

MALLA REDDY UNIVERSITY

R22 - II YEAR / I SEM

(MR22-1CS0146)

OBJECT ORIENTED SOFTWARE ENGINEERING

A. Y. 2023 – 2024

UNIT–III

Design Engineering: Design process and design quality, design concepts, the design model.

Creating an architectural design: software architecture, data design, architectural styles and patterns, architectural design, conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams.

Design Engineering

Design engineering encompasses the **set of principles, 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.

What is design?

Design is what virtually 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.

Why is it important?

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.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence. Another **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.

1) DESIGN PROCESS AND DESIGN QUALITY:

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.

Quality guidelines:

In order 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
 - a. has been created using recognizable architectural styles or patterns
 - b. is composed of components that exhibit good design characteristics and
 - c. 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 representation 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 interface 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.

These design guidelines are not achieved by chance. Design engineering encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

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

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security.

Another may demand performance with particular emphasis on processing speed.

A third might focus on reliability.

2) 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.”

I. 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.

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**.

II. 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 events.

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.

III. Patterns:

Brad Appleton defines a **design pattern** in the following manner: “a pattern is a named nugget of inside which conveys that essence of a proven solution to a recurring problem within a certain context amidst competing concerns.” Stated in another way, a design pattern describes a design structure that solves a particular design 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 capable to the current work,
- 2) Whether the pattern can be reused,
- 3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

IV. Modularity:

Software architecture and design patterns embody **modularity**; 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 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.

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.

Under modularity or over modularity should be avoided. We modularize a design so that development can be more easily planned; software increment can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

V. 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.

VI. 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 function 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.

VII. 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.

VIII. 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.

IX. Design classes:

The software team must define a set of design classes that

1. Refine the analysis classes by providing design detail that will enable the classes to be implemented, and
2. 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.” They define **four characteristics of a well- formed design class.**

Complete and sufficient: A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness: Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

High cohesion: A cohesive design class has a small, focused set of responsibilities and single- mindedly applies attributes and methods to implement those responsibilities.

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 neighboring classes.

THE DESIGN MODEL:

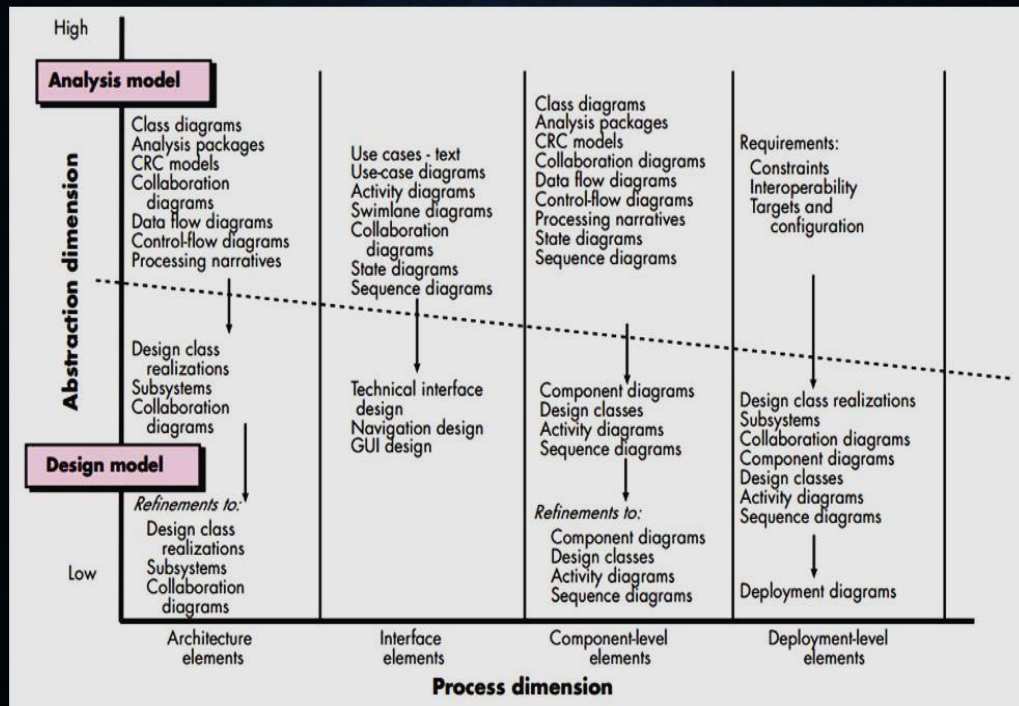
- The design model can be viewed into different dimensions.
- The process dimension indicates the evolution of the design model as design tasks are executed as a part of the software process.

The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

The elements of the design model use many of the same UML diagrams that were used in the analysis model. The difference is that these diagrams are refined and elaborated as a path of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and the interface between the components and with the outside world are all emphasized.

It is important to mention however, that model elements noted along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.

DIMENSIONS OF THE DESIGN MODEL



i. Data design elements:

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. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. 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 criterion of high-quality applications.
- ✓ At the **application level**, the translation of a data model 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.

ii. Architectural design elements:

The *architectural design* for software is the equivalent to the floor plan of a house. The architectural model is derived from three sources.

- a. Information about the application domain for the software to be built.
- b. Specific analysis model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
- c. The availability of architectural patterns

iii. Interface design elements:

The *interface design* for software is the equivalent to a set of detailed drawings for the doors, windows, and external utilities of a house.

The interface design elements for software tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture. There are 3 important elements of interface design:

- a. The user interface(UI);
- b. External interfaces to other systems, devices, networks, or other produces or consumers of information; and
- c. Internal interfaces between various design components.

These interface design elements allow the software to communicated externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design is a major software engineering action.

The design of a UI incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. The design of external interfaces should incorporate error checking and appropriated security features.

UML defines an *interface* in the following manner:” an interface is a specifier for the externally-visibleoperations of a class, component, or other classifier without specification of internal structure.”

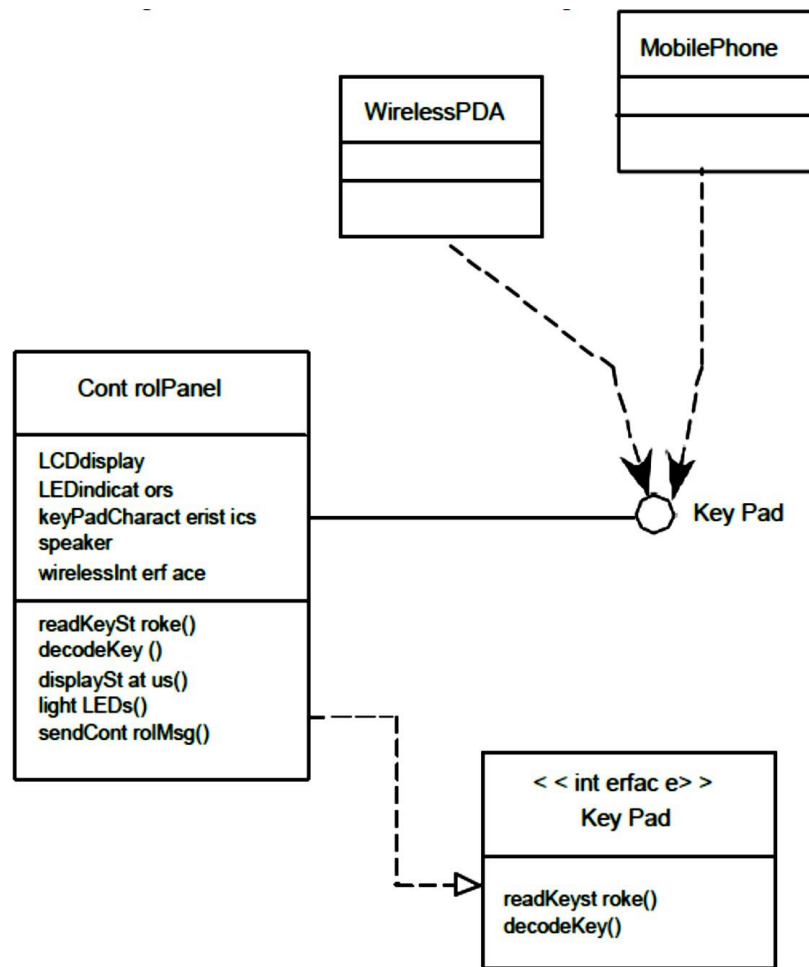
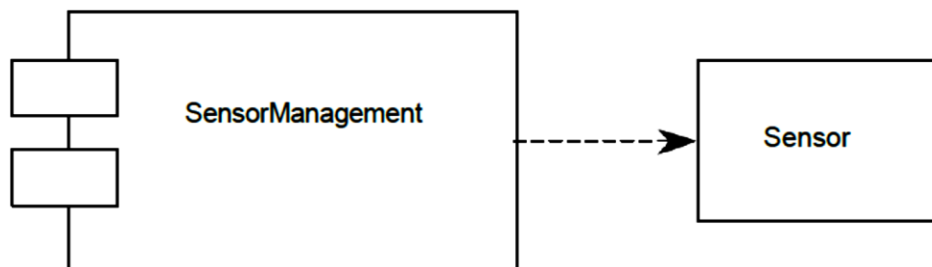


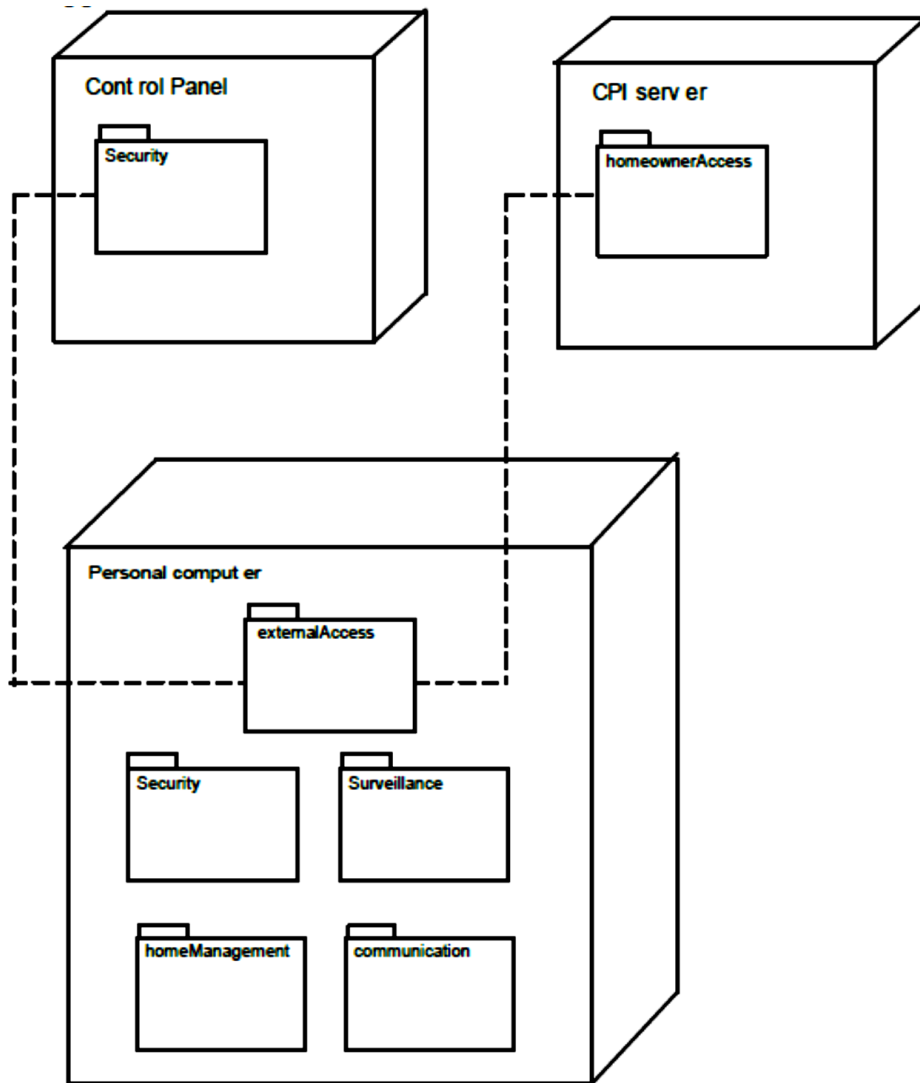
Figure 9 . 6 UML interface representation for **ControlPanel**

- iv. **Component-level design elements:** The component-level design for software is equivalent to a set of detailed drawings.

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.



- v. **Deployment-level design elements:** Deployment-level design elements indicated how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

Figure 9 . 8 UML deployment diagram for *SafeHome*

ARCHITECTURAL DESIGN

1) SOFTWARE ARCHITECTURE: What Is Architecture?

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers

- the architectural style that the system will take,
- the structure and properties of the components that constitute the system, and
- the interrelationships that occur among all architectural components of a system.

The architecture is a representation that enables a software engineer to

- (1) analyze the effectiveness of the design in meeting its stated requirements,
- (2) consider architectural alternatives at a stage when making design changes is still relatively easy,
- (3) reducing the risks associated with the construction of the software.

The design of software architecture considers two levels of the design pyramid

- data design
 - architectural design.
- ✓ Data design enables us to represent the data component of the architecture.
 - ✓ Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

Why Is Architecture Important?

Bass and his colleagues [BAS98] 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”

2) DATA DESIGN:

The data design activity translates data objects as part of the analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

- 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.

2.1) Data design at the Architectural Level:

The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross functional.

To solve this challenge, the business IT community has developed *data mining* techniques, also called *knowledge discovery in databases* (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. An alternative solution, called a *data warehouse*, adds an additional layer to the data architecture. a data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business.

2.2) Data design at the Component Level:

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. The following set of principles for data specification:

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and the operations to be performed on each should be identified.
3. A data dictionary should be established and used to define both data and program design.
4. Low-level data design decisions should be deferred until late in the design process.
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

3) ARCHITECTURAL STYLES AND PATTERNS:

The builder has used an *architectural style* as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod).

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, coordinations 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.

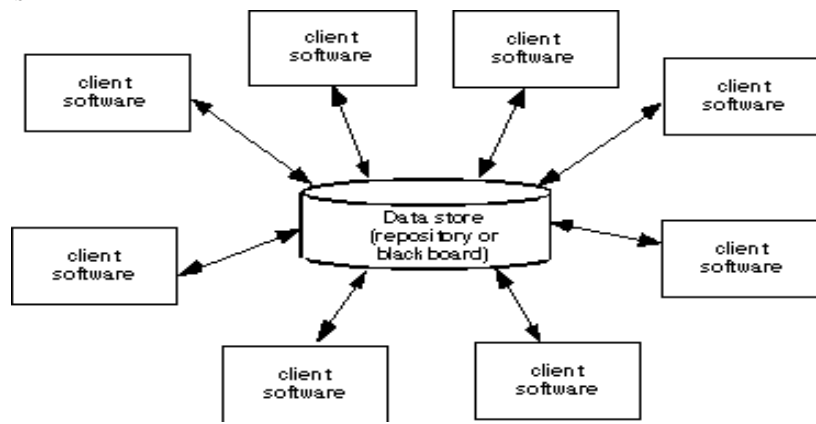
An *architectural pattern*, like an architectural style, imposes a transformation the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

- (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.
- (3) Architectural patterns tend to address specific behavioral issues within the context of the architectural.

3.1) A Brief Taxonomy of Styles and Patterns Data-centered architectures:

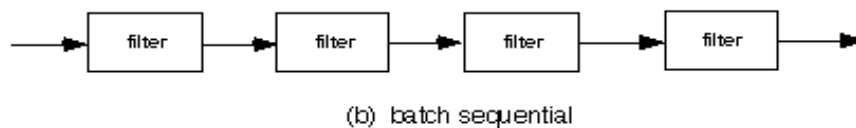
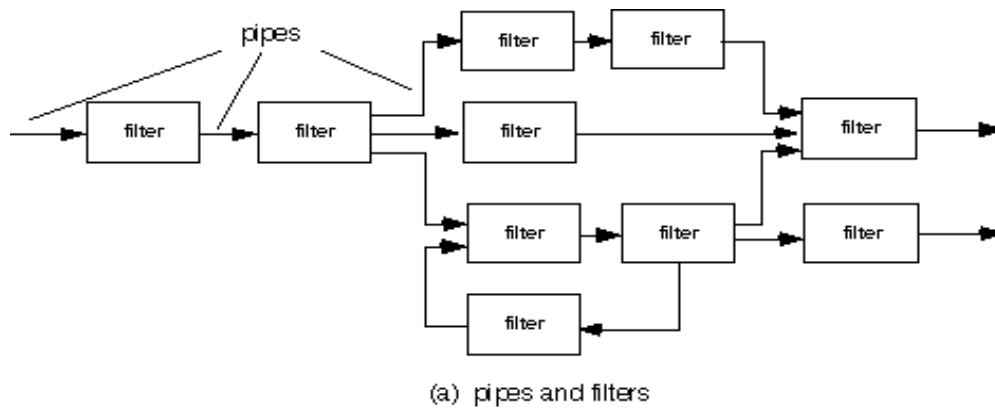
A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. A variation on this approach transforms the repository into a “blackboard” that sends notification to 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 can be 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



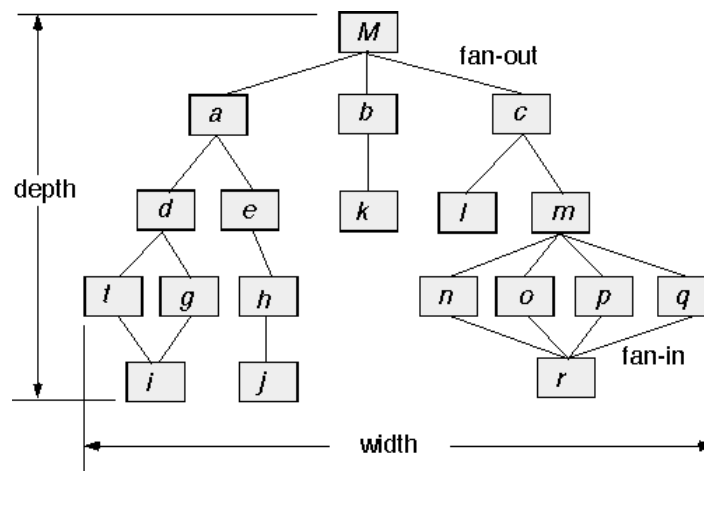
Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A **pipe and filter pattern** has a set of components, called **filters**, connected by pipes that transmit data from one component 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 of a specified form.

If the data flow degenerates into a single line of transforms, it is termed **batch sequential**. This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it.



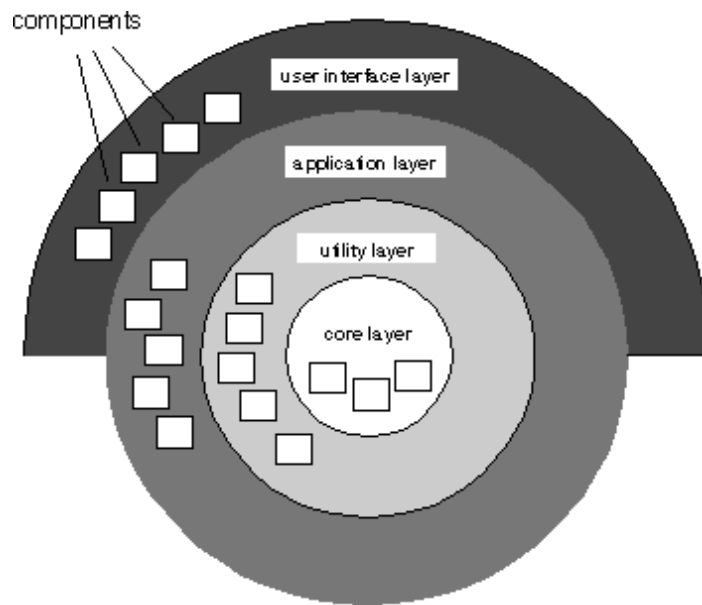
Call and return architectures. This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of substyles [BAS98] exist within this category:

- **Main program/subprogram architectures.** This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components. Figure 13.3 illustrates an architecture of this type.
- **Remote procedure call architectures.** The components of a main program/ subprogram architecture are distributed across multiple computers on a network



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 is accomplished via message passing.

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 14.3. 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.



3.2) Architectural Patterns:

An **architectural pattern**, like an architectural style, imposes a transformation the design of 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.
- (3) Architectural patterns tend to address specific behavioral issues within the context of the architectural.

The architectural patterns for software define a specific approach for handling some behavioral characteristics of the system

Concurrency—applications must handle multiple tasks in a manner that simulates parallelism

- *operating system process management pattern*
- *task scheduler pattern*

Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:

- a **database management system** pattern that applies the storage and retrieval capability of a DBMS to the application architecture
- an **application level persistence** pattern that builds persistence features into the application architecture

Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment

- A **broker** acts as a ‘middle-man’ between the client component and a server component.

Organization and Refinement:

The design process often leaves a software engineer with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions provide insight into the architectural style that has been derived:

Control.

- ✓ How is control managed within the architecture?
- ✓ Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
- ✓ How do components transfer control within the system?
- ✓ How is control shared among components?

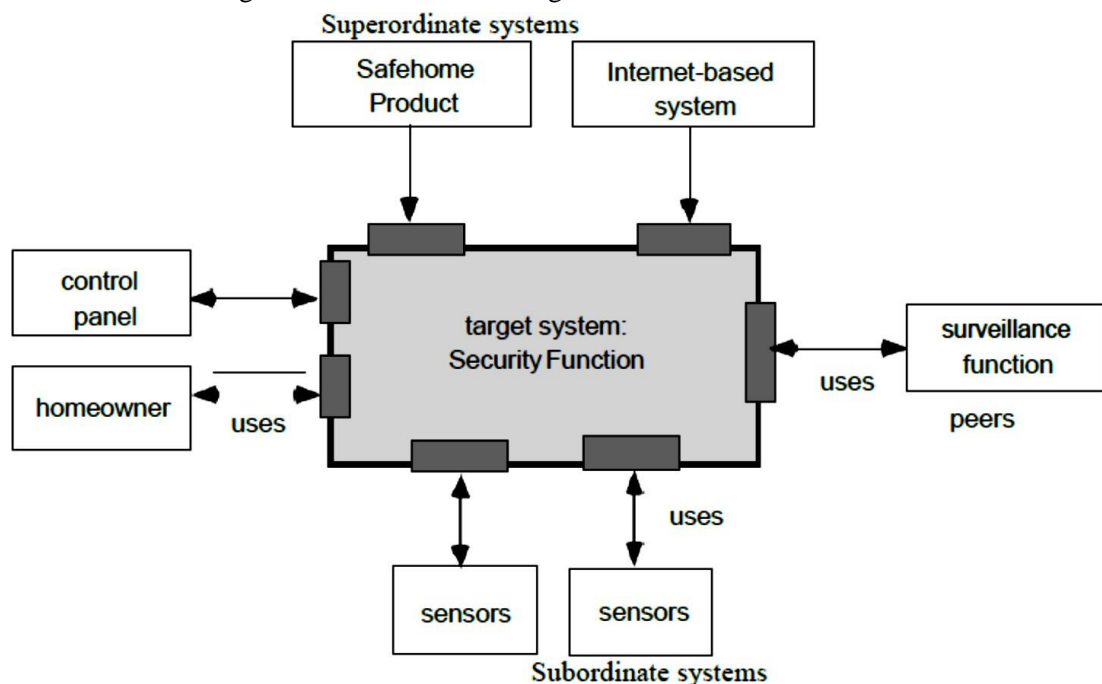
Data.

- ✓ How are data communicated between components?
- ✓ Is the flow of data continuous, or are data objects passed to the system sporadically?
- ✓ What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)?
- ✓ Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?
- ✓ How do functional components interact with data components?
- ✓ Are data components *passive* or *active* (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

4) ARCHITECTURAL DESIGN:

I 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 the figure



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

Actors -those entities that interact with the target system by producing or consuming information that is necessary for requisite processing

II Defining Archetypes:

An archetype is a class or pattern that represents a core abstraction that is critical to the design of architecture for the target system. In general, a relative small set of archetypes is required to design even relatively complex systems.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the analysis model. In safe home security function, the following are the archetypes:

- **Node:** Represent 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 indicators.
- **Detector:** An abstraction that encompasses all sensing equipment that feeds information into the target system
- **Indicator:** An abstraction that represents all mechanisms for indication 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.

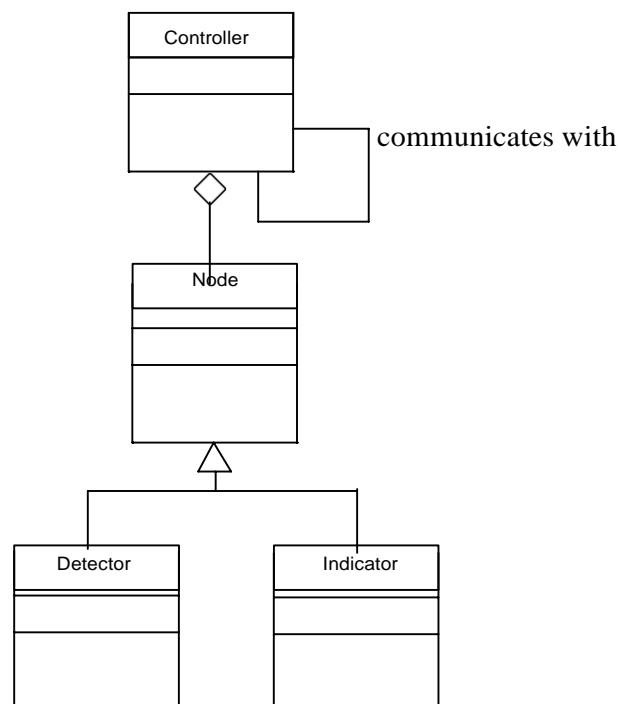


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

III Refining the Architecture into Components:

As the architecture is refined into components, the structure of the system begins to emerge. The architectural designer begins with the classes that were described as part of the analysis model. These analysis classes represent entities within the application domain that must be addressed within the software

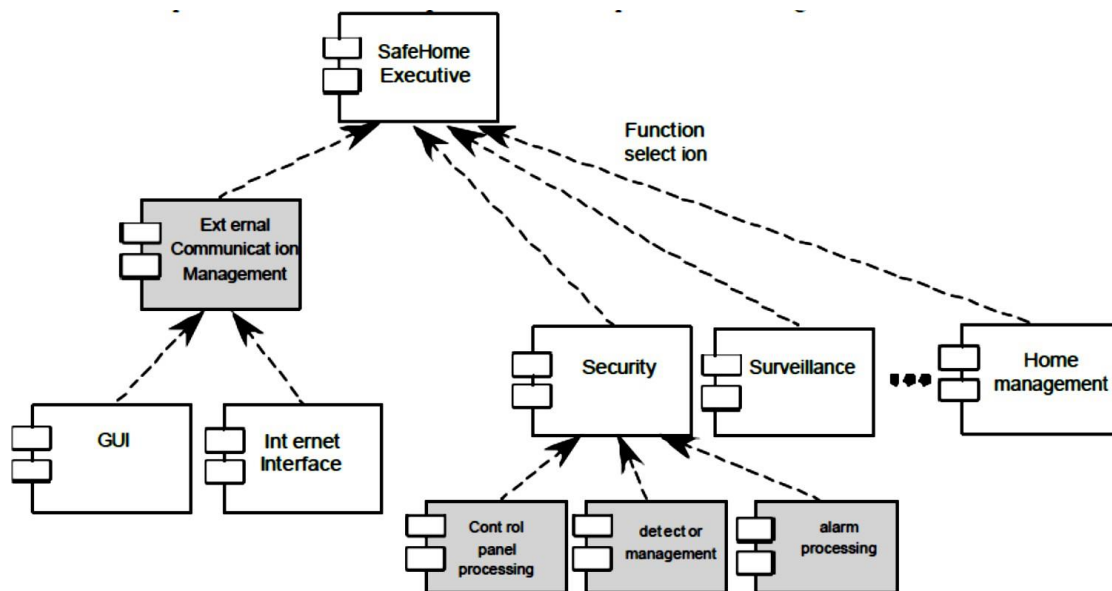
architecture. Hence, the application domain is one source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application domain.

For eg: memory management components, communication components database components, and task management components are often integrated into the software architecture.

In the *safeHome* security function example, we might define the set of top-level components that address the following functionality:

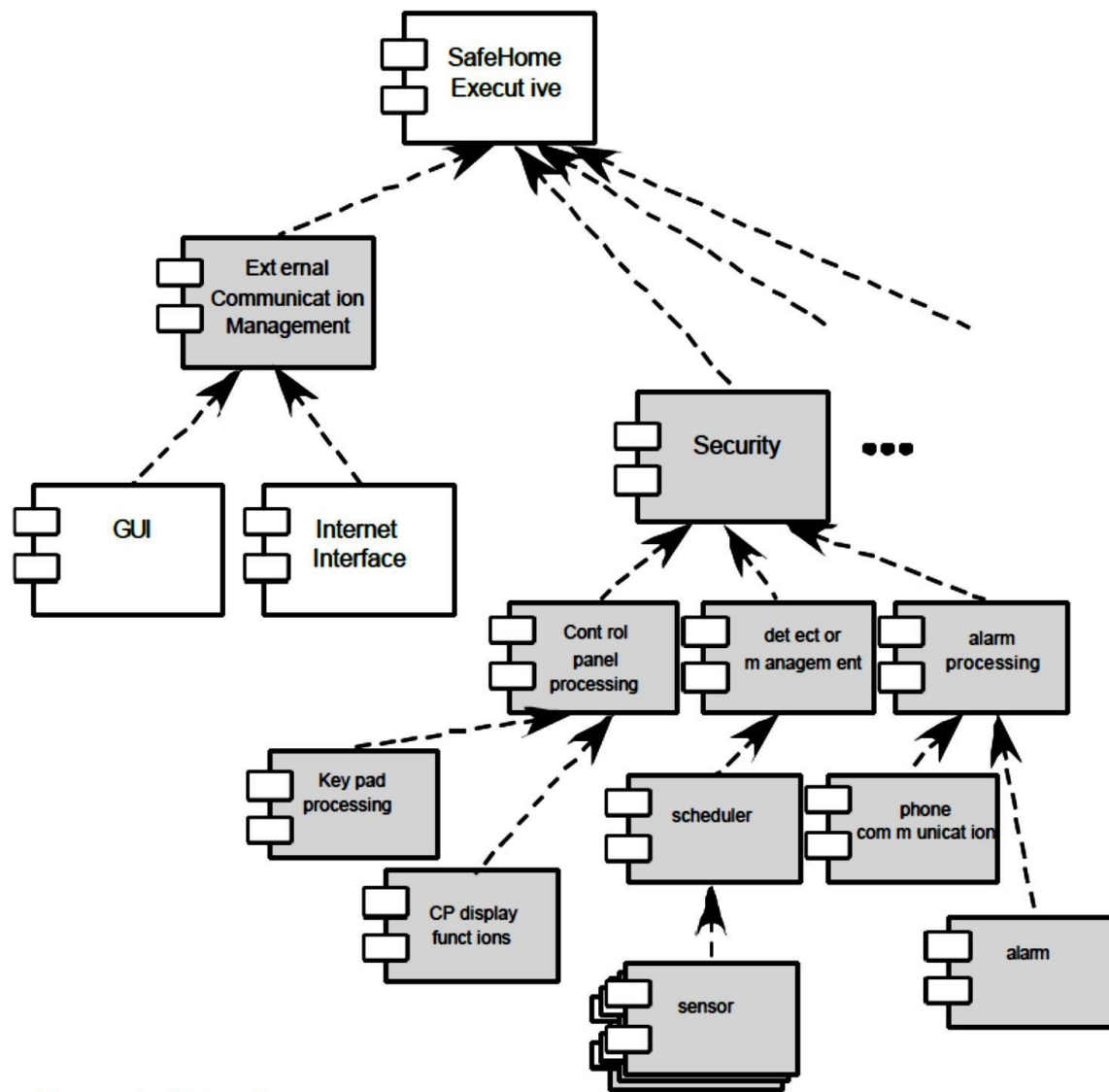
- *External communication management*- coordinates communication of the security function with external entities
- *Control panel processing*- manages all control panel functionality.
- *Detector management*- coordinates access to all detectors attached to the system.
- *Alarm processing*- verifies and acts on all alarm conditions.

Design classes would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design.



Component Structure

- IV Describing Instantiations of the System:** An actual instantiation of the architecture means the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.



A Conceptual Model of UML

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by the Object Management Group.

- UML stands for **Unified Modeling Language**.
- UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- UML is a pictorial language used to make software blueprints.
- UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.

To understand the conceptual model of UML, first we need to clarify what is a conceptual model? and why a conceptual model is required?

- A conceptual model can be defined as a model which is made of concepts and their relationships.
- A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other.

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements –

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

Object-Oriented Concepts

UML can be described as the successor of object-oriented (OO) analysis and design.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form a hierarchy to model the real-world system. The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.

Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.

UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are representation of object-oriented concepts only. Thus, before learning UML, it becomes important to understand OO concept in detail.

Following are some fundamental concepts of the object-oriented world –

- **Objects** – Objects represent an entity and the basic building block.
- **Class** – Class is the blue print of an object.
- **Abstraction** – Abstraction represents the behavior of an real world entity.
- **Encapsulation** – Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
- **Inheritance** – Inheritance is the mechanism of making new classes from existing ones.
- **Polymorphism** – It defines the mechanism to exists in different forms.

OO Analysis and Design

OO can be defined as an investigation and to be more specific, it is the investigation of objects. Design means collaboration of identified objects.

Thus, it is important to understand the OO analysis and design concepts. The most important purpose of OO analysis is to identify objects of a system to be designed. This analysis is also done for an existing system. Now an efficient analysis is only possible when we are able to start thinking in a way where objects can be identified. After identifying the objects, their relationships are identified and finally the design is produced.

The purpose of OO analysis and design can described as –

- Identifying the objects of a system.
- Identifying their relationships.
- Making a design, which can be converted to executables using OO languages.

There are three basic steps where the OO concepts are applied and implemented. The steps can be defined as

OO Analysis → OO Design → OO implementation using OO languages

The above three points can be described in detail as –

- During OO analysis, the most important purpose is to identify objects and describe them in a proper way. If these objects are identified efficiently, then the next job of design is easy. The objects should be identified with responsibilities. Responsibilities are the functions performed by the object. Each and every object has some type of responsibilities to be performed. When these responsibilities are collaborated, the purpose of the system is fulfilled.
- The second phase is OO design. During this phase, emphasis is placed on the requirements and their fulfilment. In this stage, the objects are collaborated according to their intended association. After the association is complete, the design is also complete.
- The third phase is OO implementation. In this phase, the design is implemented using OO languages such as Java, C++, etc.

Role of UML in OO Design

UML is a modeling language used to model software and non-software systems. Although UML is used for non-software systems, the emphasis is on modeling OO software applications. Most of the UML diagrams discussed so far are used to model different aspects such as static, dynamic, etc. Now whatever be the aspect, the artifacts are nothing but objects.

If we look into class diagram, object diagram, collaboration diagram, interaction diagrams all would basically be designed based on the objects.

Hence, the relation between OO design and UML is very important to understand. The OO design is transformed into UML diagrams according to the requirement. Before understanding the UML in detail, the OO concept should be learned properly. Once the OO analysis and design is done, the next step is very easy. The input from OO analysis and design is the input to UML diagrams.

Basic Structural Modeling

Structural modeling captures the static features of a system. They consist of the following –

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Composite structure diagram
- Component diagram

Structural model represents the framework for the system and this framework is the place where all other components exist. Hence, the class diagram, component diagram and deployment diagrams are part of structural modeling. They all represent the elements and the mechanism to assemble them.

The structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram.

Class Diagram:

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object-oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as –

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

How to Draw a Class Diagram?

Class diagrams are the most popular UML diagrams used for construction of software applications. It is very important to learn the drawing procedure of class diagram.

Class diagrams have a lot of properties to consider while drawing but here the diagram will be considered from a top level view.

Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. A collection of class diagrams represent the whole system.

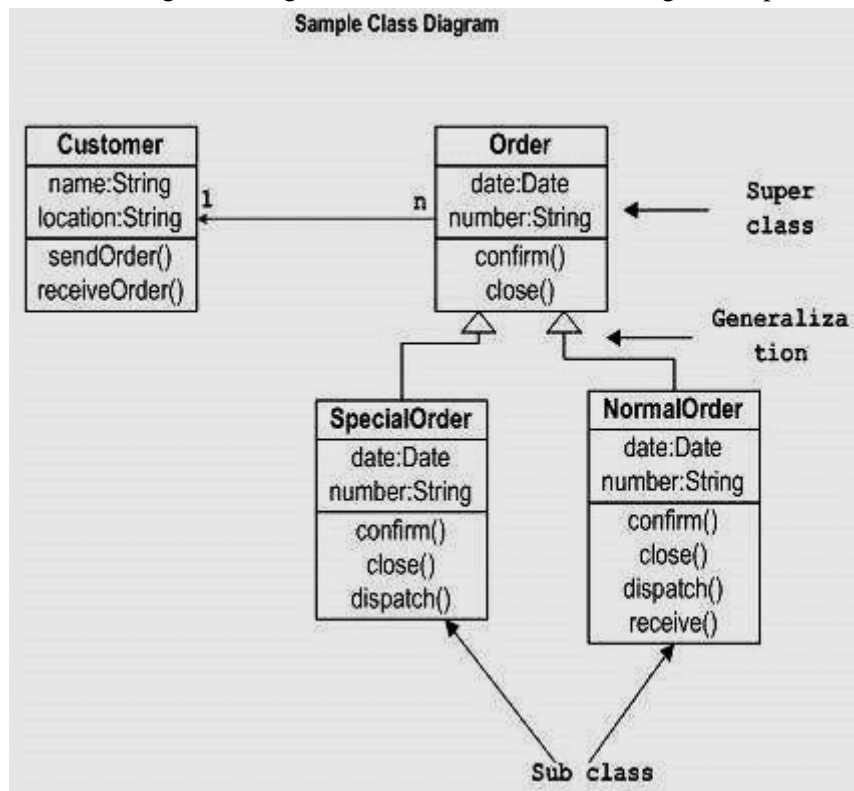
The following points should be remembered while drawing a class diagram –

- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified
- For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
- Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
- Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.

The following diagram is an example of an Order System of an application. It describes a particular aspect of the entire application.

- First of all, Order and Customer are identified as the two elements of the system. They have a one-to-many relationship because a customer can have multiple orders.
- Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
- The two inherited classes have all the properties as the Order class. In addition, they have additional functions like dispatch () and receive ().

The following class diagram has been drawn considering all the points mentioned above.



In a nutshell it can be said, class diagrams are used for –

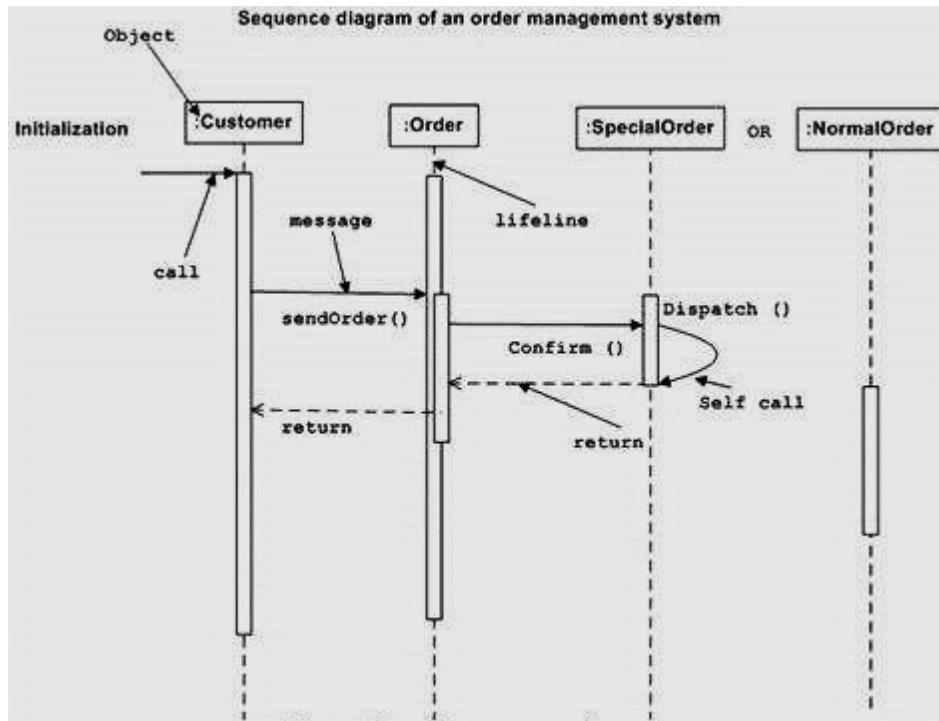
- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

The Sequence Diagram

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is *sendOrder()* which is a method of *Order* object. The next call is *confirm()* which is a method of *SpecialOrder* object and the last call is *Dispatch()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.

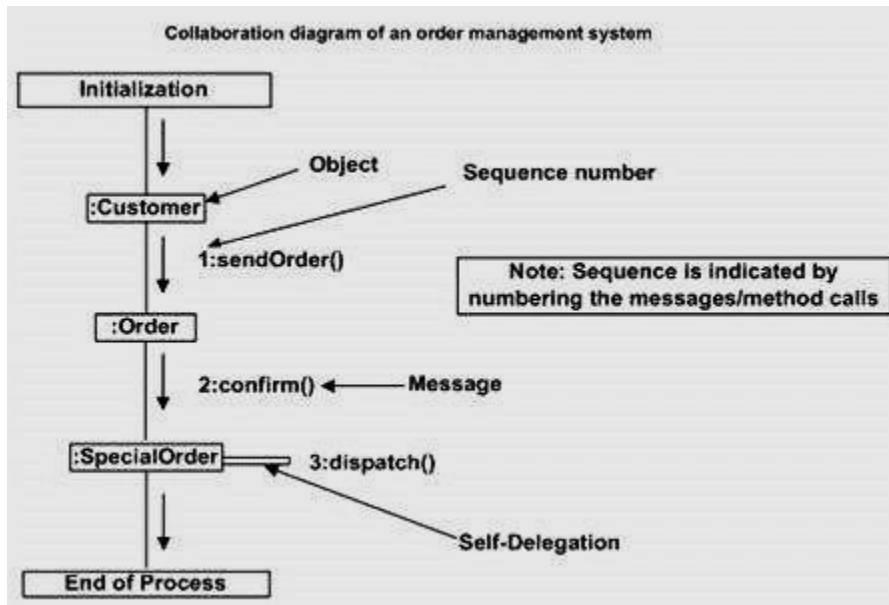


The Collaboration Diagram

The second interaction diagram is the collaboration diagram. It shows the object organization as seen in the following diagram. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization.

To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used.



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:

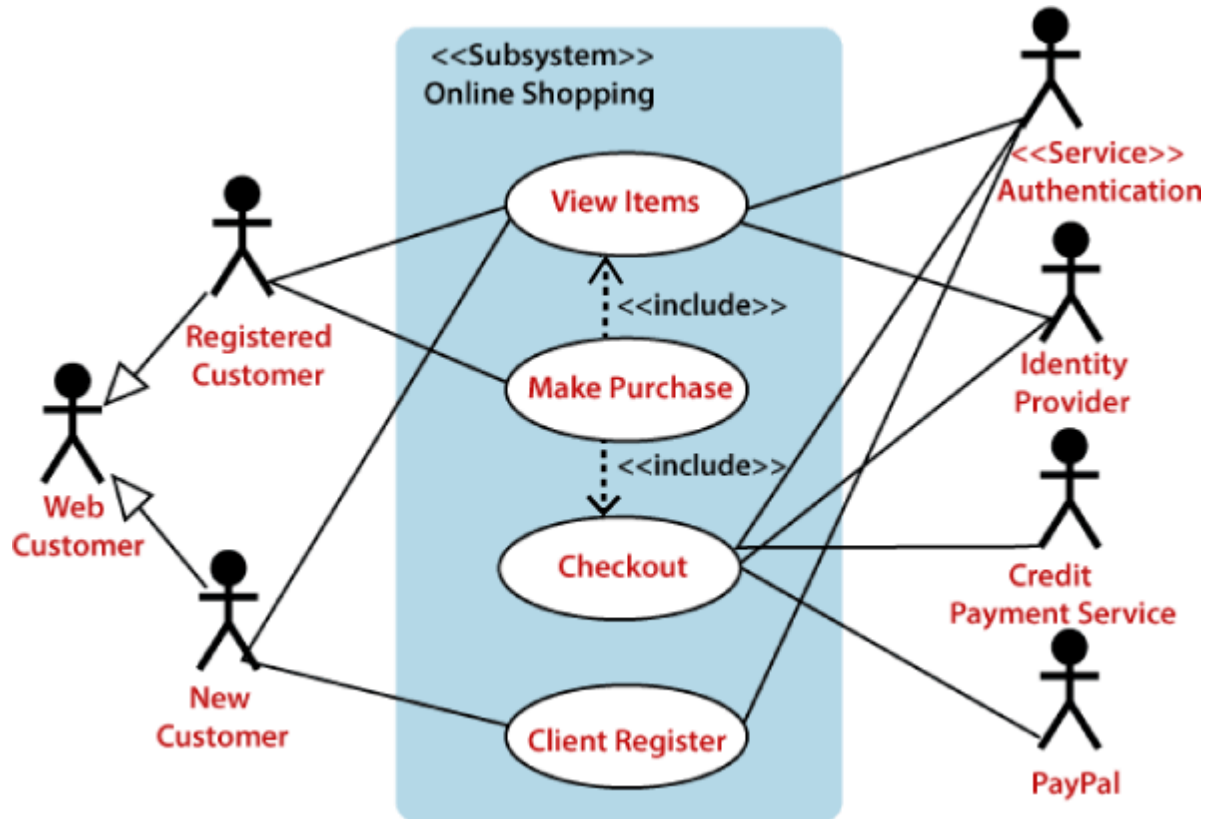
1. A pertinent and meaningful name should be assigned to the actor or a use case of a system.
2. The communication of an actor with a use case must be defined in an understandable way.
3. Specified notations to be used as and when required.

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

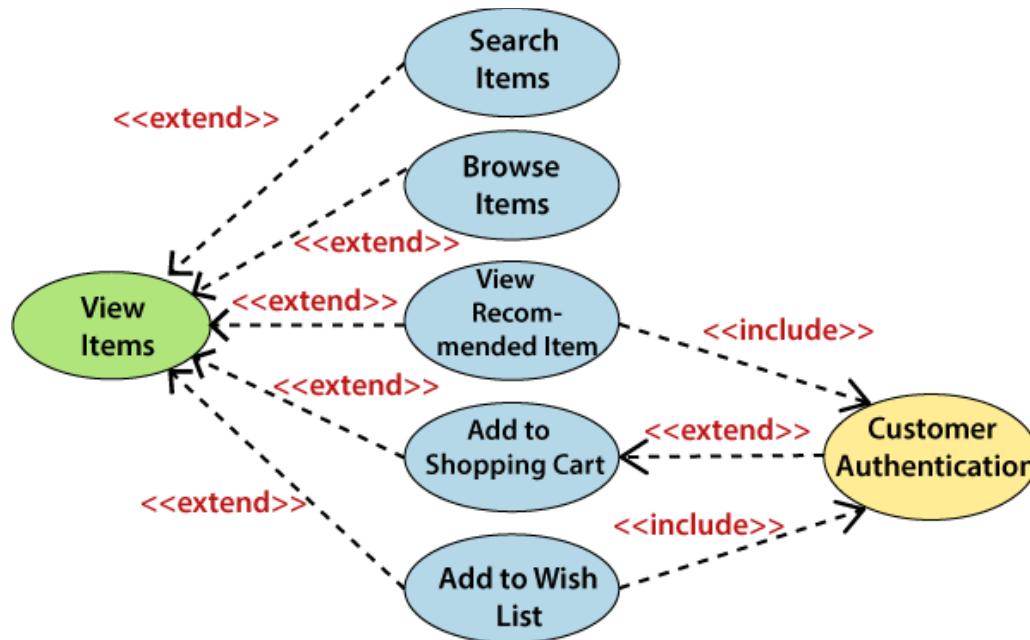
A use case diagram depicting the Online Shopping website is given below.

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.



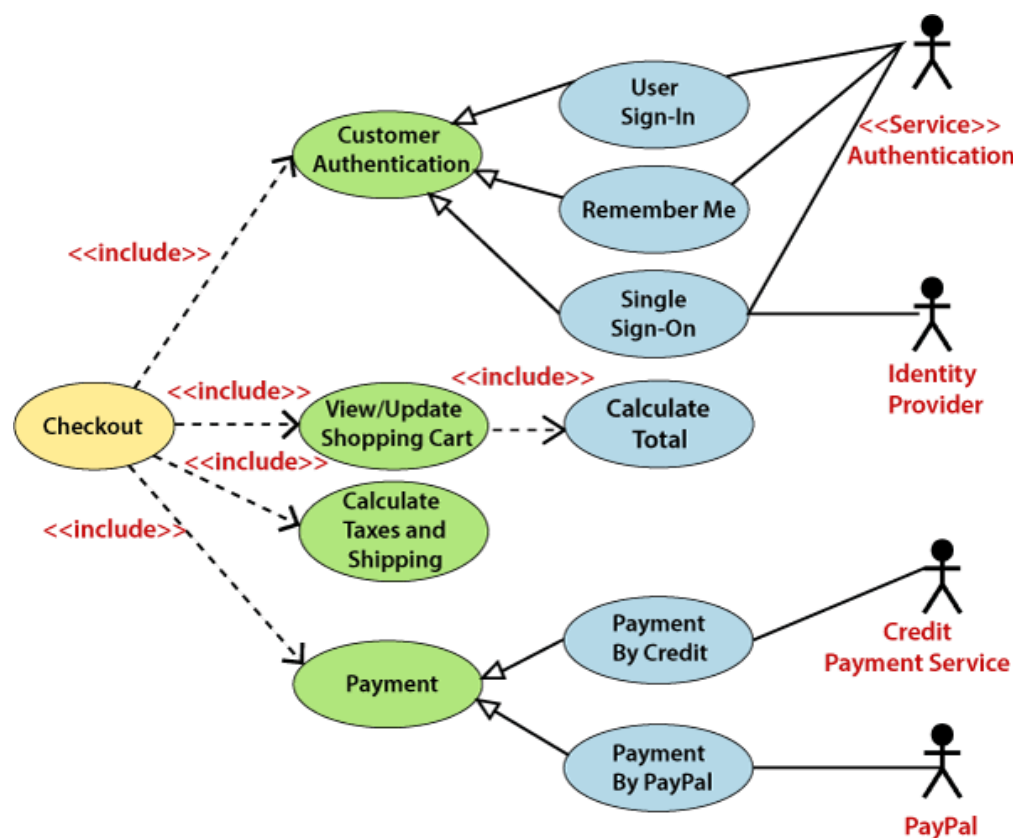
The **View Items** is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item. The View Items is further extended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



Important tips for drawing a Use Case diagram

Following are some important tips that are to be kept in mind while drawing a use case diagram:

1. A simple and complete use case diagram should be articulated.
2. A use case diagram should represent the most significant interaction among the multiple interactions.
3. At least one module of a system should be represented by the use case diagram.
4. If the use case diagram is large and more complex, then it should be drawn more generalized.

Component diagrams

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose of Component Diagrams

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as –

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

How to Draw a Component Diagram?

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries, etc

The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

Initially, the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.

This diagram is very important as without it the application cannot be implemented efficiently. A well-prepared component diagram is also important for other aspects such as application performance, maintenance, etc.

Before drawing a component diagram, the following artifacts are to be identified clearly –

- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

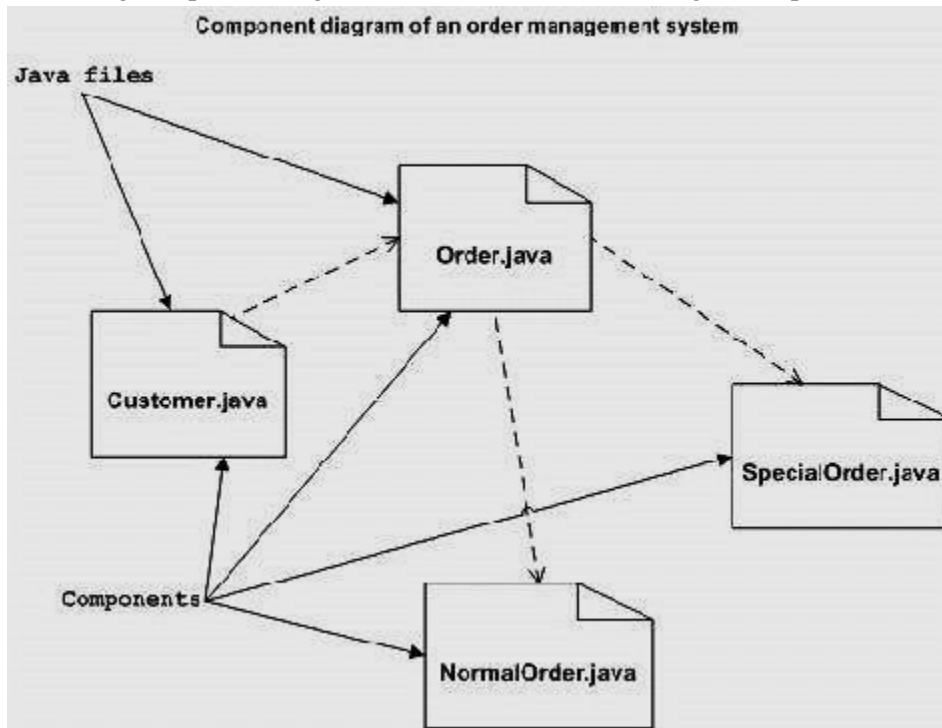
After identifying the artifacts, the following points need to be kept in mind.

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing the using tools.
- Use notes for clarifying important points.

Following is a component diagram for order management system. Here, the artifacts are files. The diagram shows the files in the application and their relationships. In actual, the component diagram also contains dlls, libraries, folders, etc.

In the following diagram, four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far as it is drawn for completely different purpose.

The following component diagram has been drawn considering all the points mentioned above.



Where to Use Component Diagrams?

Component diagrams are special type of UML diagrams used for different purposes.

These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system.

Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.

As we have already discussed, those components are libraries, files, executables, etc. Before implementing the application, these components are to be organized. This component organization is also designed separately as a part of project execution.

Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details

Component diagrams can be used to –

- Model the components of a system.
- Model the database schema.
- Model the executables of an application.
- Model the system's source code.