

CHAPTER - 3

Software Modeling

3.1 Software Engineering Principles and Practice

Software engineering practice is a broad array of *principles*, *concepts*, *methods*, and *tools* that we must consider as software is planned and developed. Principles that guide practice establish a foundation from which software engineering is conducted. The software process provides everyone involved in the creation of a computer-based system or product with a road map for getting to a successful destination. Principles instructs us on how to drive, where to slow down, and where to speed up.

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods. At the *process level*, core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products. At the *level of practice*, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

Principles That Guide Practice

Software engineering practice has a single overriding goal—to deliver on-time, high quality, operational software that contains functions and features that meet the needs of all stakeholders. To achieve this goal, we should adopt a set of core principles that guide all technical work. These principles have merit regardless of the analysis and design methods, the construction techniques (e.g., programming language, automated tools), or the verification and validation approach chosen. The following set of core principles are fundamental to the practice of software engineering:

Principle 1. *Divide and conquer.* A large problem is easier to solve if it is subdivided into a collection of elements (or *concerns*). Ideally, each concern delivers distinct functionality that can be developed, and in some cases validated, independently of other concerns.

Principle 2. *Understand the use of abstraction.* An abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase. In analysis and design work, a software team normally begins with models that represent high levels of abstraction (e.g., a spreadsheet) and slowly refines those models into lower levels of abstraction (e.g., a *column* or the *SUM* function).

Principle 3. *Strive for consistency.* Whether it's creating a requirements model, developing a software design, generating source code, or creating test cases, the principle of consistency suggests that a familiar context makes software easier to use. As an *example*, consider the design of a user interface for a WebApp. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons all help to make the interface ergonomically sound.

Principle 4. *Focus on the transfer of information.* Software is about information transfer—from a database to an end user, legacy system to a WebApp, operating system to an application, etc. In every case of information flows, there are opportunities for error, or omission, or ambiguity. This principle focus on pay special attention to the analysis, design, construction, and testing of interfaces.

Principle 5. Build software that exhibits effective modularity. Any complex system can be divided into modules (components), each module should focus exclusively on one well-constrained aspect of the system—it should be cohesive in its function and/or constrained in the content it represents. Additionally, modules should be interconnected in a relatively simple manner—each module should exhibit low coupling to other modules, to data sources, and to other environmental aspects.

Principle 6. Look for patterns. The goal of patterns within the software community is to create a body of knowledge for future planning which defines our understanding of good architectures that meet the needs of their users.

Principle 7. When possible, represent the problem and its solution from a number of different perspectives. When a problem and its solution are examined from a number of different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered.

Principle 8. Remember that someone will maintain the software. Over the long term, software will be corrected as defects are uncovered, adapted as its environment changes, and enhanced as stakeholders request more capabilities. These maintenance activities can be facilitated if solid software engineering practice is applied throughout the software process.

3.2 Requirement Engineering

Designing and building computer software is challenging, creative, and just plain fun. Requirements engineering tasks are conducted to establish a solid foundation for design and construction. Thus, it builds a bridge to design and construction. It occurs during the communication and modeling activities that have been defined for the generic software process.

Requirements engineering involves set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants, and how end users will interact with the software. In more detail manner, requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. The steps for requirement engineering can be illustrated as follows:

1. **Inception:** A task that defines the scope and nature of the problem to be solved.
2. **Elicitation:** A task that helps stakeholders define what is required. Problems of scope, problems of understanding, and problems of volatility are three problems that are encountered as elicitation occurs.
3. **Elaboration:** The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.
4. **Negotiation:** what are the priorities, what is essential, when is it required?
5. **Specification:** The term *specification* means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

3.3 Requirements elicitation

At project inception, stakeholders establish basic problem requirements, define overriding project constraints, and address major features and functions that must be present for the system to meet its objectives. This information is refined and expanded during elicitation—a requirements gathering activity that makes use of facilitated meetings, quality function deployment (QFD), and the development of usage scenarios. There are some problems encountered as elicitation occurs.

- **Problems of scope:** The problems due to ill-defined system boundary, unnecessary technical detail specified by customer etc.
- **Problems of understanding:** The problems due to customers are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, etc.
- **Problems of volatility:** The requirements change over time.

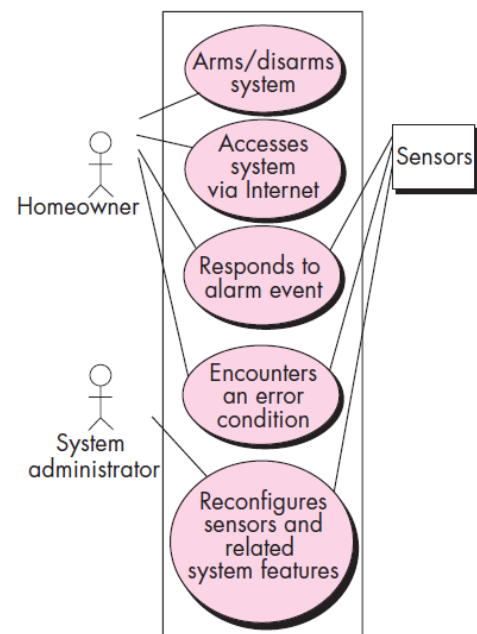
To help overcome these problems, **collaborating requirements gathering** technique is useful, which have following basic guidelines:

- Meetings are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A “facilitator” (can be a customer, a developer, or an outsider) controls the meeting.
- A “definition mechanism” (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD concentrates on maximizing customer satisfaction from the software engineering process.

Sometimes, the developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called **use cases**, provide a description of how the system will be used.

The general **work products** produced as a consequence of requirements elicitation will be: a statement of need and feasibility, a list of customers, users, and other stakeholders who participated, description of the system's technical environment, any prototypes developed to better define requirements, etc.



UML use case diagram for *SafeHome*

3.4 Developing Requirement Model

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as we learn more about the system to be built, and other stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time.

There are many different ways to look at the requirements for a computer-based system. The most common elements of the requirements model are describe below:

- **Scenario-based elements:** The system is described from the user's point of view using a scenario-based approach. For example, basic use cases and their corresponding use-case diagrams, and UML activity diagram.
- **Class-based elements:** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram

- **Behavioral elements:** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior. The *state diagram* is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A *state* is any externally observable mode of behavior. In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event.
- **Flow-oriented elements:** Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. For example, data flow diagram (DFD) and control flow diagram (CFD)

3.5 Requirement Validation

Each requirement and the requirements model as a whole are validated against customer need to ensure that the right system is to be built. *Requirements validation* examines the specification to ensure that all system requirements have been stated unambiguously, inconsistencies are removed, errors have been detected and corrected and the work products are ready for the process, the project, and the product. The primary requirements validation mechanism is the *formal technical review*.

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. A review of the requirements model addresses the following questions:

- Is each requirement consistent with the overall objectives for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

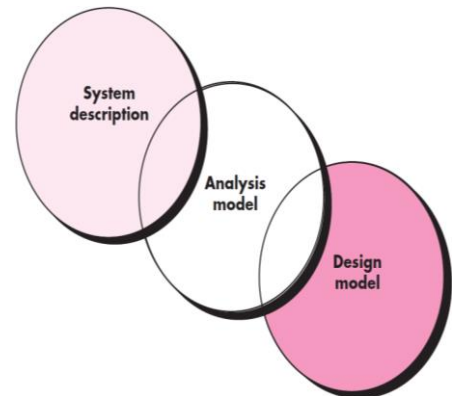
3.6 Requirement Analysis

Requirements analysis results in the specification of software’s operational characteristics, indicates software’s interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows us to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering. The requirements modeling action results in one or more of the following types of models:

- *Scenario-based models* of requirements from the point of view of various system “actors”
- *Data models* that depict the information domain for the problem

- *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as it moves through the system
- *Behavioral models* that depict how the software behaves as a consequence of external “events”

The requirements model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software’s application architecture, user interface, and component-level structure. This relationship is illustrated in figure.



Domain Analysis

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. Similarly, object-oriented domain analysis is the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

Domain analysis doesn’t look at a specific application, but rather at the domain in which the application resides. The intent is to identify common problem solving elements that are applicable to all applications within the domain. Figure illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

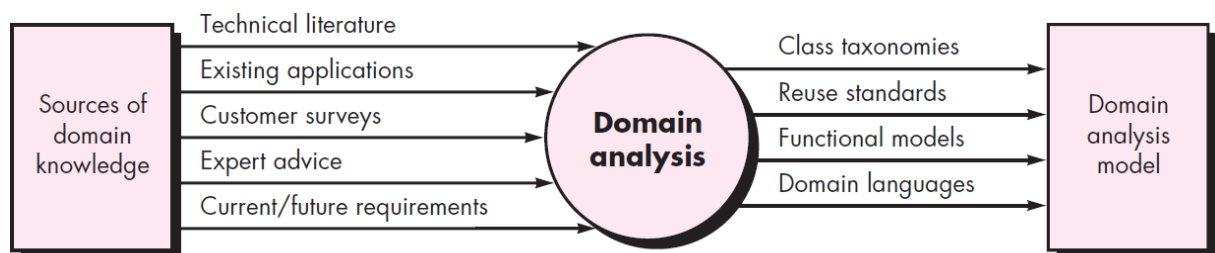


Fig. Input and output for domain analysis

Requirements Modeling Approaches

Software engineering basically have two approaches of requirements modeling, called *structured analysis*, and *object-oriented analysis*. *Structured analysis* considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system. *Object-oriented analysis* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

Each element of the requirements model presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the

manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

The negotiation among customer, users, and stakeholder is also important because if they cannot agree on defined requirements then the risk of failure is very high. Thus, the best negotiation strives for a “Win-Win” result.

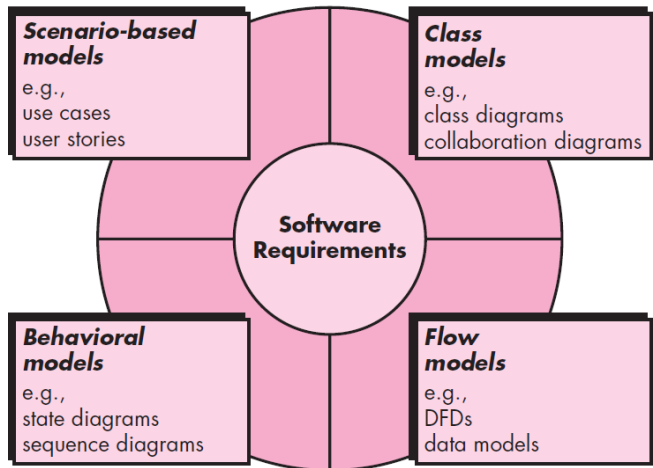


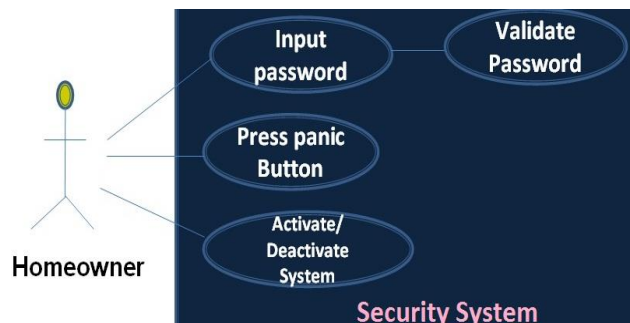
Figure: Elements of the analysis model

3.7 Scenario based modeling

Although the success of a computer-based system or product is measured in many ways, user satisfaction resides at the top of the list. If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design models. Hence, requirements modeling with UML begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

Use cases Diagram

Writing use cases is an excellent technique to understand and describe requirements. Informally, they are stories of using a system to meet goals. The essence is discovering and recording functional requirements by writing stories of using a system to help fulfill various stakeholder goals. Let's take an example of “*Home security system*” and we can analyze various use-case scenarios on the basis of description of depth.



Different formats of use cases for home security system

1. Brief use case

It describes *one-paragraph summary*, usually of the main success scenario. The homeowner arrives and presses panic button and when the system asks for entering the required password, the homeowner enters. After the validation of password, the authenticated user can activate or deactivate the system.

2. Casual use case

It is also called informal paragraph format where multiple paragraphs may covers the various scenarios. For the given example:

- **Main Success Scenario:** The homeowner can does appropriate modification on the system after the password validation.
- **Alternate Scenarios:**
 - When the entered password is not matched, the homeowner must re-enter the password.

- He can ask other home member for the proper password if there is modification on password.

3. Fully dressed use case

It is the most elaborate view of use case where all steps and variations are written in detail. There are also some supporting sections, such as preconditions and success guarantees.

A. Primary Actor: Homeowner

B. Stakeholders and Interests:

- **Homeowner:** Wants accurate, fast entry, and no password errors, failure free system response.
- **Home member:** Wants to set new password, know about the system controlling mechanism.
- **System:** Can damage with variation of electric flow, hang due to unnecessary activities, Cannot read the input signal.
- **Society:** Wants to maintain the harmony on society.
- **Introducers:** wants to break the authority.

C. Pre-conditions: Preconditions state what must always be true before beginning a scenario in the use case. A precondition implies a scenario of another use case that has successfully completed, such as logging in, or “cashier is identified and authenticated”. For example: *The system must on; homeowner must know the system password.*

D. Success Guarantee (Post conditions): It states what must be true on successful completion of the use case. Either the main success scenario or some alternate path. The guarantee should meet the needs of all stakeholders. It mainly includes 1) the break or formation of relationship, 2) attribute modification & 3) object installation (i.e. creation of new object to do the assigned work). For sale system, the post-conditions will be: Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Similarly, for the safe home system, the post-conditions may be:

- The homeowner must enter the home or leaves the home by de-activating or activating the system.
- The system must active or de-active the operation, re-set new password.

E. Main Success Scenario

1. Homeowner arrives to the security system.
2. Enter the valid password or scan finger print.
3. System matches the password or identifies the homeowner.
4. The system gives the permission to the authenticated user to do the changes on system.
5. User does the operation that he/she want.
6. User exit from system after required modification on system.
7. System saves the changes does by user.

F. Extensions (or Alternate Flows): They indicate all the other scenarios or branches, both success and failure. They are also known as "Alternative flows." For the given example, the alternate flow may be:

- User may forget the system password.
- System unable to recognize the entered password.
- System unable to save the modification does by the user.

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

- Touch screen UI on a large flat panel monitor.
- Text must be visible from 1 meter.
- Password authorization response within 30 seconds 90% of the time.
- Language internationalization on the text displayed.

H. Technology and Data Variations List:

- Password entered by touch panel or keyboard.
- Touch panel may be any UPC, EAN, JAN, or SKU scheme.
- Temperature recoding from the LM35 temperature sensor.

I. Frequency of Occurrence: Could be nearly continuous.

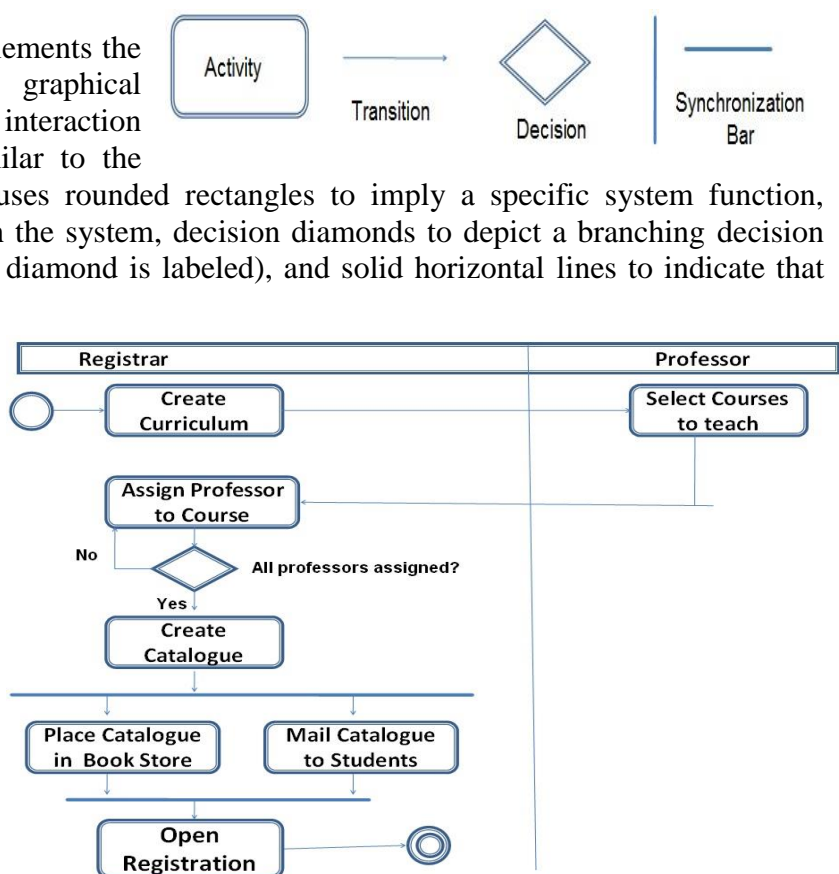
- What are the temperature variations?
- Explore the remote control issue.
- What customization is needed for other security system?
- Can the homeowner directly use the fingerprint or any other scanner?

Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the

flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring.

Use activity diagrams to specify, construct, and document the dynamics of a society of objects, or to model the flow of control of an operation. Whereas interaction diagrams emphasize the flow of control from object to object, activity diagrams emphasize the flow of control from activity to activity. *An activity is an ongoing non-atomic execution within a state machine.*



Note: Swimlane Diagrams

The UML *swimlane diagram* is a useful variation of the activity diagram and allows us to represent the flow of activities described by the use case and at the same time indicate which actor (if there are multiple actors involved in a specific use case) or analysis class (discussed later in this chapter) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

3.8 Data modeling

Data modeling defines the primary data objects to be processed by the system, the composition of each data object and their attributes, current location of objects, the relationships between each object and other objects, and the processes to transform each object. Data modeling methods make use of the entity relationship diagram that defines all data that are entered, stored, transformed, and produced within an application.

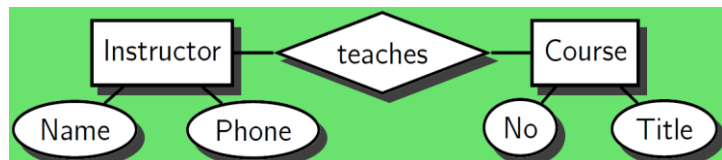
Entity Relationship Diagram (ERD)

The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships (*data object type hierarchies* and *associative data object*). ERD uses graphical notation, three basic elements in ER models are:

- Entities are the "things" about which we seek information (**Rectangle**).
- Attributes are the data we collect about the entities (**Oval**).
- Relationships provide the structure needed to draw information from multiple entities (Line and Diamond).

For example: ER-diagram about instructors and courses.

- Instructors teach courses.
- Instructors have a name and a phone number.
- Courses are numbered and have titles.



Cardinality: Specification of the number of occurrences of one object that can be related to the number of occurrences of another object.

Possible relationships are:

- One-to-One
- One-to-Many
- Many-to-Many

Modality: Specifies whether the relationship is optional or mandatory.

- Modality is 0 if relationship is optional (represented by dotted line in ERD)
- Modality is 1 if relationship is mandatory (represented by straight line in ERD)

"Every member has a spouse"

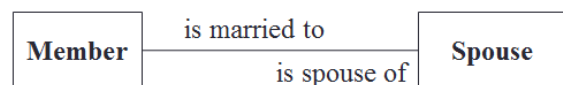


Fig. One-to-One Relationship

"Every member has one or more spouses"

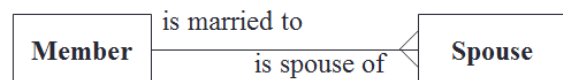


Fig. One-to-Many Relationship

"Every parent has one or more children, and every child has one or more parent"

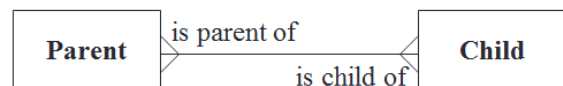


Fig. Many-to-Many Relationship

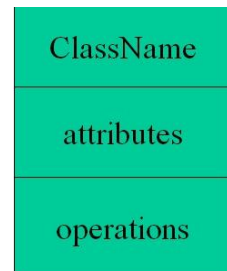
3.9 Class based modeling

Class-based modeling represents the objects that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects, attributes, operations, class responsibility collaborator (CRC) models, collaboration diagrams, and packages.

1. Class diagram

A *class diagram* is a *static model* that shows the classes and the relationships among classes that remain constant in the system over time. The class diagram depicts classes, which include both behaviors and states, with the relationships between the classes. The main building block of a class diagram is the *class*, which stores and manages information in the system. During analysis, classes refer to the people, places, events, and things about which the system will capture information. Later, during design and implementation, classes can refer to implementation-specific artifacts such as windows, forms, and other objects used to build the system.

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle, with the class's name at top, attributes in the middle, and operations at the bottom. The class is only a conceptual model (or blue print) of objects that does not appear on run time as the objects. Thus, an object is a run time entity that has physical appearance at run time but the class has not. 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.



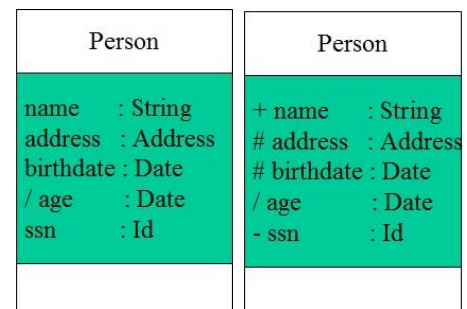
A. Class Names

The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

B. Class Attributes

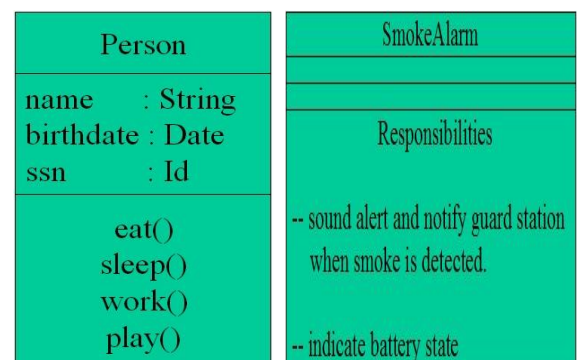
An *attribute* is a named property of a class that describes the object being modeled. In the class diagram, attributes appear in the second compartment just below the name-compartment. A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For e.g., a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in: */ age: Date*

Attributes can be: *+ public, # protected, - Private & / derived*



C. Class Operations

Operations describe the class behavior and appear in the third compartment. You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type. A class may also include its responsibilities in a class diagram. A responsibility is a contract or obligation of a class to perform a particular service.



D. Relationships

In UML, object interconnections (logical or physical), are modeled as relationships. There are three kinds of relationships in UML: dependencies, generalizations, and associations.

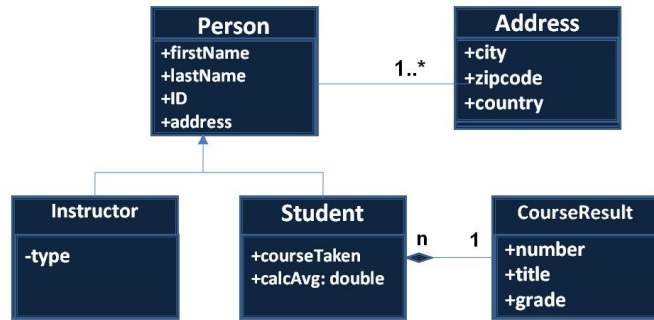
- **Association** (Loosely bounded) – Two classes are associated if one class has to know about the other.

- **Aggregation** (Belong to) – An association is one in which one class belongs to a collection in the other.

- **Generalization** (Inheritance) – An inheritance link indicating one class is a base class of the other.

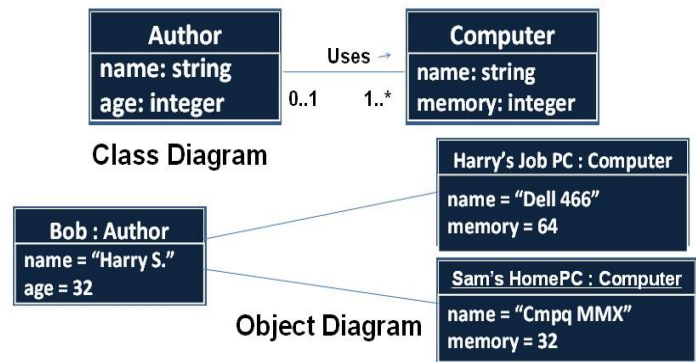
- **Dependency** – A labeled dependency between classes (such as friend, classes)

- **Composition** – Has a relation

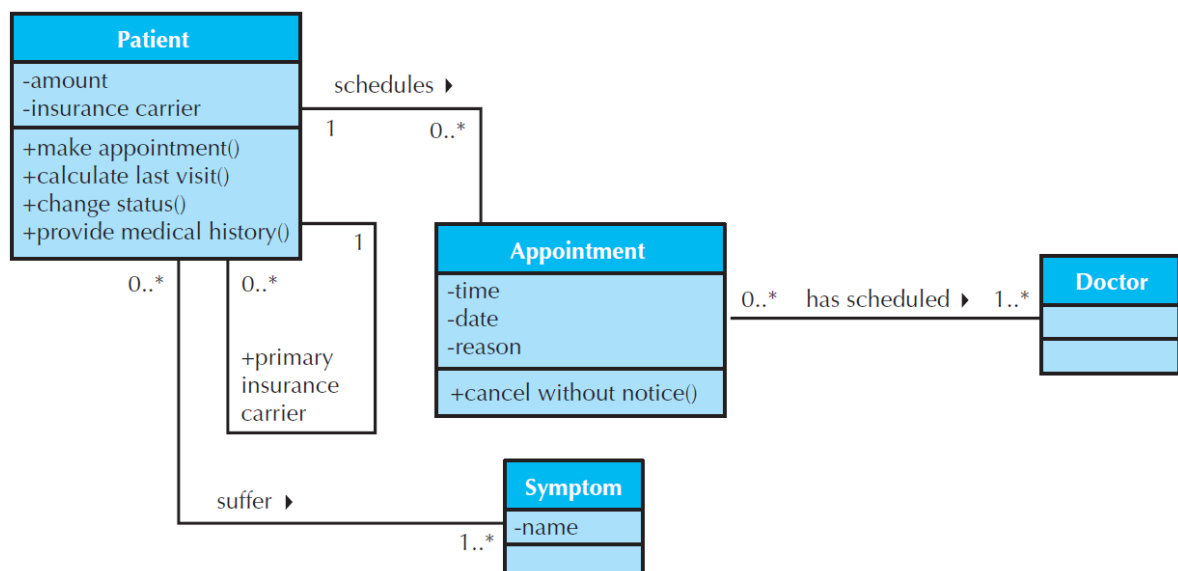


2. Object Diagram

An *object diagram* shows a set of objects and their relationships. It is used to illustrate data structures, the static snapshots of instances of the things found in class diagrams. Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the perspective of real or prototypical cases.



An object diagram is essentially an instantiation of all or part of a class diagram. Instantiation means to create an instance of the class with a set of appropriate attribute values. Object diagrams can be very useful when trying to uncover details of a class. Generally speaking, it is easier to think in terms of concrete objects (instances) rather than abstractions of objects (classes).

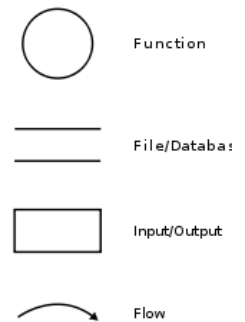


3.10 Flow oriented modeling

Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies hardware, software, and human elements to transform it; and produces output in a variety of forms. Regardless of size and complexity, the *flow model* for any computer-based system can be created. Data flow diagram (DFD) and Control flow diagram (CFD) are two common flow models. DFD or CFD have various graphical notations. A rectangle is used to represent an *external entity* such as hardware, a person, another program or another system that produces or receives information. A circle (or *bubble*) represents a *process* or *transform* that is applied to data (or control) and changes it in some way. An arrow represents one or more *data items* (data objects). The double line represents a data store.

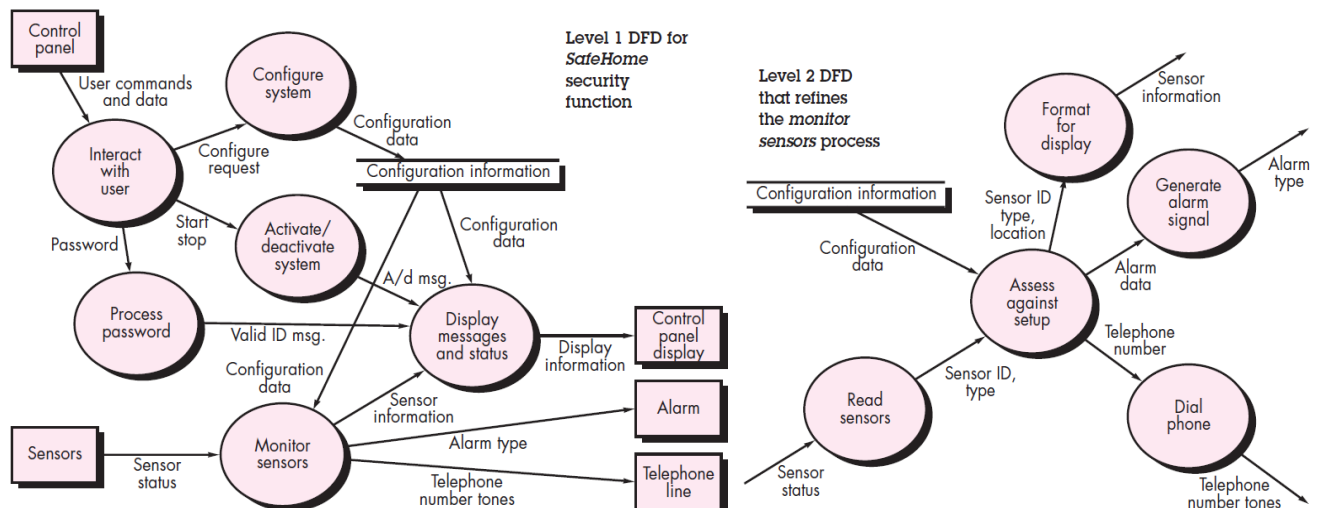
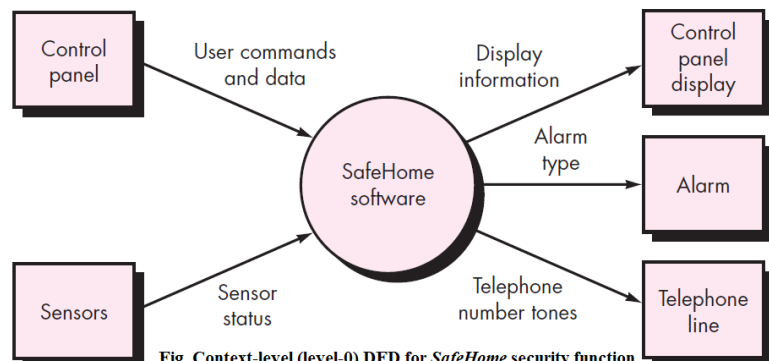
Data Flow Diagrams (DFD)

The DFD takes an input-process-output view of a system. A *data flow diagram* is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram, also known as a *data flow graph* or a *bubble chart*. The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail.



A level 0 DFD, also called a *fundamental system model* or a *context model*, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. The system of level-0 DFD is decomposed into lower-level DFD(Level-1) with the set of processes, data stores, and the data flows between these processes and data stores. Each process of level-1 is further decomposed into an even lower level diagram containing its sub-processes. This approach continues on subsequent sub-processes until the necessary and sufficient level of detail is reached which is called **primitive process**.

For example: DFD for SafeHome



The Control Specification (CSPEC) and Process Specification (PSPEC)

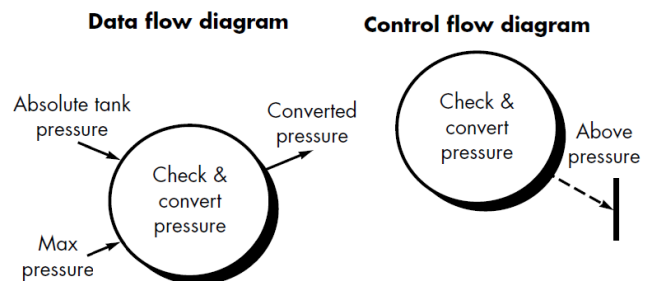
A *control specification* represents the behavior of the system (at the level from which it has been referenced) in two different ways. The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior.

The *process specification* is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm, mathematical equations, tables, or UML activity diagrams. The PSPEC is a “mini-specification” for each transform at the lowest refined level of a DFD.

Control Flow Diagrams (CFD)

The dashed arrow is once again used to represent control or event flow. Therefore, a *control flow diagram* contains the same processes as the DFD, but shows control flow, rather than data flow. The CFD shows how events move through a system. The *control specification* (CSPEC) indicates how software behaves as a consequence of events and what processes come into play to manage events.

Data flow diagrams are used to represent data and the processes that manipulate it. Control flow diagrams show how events flow among processes and illustrate those external events that cause various processes to be activated. The interrelationship between the process and control models is shown schematically in figure.



3.11 Behavioral modeling

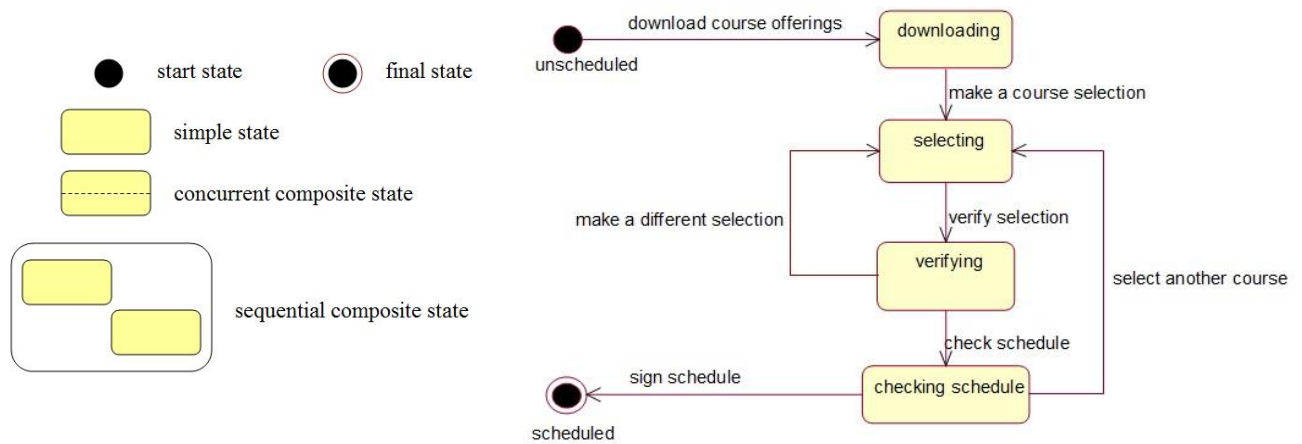
The *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within the system.
2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
3. Create a sequence for each use case.
4. Build a state diagram for the system.
5. Review the behavioral model to verify accuracy and consistency.

The UML's behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system where the dynamic aspects of a system represent its changing parts. The behavioral diagram used to show how the system evolves over time (responds to requests, events etc.). For a house, the airflow through room is dynamic aspect & similarly the flow of messages over time due to the physical movement of components across a network is dynamic aspect for the software system. The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

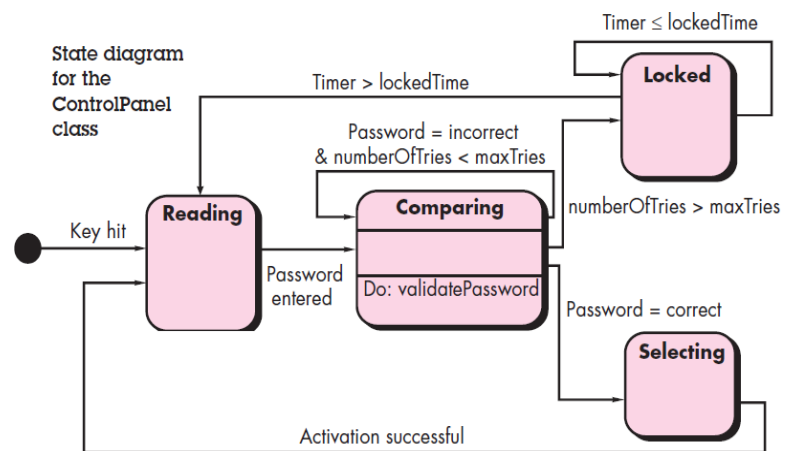
A. State chart Diagram

A *state-chart diagram* shows a state machine, consisting of states, transitions, events, and activities. They are especially important in modeling the behavior of an interface, class, or collaboration. State-chart diagrams emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.



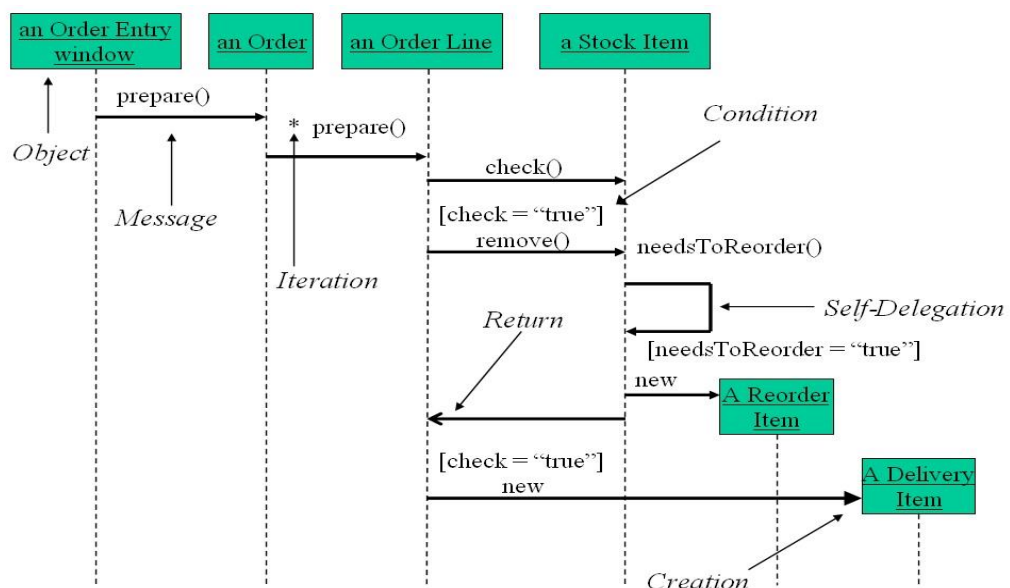
The **state transition diagram** (STD) represents the behavior of a system by depicting its states and the events that cause the system to change state. In addition, the STD indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

A state is any observable mode of behavior. For example, states for a monitoring and control system for pressure vessels might be *monitoring state*, *alarm state*, *pressure release state*, and so on. Each of these states represents a mode of behavior of the system. A state transition diagram indicates how the system moves from state to state.



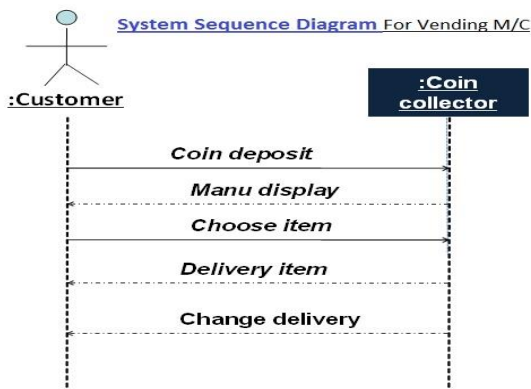
B. Sequence Diagram

A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. It shows a set of objects and the messages sent and received by those objects. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.



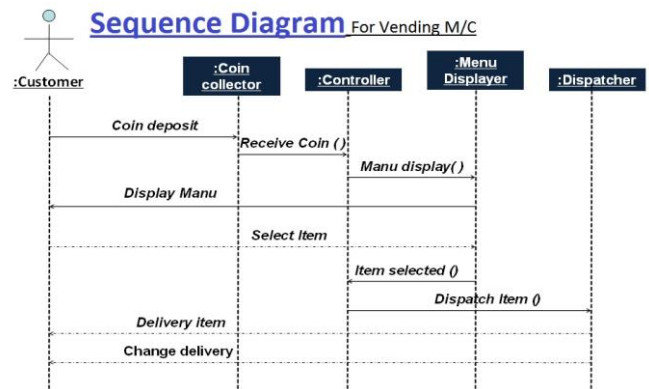
System sequence Diagram

1. It is an artifact of OOA that shows the collection of responsibility towards entire system from user. ("WHAT" prospective)
2. It shows the services provided by the entire system without participation of collaborating objects.
3. It simply identifies the possible interacting messages between a user and overall system body.
4. E.g. of Vending Machine



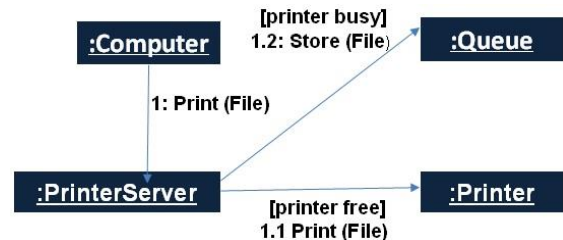
Sequence Diagram

1. It is an artifact of OOD obtained by elaborating the system sequence diagram of OOA which assign the responsibility on the many collaborating objects. ("HOW" prospective)
2. It shows the services provided by many collaborating objects that shape overall system operation. These collaborating objects are from the decomposition of entire system of system sequence diagram.
3. We assign the responsibility on collaborating objects using patterns or GRAPS and also set the interacting messages among them in general meaningful & language independent way.
4. E.g. Vending Machine

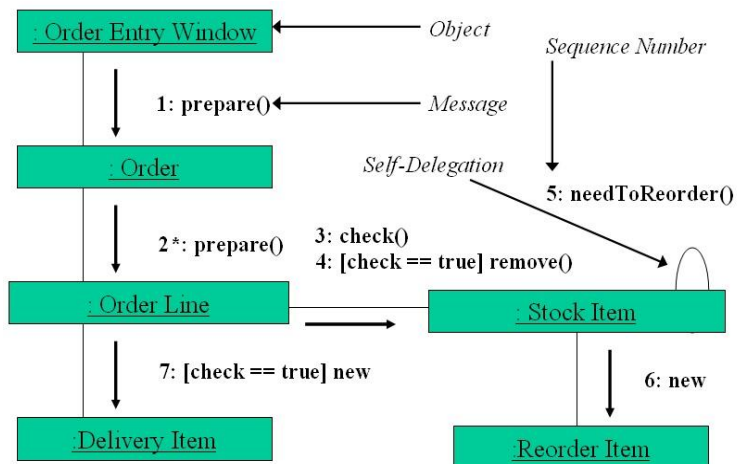


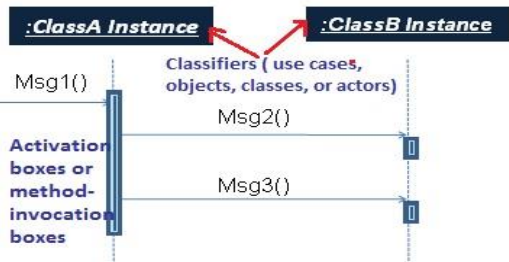
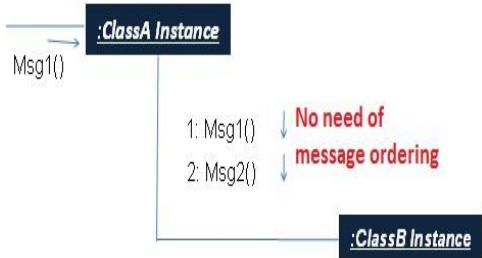
C. Collaboration or Communication Diagram

A collaboration diagram emphasizes the relationship of the objects that participate in an interaction. Unlike a sequence diagram, you don't have to show the lifeline of an object explicitly in a collaboration diagram. The sequences of events are indicated by sequence numbers preceding messages.



A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects. Sequence and collaboration diagrams are isomorphic, meaning that you can convert from one to the other without loss of information using any tools such as Rational rose.



Sequential diagram	Collaboration diagram
<ol style="list-style-type: none"> 1. It focuses on the dynamic aspect of an object. 2. It defines more clearly the chronological order of messages with respect to time on object from other object. 3. It only shows the message of one instant of time between the objects. 4. It has no provision to show all possible the relationship on an object. 5. For example: 	<ol style="list-style-type: none"> 1. It focuses on the structural aspect of one object to other. 2. It can define the total possible messages passed from one object to another but the order of message is not important. 3. It can show the total responsibility of an object. 4. It can show one or many possible relation of on objects. 5. For example: 

3.12 Design process

Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Design is the place where software quality is established. The design process moves from a “big picture” view of software to a more narrow view that defines the detail required to implement a system. The process begins by focusing on architecture. Subsystems are defined; communication mechanisms among subsystems are established; components are identified, and a detailed description of each component is developed. In addition, external, internal, and user interfaces are designed.

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. 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 requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- 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.

In order to evaluate the quality of a design representation, all the members of the software team must establish technical criteria for good design. Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability.

Generic task set for Design

1. Examine the information domain model, and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture: Be certain that each subsystem is functionally cohesive. Design subsystem interfaces. Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components: Translate analysis class description into a design class. Check each design class against design criteria; consider inheritance issues. Define methods and messages associated with each design class. Evaluate and select design patterns for a design class or a subsystem. Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface: Review results of task analysis. Specify action sequence based on user scenarios. Create behavioral model of the interface. Define interface objects, control mechanisms. Review the interface design and revise as required.
7. Conduct component-level design. Specify all algorithms at a relatively low level of abstraction. Refine the interface of each component. Define component-level data structures. Review each component and correct all errors uncovered.
8. Develop a deployment model.

3.13 Design concepts

Design concepts have evolved over the first 60 years of software engineering work. They describe attributes of computer software that should be present regardless of the software engineering process that is chosen, the design methods that are applied, or the programming languages that are used. A brief overview of important software design concepts can be describe below that span both traditional and object-oriented software development.

Abstraction

Software engineering possess many levels of abstraction. 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. In any levels of abstraction we work with two abstraction: *procedural* and *data* abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. A *data abstraction* is a named collection of data that describes a data object. For example, in the context of the procedural abstraction *open*, we can define a data abstraction called *door*.

Architecture

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. Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.
- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

- *Structural 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 of 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.

Patterns

In object-oriented idioms, the pattern is a problem solving technique by assigning some or specific responsibilities on an object. For example:

<i>Pattern Name</i>	Information Expert
<i>Solution</i>	Assign a responsibility to the class that has the information needed to fulfill it
<i>Problem It Solves</i>	What is a basic principle by which to assign

The intent of each design pattern is to provide a description that enables a designer to determine

- Whether the pattern is applicable to the current work,
- Whether the pattern can be reused (hence, saving design time), and
- Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve. Separation of concerns is expressed in other related design concepts: modularity, aspects, functional independence, and refinement.

Modularity

Modularity is the single attribute of software that allows a program to be intellectually manageable. Software is divided into separately named and addressable components, sometimes called *modules*, which are integrated to satisfy problem requirements. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

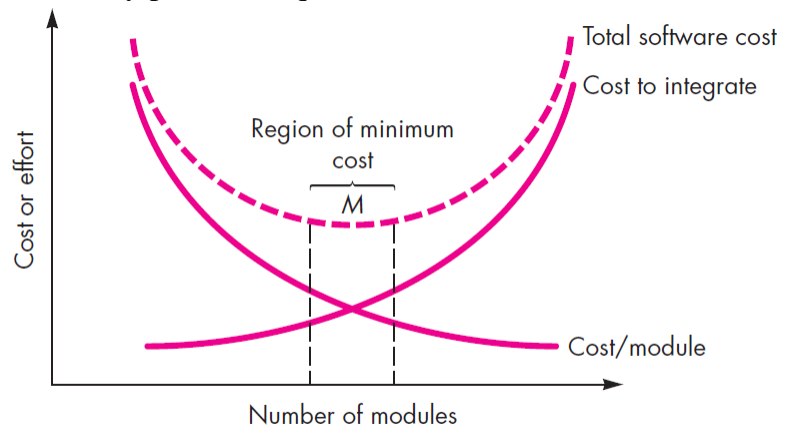


Fig. Modularity and software cost

Information Hiding

The concept of modularity leads you to a fundamental question: “How do I decompose a software solution to obtain the best set of modules?” The principle of information hiding suggests that modules be “characterized by design decisions that (each) hides from all others.” In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

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 procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence

Functional independence is a key to good design, and design is the key to software quality. The concept of functional independence is a direct outgrowth of separation of concerns, 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. Independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified. Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.

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.

- **High Cohesion:** High cohesion means assigning only one or less responsibility to an object so that it can be re-useable in many other systems also. Also, assigning some specific job to any system helps for efficiency of work increase.
- **Low Coupling:** Low coupling means less number of relations to other object (less dependency) that helps to isolate the object as re-usable component. Strongly coupled classes are undesirable; they suffer from the problems of: “hard to comprehend, reuse, maintain and change”.

Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached. Refinement is actually a process of *elaboration*. Abstraction and refinement are complementary concepts. Abstraction enables us to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details. Refinement helps us to reveal low-level details as design progresses. Both concepts allow us to create a complete design model as the design evolves.

Aspects

An *aspect* is a representation of a crosscutting concern. A crosscutting concern is some characteristic of the system that applies across many different requirements. Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* “if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account”.

An aspect is implemented as a separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout many components. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

Refactoring

An important design activity suggested for many agile methods, *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. 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.

3.14 Design Model

The design model can be viewed in two different dimensions as illustrated in figure. The *process dimension* indicates the evolution of the design model as design tasks are executed as 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 dashed line indicates the boundary between the analysis and design models.

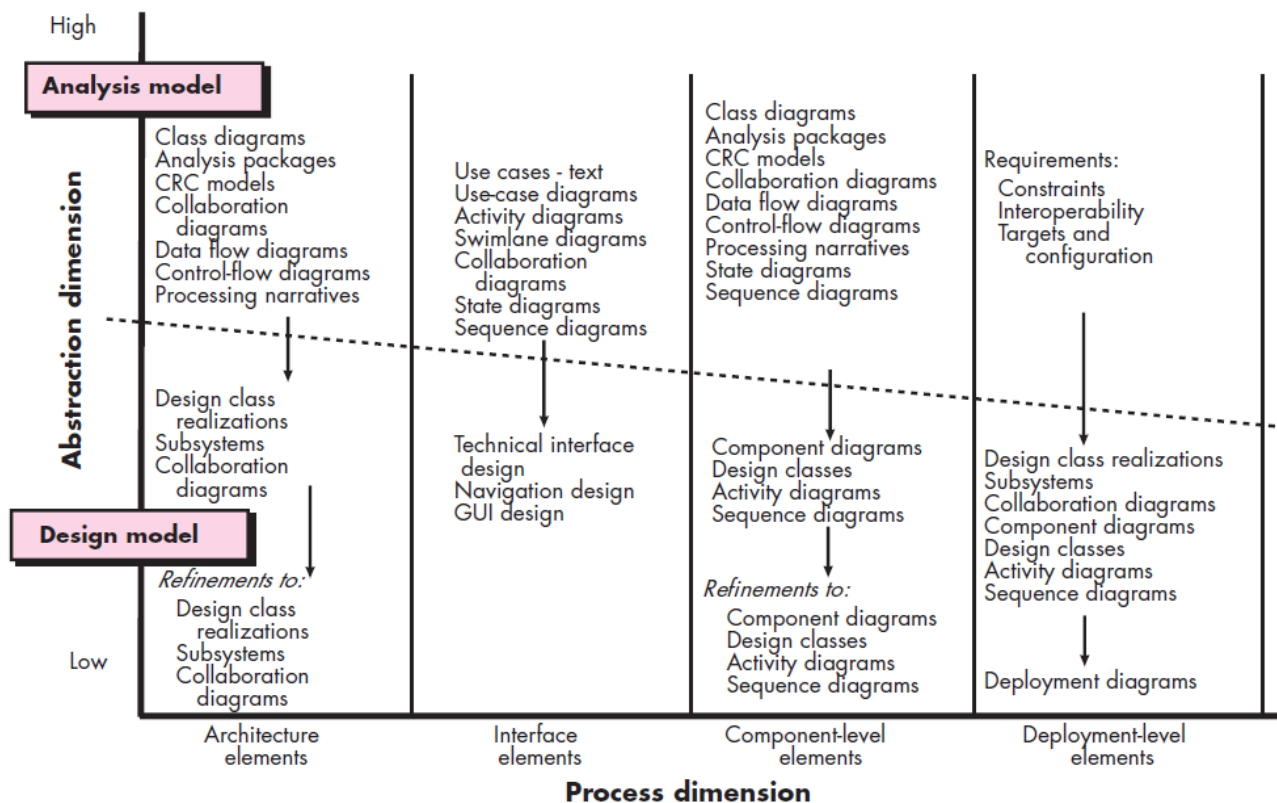


Fig. Dimensions of the design model

The elements of the design model use many of the same UML diagrams that were used in the analysis model. These diagrams are refined and elaborated as part of design.

Analysis Model		Design Model
classes	↔	objects
attributes	↔	data structures
methods	↔	algorithms
relationships	↔	messaging
behavior	↔	control

OOA	OOD
<ul style="list-style-type: none"> ▪ Analysis: Analysis emphasizes an investigation of the problem and requirements, rather than a solution. For example, if a new computerized library information system is desired, how will it be used? ▪ Object-oriented analysis: During object-oriented analysis, there is an emphasis on finding and describing the objects—or concepts—in the problem domain. For example, in the case of the library information system, some of the concepts include Book, Library, and Patron. <ul style="list-style-type: none"> - Investigation of the requirements, concepts, and operations related to a system. ▪ Essential use cases: They are created during early stage of eliciting the requirements and independent of the design decisions. For example: <ul style="list-style-type: none"> • Actor action: Cashier record identifier for each item. If there is more than one of the same, the cashier enters quantity. • System response: Determine the item price and add the item info to the running sales transaction description price of the current item in item presented. 	<ul style="list-style-type: none"> ▪ Design: Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Ultimately, designs can be implemented. ▪ Object-oriented design: During object-oriented design, there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a Book software object may have a title, attribute and a getChapter () method <ul style="list-style-type: none"> - Designing a solution for this iteration in terms of collaborating software objects. ▪ Real use cases: Make the essential use case more concrete by specifying the particular technology and methods. They are developed only after the design decisions have been made. For example: <ul style="list-style-type: none"> • Actor action: For each item cashier types UPC. In the UPC input field of window then they press “enter item” button with the mouse or keyboard. • System response: Display item price and add the item information to the running sales transaction the description and price of current item are displayed.

1. Data Design or Architecting

Data design creates a model of data and/or information that is represented at a high level of abstraction (the customer/user’s view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model 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. In every case, data design plays an important role.

2. Architectural Design

The *architectural design* for software is the equivalent to the floor plan of a house that give us the overall view of the building. Similarly, the software 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.

Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method. The architectural model is derived from three sources:

1. Information about the application domain for the software to be built;
2. Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and
3. The availability of architectural styles and patterns.

Architectural Style

As the various architecture of building construction, the software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

1. A set of components (e.g., a database, computational modules) that perform a function required by a system;
2. A set of connectors that enable “communication, coordination and cooperation” among components;
3. Constraints that define how components can be integrated to form the system; and
4. Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety; (2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency); (3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

A. Data-centered architectures: A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure illustrates a typical data-centered style.

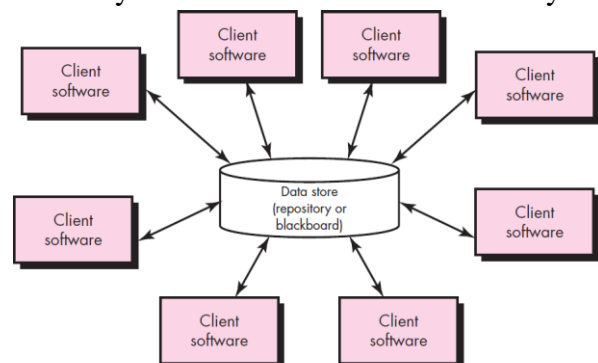
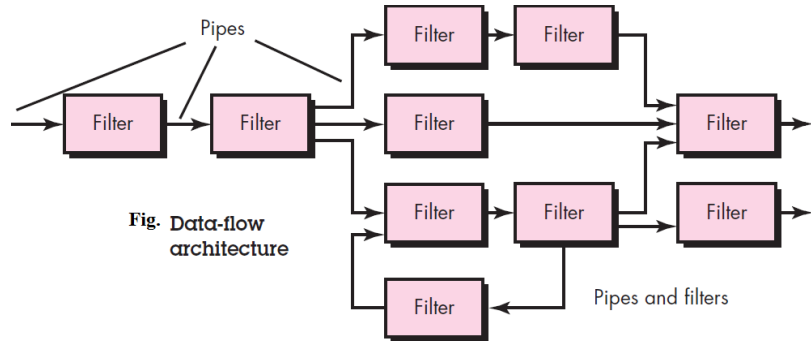


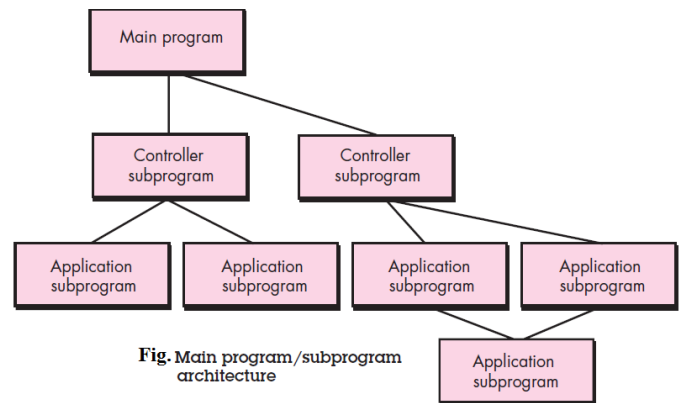
Fig. Data-centered architecture

B. 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 (to the next filter) of a specified form.



C. Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub-styles exist within this category:

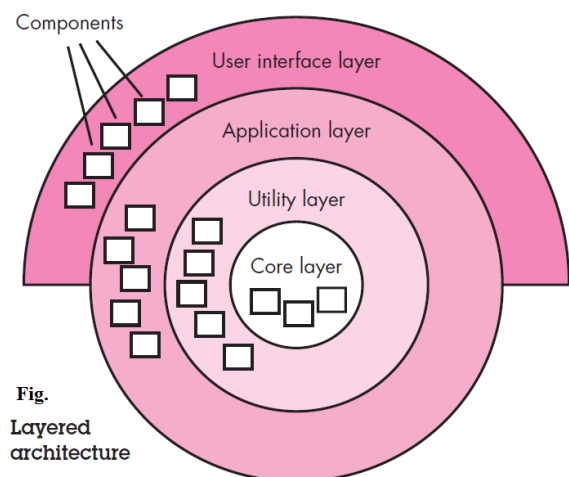
- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components. Figure illustrates an architecture of this type.



- *Remote procedure call architectures.* The components of a main program/subprogram architecture are distributed across multiple computers on a network.

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

E. Layered architectures: 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. Interface Design

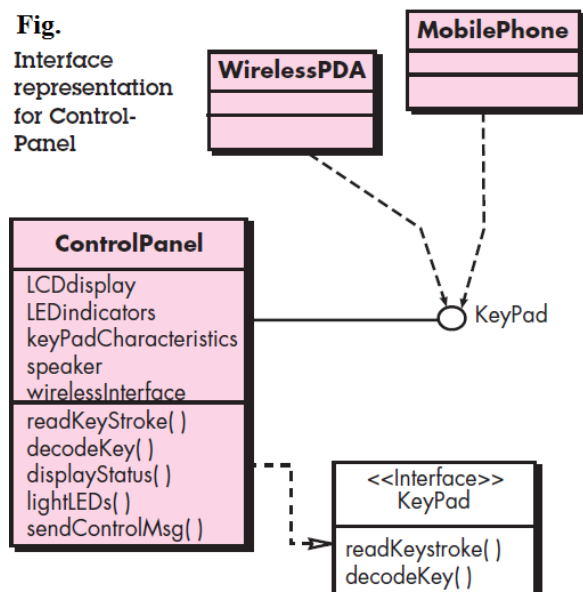
The interface design for software depicts information flows into and out of the system and how it is communicated among the components defined as part of the architecture. There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components. These interface designs allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

A. User Interface (UI) design

User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype. Three important principles guide the design of effective user interfaces: (1) place the user in control, (2) reduce the user's memory load, and (3) make the interface consistent. To achieve an interface that abides by these principles, an organized design process must be conducted.

If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user's perception of the software.

For example, the *SafeHome* security function makes use of a control panel that allows a homeowner to control certain aspects of the security function. In an advanced version of the system, control panel functions may be implemented via a wireless PDA or mobile phone. The interface representation for such a system can be shown in figure.



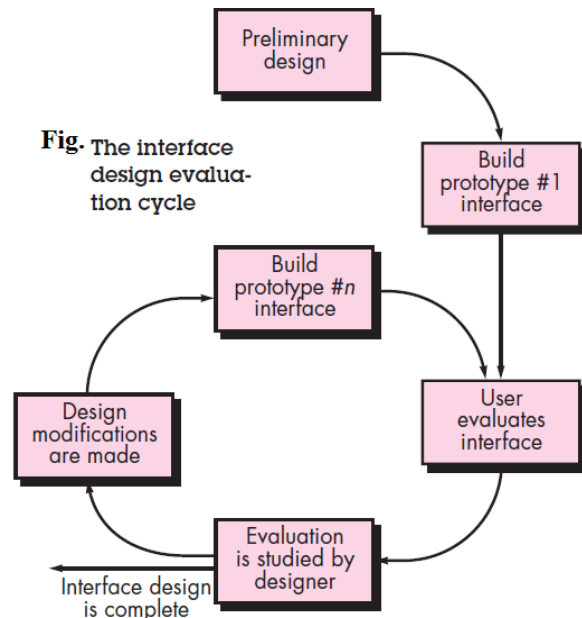
B. Interface design steps

User interface design begins with the identification of user, task, and environmental requirements. Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step. Although many different user interface design models have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis, define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

C. Interface design evaluation

Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal “test drive,” in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users. The user interface evaluation cycle takes the form shown in figure.



4. Component Level Design

A *component* is a modular building block for computer software that encapsulates implementation and exposes a set of interfaces. The component-level design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

The *component-level design* for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

Component Diagram

The component diagram is used to model the physical aspects of an OO system. It shows the software components of a system & how they are related to each other. Thus, they show the organization and dependencies between a set of components. Component diagrams are related to class diagrams in that a component *typically maps to one or more classes, interfaces, or collaborations*. Use component diagrams to model the *static implementation view* of a system. This involves modeling the physical things that reside on a node, such as executable, libraries, tables, files, and documents.



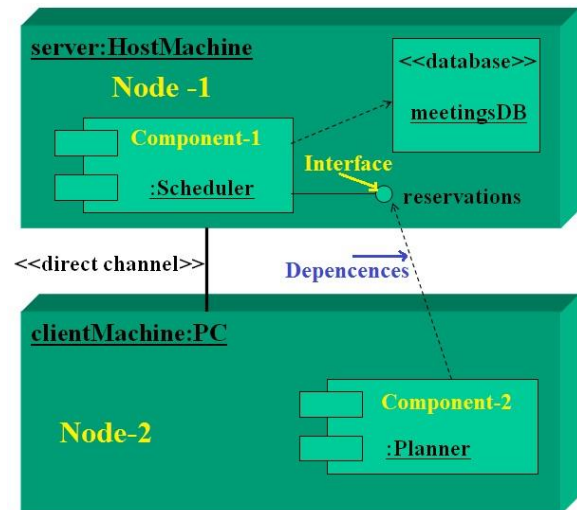
5. Deployment Level Design

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. Deployment diagrams begin in descriptor form, where the deployment environment is described in general terms. Later, instance form is used and elements of the configuration are explicitly described.

Deployment Diagram

The deployment diagram is also used to model the physical aspects of an OO system. A *deployment diagram* shows a set of *nodes* and their *relationships*. They show the configuration of *run-time processing* nodes and the components that live on them. Use deployment diagrams to model the *static deployment* view of a system or architecture.

This involves modeling the topology of the hardware on which the system executes. Deployment diagrams are *related to component diagrams* in that a node typically encloses one or more components.



3.15 Introduction to Design Patterns

Design patterns provide a codified mechanism for describing problems and their solution in a way that allows the software engineering community to capture design knowledge for reuse. A pattern describes a problem, indicates the context enabling the user to understand the environment in which the problem resides, and lists a system of forces that indicate how the problem can be interpreted within its context and how the solution can be applied.

Coplien characterizes an effective design pattern in the following way:

- *It solves a problem*: Patterns capture solutions, not just abstract principles or strategies.
- *It is a proven concept*: Patterns capture solutions with a track record, not theories or speculation.
- *The solution isn't obvious*: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
- *It describes a relationship*: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- *The pattern has a significant human component (minimize human intervention)*. All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.