TEEGALA KRISHNA REDDY ENGINEERING COLLEGE
(UGC-Autonomous)
Approved by AICTE, Affiliated by JNTUH, Accreditated by NAAC- 'A' Grade
Medbowli, Meerpet, Balapur, Hyderabad, Telangana- 500097
Mob: 8498085218. principal@tkrec.ac.in, ace@tkrec.ac.in

Software Engineering
(22AI301PC)

YEAR & SEM	**:**	**B. Tech II YEAR I SEM**
SUBJECT	**:**	**SOFTWARE ENGINEERING**
REGULATION	**:**	**R22**
DEPORTMENT & SEC	**:**	**AIML - C**
NAME OF THE FACULTY	**:**	**Mr. N NARESH**

UNIT 3 SYLLABUS

- **Design Engineering:** Design process and Design quality, Design concepts, the design model, Pattern based software design.
- **Creating an architectural design:** software architecture, Data design, Architectural styles and Patterns, Architectural Design, assessing alternative architectural designs, mapping data flow into software architecture.
- **Modeling component-level design:** Designing class-based components, conducting component-level design, object constraint language, designing conventional components.
- **Performing User interface design:** Golden rules, User interface analysis and design, interface analysis, interface design steps, Design evaluation.

DESIGN ENGINEERING:

- Design engineering consist set of principles, concepts and practices that lead to development of high-quality system.
- Design is a core engineering activity.
- Design creates a representation or model of the software.
- Design model provides details about S/W architecture, interfaces and components that are necessary to implement the system.
- Quality is assessed and established during Design.
- The goal of design engineering is to produce a model or design that exhibit firmness, commodity and delight.
- Design engineering for software changes continually as new methods, better analysis and understanding evolves.

Design with context of software engineering:

- Design is applied regardless of the software process model that is used.
- Design sets the stage for construction (code generation and testing).
- The architectural design defines relationship between major elements of the software, architectural styles and design patterns.

- They are used to achieve requirements defined for system.
- The interface design describes how software communicates with systems that operate with it.
- During design we make decisions that will affect success of software construction and maintenance.
- Importance of software design in single word is – “quality”.
- Design provides representations of software that can be analyzed for quality.
- Design is the only way that we can translate customer’s requirements into a finished software product.
- Design serves as the foundation for all SE and software support activities.
- Without design risk of building an unstable system.

Unstable system is the system that: –

- Fails when small changes are done
- That may be difficult to test
- Whose quality cannot be assessed during process i.e. until big amount is already spent.

DESIGN PROCESS AND DESIGN QUALITY:

- Software design is an iterative process by which requirements are converted into “blueprint” for constructing the software.
- Throughout the design process, the quality of evolving design is assessed with series of formal technical reviews.

Three characteristics that are guide for good design:

- The design must implement all implicit requirements desired by the customer.
- The design must be readable, understandable for those who generate code, and for those who test and support the software.
- The design should provide complete picture of software, addressing data, functions and behavior.

QUALITY GUIDELINES:

- To evaluate the quality of a design representation, we must establish technical criteria for good design.

Following are some quality guidelines:

- A design should be modular; that is software should be logically partitioned into elements or subsystems.
- A design should contain different representation of data, architecture, interfaces and components.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to appropriate data structures for the classes to be implemented.
- A design should lead to components that exhibit Independent functional characteristics.
- A design should lead to interfaces that reduce complexity of connections between components.
- A design should be represented using notation which effectively communicates its meaning.
- These guidelines are not achieved by chance.
- Design engineering gives good design by applying design principles, systematic methodology, and by reviews.

QUALITY ATTRIBUTES:

- HP developed a set of software quality attributes.
- **Known as FURPS quality attributes which represent a target for all software design:**

Functionality: -- It is assessed by:

* Features and capabilities of programs.

* Security of the overall system, functions that are delivered.

Usability: -- It is assessed by:

* User-friendliness

* Aesthetics

* Consistency

* Documentation

Reliability: -- It is evaluated by:

* Measuring the frequency and severity of failure.

* MTTF (mean-time-to-failure), accuracy of output results.

* Ability to recover from failure, and predictability of the program.

Performance: -- It is measured by:

* Processing speed, Response time.

* Resource consumption, Throughput and efficiency.

Supportability: -- It combines:

* Extensibility

* Adaptability

* Serviceability

* Maintainability

* Testability

* Compatibility

* Configurability

MODULE AND COMPONENT:

MODULE:

- A module is a software component or part of a program that contains one or more routines.
- One or more independently developed modules make up a program.
- An enterprise-level software application may contain several different modules, and each module serves unique and separate business operations.
- Modules make a programmer's job easy by allowing the programmer to focus on only one area of the functionality of the software application.
- Modules are typically incorporated into the program (software) through interfaces.
- A classic example of a module-based application is Microsoft Word, which contains modules incorporated from Microsoft Paint that help users create drawings or figures.

COMPONENT:

- In programming and engineering disciplines, a component is an identifiable part of a larger program or construction.
- Usually, a component provides a particular function or group of related functions.
- In programming design, a *system* is divided into *components* that made of *modules*.

- *Component test* means testing all related modules that form a component as a group to make sure they work together.
- In object-oriented programming, a component is a reusable program building block that can be combined with other components in the same or other computers in a distributed network to form an application.
- Examples of a component include: a single button in a graphical user interface, a small interest calculator, an interface to a database manager.
- Components can be deployed on different servers in a network and communicate with each other for needed services.

DESIGN CONCEPTS:

- The software engineer has to recognize the difference between getting a program to work, and getting it right.
- Following fundamental software design concepts provide the necessary framework for "getting it right".

THE DESIGN CONCEPTS ARE:

- 1. Abstraction
- 2. Architecture
- 3. Patterns
- 4. Modularity
- 5. Information Hiding
- 6. Functional Independence
- 7. Refinement
- 8. Re-factoring
- 9. Design Classes

1. ABSTRACTION:

- Many levels of abstraction are there.
- **At Highest level of abstraction** : Solution is stated in broad terms using the language of environment
- **At Lower levels of abstraction:** More detailed description of the solution is provided.
- **Procedural abstraction:** Refers to a sequence of instructions that a specific and limited function.

- An example of procedural abstraction would be the word "**open**" for the "**door**".
- "Open" has sequence of procedural steps (e.g. walk to door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)
- **Data abstraction:** Collection of data that describe a data object.
- An example of data abstraction would be "door".
- Data abstraction for door set of attributes that describe door (e.g. door type, swing, direction, opening mechanism, weight, dimensions).
- Procedural abstraction "**open"** make use of information contained in the attributes of data abstraction "**door".**

2. **ARCHITECTURE**:

- In simplest form, architecture is the structure or organization of program components (modules), structure of data used by the components and their interconnection.
- **Architecture Models :** Architecture can be represented using following different models :

(A). **Structural Models:** Represent architecture as an organized collection of program components.

(b). **Framework Models:**

-- Represents the design in more abstract way.

(c). **Dynamic Models:**

-- Represents the behavioral aspects of architecture indicating configuration change as a function of external events.

(d). **Process Models:**

-- Focus on the design of the business or technical process.

(E). **Functional Models:** Can be used to represent functional hierarchy of system.

3. PATTERNS:

- Design pattern describes a structure that solves a particular design problem.

Provides a description that enables a designer to determine the followings:

(a). whether the pattern is applicable to the current work.

(b). whether the pattern can be reused (saving desired time)

(c). whether the pattern can serve as a guide for developing functionally or structurally different pattern.

4. MODULARITY:

- Divides software into separately named components, sometimes called modules.
- Modules are integrated to satisfy problem requirements
- Modularity is the single attribute of software that allows a program to be manageable.
- Consider two problems p1 and p2.
- If the complexity of p1 is cp1 and of p2 is cp2 then effort to solve p1=cp1 and effort to solve p2=cp2.If cp1>cp2, then ep1>ep2.
- The complexity of two problems when they are combined is often greater than the sum of the complexity when each is taken separately.
- Based on Divide and Conquer strategy: it is easier to solve a complex problem when broken into sub-modules.
- Under modularity or over modularity should be avoided.
- We modularize the design so that development can be easily planned, changes can be easily done, testing can be conducted more efficiently, and long term maintenance can be conducted.

5. INFORMATION HIDING:

- Information contained within a module is inaccessible to other modules who do not need such information.
- Details need not be known by users of the module.
- Effective modularity achieved by defining independent modules.
- These modules communicate with one another only that information necessary to achieve S/W function.
- Provides the greatest benefits when modifications are required during testing and later during software maintenance.
- Most data and procedure are hidden from other parts of software.
- So errors introduced during modification are less likely to propagate to other location within the S/W.

6. FUNCTIONAL INDEPENDENCE:

- Achieved by developing a module with “single minded” function and an “oppose” to interaction with other modules.

- Easier to develop independent modules and simple interface.

Independent modules are easier to maintain because:

- Secondary effects caused by design or code modification are limited.
- Error propagation is reduced.
- Reusable modules are possible.
- Functional independence is a key to good design, and design is the key to software quality.

Independency is assessed by two quantitative criteria:

- **(1) Cohesion**
- **(2) Coupling**

Cohesion:

- It is relative functional strength of a module.
- A cohesive module performs a single task requiring little interaction with other components.
- Simply, cohesive module should do just one thing.

Coupling:

- It is the measure of interconnection among modules.
- It is indication of degree up to which a module is connected to other modules and outside world.
- For good design coupling should be low and cohesion should be high.
- Coupling depends on interface complexity, point at which entry is made to module, and which data pass across interface.

7. REFINEMENT & REFACTORING:

REFINEMENT:

- It is a top-down design strategy proposed by Nicklaus Wirth.
- Refinement is actually Process of elaboration from high level abstraction to the lowest level abstraction.
- High level abstraction begins with a statement of functions.
- Statement describes function but provides no information about internal working of function or internal structure of data.

- Refinement causes the designer to elaborate original statement.
- Provide more and more details at successive level of abstractions.
- Abstraction and refinement are complementary concepts.

REFACTORING:

- Organization technique that simplifies the design of a component without changing its function or behavior.
- Examines for redundancy, unused design elements and inefficient or unnecessary algorithms.

8. DESIGN CLASSES:

- Class represents a different layer of design architecture.

Five types of Design Classes:

- **1. User interface class :**
- -- Defines all abstractions that are necessary for human computer interaction
- **2. Business domain class :**
- -- Refinement of the analysis classes that identity attributes and services to implement some of business domain
- **3.Process class :**
- -- Implements lower level business abstractions required to fully manage the business domain classes
- **4.Persistent class :**
- -- Represent data stores that will persist beyond the execution of the software
- **5.System class :**
- -- Implements management and control functions to operate and communicate within the computer environment and with the outside world.

THE DESIGN MODEL:

The design model can be viewed in two different dimensions:

- Process dimension and Abstract dimension.
- Process dimension indicates the evolution of the design model as design tasks are executed as part of software process.
- Abstraction dimension represents the level of details as each element of the analysis model is transformed into design model.

Data Design elements:

- -- Data design creates a model of data that is represented at a high level of abstraction (user's view of data).
- -- This data model is refined to more specific representation for processing by the computer base system.
- -- Translation of data model into a data base is important to achieve business objective of a system.

Architectural design elements:

- The architectural design for software is equivalent to floor plan.
- Floor plan gives overall layout of the rooms, their size, shape, doors and windows to move in and out of the rooms.
- It gives us an overall view of the house.
- So, architectural design elements give an overall view of software.

Architectural model is derived from three sources:

- (1) Information about the application of the software
- (2) Analysis model such as dataflow diagrams or analysis classes
- (3) Architectural patterns and styles

Interface Design elements:

- Interface design for software is equivalent to a set of drawings for doors, windows, and external utilities of house.
- Drawings shows shape and size of doors and windows, the way in which they operate, the way in utilities connections (water, electrical, gas, telephone) are distributed to rooms shown in floor plan.
- Detailed drawings for doors, windows, and utilities tell us how things and information flow in and out of house and within the rooms.

- Interface design elements for software tell how information flows in and out of system and how it is communicated among components.

There are three elements of interface design:

- (1) User interface
- (2) External interfaces to other systems, devices, networks etc
- (3) Internal interfaces between various components

Component level design elements:

- --Fully describe the internal details of each software component.
- It is equal to set of detailed drawings for each room in a house.
- These drawings show wiring and plumbing in each room, switches, sinks, showers, tubs etc.
- They also describe flooring, moldings to be applied, and other detail related with the room.
- UML diagram can be used for component.
- In fig. a component sensor management (part of safe home security function) is represented.
- Sensor management component performs all functions of safe home sensors like monitoring and configuring them.

Deployment level design elements:

- -- Indicates how software functionality and subsystem will be allocated within the physical computing environment.
- -- UML deployment diagram is developed and refined.
- For e.g. Elements of the software product are configured to operate within three primary computing environments: a home – based PC, the Safe-Home control panel, and a server housed at CPI Corp.
- During design, a UML deployment diagram is developed and then refined shown in figure.
- In figure, three computing environments are shown.
- The subsystems within each computing element are indicated.
- For e.g., the personal computer has subsystems that implement security, surveillance, home management and communication.

- External access subsystem manages all attempts to access Safe-Home system from an external source.
- This deployment diagram is in “descriptor form”.
- Deployment diagram shows computing environment but does not indicate configuration details.
- For e.g. Personal computer is not further identified.
- Details are provided when deployment diagram are revisited in “instance form” during later stages.

PATTERN BASED SOFTWARE DESIGN:

- A Software engineer should look for every opportunity to reuse existing design patterns rather than creating new.

Describing a Design Pattern:

- Mature engineering discipline makes use of thousand design patterns.
- For e.g. Mechanical engineer uses a two step, key shaft as a design pattern.
- Inherent in the pattern are attributes and operations.
- An electrical engineer uses an integrated circuit to solve a specific element of a new problem.
- Design Pattern is described using Template.

Design Pattern Template:

- Pattern Name
- Intent
- Also known as
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Related Patterns
- **A description of the design pattern may also consider a set of design forces.**

- Design forces describe nonfunctional requirements associated with software.
- Design force also defines constraints that restrict manner in which design is to be implemented.
- Design force describes environment and conditions that exist to make pattern applicable.

Using Patterns in Design:

- Design patterns can be used throughout software design.
- The problem description is examined at various level of abstraction to determine whether it is amenable to one or more following types of design patterns like architectural patterns, design patterns, idioms.

Frameworks:

- Sometimes it is necessary to provide an implementation specific infrastructure called framework, for design work.
- So, designer can select “reusable mini architecture”.
- Framework is not an architectural pattern, but it is a skeleton with collection of “plug points”.
- Plug points enable a designer to integrate problem specific classes or functionality.

CREATING AN ARCHITECTURAL DESIGN:

SOFTWARE ARCHITECTURE:

- The software architecture is the structure of the system, which comprise software components, the properties of those components and the relationship among all components.
- Software Architecture is not the operational software.

It is a representation that enables a software engineer to:

- Analyze the effectiveness of design in meeting its requirements.
- Consider architectural alternative at a stage when making design changes is still relatively easy.
- Reduces the risk associated with the construction of the software.

Why is software Architecture Important?

Three key reasons:

--Representations of software architecture enable communication and understanding between stakeholders.

--Highlights early design decisions to create an operational entity.

--Constitutes a model of software components and their interconnection.

DATA DESIGN:

DATA DESIGN AT ARCHITECTURE LEVEL:

It translates data objects into:

- **Data structures at programming level or component level.**
- **Data base at application level.**
- **Data warehouse at business level :**
- The challenge is to extract useful information from large (thousands GB) data environment.
- To solve this, Data mining technique has been developed, also called knowledge discovery in databases (KDD).
- It extracts appropriate business-level information.
- Many factors make data mining difficult with existing environment
- The alternative solution is called data warehouse.
- A data warehouse is a large, independent database that uses the data that are stored in databases that serve the set of applications required by a business.

DATA DESIGN AT COMPONENT LEVEL:

- Focuses on representation of data structures that are directly used by one or more software components.
- **Wasserman proposed set of principles used to specify and design such data structures.**
- **In actual, data design begins during creation of analysis model.**

Principles for data specification:

- 1. Proper selection of data objects and data models.
- 2. Identification of attribute, functions and encapsulation of these within a class.
- 3. Representation of the content of each data object. Class diagrams may be used

- 4. Refinement of data design elements from requirement analysis to component level design elements.
- 5.Information hiding
- 6. A library of useful data structures and operations be developed.
- 7. Software design and PL should support the specification of abstract data types.
- These principles are basis for component level data design.

ARCHITECTURAL STYLES:

- When a builder uses a word “central apartment” to describe the house, we get a general image of what a house will look like.
- Builder has used an architectural style as descriptive mechanism to differentiate from other styles.
- Architectural style is also a template for construction.
- Further details of house must be defined, final dimensions must be specified, customized features may be added, and building materials are to be determined.
- But the style “central apartment “guides the builder in his work.
- The software also exhibits one or many architectural styles.

Each style describes a system category that encompasses:

- (1) A set of *components* (e.g. Database, modules) that perform functions required by a system.
- (2) A set of *connectors* that enables “communication, coordination and cooperation” among components.
- (3) C*onstraints* that define how components can be integrated to form the system.
- (4) S*emantic models* to understand the overall properties of a system

TAXONOMY OF ARCHITECTURAL STYLES:

- **Although many computer based systems have been created over past 50 years, following are number of architectural styles :**

Data-centered Architecture:

- A data store (e.g. A file or database) resides at centre of this architecture and is used by other components that update, add, delete, or modify data within store.

- Next figure shows a data-centered style in which client software uses a central repository.
- Client software uses the data independent of any changes to the data of other client software.
- Data centered architecture also promotes integratibility.
- It means, existing components can be changed and new client components added to architecture without concern of other clients.
- Also data is be passed among clients using blackboard mechanism in which blackboard component transfer information between clients.
- Client components independently execute processes.

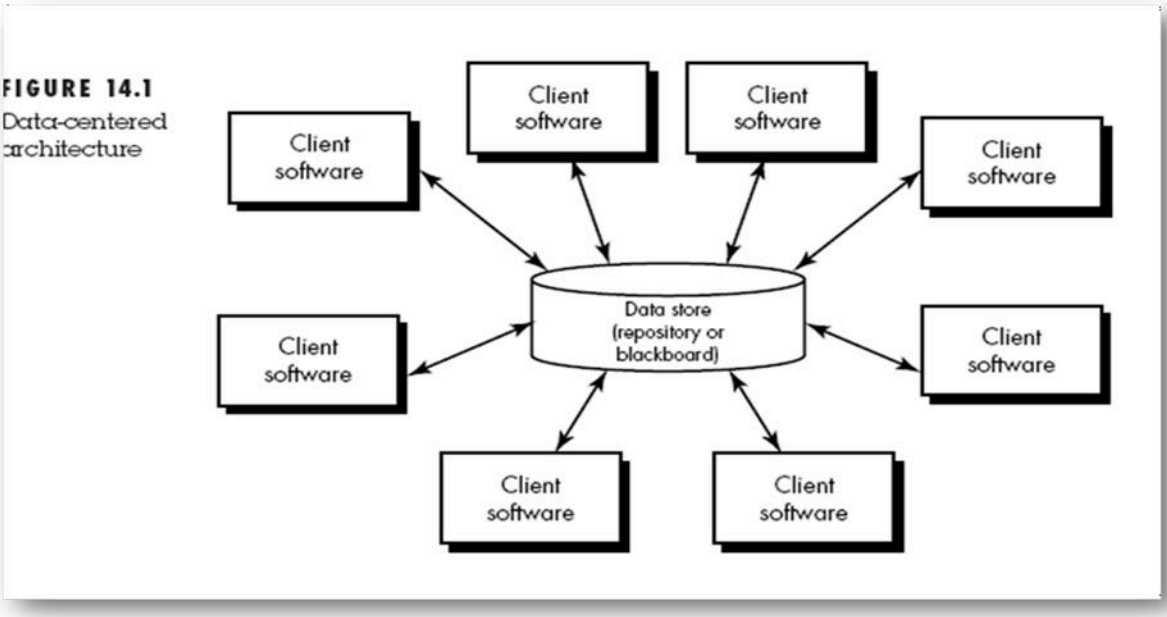
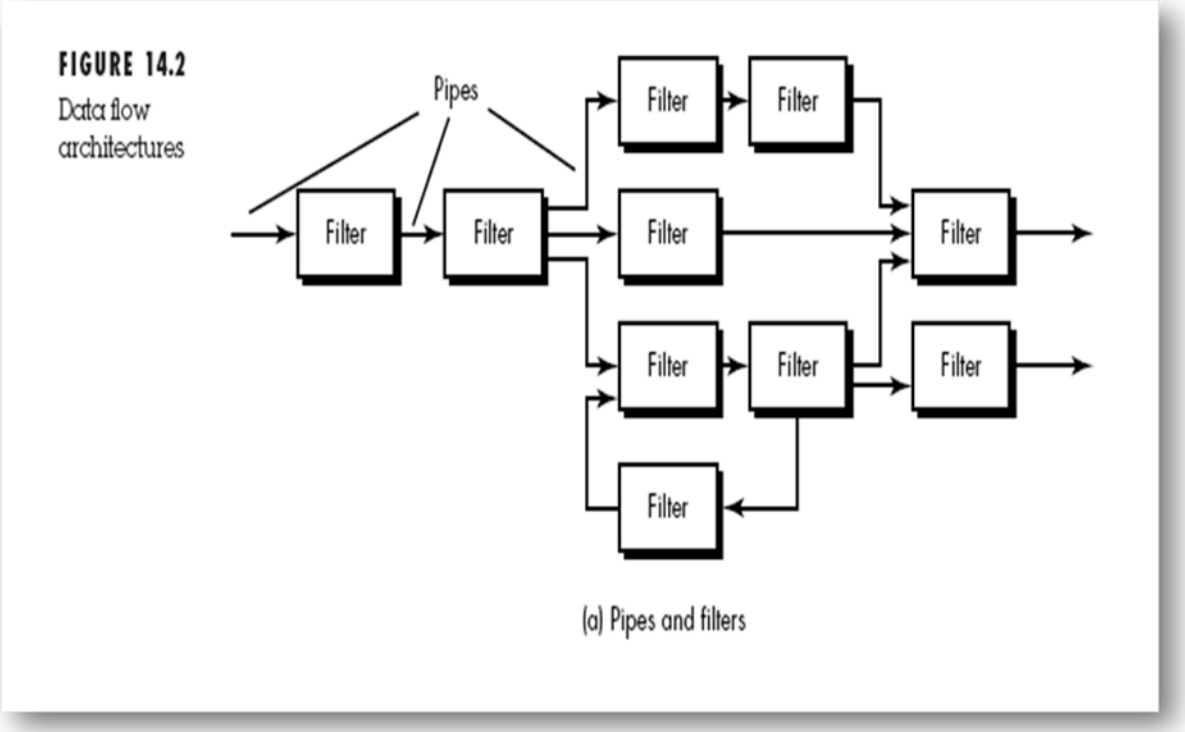


FIGURE 14.1
Data-centered architecture

Data-flow architecture:

- Shows the flow of input data, its components and output data.
- Structure is also called pipe and Filter structure.
- It has a set of components called filters connected by pipes.
- Pipe provides path for flow of data from one component to next.
- Filters expect data input of a certain form, and produces data output(to the next filter) of a particular form.
- Filters manipulate data and work independently of neighboring filter.
- If data flow degenerates into a single line of transform, it is termed as batch sequential.

- This structure accepts a batch of data and then applies a series of sequential components.



Call and return architectures:

- This architecture style enable software engineer to achieve a structure that is easy to modify and scale.

Two sub styles are:

- (1) **Main program/sub program architecture :**
- -- This classic program structure decomposes function into control hierarchy where "main" program invokes a number of components, which in turn invoke still other components.
- (2) **Remote procedure call architecture :**
- -- Components of main program/subprogram are distributed across computers over network.

Object-oriented Architecture:

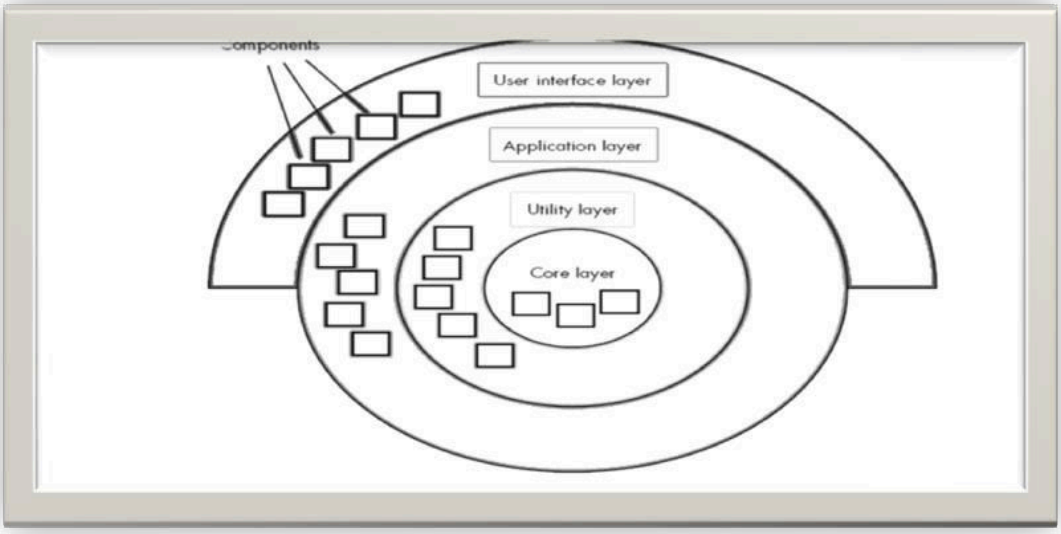
- The components of a system encapsulate data and the operations that are applied to manipulate the data.

- Communication and coordination between components is done by message passing.

Layered architecture:

- A number of different layers are defined
- At Inner Layer (interface with OS), Component perform operating system interfacing.
- Intermediate Layer provides Utility services and application software functions.
- At Outer Layer (User interface), Component service user interface operations.
- Next figure is showing the layered architecture.

FIG: Layered Architecture



ARCHITECTURAL PATTERNS:

- If a house builder decides to construct a centre –hall colonial, there is a single architectural style that can be applied.
- Once decision on architecture of house is made, style is imposed on the design.
- Architectural pattern is little bit different.
- Every architectural style for houses has a kitchen pattern.
- The kitchen pattern defines need for placement of basic kitchen appliances, the need for sink, and need for cabinets.

- In addition, pattern specifies need for counter tops, lighting, wall switches, flooring, so on.
- A template that specifies approach for some behavioral characteristics of the system.
- Patterns are imposed on the architectural styles :

Pattern Domains:

- **1.Concurrency :**
  - --Handles multiple tasks that simulate parallelism.
  - **--Approaches(Patterns):**
  - (a) Operating system process management pattern provides built-in OS features that allow components to execute concurrently.
  - (b) A task scheduler pattern at application level.
  - It contains set of active objects that each contains a tick() operation.
  - **2.Persistence:**
  - --Data survives past the execution of the process.
  - **--Approaches (Patterns) :**
  - (a) Data base management system pattern that applies the storage and retrieval capability of a DBMS.
  - (b) Application Level persistence Pattern(word processing software)
  - **3.Distribution :**
  - -- The distribution problem addresses the communication of system in a distributed environment.
  - There are two elements to this problem :
  - 1. The way in which entities connect to each other.
  - 2. Nature of communication that occurs.
  - **--Approaches(Patterns) :**
  - (a) Broker Pattern :
  - -- Acts as middleman between client and server.
  - Client send message to broker and broker completes the connection.

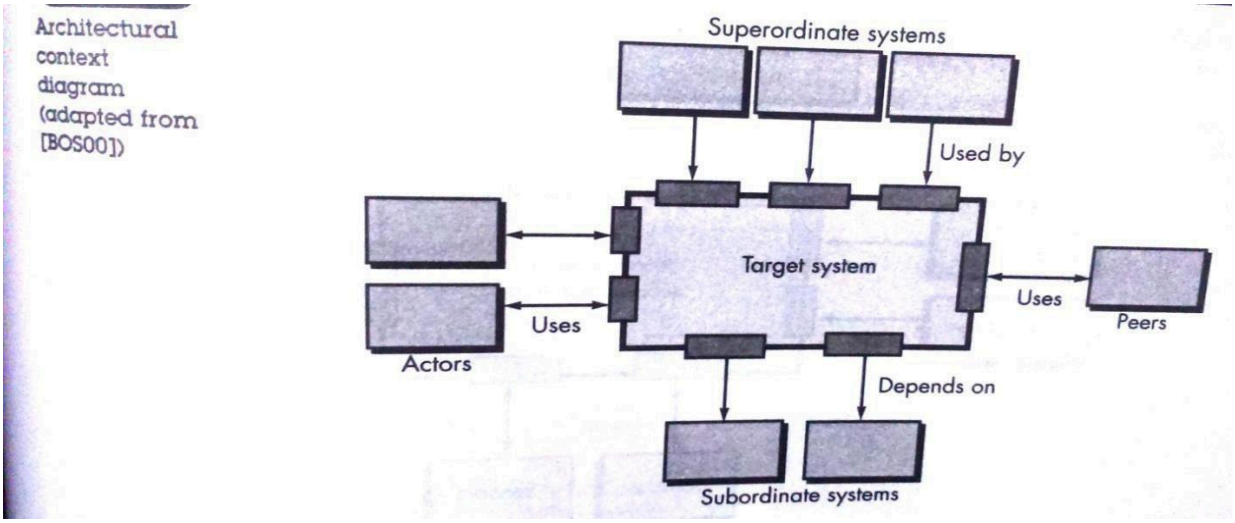
ARCHITECTURAL DESIGN:

- When architectural design begins, software must be put into context.

- Means, the design should define external entities, (other systems, devices, and people).

Representing the system in Context:

- We know that a system engineer must model context.
- At architecture design level, software architect uses an architectural context diagram ACD to model the manner in which software interacts with external entities.
- **Figure shows the architectural context diagram**



- **Super ordinate systems**: systems that use target system.
- **Subordinate systems** : systems that are used by target system and provides data necessary to complete target system functions
- **Peer level system** : systems that interact on peer to peer to basis
- **Actors:** Entities that interact with target system by producing or consuming information
- To understand use of the ACD we consider the ***Safehome*** product for home security function shown in **next figure**.
- At architectural design, details of each interface must be specified.
- And data flow into and out of the target system must be identified at this stage.

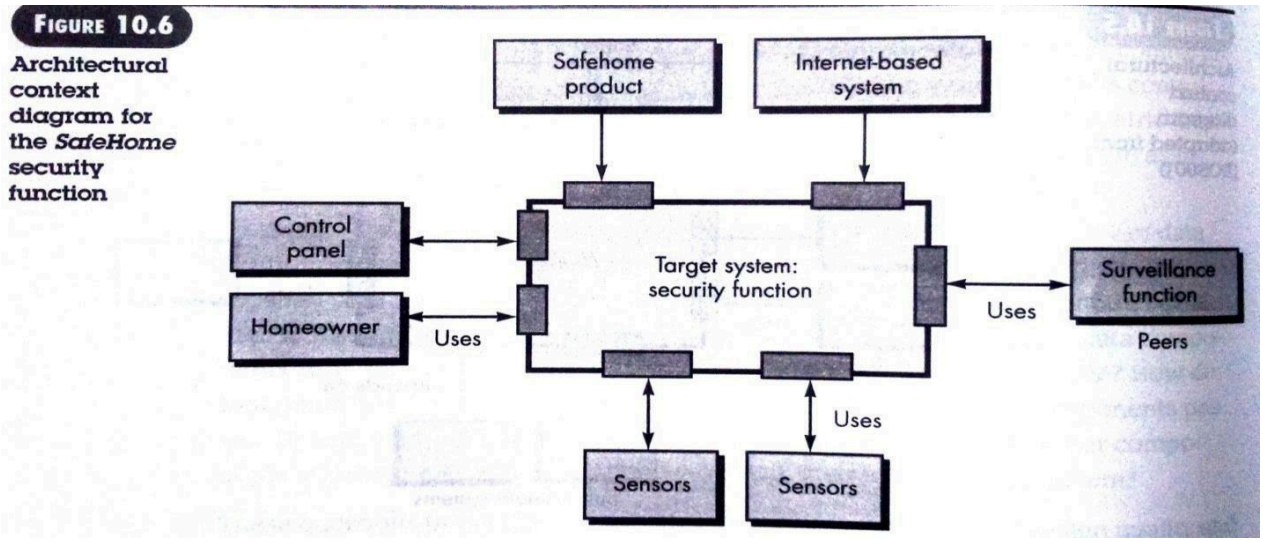
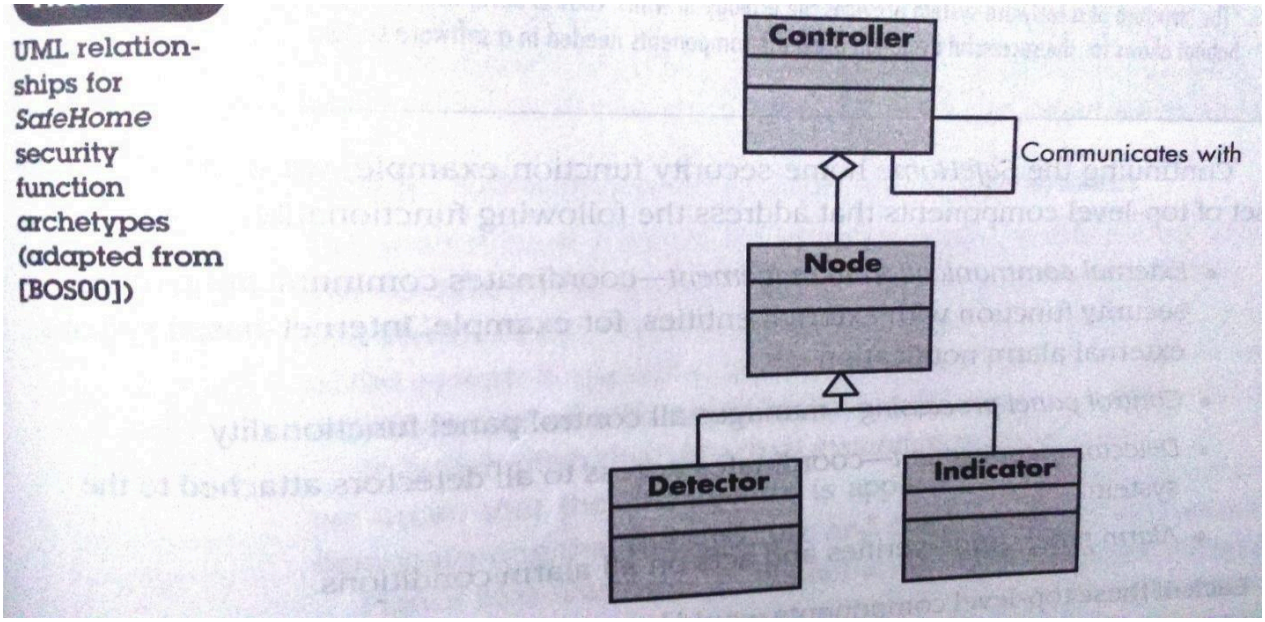


FIGURE 10.6
Architectural context diagram for the *SafeHome* security function

Defining Archetypes:

- An archetype is a class or pattern that represents core abstraction that is critical to the design of architecture of target system.
- Small archetypes are required to design even complex systems.
- For ***safehome*** product home security function, we define following archetypes :
- **Node :** Collection of input and output elements(sensors, alarm indicators)
- **Detector:** An abstraction that represents all sensing equipment that feeds information into target system.
- **Indicator:** An abstraction that represents all mechanisms (alarm siren, flashing lights, bell) which indicates alarm conditions.
- **Controller:** Mechanisms that allows arming or disarming of node.
- Each archetype is shown in **figure** using UML notation.



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

Refining the Architecture into components:

- As software architecture is refined into components, structure of system starting to emerge.
- Application domain and infrastructure domain are two sources for deriving and refinement of components.
- The architecture must have infrastructure components that enable application components.
- For example, memory management components, communication components, database components, task management components.
- For ***SafeHome*** home security function example, we define top level components that provide following functionality :
- **External communication management:** provides communication of security function with external entities.
- **Control panel processing** : manages all control panel functionality
- **Detector management:** coordinates access to all detectors attached.
- **Alarm processing:** Verifies and acts on alarm conditions.
- **Next figure** shows the architectural structure for ***safehome*** with top level components using UML component diagram.
- In fig. transactions are acquired by external communication management that process the safehome GUI and internet interface

- This is managed by safehome executive component which select proper function.
- Control panel processing component interacts with homeowner to arm or disarm security function.
- Detector management component polls sensors to detect alarm condition.
- Alarm processing component produces output when alarm detected

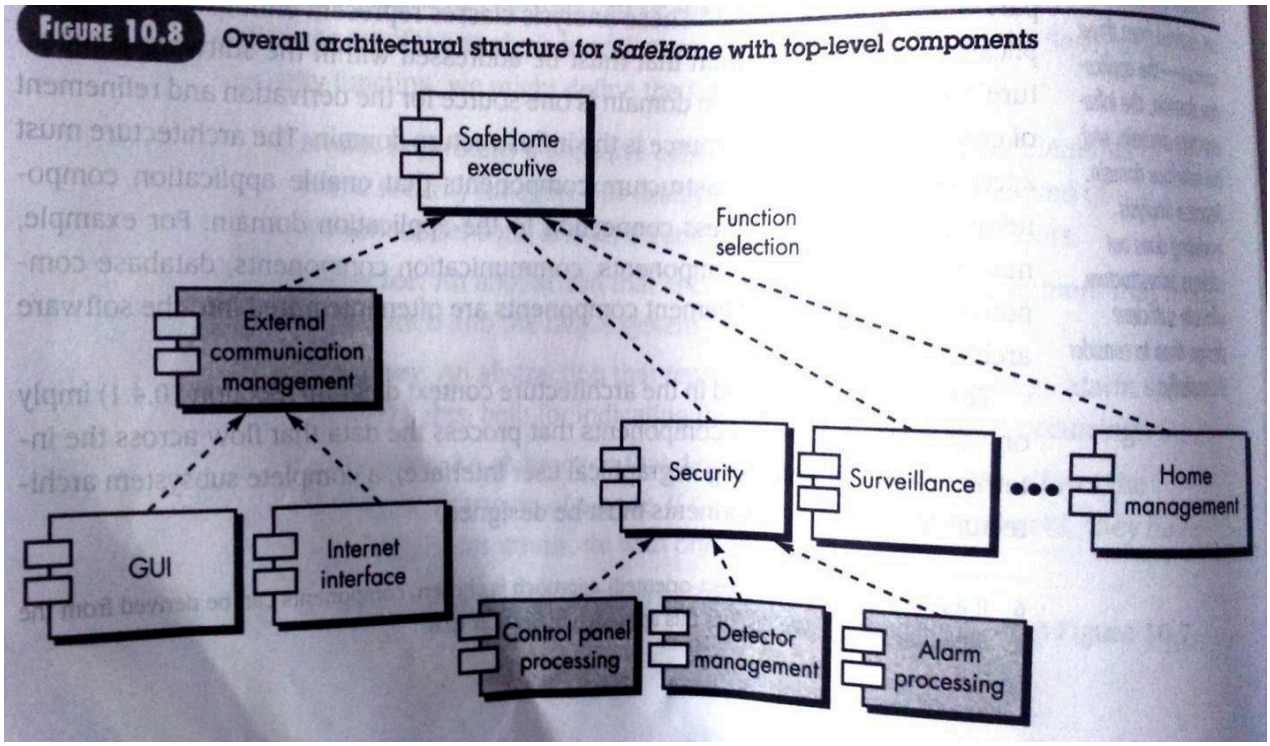


FIGURE 10.8 Overall architectural structure for *SafeHome* with top-level components

ASSESSING ALTERNATIVE ARCHITECTURAL DESIGN:

- **Design gives number of architectural of architectural alternatives that are assessed to determine most appropriate for problem.**

1. An architectural Trade-off analysis Method:

- The Software Engineering Institute SEI has developed architecture trade off analysis method (ATAM) that establishes an iterative process for software architectures.

The design analyses that follow are performed iteratively:

- 1. **Collect scenarios:** A set of use cases is developed to represent the system from user's point of view.
- **2. Elicit requirements, constraints, and environment description:** This is used for that all stakeholders concerns have been addressed.
- **3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:**
- **4. Evaluate quality attributes:** Quality attributes for architectural design assessment include reliability, performance, security maintainability, flexibility, testability, portability, reusability.
- **5. Identify the sensitivity of quality attributes:**
- **6. Critique candidate architectures using sensitivity analysis:**
- These six steps represent first ATAM iteration.

2. Architectural Complexity:

- A useful technique for assessing the complexity of architecture is to consider dependencies between components.
- **Sharing dependencies:** Represent relationship among consumers who use same resource or producers who produce for same consumers.
- For ex. For two components u and v, if u and v refer to same global data.
- **Flow dependencies:** represent dependence relationship between producers and consumers of resources.
- For ex. For two components u and v , if u must complete before control flows into v , or if u communicates with v.
- **Constrained dependencies**: represent constraints on relative flow of control among set of activities.
- For ex. For two components u and v, if u and v cannot execute at same time.

3. Architectural Description Languages:

- ADL provides a syntax and semantics for describing software architecture.
- ADL should provide the designer with ability to decompose architectural component, compose individual components into large architectural blocks.

Importance of Modeling:

- Modeling is central part of all the activities that lead to the deployment of good software.
- We build the models to specify the desired structure and behavior of our system.

- We build the models to better understand the system which we are developing.
- To visualize and control the system architecture.
- Modeling is well accepted engineering technique.
- Model provides the blueprints of system.
- To overcome the drawback of managing large documents.
- Models are easy to understand.
- Various types of diagrams used by developer to start the project after getting all requirements.

- **Through Modeling we achieve four Aims:**
- Model helps us to visualize a system as it is or as we want to be.
- A model helps us to specify the structure or behavior of a system.
- Models give us a template that guides us in constructing a system.
- Models document the decisions we have made.
- **Object oriented modeling:**
- In this approach of software development, main building block of all software systems is object or class.
- An object is a thing; a class is a description of set of common objects.
- Every object has identity, state and behavior**.**

SOFTWARE ENGINEERING LAB

Introduction to UML:

- UML stands for Unified Modeling Language.
- UML is a standard or pictorial language for writing, making or drawing software blueprints.
- UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.
- Modeling language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system.

UML is a language for:

- Specifying the artifacts of software systems.
- Visualizing the artifacts of software systems.
- Constructing the artifacts of software systems.
- Documenting the artifacts of software systems.
- UML was created by Object Management Group (OMG) and UML
- Specification draft was proposed to the OMG in January 1997.
- UML is different from the other common programming languages

Where the UML can be used:

- UML is used primarily for software –intensive systems.
- It has been used effectively for domain like:
- Enterprise information systems
- Banking and Financial services
- Telecommunications
- Transportation
- Defense and Aerospace
- Retail
- Medical Electronics
- Scientific
- Distributed Web based services

CONCEPTUAL MODEL OF UML :

Structural things:

- The Structural things define the static part of the model.
- They represent physical and conceptual elements.
- The Structural things are the nouns of UML models.
- Define the static part of the model.
- They represent physical and conceptual elements.

Following are the brief descriptions of 7 kinds of structural things:

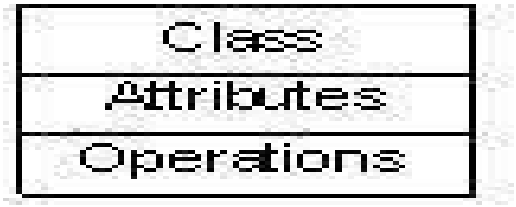
- **Class**
- **Interface**
- **Collaboration**
- **Use case**
- **Active classes**
- **Components**
- **Nodes**

Class:

- Class represents set of objects having same responsibilities, attributes, operations, relationships.
- Class implements one or more interfaces.
- Represented as a rectangle includes name, attributes and operations.
- UML class is represented by the diagram shown below.

The diagram is divided into four parts:

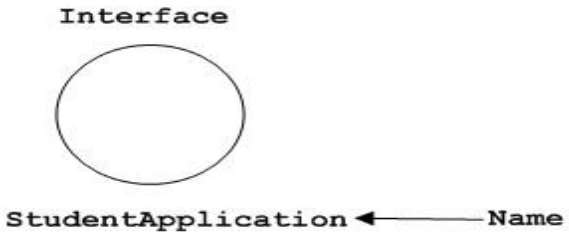
- The top section is used to name the class.
- The second one is used to show the attributes of the class.
- The third section is used to describe operations performed by class.
- The fourth section is optional to show any additional components.
- Class represents set of objects having similar responsibilities.

- 

Interface :

- Interface defines a set of operations which specify the responsibility of a class.

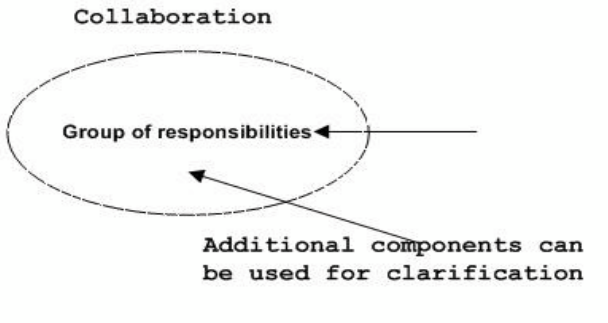
- Interface is represented by a circle as shown below. It has a name which is generally written below the circle.



- Interface is used to describe functionality without implementation.
- . Interface defines a set of operations which specify the responsibility or services of a class.
- Interface describes behavior of class.
- It is attached to the class or component that realizes the interface.

Collaboration:

- Collaboration defines interaction between elements
- Collaboration is represented by a dotted eclipse as shown below. It has a name written inside the eclipse.



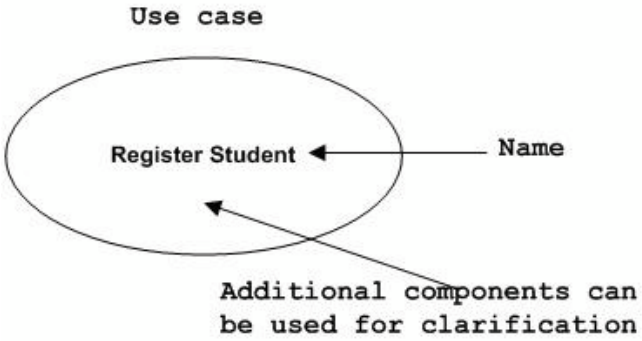
- Collaboration represents responsibilities. Generally responsibilities are in a group.
- Collaboration defines interaction between elements.
- It defines roles and other elements that work together.

Use case:

- Use case represents a set of actions performed by a system for a specific goal.

USE CASE notation

- Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities.

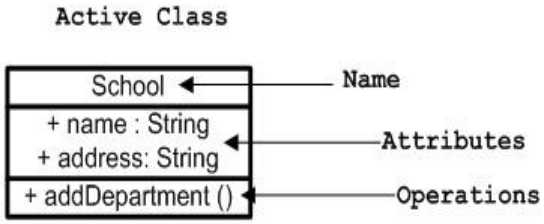


- Use case is used to capture high level functionalities of a system.

- Use case represents a set of series of actions performed by a system for a specific goals of the actor.
- It is realized by collaboration.
- 

Active class Notation:

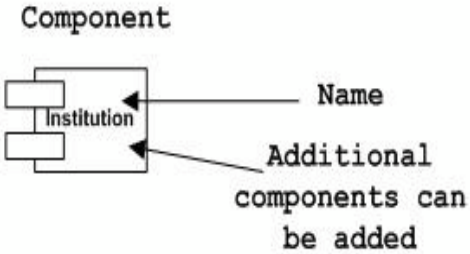
- Active class looks similar to a class with a solid border. Active class is generally used to describe concurrent behavior of a system.

- 

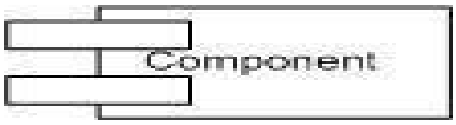
- Active class is used to represent concurrency in a system.

Component:

- Component describes physical part of a system.
- A component in UML is shown as below with a name inside. Additional elements can be added wherever required.

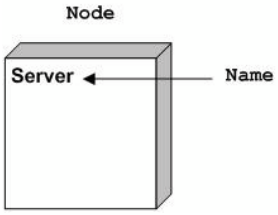


- Component is used to represent any part of a system for which UML diagrams are made.
- Component describes physical and replaceable part of a system.
- Represented as a rectangle with tabs, including only its name.



Node:

- A node can be defined as a physical element that exists at run time
- A node represents a physical component of the system.



- Node is used to represent physical part of a system like server, network etc.
- A node can be defined as a physical element that exists at run time and represents a computational resource.
- It is represented as a cube, including only its name.

Behavioral things :

A behavioral thing consists of the dynamic parts of UML models.

These are the verbs of a model representing behavior over time and space.

These features include interactions and state machines.

Interactions can be of two types:

- Sequential (Represented by sequence diagram)
- Collaborative (Represented by collaboration diagram)

Interaction Notation:

Interaction is basically message exchange between two UML components. The following diagram represents different notations used in an interaction.

Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.

State machine :

State machine is useful when the state of an object in its life cycle is important.

It defines the sequence of states an object goes through in response to events.

Events are external factors responsible for state change.

It defines the sequence of states an object goes through in response to events.

Events are external factors responsible for state change.

A state machine involves number of other elements including states; transition from state to state, events (things that allow the transition) and activities.

Grouping things:

They are the organizational parts of UML.

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available

Package:

Package is the only one grouping thing available for gathering structural **and behavioral things**

It is a general purpose mechanism for organizing elements into groups.

Represented as a tabbed folder, including only its name and sometimes its contents.

Example: Java Package, Framework.

Annotational things:

They are the explanatory parts of UML models.

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** is the only one Annotational thing available

Note:

A note is used to render comments, constraints etc of an UML element.

Note is the only one Annotational thing available.

A note is used to render comments, constraints etc of an UML

element. Ex. Return copy of self.

(2) Relationships:

Relationships are another most important building block of UML.

It shows how elements are associated with each other and this association describes the functionality of an application.

A model is not complete unless the relationships between elements are described properly. The *Relationship gives* a proper meaning to an UML model.

There are four kinds of relationships available.

- Dependency
- Association
- Generalization.
- Realization

Dependency:

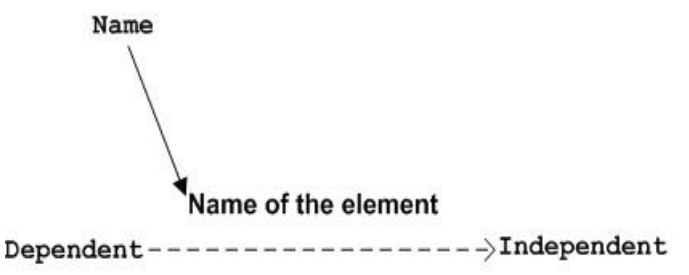
Dependency is a relationship between two things in which change in one element also affects the other one.

Dependency is a relationship between two things in which change in one element also affects the other one.

Represented by dashed line, possibly directed.

It describes the dependent elements and the direction of dependency.

The arrow head represents the independent element and the other end the dependent element.



Association:

Association is basically a set of links that connects elements of an UML

model. It also describes how many objects are taking part in that relationship.

Association describes how the elements in an UML diagram are associated.

In simple word it describes how many elements are taking part in an interaction.

The two ends represent two associated elements as shown below.

Represented as a solid line, possibly directed.

Example: Employer Employee.

Student No. of

subject

Generalization:

Generalization can be defined as a relationship which connects a specialized element with a generalized element.

It basically describes inheritance relationship in the world of

objects. It is parent and child relationship.

Generalization is represented by an arrow with hollow arrow head as shown below.

One end represents the parent element and the other end child element.

Generalization is used to describe parent-child relationship of two elements of a system.

It basically describes inheritance relationship in the world of objects.

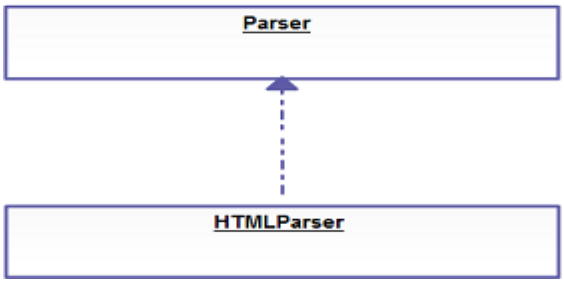
Graphically represented by a solid line with a hollow arrowhead pointing to the parent.

Realization:

Realization can be defined as a relationship in which two elements are connected.

One element describes some responsibility which is not implemented and the other one implements them.

This relationship exists in case of interfaces



(3) Diagrams:

Diagram is the graphical presentation of a set of elements, represented as a connected graph of vertices (things) and arcs (relationships).

Each UML diagram is designed to allow developers and customers view a software system from a different perspective.

UML diagrams commonly created in visual modeling tools.

UML includes 9 kinds of diagrams:

Use case diagram, Class diagram

Activity diagram

Object diagram

Sequence diagram

Collaboration Diagram

State chart diagram

Component diagram

Deployment diagram

Use Case Diagram:

It shows a set of use cases and actors and their relationships.

Use case diagrams address the static use case view of a system.

These diagrams are important in organizing and modeling behavior of a system.

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

Use case diagrams address the static use case view of a system.

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

Class diagrams:

A class diagram shows a set of classes, interfaces, and collaborations and their relationships.

These diagrams are the most common diagram found in modeling object-oriented systems.

Class diagrams address the static design view of a system.

Class diagrams that include active classes address the static process view of a system.

Object diagrams:

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

Object diagrams represent static snapshots of instances of the things found in class diagrams.

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

Interaction diagrams

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them.

Interaction diagrams address the dynamic view of a system.

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages; a

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

Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

State chart diagrams

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

State chart diagrams address the dynamic view of a system.

They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Activity diagrams

An activity diagram is a special kind of a state chart diagram that shows the flow from activity to activity within a system.

Activity diagrams address the dynamic view of a system.

They are especially important in modeling the function of a system and emphasize the flow of control among objects.

Component diagrams

A component diagram shows the organizations and dependencies among a set of components.

Component diagrams address the static implementation view of a system.

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

Deployment diagrams

A deployment diagram shows the configuration of run-time processing nodes and the components that live on them.

Deployment diagrams address the static deployment view of architecture.

They are related to component diagrams in that a node typically encloses one or more components.

This is not a closed list of diagrams.

Tools may use the UML to provide other kinds of diagrams, although these nine are by far the most common you will encounter in practice.

Rules of the UML the UML's building blocks can't simply be thrown together in a random fashion.

Like any language, the UML has a number of rules that specify what a well-formed model should look like.

A well-formed model is one that is semantically self-consistent and in harmony with all its related models.

The UML has semantic rules for

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

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

• Elided	Certain elements are hidden to simplify the view
• Incomplete	Certain elements may be missing

• Inconsistent	The integrity of the model is not guaranteed

These less-than-well-formed models are unavoidable as the details of a system unfold and churn during the software development life cycle.

The rules of the UML encourage you but do not force you to address the most important analysis, design, and implementation questions that push such models to become well-formed over time.

Common Mechanisms in the UML:

A building is made simpler and more harmonious by the conformance to a pattern of common features.

A house may be built in the Victorian or French country style largely by using certain architectural patterns that define those styles.

The same is true of the UML.

It is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications

2. Adornments

3. Common divisions

4. Extensibility mechanisms

CLASS DIAGRAM:

The class diagram is a static diagram. It represents the static view of an application.

A structural diagram that shows a set of classes, interfaces, collaborations, and their relationships.

Class Diagram models class structure and contents using design elements such as classes, packages and objects.

It also displays relationships such as containment, inheritance, associations and others.

Class diagrams are widely used to describe the types of objects in a system and their relationships

Classes are composed of three things: a name, attributes, and operations.

It gives you a static picture of the pieces in the system and of the relationships between them.

Class diagrams help the developers see and plan the structure of the system before the code is written, helping to ensure that the system is well designed from the beginning.

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.

Class diagrams are widely used to describe the types of objects in a system and their relationships.

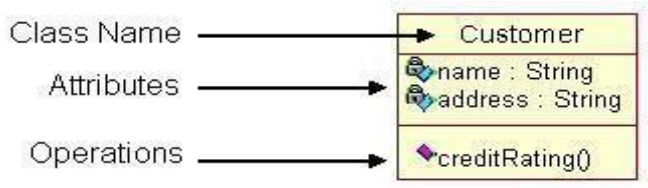
Class diagrams model class structure and contents using design elements such as classes, packages and objects.

Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation.

These perspectives become evident as the diagram is created and help solidify the

design. This example is only meant as an introduction to the UML and class diagrams.

Classes are composed of three things: a name, attributes, and operations.



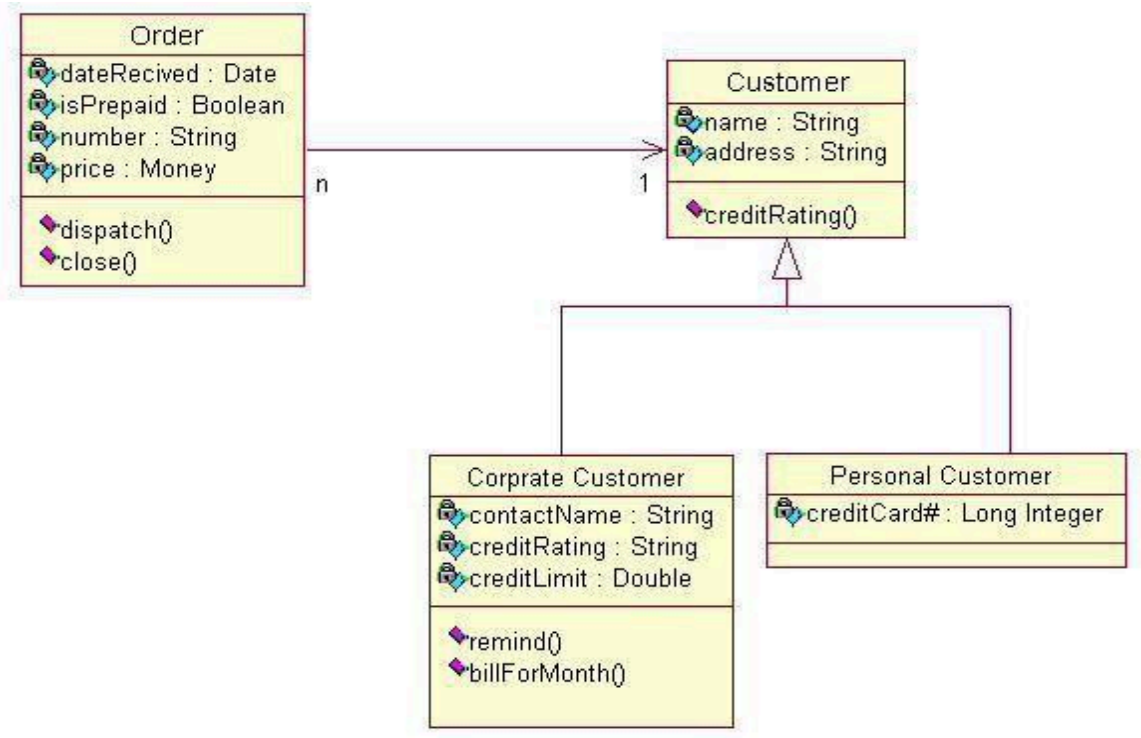
When to Use: Class Diagrams

Class diagrams are used in nearly all Object Oriented software designs.

Use them to describe the Classes of the system and their relationships to each other.

Modeling steps for Class Diagrams

1. Identity the things that are interacting with class diagram.
2. Set the attributes and operations.
3. Set the responsibilities.
4. Identify the generalization and specification classes.
5. Set the relationship among all the things.
6. Adorn with tagged values, constraints and notes.



Sequence Diagram*:*

Displays the time sequence of the objects participating in the interaction.

This consists of the vertical dimension (time) and horizontal dimension (different objects).

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.

The diagrams are read left to right and descending.

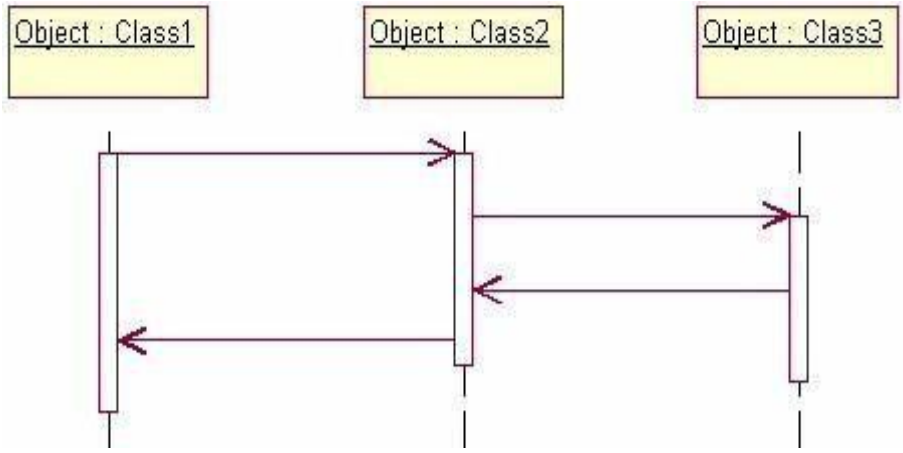
The example below shows an object of class 1 start the behavior by sending a message to an object of class 2.

Messages pass between the different objects until the object of class 1 receives the final message

Modeling steps for Sequence Diagrams

1. Set the context for the interactions, system, subsystem, classes, object or use cases.
2. Set the stages for the interactions by identifying objects which are placed as actions in interaction diagrams.
3. Lay them out along the X-axis by placing the important object at the left side and others in the next subsequent.
4. Set the lifelines for each and every object by sending create and destroy messages.

5. Start the message which is initiating interactions and place all other messages in the increasing order of items.
6. Specify the time and space constraints.
7. Set the pre and post conditions.



A behavioral diagram that shows an interaction, emphasizing the time ordering of messages.

Purpose:

1. To capture dynamic behavior of a system.
2. To describe the message flow in the system.
3. To describe structural organization of the objects.
4. To describe interaction among objects.

Contents of a Sequence Diagram

- Objects
- Focus of control
- Messages
- Life

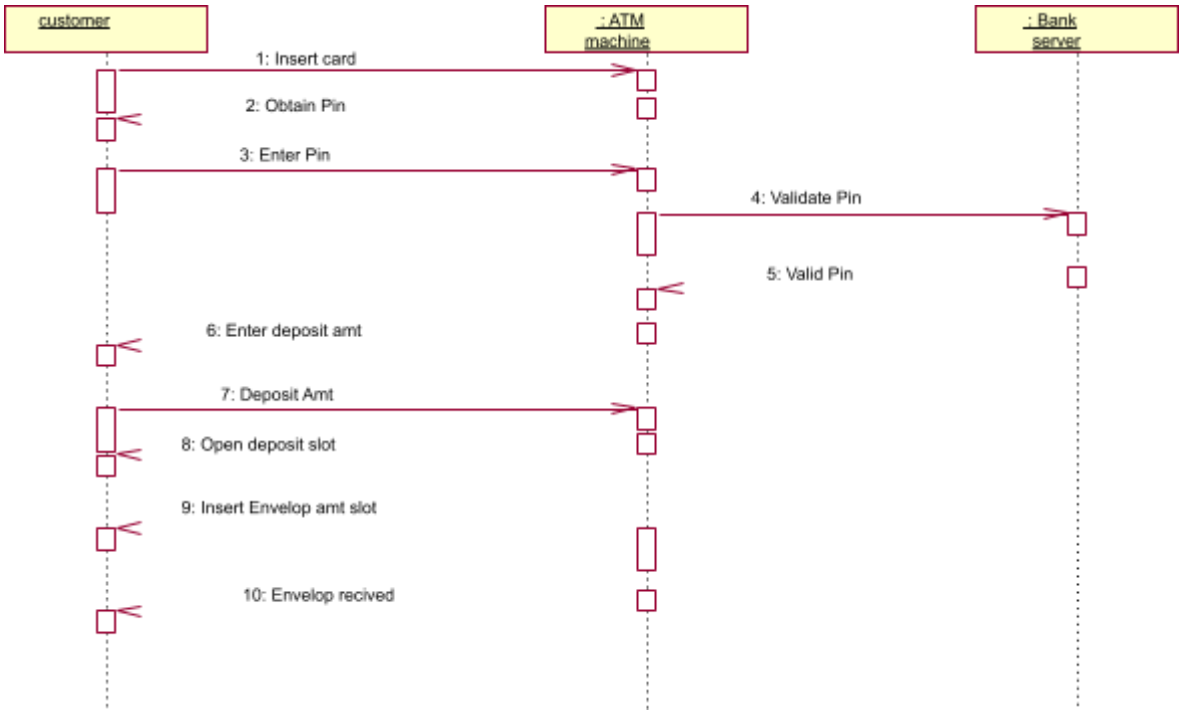
line

Contents

The sequence diagram captures the time sequence of message flow from one object to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.

Sequence diagrams are usually created to show the flow of functionality and control throughout the objects in the system

Sequence diagrams are usually created to show the flow of functionality and control throughout the objects in the system



SEQUENCE DIAGRAM FOR ATM DEPOSIT

Collaboration diagrams:

A behavioral diagram that shows an interaction, emphasizing the structural organization of the objects that send and receive massages.

Collaboration diagrams are not time-based, but show the interactions of the objects as a whole.

They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers.

Sequencing of messages is shown on a Collaboration diagram by numbering the messages.

As the name suggests, they show the sequence of the messages as they are passed between the objects

Contents of a Collaboration Diagram

- Objects
- Links
- Messages

Collaboration diagrams are also relatively easy to draw.

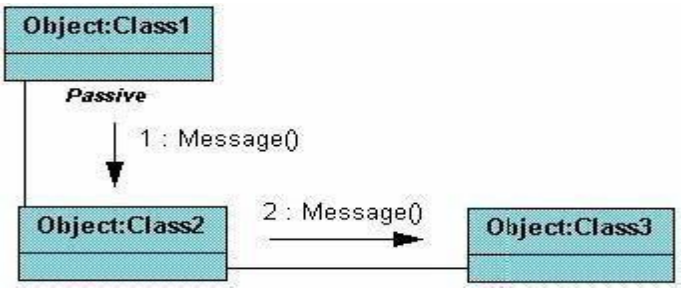
They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers.

As the name suggests, they show the sequence of the messages as they are passed between the objects.

There are many acceptable sequence numbering schemes in UML.

Modeling steps for Collaboration Diagrams

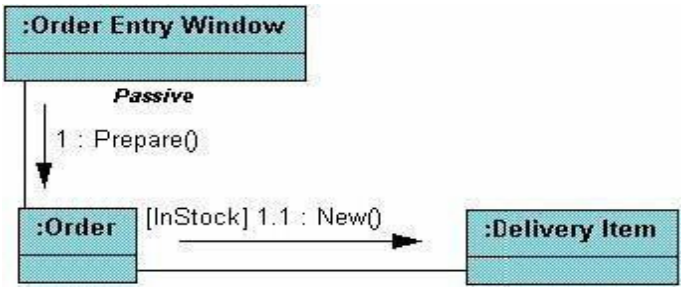
1. Set the context for interaction, whether it is system, subsystem, operation or class or one scenario of use case or collaboration.
2. Identify the objects that play a role in the interaction. Lay them as vertices in graph, placing important objects in centre and neighboring objects to outside.
3. Set the initial properties of each of these objects. If the attributes or tagged values of an object changes in significant ways over the interaction, place a duplicate object, update with these new values and connect them by a message stereotyped as become or copy.
4. Specify the links among these objects.
5. Lay the association links first represent structural connection lay out other links and adorn with stereotypes.
6. Starting with the message that initiates this interaction, attach each subsequent message to appropriate link, setting sequence number as appropriate.

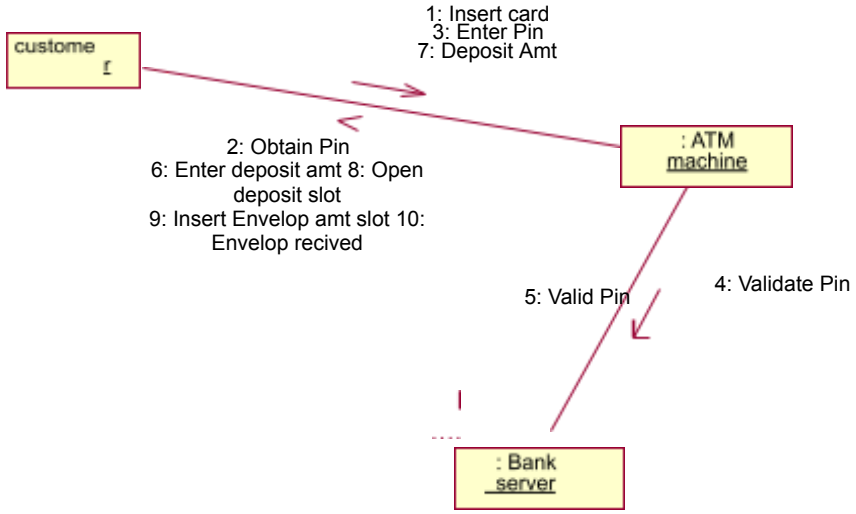


The example below shows a simple collaboration diagram for the placing an order use case.

This time the names of the objects appear after the colon, such as: Order Entry Window following the objectName: className naming convention.

This time the class name is shown to demonstrate that all of objects of that class will behave the same way.





DEPOSIT COLLABORATION DIAGRAM

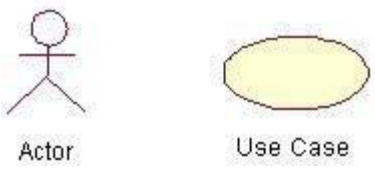
Use Case Diagram :

Displays the relationship among actors and use cases

A use case is a set of scenarios that describing an interaction between a user and a

system. A use case diagram displays the relationship among actors and use cases.

The two main components of a use case diagram are use cases and actors.



An actor is represents a user or another system that will interact with the system you are modeling.

A use case is an external view of the system that represents some action the user might perform in order to complete a task.

When to Use: Use Cases Diagrams

Use cases are used in almost every project.

A behavioral diagram that shows a set of use cases (actions) and actors and their relationships.

To model a system the most important aspect is to capture the dynamic behavior.

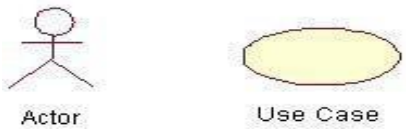
Use case diagram is dynamic in nature

Looking at a Use Case diagram, you should easily be able to tell what the system will do and who will interact with it.

A use case is a set describing an interaction between a user and a system.

The two main components of a use case diagram are use cases and

actors.



These are helpful in exposing requirements and planning the project.

During the initial stage of a project most use cases should be defined, but as the project continues more might become visible.

Modeling steps for Use case Diagram

1. Draw the lines around the system and actors lie outside the system.
2. Identify the actors which are interacting with the system.
3. Separate the generalized and specialized actors.

4. Identify the functionality the way of interacting actors with system and specify the behavior of actor.
5. Functionality or behavior of actors is considered as use cases.
6. Specify the generalized and specialized use cases.
7. Se the relationship among the use cases and in between actor and use cases.
8. Adorn with constraints and notes.
9. If necessary, use collaborations to realize use cases.

How to Draw: Use Cases Diagrams

Use cases are a relatively easy UML diagram to draw, but this is a very simplified example.

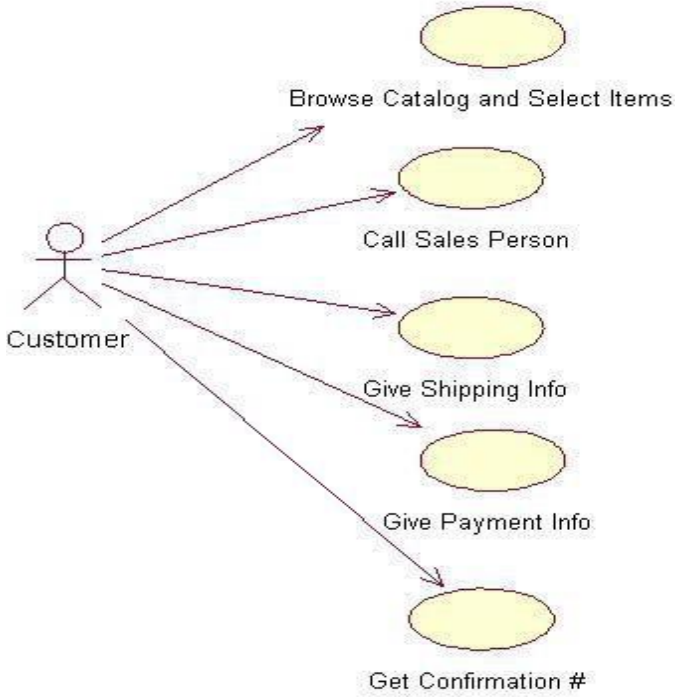
This example is only meant as an introduction to the UML and use cases.

Start by listing a sequence of steps a user might take in order to complete an action.

For example a user placing an order with a sales company might follow these steps.

1. Browse catalog and select items.
2. Call sales representative.
3. Supply shipping information.
4. Supply payment information.
5. Receive conformation number from salesperson.

These steps would generate this simple use case diagram:



This example shows the customer as a actor because the customer is using the ordering system.

The diagram takes simple steps listed above and shows them as actions customer might perform.

The salesperson could also be included in this use case diagram because the salesperson is also interacting with the ordering system.

From this simple diagram the requirements of the ordering system can easily be derived.

So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors

USE CASE DIAGRAM OF ATM:

COMPONENT DIAGRAM:

Component diagrams are basically used to model static view of the system.

This can be achieved by modeling various physical components like libraries, tables, and files etc. which are residing within a node.

Component diagrams are very essential for constructing executable systems.

Graphical representation of component diagrams basically include collection of vertices and

arcs. Displays the high level packaged structure of the code itself.

Dependencies among components are shown, including source code components, binary code components, and executable components.

Some components exist at compile time, at link time, at run times well as at more than one time.

Component diagrams show the software components of a system and how they are related to each other.

These relationships are called dependencies.

Component diagrams are basically used to model static view of the system.

This can be achieved by modeling various physical components like libraries, tables, files etc. which are residing within a node.

Component diagrams show software components of system and how they related to each other.

These relationships are called dependencies.

The component diagram contains components and dependencies.

Components represent the physical packaging of a module of code.

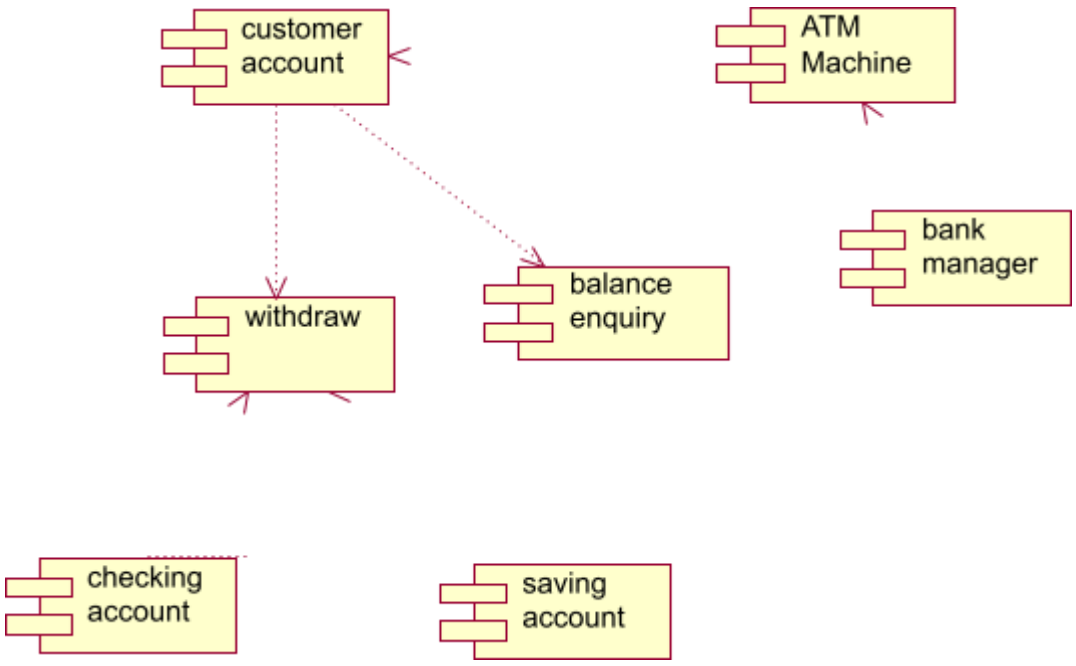
The dependencies between the components show how changes made to one component may affect the other components in the system.

Dependencies in a component diagram are represented by a dashed line between two or more components.

Component diagrams can also show the interfaces used by the components to communicate to each other.

Modeling steps for Component Diagrams

1. Identify the component libraries and executable files which are interacting with the system.
2. Represent this executables and libraries as components.
3. Show the relationships among all the components.
4. Identify the files, tables, documents which are interacting with the system.
5. Represent files, tables, documents as components.
6. Show the existing relationships among them generally dependency.
7. Identify the seams in the model.
8. Identify the interfaces which are interacting with the system.
9. Set attributes and operation signatures for interfaces.
10. Use either import or export relationship in b/w interfaces & components.
11. Identify the source code which is interacting with the system.
12. Set the version of the source code as a constraint to each source code.
13. Represent source code as components.
14. Show the relationships among components.



DEPLOYMENT DIAGRAM:

Deployment diagrams show the physical relationship between hardware and software in a system.

A Deployment diagram shows all of the nodes on the network, the connections between them, and the processes that will run on each one.

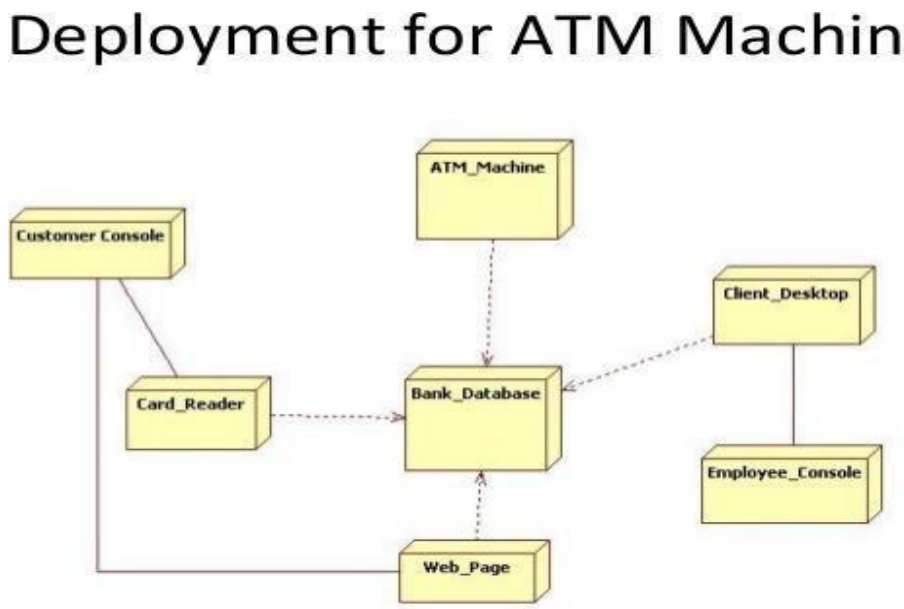
Deployment diagrams illustrate the physical distribution of a system.

These diagrams show the processors, devices, connections, and processes involved in the system

The deployment diagram contains nodes and connections.

A node usually represents a piece of hardware in the system.

A connection depicts the communication path used by the hardware to communicate



ACTIVITY DIAGRAM:

Activity diagram is basically a flow chart to represent the flow form one activity to another.

Activity diagrams describe the workflow behavior of a system.

Activity diagrams are similar to state diagrams because activities are the state of doing something.

An activity diagram represents the execution state of a mechanism as a sequence of steps grouped sequentially as parallel control flow branches.

Activity diagrams describe the workflow behavior of a system.

Activity diagrams are similar to state diagrams because activities are state of doing something.

The diagrams describe the state of activities by showing the sequence of activities performed.

Activity diagrams can show activities that are conditional or parallel.

This diagram focuses on flows driven by internal processing.

The diagrams describe the state of activities by showing the sequence of activities performed.

Activity diagrams can show activities that are conditional or parallel.

The activity can be described as an operation of the system.

So the control flow is drawn from one operation to another.

This flow can be sequential, branched or concurrent.

Activity diagrams deals with all type of flow by using elements like fork, join etc.

Fork: A fork represents the splitting of a single flow of control into two or more concurrent Flow of control.

A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.

Join: A join represents the synchronization of two or more concurrent flows of

control. A join may have two or more incoming transition and one outgoing transition.

Above the join the activities associated with each of these paths continues in parallel.

Branching: A branch specifies alternate paths takes based on some Boolean expression.

Branch is represented by diamond Branch may have one incoming transition and two or more outgoing one on each outgoing transition.

Swim lane: Swim lanes are useful when we model workflows of business processes to partition the activity states on an activity diagram into groups.

Each group representing the business organization responsible for those activities, these groups are called Swim lanes

Modeling steps for Activity Diagrams

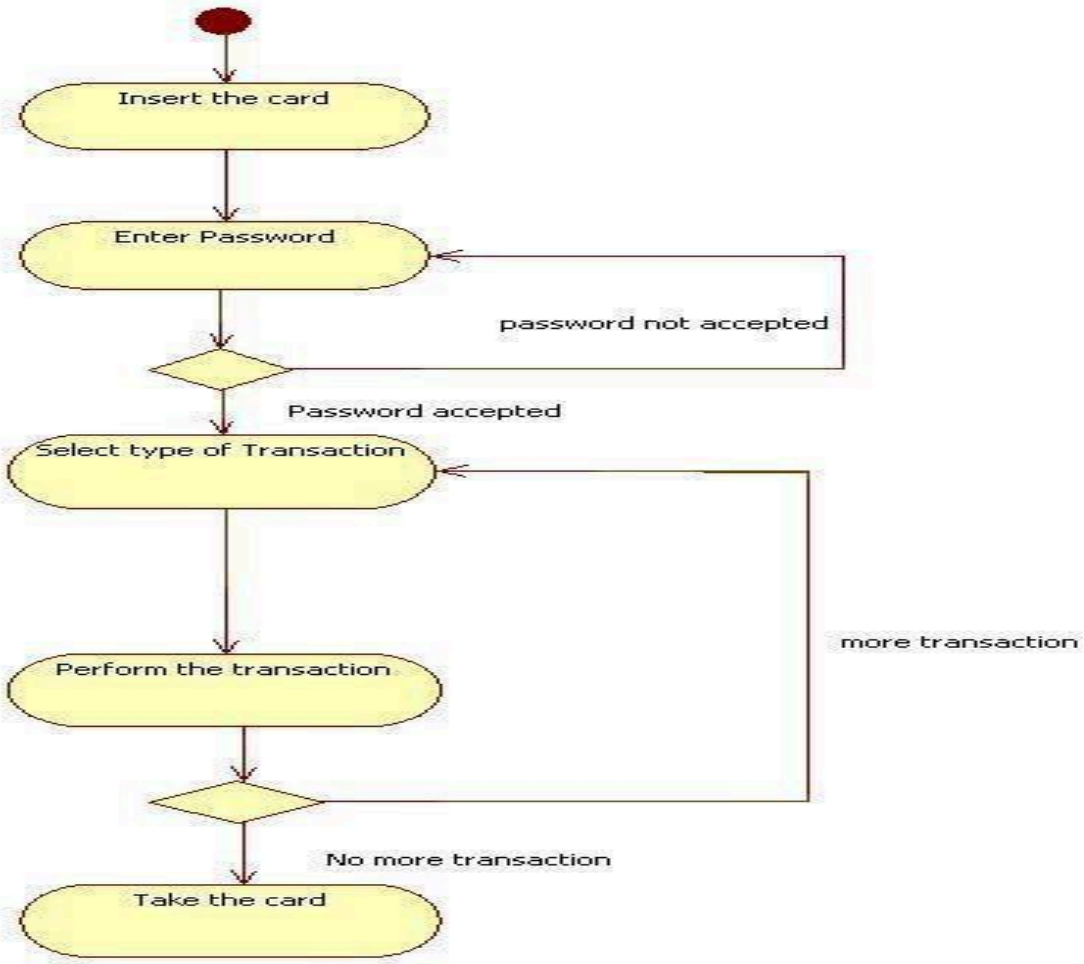
1. Select the object that has high level responsibilities.
2. These objects may be real or abstract. In either case, create a swim lane for each important object.
3. Identify the precondition of initial state and post conditions of final state.
4. Beginning at initial state, specify the activities and actions and render them as activity states or action states.
5. For complicated actions, or for a set of actions that appear multiple times, collapse these states and provide separate activity diagram.
6. Render the transitions that connect these activities and action states.
7. Start with sequential flows; consider branching, fork and joining.
8. Adorn with notes tagged values and so on.

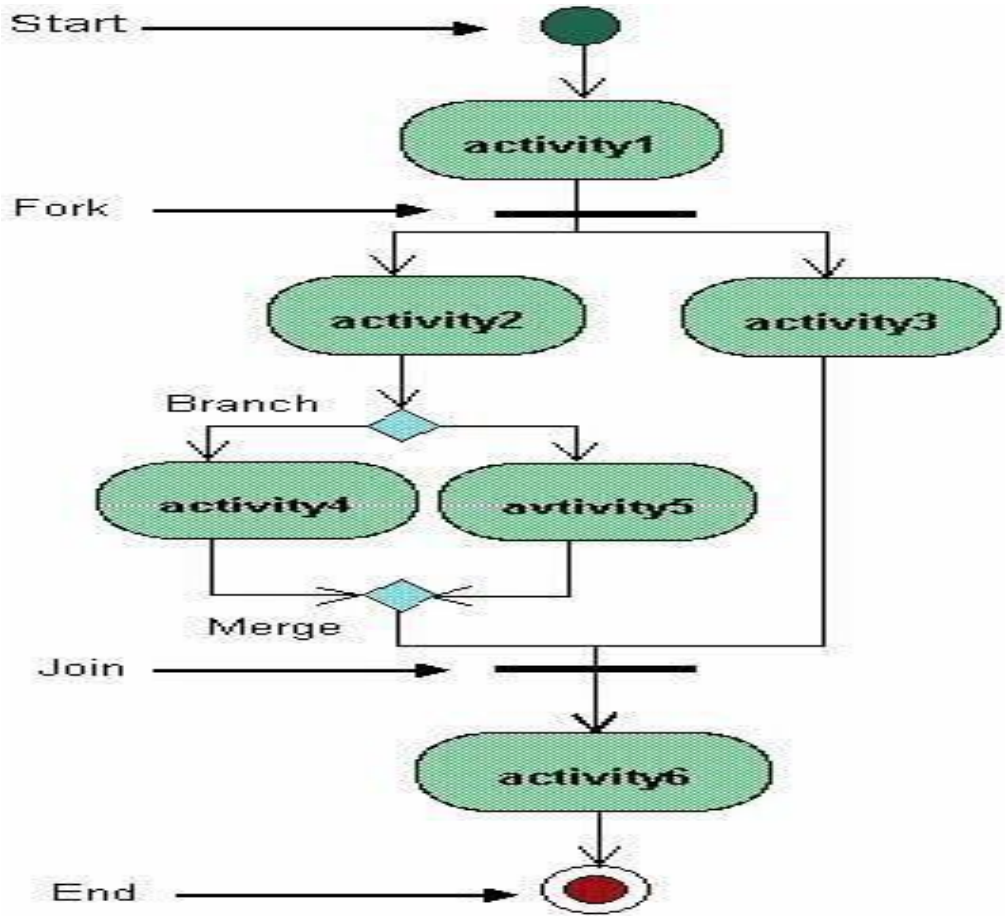
When to Use: Activity Diagrams

Activity diagrams should be used in conjunction with other modeling techniques such as interaction diagrams and state diagrams.

Main reason to use activity diagrams is to model the workflow behind system being designed.

Activity Diagrams are also useful for analyzing a use case by describing what actions need to take place and when they should occur.




Start
activity1
Fork
activity2
activity3
Branch
activity4
avtivity5
Merge
Join
activity6
End