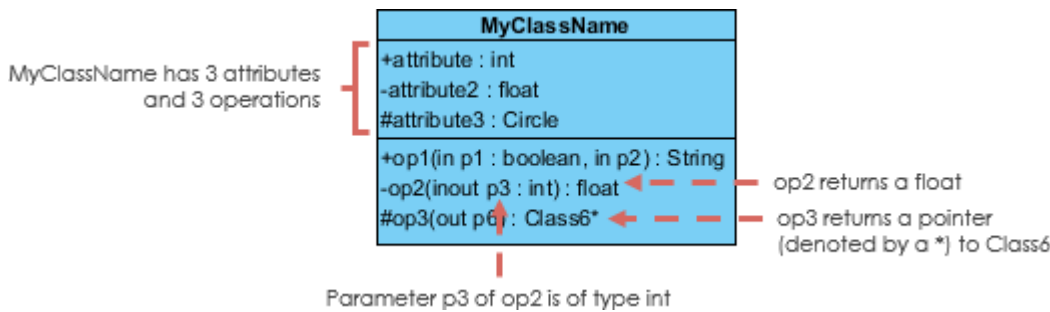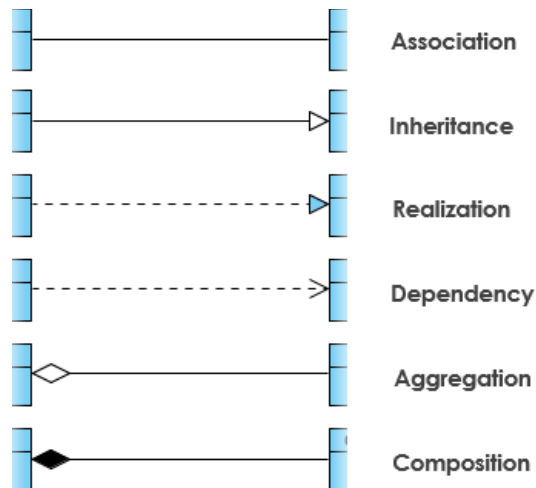

Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.
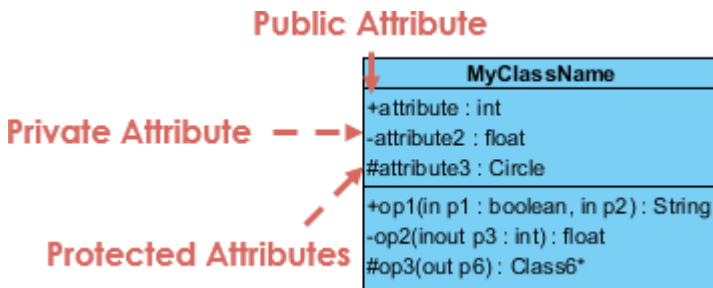
- + denotes public attributes or operations.
- - denotes private attributes or operations
- # denotes protected attributes or operations

Relationships between classes

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below. If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:
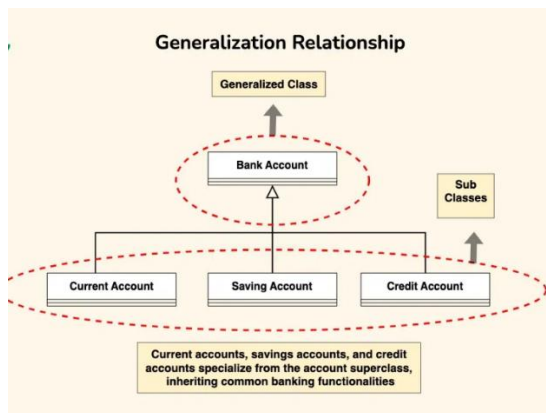
Generalization(Inheritance)

Inheritance represents an "is-a" relationship between classes, where one class (the subclass or child) inherits the properties and behaviors of another class (the superclass or parent). Inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.

In the example of bank accounts, we can use generalization to represent different types of accounts such as current accounts, savings accounts, and credit accounts.

The Bank Account class serves as the generalized representation of all types of bank accounts, while the subclasses (Current Account, Savings Account, Credit Account) represent specialized versions that inherit and extend the functionality of the base class.
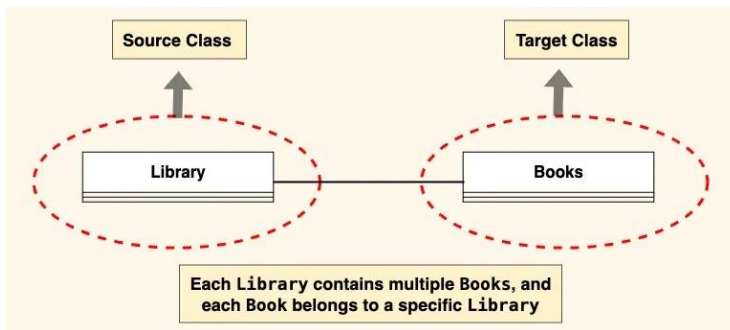

Association

An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class. Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.

Let's understand association using an example:

Let's consider a simple system for managing a library. In this system, we have two main entities: Book and Library. Each Library contains multiple Books, and each Book belongs to a specific Library. This relationship between Library and Book represents an association.
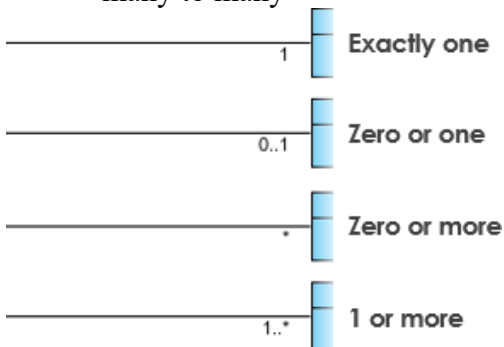
The "Library" class can be considered the source class because it contains a reference to multiple instances of the "Book" class. The "Book" class would be considered the target class because it belongs to a specific library.

Source Class   Target Class

Library   Books

Each Library contains multiple Books, and each Book belongs to a specific Library

Cardinality

Cardinality is expressed in terms of:
- one to one
- one to many
- many to many

1   Exactly one

0..1   Zero or one

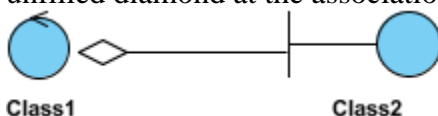*   Zero or more

1..*   1 or more

Aggregation

A special type of association.
- It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
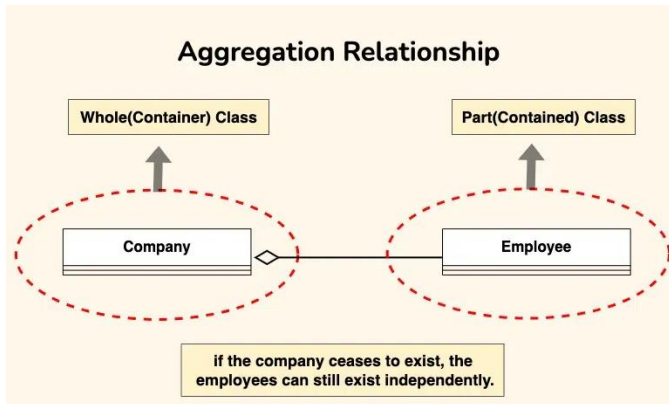- Objects of Class1 and Class2 have separate lifetimes.

The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.

Class1   Class2

Aggregation is a specialized form of association that represents a "whole-part" relationship. It denotes a stronger relationship where one class (the whole) contains or is composed of another class (the part). Aggregation is represented by a diamond shape on the side of the whole class. In this kind of relationship, the child class can exist independently of its parent class.
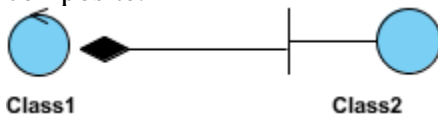
Let's understand aggregation using an example:
The company can be considered as the whole, while the employees are the parts. Employees belong to the company, and the company can have multiple employees. However, if the company ceases to exist, the employees can still exist independently.

Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.
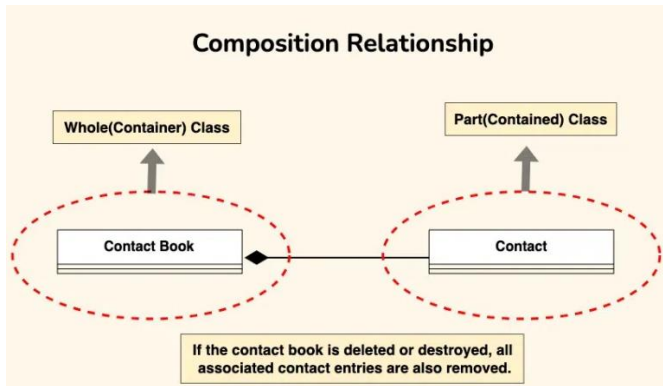

Composition is a stronger form of aggregation, indicating a more significant ownership or dependency relationship. In composition, the part class cannot exist independently of the whole class. Composition is represented by a filled diamond shape on the side of the whole class.

Let's understand Composition using an example:
Imagine a digital contact book application. The contact book is the whole, and each contact entry is a part. Each contact entry is fully owned and managed by the contact book. If the contact book is deleted or destroyed, all associated contact entries are also removed.

This illustrates composition because the existence of the contact entries depends entirely on the presence of the contact book. Without the contact book, the individual contact entries lose their meaning and cannot exist on their own.

Composition Relationship

Whole(Container) Class

Part(Contained) Class

Contact Book

Contact

If the contact book is deleted or destroyed, all
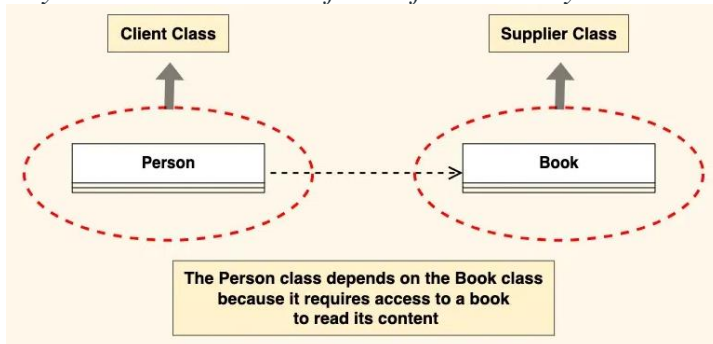associated contact entries are also removed.

Dependency Relationship

A dependency exists between two classes when one class relies on another, but the relationship is not as strong as association or inheritance. It represents a more loosely coupled connection between classes. Dependencies are often depicted as a dashed arrow.

Let's consider a scenario where a Person depends on a Book.

- **Person Class:** Represents an individual who reads a book. The Person class depends on the Book class to access and read the content.
- **Book Class:** Represents a book that contains content to be read by a person. The Book class is independent and can exist without the Person class.

The Person class depends on the Book class because it requires access to a book to read its content. However, the Book class does not depend on the Person class; it can exist independently and does not rely on the Person class for its functionality.

Client Class

Supplier Class

Person

Book

The Person class depends on the Book class
because it requires access to a book
to read its content

Realization (Interface Implementation)

Realization indicates that a class implements the features of an interface. It is often used in cases where a class realizes the operations defined by an interface. Realization is depicted by a dashed line with an open arrowhead pointing from the implementing class to the interface.
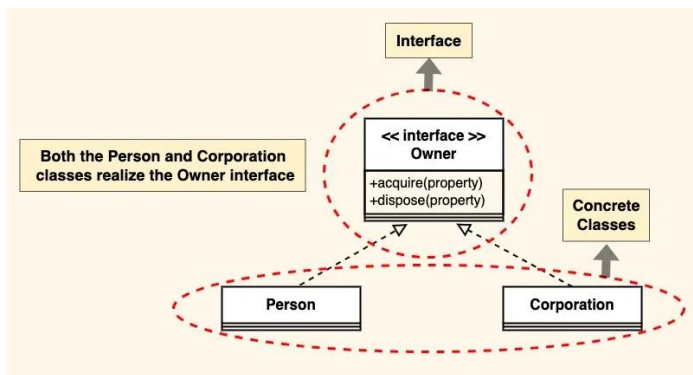
Let's consider the scenario where a "Person" and a "Corporation" both realizing an "Owner" interface.

- **Owner Interface:** This interface now includes methods such as "acquire(property)" and "dispose(property)" to represent actions related to acquiring and disposing of property.
- **Person Class (Realization):** The Person class implements the Owner interface, providing concrete implementations for the "acquire(property)" and "dispose(property)" methods. For instance, a person can acquire ownership of a house or dispose of a car.

- **Corporation Class (Realization):** Similarly, the Corporation class also implements the Owner interface, offering specific implementations for the "acquire(property)" and "dispose(property)" methods. For example, a corporation can acquire ownership of real estate properties or dispose of company vehicles.

Both the Person and Corporation classes realize the Owner interface, meaning they provide concrete implementations for the "acquire(property)" and "dispose(property)" methods defined in the interface.

In this design, any instance of a Person or Corporation can be treated as an Owner, meaning they can acquire and dispose of property. This allows for polymorphic behavior where functions or methods can operate on an Owner type, regardless of whether it is a Person or a Corporation.

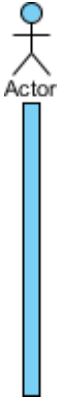## Class Diagram Example: Order System
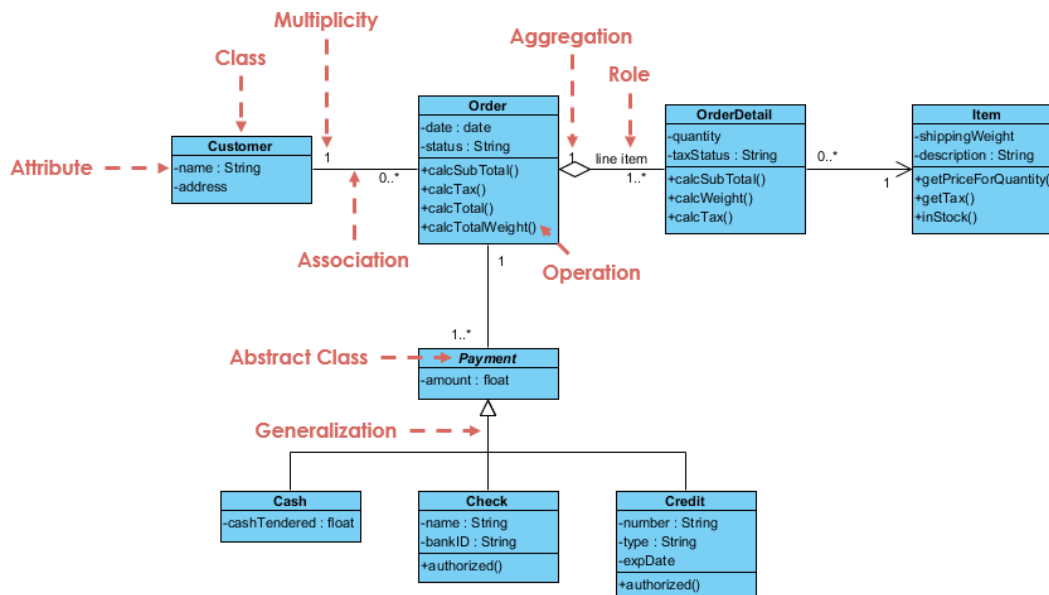
3. Explain about Sequence Diagram

Sequence Diagram

The sequence diagram represents the flow of messages in the system and is also termed as an event diagram. It helps in envisioning several dynamic scenarios. It portrays the communication between any two lifelines as a time-ordered sequence of events, such that these lifelines took part at the run time. In UML, the lifeline is represented by a vertical bar, whereas the message flow is represented by a vertical dotted line that extends across the bottom of the page. It incorporates the iterations as well as branching.
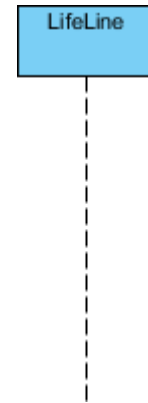
Purpose of a Sequence Diagram

1. To model high-level interaction among active objects within a system.
2. To model interaction among objects inside a collaboration realizing a use case.
3. It either models generic interactions or some certain instances of interaction.
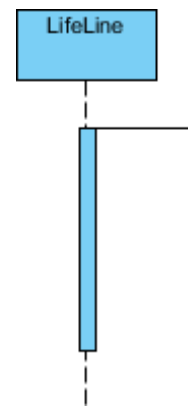
Notations of a Sequence Diagram

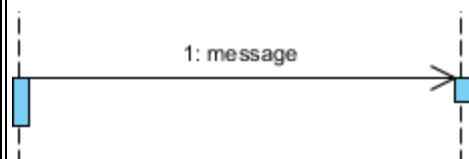| Notation Description	Visual Representation
Actor • a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data) • external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). • represent roles played by human users, external hardware, or other subjects. Note that: • An actor does not necessarily represent a specific physical entity but merely a particular role of some entity • A person may play the role of several different actors and, conversely, a given actor may be played by multiple different person.	 Actor

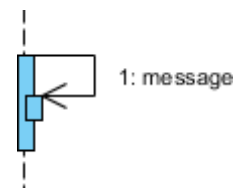|	
Lifeline • A lifeline represents an individual participant in the Interaction.	
Activations • A thin rectangle on a lifeline) represents the period during which an element is performing an operation. • The top and the bottom of the of the rectangle are aligned with the initiation and the completion time respectively	
Call Message • A message defines a particular communication between Lifelines of an Interaction. • Call message is a kind of message that represents an invocation of operation of target lifeline.	
Return Message • A message defines a particular communication between Lifelines of an Interaction. • Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.	
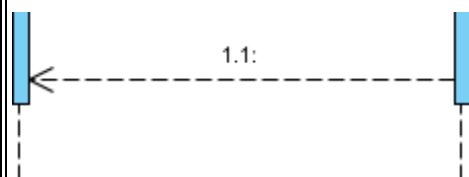
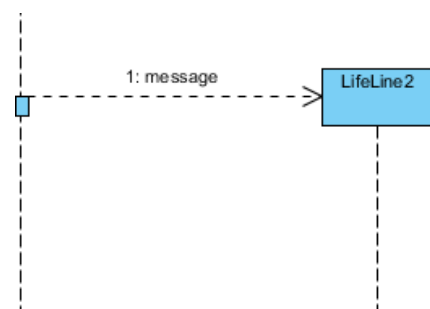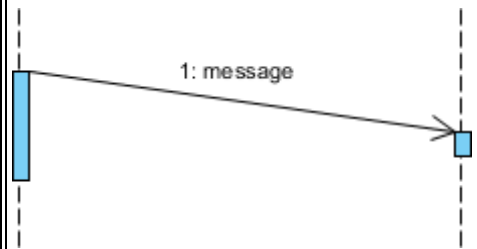|	
Self Message • A message defines a particular communication between Lifelines of an Interaction. • Self message is a kind of message that represents the invocation of message of the same lifeline.	1: message
Recursive Message • A message defines a particular communication between Lifelines of an Interaction. • Recursive message is a kind of message that represents the invocation of message of the same lifeline. It's target points to an activation on top of the activation where the message was invoked from.	1: message
Create Message • A message defines a particular communication between Lifelines of an Interaction. • Create message is a kind of message that represents the instantiation of (target) lifeline.	1: message LifeLine2
Destroy Message • A message defines a particular communication between Lifelines of an Interaction. • Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.	1: message
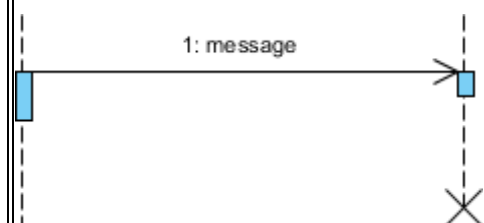
|	
Duration Message • A message defines a particular communication between Lifelines of an Interaction. • Duration message shows the distance between two time instants for a message invocation.	1: message
Note A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.	

Example of a Sequence Diagram

Figure 6.14
Sequence diagram of
ATM withdrawal
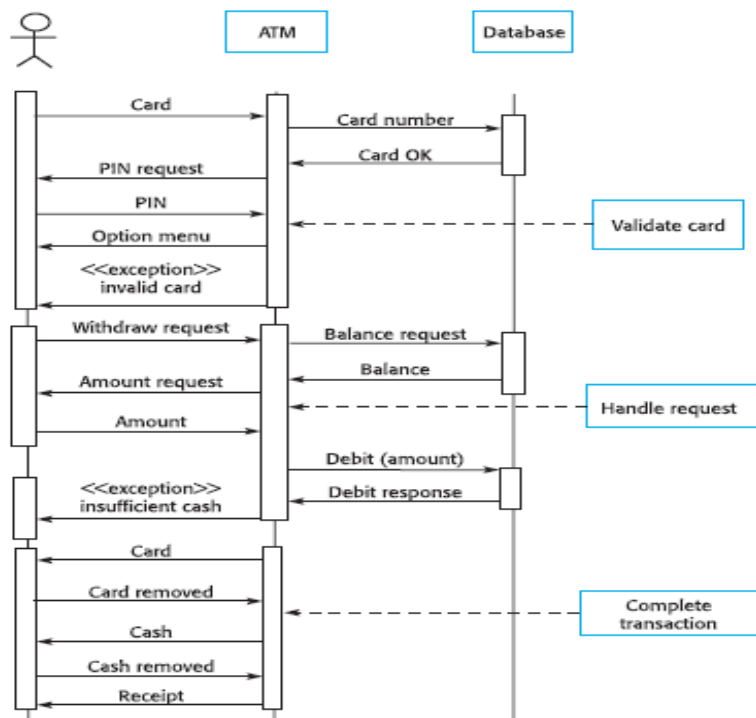
4. **Explain about**
   Collaboration diagram
   Use case diagram
   Component diagram

Collaboration Diagram

The collaboration diagram is used to show the relationship between the objects in a system. Both the sequence and the collaboration diagrams represent the same information but differently. Instead of showing the flow of messages, it depicts the architecture of the object residing in the system as it is based on object-oriented programming. An object consists of several features. Multiple objects present in the system are connected to each other. The collaboration diagram, which is also known as a communication diagram, is used to portray the object's architecture in the system.

Notations of a Collaboration Diagram

Following are the components of a component diagram that are enlisted below:
1. **Objects:** The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.
   In the collaboration diagram, objects are utilized in the following ways:
   - The object is represented by specifying their name and class.
   - It is not mandatory for every class to appear.
   - A class may constitute more than one object.
   - In the collaboration diagram, firstly, the object is created, and then its class is specified.
   - To differentiate one object from another object, it is necessary to name them.
2. **Actors:** In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
3. **Links:** The link is an instance of association, which associates the objects and actors. It portrays a relationship between the objects through which the messages are sent. It is represented by a solid line. The link helps an object to connect with or navigate to another object, such that the message flows are attached to links.
4. **Messages:** It is a communication between objects which carries information and includes a sequence number, so that the activity may take place. It is represented by a labeled arrow, which is placed near a link. The messages are sent from the sender to the receiver, and the direction must be navigable in that particular direction. The receiver must understand the message.

When to use a Collaboration Diagram?

The collaborations are used when it is essential to depict the relationship between the object. Both the sequence and collaboration diagrams represent the same information, but the way of portraying it quite different. The collaboration diagrams are best suited for analyzing use cases. Following are some of the use cases enlisted below for which the collaboration diagram is implemented:
1. To model collaboration among the objects or roles that carry the functionalities of use cases and operations.
2. To model the mechanism inside the architectural design of the system.

3. To capture the interactions that represent the flow of messages between the objects and the roles inside the collaboration.
4. To model different scenarios within the use case or operation, involving a collaboration of several objects and interactions.
5. To support the identification of objects participating in the use case.
6. In the collaboration diagram, each message constitutes a sequence number, such that the top-level message is marked as one and so on. The messages sent during the same call are denoted with the same decimal prefix, but with different suffixes of 1, 2, etc. as per their occurrence.

Steps for creating a Collaboration Diagram

1. Determine the behavior for which the realization and implementation are specified.
2. Discover the structural elements that are class roles, objects, and subsystems for performing the functionality of collaboration.
   o Choose the context of an interaction: system, subsystem, use case, and operation.
3. Think through alternative situations that may be involved.
   o Implementation of a collaboration diagram at an instance level, if needed.
   o A specification level diagram may be made in the instance level sequence diagram for summarizing alternative situations.

Example of a Collaboration Diagram
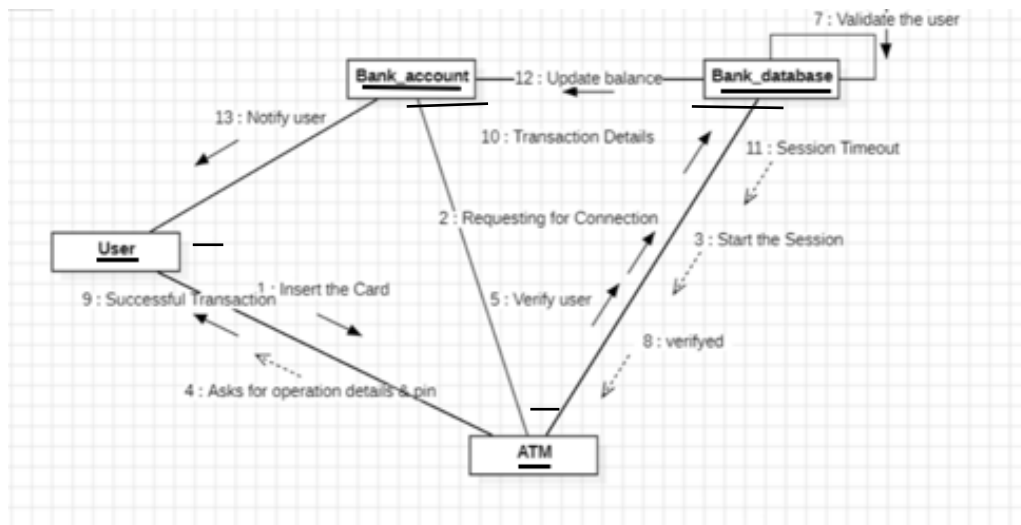
Benefits of a Collaboration Diagram

1. The collaboration diagram is also known as Communication Diagram.
2. It mainly puts emphasis on the structural aspect of an interaction diagram, i.e., how lifelines are connected.
3. The syntax of a collaboration diagram is similar to the sequence diagram; just the difference is that the lifeline does not consist of tails.
4. The messages transmitted over sequencing is represented by numbering each individual message.
5. The collaboration diagram is semantically weak in comparison to the sequence diagram.
6. The special case of a collaboration diagram is the object diagram.

7. It focuses on the elements and not the message flow, like sequence diagrams.
8. Since the collaboration diagrams are not that expensive, the sequence diagram can be directly converted to the collaboration diagram.
9. There may be a chance of losing some amount of information while implementing a collaboration diagram with respect to the sequence diagram.

The drawback of a Collaboration Diagram
1. Multiple objects residing in the system can make a complex collaboration diagram, as it becomes quite hard to explore the objects.
2. It is a time-consuming diagram.
3. After the program terminates, the object is destroyed.
4. As the object state changes momentarily, it becomes difficult to keep an eye on every single that has occurred inside the object of a system.

Use Case Diagram

A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. It depicts the high-level functionality of a system and also tells how the user handles a system.

Purpose of Use Case Diagrams

The main purpose of a use case diagram is to portray the dynamic aspect of a system. It accumulates the system's requirement, which includes both internal as well as external influences. It invokes persons, use cases, and several things that invoke the actors and elements accountable for the implementation of use case diagrams. It represents how an entity from the external environment can interact with a part of the system.

Following are the purposes of a use case diagram given below:
1. It gathers the system's needs.
2. It depicts the external view of the system.
3. It recognizes the internal as well as external factors that influence the system.
4. It represents the interaction between the actors.

How to draw a Use Case diagram?

It is essential to analyze the whole system before starting with drawing a use case diagram, and then the system's functionalities are found. And once every single functionality is identified, they are then transformed into the use cases to be used in the use case diagram.

After that, we will enlist the actors that will interact with the system. The actors are the person or a thing that invokes the functionality of a system. It may be a system or a private entity, such that it requires an entity to be pertinent to the functionalities of the system to which it is going to interact.

Once both the actors and use cases are enlisted, the relation between the actor and use case/ system is inspected. It identifies the no of times an actor communicates with the system. Basically, an actor can interact multiple times with a use case or system at a particular instance of time.

Following are some rules that must be followed while drawing a use case diagram:
1. A pertinent and meaningful name should be assigned to the actor or a use case of a system.
2. The communication of an actor with a use case must be defined in an understandable way.
3. Specified notations to be used as and when required.
4. The most significant interactions should be represented among the multiple no of interactions between the use case and actors.

Example of a Use Case Diagram

A use case diagram depicting the Online Shopping website is given below.Here the Web Customer actor makes use of any online shopping website to purchase online. The top-level uses are as follows; View Items, Make Purchase, Checkout, Client Register. The **View Items** use case is utilized by the customer who searches and view products. The **Client Register** use case allows the customer to register itself with the website for availing gift vouchers, coupons, or getting a private sale invitation. It is to be noted that the **Checkout** is an included use case, which is part of **Making Purchase,** and it is not available by itself.
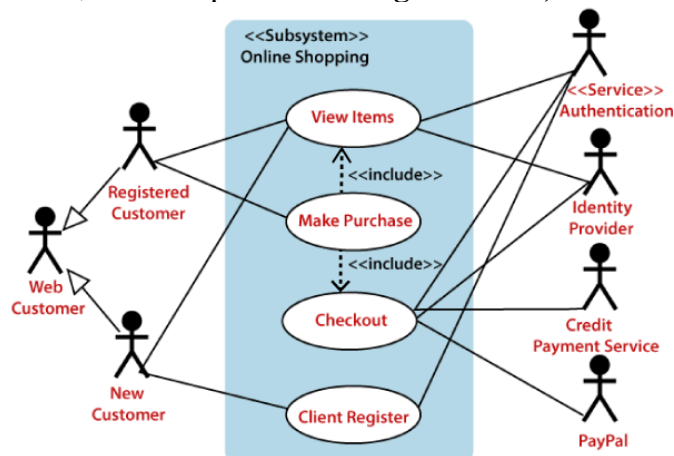

The **View Items** is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item. The View Items is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item.

Both **View Recommended Item** and **Add to Wish List** include the Customer Authentication use case, as they necessitate authenticated customers, and simultaneously item can be added to the shopping cart without any user authentication.
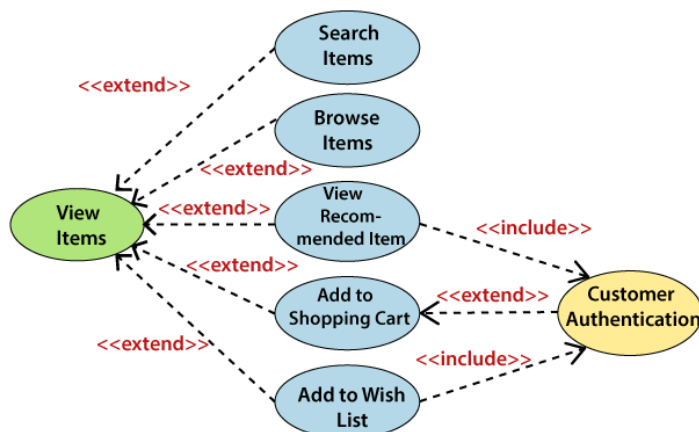
Similarly, the **Checkout** use case also includes the following use cases, as shown below. It requires an authenticated Web Customer, which can be done by login page, user authentication cookie ("Remember me"), or Single Sign-On (SSO). SSO needs an external identity provider's participation, while Web site authentication service is utilized in all these use cases.

The Checkout use case involves Payment use case that can be done either by the credit card and external credit payment services or with PayPal.
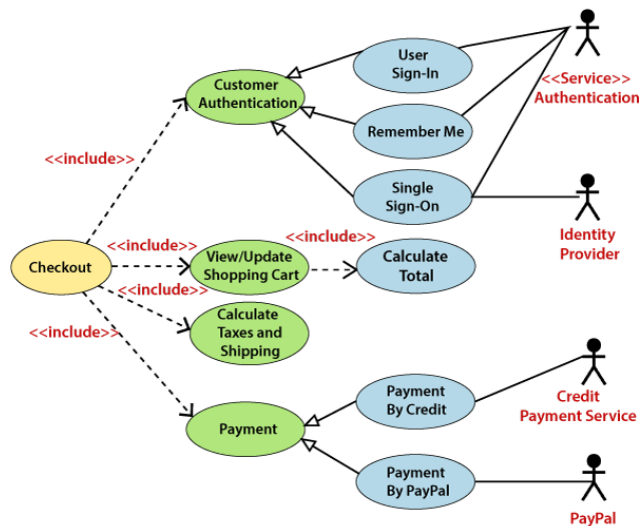
USE CASE DIAGRAM RELATIONSHIPS

In a Use Case Diagram, relationships play a crucial role in depicting the interactions between actors and use cases. These relationships provide a comprehensive view of the system's functionality and its various scenarios. Let's delve into the key types of relationships and explore examples to illustrate their usage.

Association Relationship

The Association Relationship represents a communication or interaction between an actor and a use case. It is depicted by a line connecting the actor to the use case. This relationship signifies that the actor is involved in the functionality described by the use case.

Example: Online Banking System

- **Actor:** Customer
- **Use Case:** Transfer Funds
- **Association:** A line connecting the "Customer" actor to the "Transfer Funds" use case, indicating the customer's involvement in the funds transfer process.
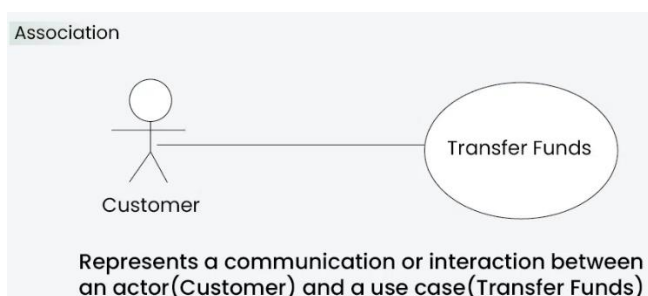

Include Relationship

The Include Relationship indicates that a use case includes the functionality of another use case. It is denoted by a dashed arrow pointing from the including use case to the included use case. This relationship promotes modular and reusable design.

Example: Social Media Posting

- **Use Cases:** Compose Post, Add Image
- **Include Relationship:** The "Compose Post" use case includes the functionality of "Add Image." Therefore, composing a post includes the action of adding an

"Compose Post" includes the functionality of another "Add Image"
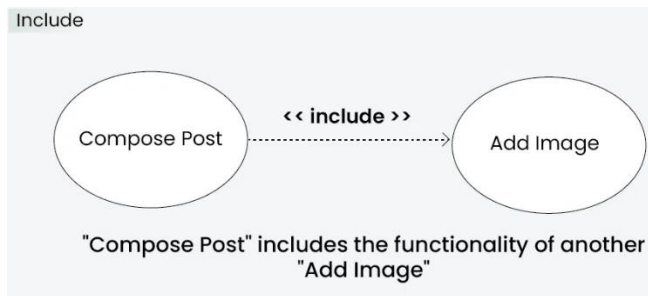
Extend Relationship

The Extend Relationship illustrates that a use case can be extended by another use case under specific conditions. It is represented by a dashed arrow with the keyword "extend." This relationship is useful for handling optional or exceptional behavior.

Example: Flight Booking System

- **Use Cases:** Book Flight, Select Seat
- **Extend Relationship:** The "Select Seat" use case may extend the "Book Flight" use case when the user wants to choose a specific seat, but it is an optional step.

The "Select Seat" use case may extend the "Book Flight" use case when the user wants to choose a specific seat
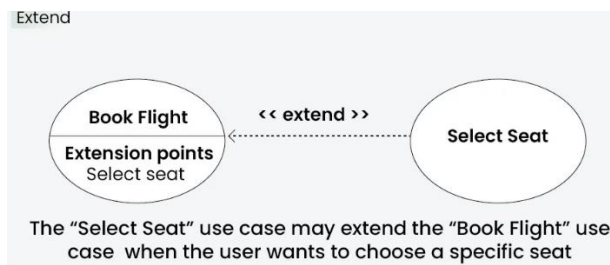
Generalization Relationship

The Generalization Relationship establishes an "is-a" connection between two use cases, indicating that one use case is a specialized version of another. It is represented by an arrow pointing from the specialized use case to the general use case.

Example: Vehicle Rental System

- **Use Cases:** Rent Car, Rent Bike
- **Generalization Relationship:** Both "Rent Car" and "Rent Bike" are specialized versions of the general use case "Rent Vehicle."


Both "Rent Car" and "Rent Bike" are specialized versions of the general use case "Rent Vehicle."
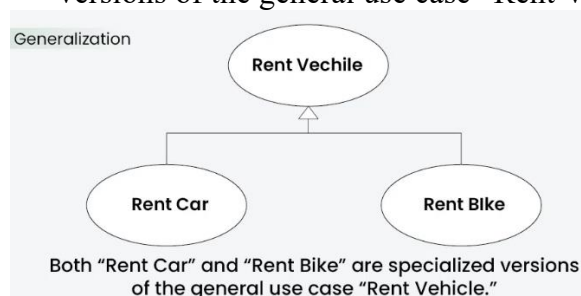
Component Diagram

A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable. It models the physical view of a system such as executables, files, libraries, etc. that resides within the node.

It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

Notation of a Component Diagram

a) A component   Component-name
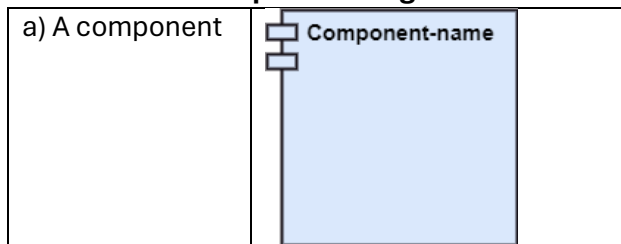
Purpose of a Component Diagram

Since it is a special kind of a UML diagram, it holds distinct purposes. It describes all the individual components that are used to make the functionalities, but not the functionalities of the system. It visualizes the physical components inside the system. The components can be a library, packages, files, etc.

The component diagram also describes the static view of a system, which includes the organization of components at a particular instant. The collection of component diagrams represents a whole system.

The main purpose of the component diagram are enlisted below:

1. It envisions each component of a system.
2. It constructs the executable by incorporating forward and reverse engineering.
3. It depicts the relationships and organization of components.

Why use Component Diagram?

The component diagrams have remarkable importance. It is used to depict the functionality and behavior of all the components present in the system, unlike other diagrams that are used to represent the architecture of the system, working of a system, or simply the system itself.

In UML, the component diagram portrays the behavior and organization of components at any instant of time. The system cannot be visualized by any individual component, but it can be by the collection of components.

Following are some reasons for the requirement of the component diagram:

1. It portrays the components of a system at the runtime.
2. It is helpful in testing a system.
3. It envisions the links between several connections.

When to use a Component Diagram?

It represents various physical components of a system at runtime. It is helpful in visualizing the structure and the organization of a system. It describes how individual components can together form a single system. Following are some reasons, which tells when to use component diagram:

1. To divide a single system into multiple components according to the functionality.
2. To represent the component organization of the system.

How to Draw a Component Diagram?

The component diagram is helpful in representing the physical aspects of a system, which are files, executables, libraries, etc. The main purpose of a component diagram is different from that of other diagrams. It is utilized in the implementation phase of any application.

Once the system is designed employing different UML diagrams, and the artifacts are prepared, the component diagram is used to get an idea of implementation. It plays an essential role in implementing applications efficiently.

Following are some artifacts that are needed to be identified before drawing a component diagram:

1. What files are used inside the system?
2. What is the application of relevant libraries and artifacts?
3. What is the relationship between the artifacts?

Following are some points that are needed to be kept in mind after the artifacts are identified:

1. Using a meaningful name to ascertain the component for which the diagram is about to be drawn.
2. Before producing the required tools, a mental layout is to be made.
3. To clarify the important points, notes can be incorporated.

Example of a Component Diagram

A component diagram for an online shopping system is given below:
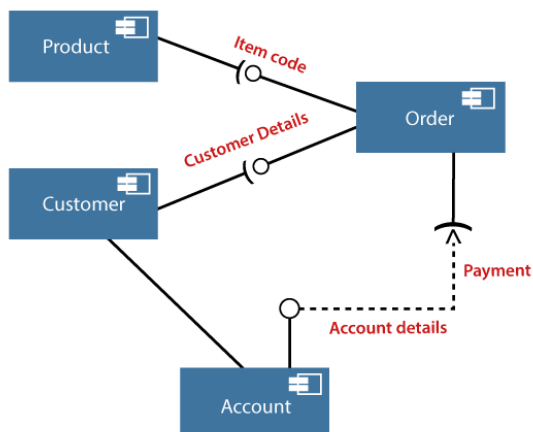
Where to use Component Diagrams?

The component diagram is a special purpose diagram, which is used to visualize the static implementation view of a system. It represents the physical components of a system, or we can say it portrays the organization of the components inside a system. The components, such as libraries, files, executables, etc. are first needed to be organized before the implementation.

The component diagram can be used for the followings:

1. To model the components of the system.
2. To model the schemas of a database.
3. To model the applications of an application.
4. To model the system's source code.