


UNIT -III

REQUIREMENTS ENGINEERING

Understanding Requirements: Requirements Engineering, Establishing the Ground-work, Requirements Gathering, Developing UseCases, Building the Analysis Model, Negotiating Requirements, Requirements Monitoring, and Validating Requirements. **Requirements Modeling - A Recommended Approach:** Requirements Analysis, Scenario-Based Modeling, Class-Based Modeling, Functional Modeling, and Behavioral Modeling.

**QUICK LOOK**

What is it? Before you begin any technical work, it's a good idea to create a set of requirements for the engineering tasks. By establishing a set of requirements, you'll gain 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.

Who does it? Software engineers and other project stakeholders (managers, customers, and end users) all participate in requirements engineering.

Why is it important? To understand what the customer wants before you begin to design and build a computer-based system. Building an elegant computer program that solves the wrong problem helps no one.

What are the steps? Requirements engineering begins with Inception (a task that defines the scope and nature of the problem to be solved). It moves onward to elicitation (a task that helps stakeholders define what is required), and then elaboration (where basic requirements are refined and modified). As stakeholders define the problem, negotiation occurs (what are the priorities, what is essential, when is it required?). Finally, the problem is specified in some manner and then reviewed or validated to ensure that your understanding of the problem and the stakeholders' understanding of the problem coincide.

What is the work product? Requirements engineering provides all parties with a written understanding of the problem. The work products may include: usage scenarios, function and feature lists, and requirements models.

How do I ensure that I've done it right? Requirements engineering work products are reviewed with stakeholders to ensure that everyone is on the same page. A word of warning: Even after all parties agree, things will change, and they will continue to change throughout the project.

Definitions:

Requirements Definition:

- * It is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements.

- * The requirements engineering process includes the set of activities that lead to the production of the requirements definition and requirements specification.

- * This document must be written, so that it can be understood by the end-user and the system customer

Requirements Gathering:

According to Pressman,

- * The broad spectrum of tasks and techniques that lead to an understanding of requirements called **Requirements Gathering**, which builds a bridge between design and construction.

- * From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity.

* In addition, it includes reports on the feasibility of the system and software specification

3.1 Requirements Engineering

Requirements engineering refers to the broad range of tasks and techniques that lead to an understanding of requirements. It is a major software engineering activity that begins during the communication activity and continues into the modeling activity. It bridges the gap between design and construction.

Requirements engineering provide,

- Understanding what the customer wants.
- Analyzing requirements.
- Assessing feasibility.
- Negotiating a reasonable solution.
- Specifying the solution unambiguously.
- Validating the specification.

It consists of **seven** distinct tasks: *inception, elicitation, elaboration, negotiation, specification, validation, and management.*

a. Inception

In general, most projects begin with the identification of a business need or the discovery of a potential new market or service. Business stakeholders define a business case for the idea, attempt to identify the breadth and depth of the market, perform a rough feasibility analysis, and identify a working description of the project's scope.

At the start of a project, you establish a basic understanding of the problem, the people who want a solution, the nature of the desired solution, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

b. Elicitation

Make inquiries with the customer about the systems or product's objectives, what is to be implemented, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a daily basis. A number of issues arise during the elicitation process.

i. *Problems of scope*

The system's boundary is unclear, or customers/users provide unnecessary technical detail that may confuse, instead of clarifying overall system objectives.

ii. *Problems of understanding*

Customers/users are unsure of what is required, have a poor understanding of the capabilities and restrictions of their computing environment, do not have a complete understanding of the problem domain, have problems communicating needs to the system engineer, ignore information that is considered "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous.

iii. *Problems of volatility*

The specifications change over time. You must approach requirements gathering in an organized manner to help overcome these issues.

c. Elaboration

During elaboration, the information gathered from the customer during inception and elicitation is expanded and refined. The goal of this task is to create a refined requirements model that identifies various aspects of system function, behavior, and information.

The creation and refinement of user scenarios that describe how the end user and other actors will interact with the system drives elaboration. Each user scenario is parsed to extract analysis classes—visible to the end user business domain entities. Each analysis class's attributes are defined, and the services required by each class are identified. The relationships and collaboration among classes are identified, and a variety of supplementary diagrams are created.

d. Negotiation

It's fairly common for different customers or users to propose competing requirements, claiming that their version is "essential for our special needs."

You must resolve these conflicts through a negotiating process. Customers, users, and other stakeholders are asked to rank requirements and then discuss any conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some level of satisfaction.

e. Specification

To different people, specification means different things. A specification can be a written document, a collection of graphical models, a formal mathematical model, a set of usage scenarios, a prototype, or any combination of these. Some argue that a "standard template" for a specification should be developed and used, arguing that this results in requirements that are presented in a consistent and thus more understandable manner.

However, when developing a specification, it is sometimes necessary to remain flexible. A written document combining natural language descriptions and graphical models may be the best approach for large systems.

f. Validation

Validation of the software requirements is the quality of the requirements specification sufficient? *For instance*, are requirements consistent, unambiguous and complete? In other words, could they be an adequate basis for designing and implementing the system. It is the process of checking that requirements defined for development, define the system that the customer really wants. To check issues related to requirements, we perform requirements validation. We usually use requirements validation to check error at the initial phase of development as the error may increase excessive rework when detected later in the development process.

In the requirements validation process, we perform a different type of test to check the requirements mentioned in the *Software Requirements Specification (SRS)*, these checks include:

- Completeness checks
- Consistency checks
- Validity checks
- Realism checks

- Ambiguity checks
- Verifiability

After requirement specifications developed, the requirements discussed in this document are validated. The user might demand illegal, impossible solution or experts may misinterpret the needs. Requirements can be the check against the following conditions -

- If they can practically implement
- If they are correct and as per the functionality and specially of software
- If there are any ambiguities
- If they are full
- If they can describe

Requirements Validation Techniques

- **Requirements reviews / inspections:** systematic manual analysis of the requirements.
- **Prototyping:** Using an executable model of the system to check requirements.
- **Test-case generation:** Developing tests for requirements to check testability.
- **Automated consistency analysis:** checking for the consistency of structured requirements descriptions.

g. Management:

Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management (SCM) techniques.

3.2 Establishing the groundwork

In an ideal world, stakeholders and software engineers would collaborate on the same team. In such cases, **requirements engineering** is simply a matter of having meaningful conversations with team members who are well-known.

We go over the steps that must be taken to lay the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving toward successful completion.

3.2.1 Identifying the stakeholders

Any person who benefits directly or indirectly from the system being developed is a stakeholder. Business operations managers, product managers, marketing people, internal and external customers, end-users, consultants, product engineers, software engineers, and support/maintenance engineers are the usual stakeholders.

Each stakeholder sees the system differently, gains different benefits when the system is successfully developed and faces different risks if the development effort fails.

3.2.2 Recognizing Multiple Viewpoints

Because there are so many different stakeholders, the system's requirements will be examined from various perspectives. Each of these stakeholders will contribute data to the requirements engineering process. As information is gathered from multiple viewpoints, emerging requirements may be inconsistent or contradictory. You should categorize all stakeholder information so that decision-makers can select an internally consistent set of system requirements for the system.

3.2.3 Working toward Collaboration

If there are five stakeholders involved in a software project, there may be five different opinions on the set of requirements. Customers must work together as well as with software engineering practitioners to create a successful system. A requirements engineer's job is to identify areas of commonality as well as areas of conflict or inconsistency. Collaboration does not always imply that requirements are set by a committee. In many cases, stakeholders collaborate by providing their perspective on requirements, but a strong "project champion" may ultimately decide on which requirements are accepted.

3.2.4 Asking the first questions

The first set of questions asked is context-free and focuses on the customer and other stakeholders, as well as the overall project goals and benefits. You could ask

- Who is the person or organization behind the request for this work?
- Who will make use of the solution?
- What is the economic value of a successful solution?
- Is there a different source for the solution you require?

These questions aid in identifying all stakeholders who will be interested in the software being developed. Moreover, the questions identify the quantifiable benefit of successful implementation as well as potential alternatives to custom software development.

The following set of questions helps in gaining a better understanding of the problem and allows the customer to express his or her thoughts on a solution:

- How would you characterize "good" output produced by a successful solution?
- To what problem(s) will this solution be applied?
- Can you describe (or show me) the business environment in which the solution will be used?
- Will there be any special performance issues or constraints that will influence how the solution is approached?

The final set of questions is concerned with the effectiveness of the communication activity itself.

- Are you qualified to respond to these questions? Is your response "official"?
- Are my queries relevant to the issue you're dealing with?
- Am I interrogating you too much?
- Can anyone else add to this?
- Do you have any further questions?

These questions will assist in "breaking the ice" and initiating the communication required for successful elicitation.

3.3 Requirements Gathering

Requirements Elicitation includes problem solving, elaboration, negotiation, and specification.

3.3.1 Collaborative Requirements Gathering

There have been numerous techniques of collaborative requirements collecting proposed.

- Meetings are held and attended by software engineers as well as other stakeholders.

- Preparation and participation rules are created.
- An agenda is proposed that is formal enough to cover all critical topics while being informal enough to allow for the free flow of ideas.
- The discussion is managed by a “facilitator.”
- A “definition mechanism” (work sheets, flip charts, etc.) is used.

The purpose is to define the problem, offer solution elements, negotiate different approaches, and specify a preliminary set of solution requirements in an environment conducive to goal achievement. Each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions while reviewing the product request in the days leading up to the meeting.

The mini-specs are offered for discussion to all stakeholders. There are additions, deletions, and extra elaboration. In some circumstances, creating mini-specs will reveal new objects, services, constraints, or performance requirements that will be added to the original listings. The team may raise an issue that cannot be handled during the meeting at any time. An issue list is kept so that these suggestions can be followed up on later.

3.3.2 Quality Function Deployment (QFD)

Quality function deployment (QFD) is a quality management technique that translates customer expectations into technical software requirements. It is focused on maximizing customer satisfaction from the software engineering process.

The QFD defines type’s categories of requirements:

i. Normal requirements

The objectives and goals that are mentioned for a product or system during customer meetings are called normal requirements. The customer is satisfied if these requirements are met. Requested forms of graphical displays, specified system functionalities, and defined levels of performance are examples of normal requirements.

ii. Expected requirements

These requirements are inherent in the product or system and may be so basic that the client does not express them explicitly. Their absence will be a major source of dissatisfaction. Expected needs include ease of human/machine communication, general operating correctness and reliability, and software installation simplicity.

iii. Exciting requirements

These characteristics go above and beyond the customer’s expectations and are absolutely delightful when they are present.

For example, software for a new mobile phone has typical functionality as well as a set of unexpected capabilities like multi-touch screen, visual voice mail, etc. that thrill every product user.

3.3.3 Usage Scenarios

As requirements are acquired, an overall vision of system functions and features emerges. However, moving into more technical software engineering activities is difficult unless you understand how these functions and features will be used by different types of end-users. To do this, engineers and users can collaborate to establish a set of scenarios that identify a thread of usage for the system to be built.

3.3.4 Elicitation Work Products

The work products generated as a result of requirements elicitation will differ according to the size of the system or product to be constructed.

The work products for the majority of systems include

- A need and feasibility statement.
- A bounded statement of the systems or product's scope.
- A list of clients, users, and other stakeholders who took part in the requirements elicitation process.
- A description of the technological environment of the system.
- A list of objectives, as well as the domain restrictions that apply to each of them.
- A set of usage scenarios that provide information about how the system or product is used under various operational conditions.
- Any prototypes created to help describe requirements more precisely.

Each of these work products is assessed by everyone who took part in the requirements elicitation process.

3.4 Developing Usecases

A use case, in essence, presents a stylized Storey about how an end user interacts with the system under certain conditions. The narrative text, an outline of activities or interactions, a template-based description, or a diagrammatic representation can all be used to tell the story. A use case, in whatever shape it takes, portrays the program or system from the perspective of the end user.

The first phase in developing a use case is to identify the “actors” who will be participating in the storey. Actors are the various people (or devices) who use the system or product in the context of the defined function and behavior. It is vital to remember that an actor and an end user are not always synonymous. Primary actors can be identified during the first iteration, and secondary actors can be identified when more about the system is learnt.

The primary actors interact in order for the system to perform the essential functions and provide the desired benefit. They interact frequently and directly with the program. Secondary actors help the system so that primary actors can fulfill their jobs. Use cases can be developed once the actors have been identified.

According to **Jacobson**, a use case should address the following questions.

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

3.5 Building the Requirements Model

The analysis model's goal is to provide a description of the informational, functional, and behavioral domains required for a computer-based system. The model evolves dynamically as you learn more about the system to be developed and other stakeholders gain a better understanding of what they actually require. As a result, the analysis model represents a snapshot of requirements at any given time.

3.5.1 Elements of the Requirements Model

There are numerous approaches to analyzing the requirements for a computer-based system. Different modes of representation require you to analyze requirements from many perspectives—a method that has a higher likelihood of uncovering omissions, inconsistencies, and ambiguity.

a. Scenario-based elements

A scenario-based approach is used to describe the system from the perspective of the user. For example, basic use cases and their related use-case diagrams evolve into more complicated template-based use cases. Scenario-based requirements model elements are frequently the first parts of the model to be produced. There are three levels of elaboration shown, ending in a scenario-based portrayal.

b. Class-based element

Each usage scenario entails a collection of objects that are modified as an actor interacts with the system. These objects are classified as classes—a collection of things with similar characteristics and behavior.

c. Behavioral elements

The behavior of a computer-based system can have a significant impact on the design and implementation techniques used. As a result, the requirements model must include modeling elements that represent behavior. The state diagram is one approach of expressing a system's behavior by illustrating its states and the events that cause the system to change state. Any externally observable form of conduct is referred to as a state. Furthermore, the state diagram indicates activities taken as a result of a specific **event**.

d. Flow-oriented elements

As data moves through a computer-based system, it is transformed. The system accepts input in a variety of formats, transforms it using functions, and produces output in a variety of forms. A control signal transmitted by a transducer, a series of numbers written by a human operator, a packet of information transmitted over a network link, or a large data file retrieved from secondary storage can all be used as input. The transform(s) may consist of a single logical comparison, a complex numerical algorithm, or an expert system's rule-inference approach.

3.5.2 Analysis Patterns

Anyone who has done requirements engineering on a number of software projects will note that some issues repeat across all projects within a certain application area. These patterns of analysis provide solutions (e.g., a class, a function, or a behavior) inside the application domain that can be reused when modeling several applications.

By referencing the pattern name, analysis patterns are integrated into the analysis model. They are also stored in a repository so that requirements engineers can find and

apply them using search facilities. Information about an analysis pattern (and other sorts of patterns) is contained in a standard template.

3.6 Negotiating and Validating Requirements

3.6.1 Negotiating requirements

The inception, elicitation, and elaboration tasks in an ideal requirement engineering setting determine customer requirements in sufficient depth to proceed to later software engineering activities. You might have to negotiate with one or more stakeholders. Most of the time, stakeholders are expected to balance functionality, performance, and other product or system attributes against cost and time-to-market.

The goal of this discussion is to create a project plan that meets the objectives of stakeholders while also reflecting the real-world restrictions (e.g., time, personnel, and budget) imposed on the software team. The successful negotiations aim for a “win-win” outcome. That is, stakeholders benefit from a system or product that meets the majority of their needs, while you benefit from working within realistic and reasonable budgets and schedules.

At the start of each software process iteration, **Boehm** defines a series of negotiating actions. Rather than defining a single customer communication activity, the following are defined:

1. Identifying the major stakeholders in the system or subsystem.
2. Establishing the stakeholders’ “win conditions.”
3. Negotiation of the win conditions of the stakeholders in order to reconcile them into a set of win-win conditions for all people involved.

3.6.2 Requirements Monitoring

Incremental development is commonplace. This means that use cases evolve, new test cases are developed for each new software increment, and continuous integration of source code occurs throughout a project. *Requirements monitoring* can be extremely useful when incremental development is used. It encompasses five tasks:

- (1) *distributed debugging* uncovers errors and determines their cause,
- (2) *run-time verification* determines whether software matches its specification,
- (3) *run-time validation* assesses whether the evolving software meets user goals,
- (4) *business activity monitoring* evaluates whether a system satisfies business goals, and
- (5) *evolution and codesign* provides information to stakeholders as the system evolves.

Incremental development implies the need for incremental validation. Requirements monitoring supports continuous validation by analyzing user goal models against the system in use. **For example**, a monitoring system might continuously assess user satisfaction and use feedback to guide incremental improvements

3.6.3 Validating Requirements

Each aspect of the requirements model is checked for consistency, omissions, and ambiguity as it is developed. The model’s requirements are prioritized by stakeholders and bundled into requirements packages that will be implemented as software increments.

The following questions are addressed by an examination of the requirements model:

- Is each requirement aligned with the overall system/product objectives?
- Were all requirements expressed at the appropriate level of abstraction? Do some criteria, in other words, give a level of technical information that is inappropriate at this stage?
 - Is the requirement truly necessary, or is it an optional feature that may or may not be critical to the system's goal?
 - Is each requirement well defined and unambiguous?
 - Is each requirement attributed? Is there a source noted for each requirement?
 - Are there any requirements that conflict with others?
 - Is each requirement attainable in the technical environment in which the system or product will be housed?
 - Is each requirement, once implemented, testable?
 - Do the requirements model accurately represent the information, functionality, and behavior of the system to be built?
 - Has the requirements model been "partitioned" in such a way that progressively more detailed information about the system is exposed?
 - Have requirements patterns been used to reduce the complexity of the requirements model?
 - Have all patterns been validated properly? Are all patterns in accordance with the requirements of the customers?

PART II

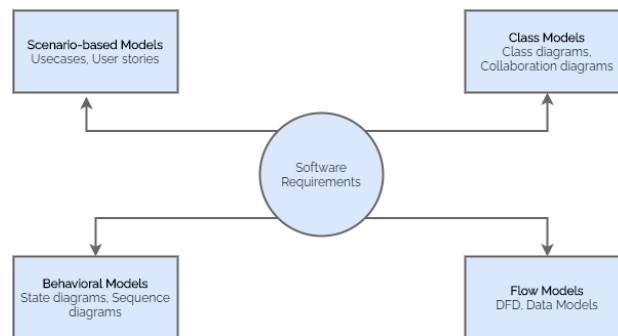
REQUIREMENTS MODELING

3.8 Requirement Analysis

Requirements analysis leads to the specification of software's operating characteristics, the identification of software's interface with other system elements, and the establishment of constraints that software must meet. Requirements analysis helps you elaborate on basic requirements generated through inception, elicitation, and negotiation requirements engineering tasks.

The action of requirements modeling produces one or more of the following types of models:

- **Scenario-based models** of requirements from the perspective of various system "actors" Scenario-based models of requirements from the perspective of various system "actors".
- **Data models** that depict the problem's information domain.
- **Class-oriented models** represent object-oriented classes (attributes and operations) and how classes collaborate to achieve system requirements.
- **Flow-oriented models** that represent the system's functional elements and how they transform data as it moves through the system.
- **Behavioral models that describe how software behaves in response to external "events"** that describe how software behaves in response to external "events"



These models provide information to a software designer that can be turned into architectural, interface, and component-level designs.

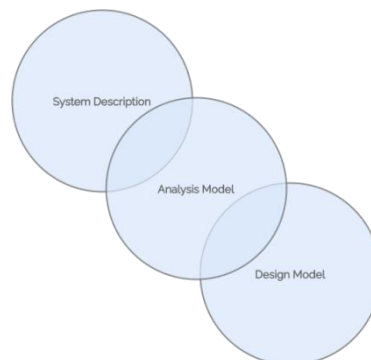
3.8.1 Overall Objectives and Philosophy

Throughout the requirements modeling process, your primary focus should be on what, not how.

The requirements model must accomplish three basic goals:

1. Describing what the client requires.
2. Establishing a foundation for the production of software design.
3. Defining a set of criteria that can be validated once the software is constructed.

The analysis model bridges the gap between a system-level description of the overall system or business functioning as it is delivered through the use of software, hardware, data, human, and other system aspects and a software design. This relationship is shown in the figure below.



3.8.2 Analysis Rules of Thumb

Arlow and Neustadt suggest a few useful rules of thumb to follow when developing the analytical model.

- **The model should concentrate on requirements that are obvious within the problem or business domain.** The level of abstraction should be moderate. “Don’t get bogged down in details” that attempt to explain how the system will function.
- **Each element of the requirements model should contribute to a broader understanding of software needs and provide insight into the system’s information domain, function, and behavior.**
- **Put off thinking about infrastructure and other nonfunctional models until design.** That is, while a database may be required, the classes required to implement it,

the functions required to access it, and the behavior that will be displayed as it is used should be considered only after the problem domain analysis is complete.

- **Keep coupling to a minimum** across the system. Relationships between classes and functions must be represented. If the level of “interconnectedness” is excessively high, however, efforts should be made to reduce it.
- **Check to see if the requirements model gives value to all stakeholders.** Each constituency has a different application for the model. Business stakeholders, for example, should use the model to validate requirements; designers should use the model as a foundation for design; and QA personnel should use the model to help prepare acceptance tests.
- **Keep the model as simple as possible.** Don’t make any new diagrams if they don’t contribute any new information. When a basic list will enough, avoid using complex notational forms.

3.8.3 Requirement Modeling Approaches

Structured Analysis

Structured analysis is one approach to requirements modeling that treats data and the processes that transform it as separate entities. Data objects are modeled in such a way that their attributes and relationships are defined. Processes that change data items are designed in such a way that it is clear how they transform data as it flows through the system.

Object-Oriented Analysis

The object-oriented analysis focuses on the definition of classes and how they interact with one another. The Unified Process and UML are both mostly object-oriented.

3.8.4 Modeling Strategies

Your understanding of software requirements grows in direct proportion to the number of different dimensions of requirements modeling. Although you may not have the time, resources, or motivation to create every representation, remember that each modeling technique provides a unique perspective on the problem. As a result, you’ll be able to evaluate whether you’ve properly specified what has to be done.

i. Flow-Oriented Modeling (The Data Flow Diagram)

The DFD analyzes a system from the viewpoint of input-process-output. In other words, data objects enter the software, are modified by processing elements, and then exit the software. Data objects are represented as labeled arrows, whereas transformations are represented as circles (also called bubbles). The DFD is displayed in a hierarchical format. That is, the first data flow model (also known as a level 0 DFD or context diagram) reflects the entire system. Subsequent data flow diagrams refine the context diagram, adding more detail with each level.

You can use the data flow diagram to create models of the information domain and the functional domain. As the DFD is refined into higher levels of detail, an implicit functional decomposition of the system is performed.

For certain applications, the data model and data flow diagram are all that is required to gain meaningful insight into software requirements. A substantial class of applications, on the other hand, is “driven” by events rather than data generate control information rather than reports or displays, and processing information with a strong emphasis on speed and

performance. For such applications, **Control flow modeling** is required, in addition to data flow modeling.

ii. Behavioral Model

The system's behavior can be represented as a function of certain events and times. The behavioral model describes how the software will respond to outside events or stimuli. Go through the following steps, to make the model.

1. Evaluate all use cases to properly understand the system's interaction sequence.
2. Recognize the events that drive the interaction sequence and understand how these events are related to specific objects.
3. For each use case, create a sequence.
4. Create a system state diagram.
5. Go over the behavioral model again to ensure its accuracy and consistency.

3.9 Elements of the Requirements Model

Requirements for a computer-based system can be seen in many different ways. Some software people argue that it's worth using a number of different modes of representation while others believe that it's best to select one mode of representation.

The specific **elements of the requirements model** are dedicated to the analysis modeling method that is to be used.

- **Scenario - based elements:**
Using a scenario-based approach, the system is described from the user's point of view. **For example**, basic use cases and their corresponding use-case diagrams evolve into more elaborate template-based use cases. Figure 1(a) depicts a UML activity diagram for eliciting requirements and representing them using use cases. There are three levels of elaboration.
- **Class-based elements:**
A collection of things that have similar attributes and common behaviors i.e., objects are categorized into classes. **For example**, a UML case diagram can be used to depict a Sensor class for the SafeHome security function. Note that the diagram lists attributes of sensors and operations that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict manner in which classes collaborate with one another and relationships and interactions between classes.
- **Behavioral elements:**
The effect of behavior of computer-based system can be seen on design that is chosen and implementation approach that is applied. Modeling elements that depict behavior must be provided by requirements model.

Class diagram for sensor

Method for representing behavior of a system by depicting its states and events that cause system to change state is state diagram. A state is an externally observable mode of behavior. In addition, the state diagram indicates actions taken as a consequence of a particular event.

To illustrate use of a state diagram, consider software embedded within safeHome control panel that is responsible for reading user input. A simplified UML state diagram is shown in figure 2.

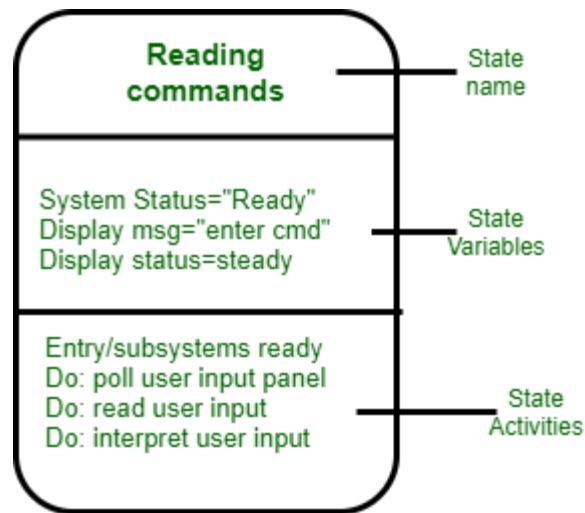


Figure 2: UML state diagram notation

- **Flow-oriented**

elements:

As it flows through a computer-based system information is transformed. System accepts input, applies functions to transform it, and produces output in a various form. Input may be a control signal transmitted by a transducer, a series of numbers typed by human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. Transform may compromise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output produces a 200-page report or may light a single LED. In effect, we can create a flow model for any computer-based system, regardless of size and complexity.

3.10 SCENARIO-BASED MODELING

A requirement modeling with UML begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

3.10.1 Creating a Preliminary Use Case:

A use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know

- (1) What to write about,
- (2) How much to write about it,
- (3) How detailed to make your description, and
- (4) How to organize the description?

What to write about? The first two requirements engineering tasks—inception and elicitation— provide you with the information you’ll need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams developed as part of requirements modeling.

3.10.2 Refining a Preliminary Use Case:

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions.

- Can the actor take some other action at this point?
- Is it possible that the actor will encounter some error condition at this point? If so, what is it?
- Is it possible that the actor will encounter some other behavior at this point? If so, what is it?

Answers to these questions result in the creation of a set of secondary scenarios that are part of the original use case but represent alternative behavior. Can the actor take some other action at this point? The answer is “yes.” Is it possible that the actor will encounter some error condition at this point? Any number of error conditions can occur as a computer-based system operates. Is it possible that the actor will encounter some other behavior at this point? Again, the answer to the question is “yes.”

In addition to the three generic questions suggested, the following issues should also be explored:

- Are there cases in which some “validation function” occurs during this use case? This implies that validation function is invoked and a potential error condition might occur.
- Are there cases in which a supporting function (or actor) will fail to respond appropriately? For example, a user action awaits a response but the function that is to respond times out.
- Can poor system performance result in unexpected or improper user actions?

3.10.3 Writing a Formal Use Case:

The informal use cases presented are sometimes sufficient for requirements modeling. However, when a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

3.11 CLASS-BASED MODELING

Class-based modeling represents the objects that the system will manipulate, the operations that will be applied to the objects to affect the manipulation, relationships 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.

3.11.1 Identifying Analysis Classes

Classes are determined by underlining each noun or noun phrase and entering it into a simple table. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space. Analysis classes manifest themselves in one of the following ways:

- *External entities* that produce or consume information to be used by a computer-based system.
- *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.

- **Occurrences or events** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- **Roles** (e.g., manager, engineer, salesperson) played by people who interact with the system.
- **Organizational units** (e.g., division, group, team) that are relevant to an application.
- **Places** (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- **Structures** (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

3.11.2 Specifying Attributes:

Attributes are the set of data objects that fully define the class within the context of the problem. Attributes describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space. To develop a meaningful set of attributes for an analysis class, you should study each use case and select those “things” that reasonably “belong” to the class.

3.11.3 Defining Operations:

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories:

- (1) Operations that manipulate data in some way.
- (2) Operations that perform a computation,
- (3) Operations that inquire about the state of an object, and
- (4) Operations that monitor an object for the occurrence of a controlling event.

These functions are accomplished by operating on attributes and/or associations. Therefore, an operation must have “knowledge” of the nature of the class’ attributes and associations. As a first iteration at deriving a set of operations for an analysis class, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class. To accomplish this, the grammatical parse is again studied and verbs are isolated. Some of these verbs will be legitimate operations and can be easily connected to a specific class.

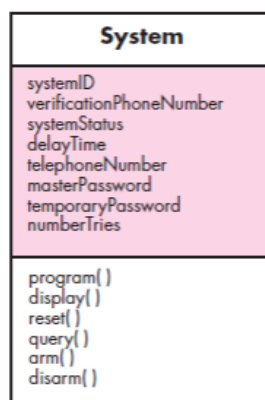


Fig 6.9: Class diagram for the System class

3.11.4 Class-Responsibility-Collaborator (CRC) Modeling:

Class-responsibility-collaborator (CRC) modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.

One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code.

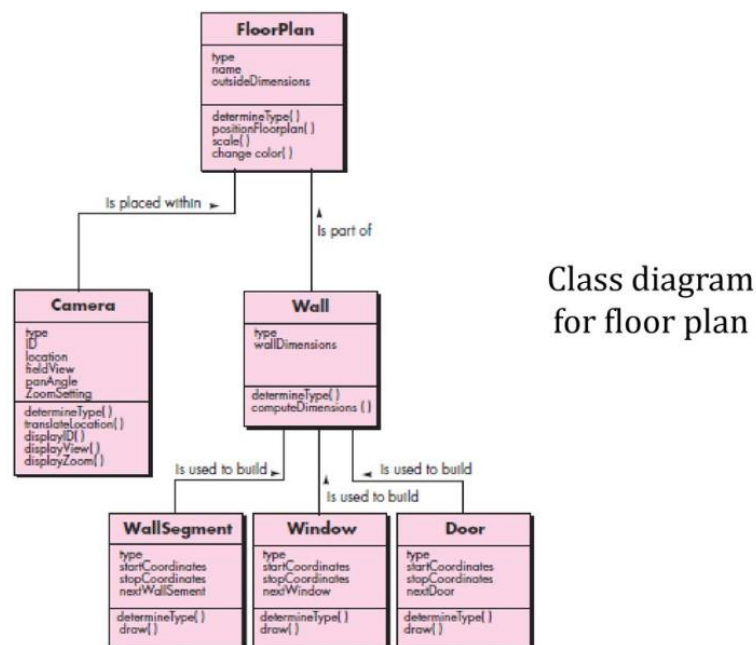
Ambler describes CRC modeling in the following way:

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card, you write the name of the class. In the body of the card, you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is “anything the class knows or does”. Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility.

In general, collaboration implies either a request for information or a request for some action.

A simple CRC index card for the FloorPlan class is illustrated in Figure 6.11. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes Wall and Camera are noted next to the responsibility that will require their collaboration.



Classes. The taxonomy of class types presented can be extended by considering the following categories:

- **Entity classes**, also called model or business classes, are extracted directly from the statement of the problem. These classes typically represent things that are to be stored in a database and persist throughout the duration of the application.

- **Boundary classes** are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. Entity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

Class: FloorPlan	
Description	
Responsibility:	Collaborator:
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	Wall
Shows position of video cameras	Camera

Fig 6.11: A CRC model index card

- Controller classes manage a “unit of work” from start to finish. That is, controller classes can be designed to manage

- (1) the creation or update of entity objects,
 - (2) the instantiation of boundary objects as they obtain information from entity objects,
 - (3) complex communication between sets of objects,
 - (4) validation of data communicated between objects or between the user and the application.
- In general, controller classes are not considered until the design activity has begun. Responsibilities.

The five guidelines for allocating responsibilities to classes:

1. System intelligence should be distributed across classes to best address the needs of the problem. Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways. “Dumb” classes can be modeled to act as servants to a few “smart” classes. To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. **Example** is aggregation of classes.
2. Each responsibility should be stated as generally as possible. This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy
3. Information and the behavior related to it should reside within the same class. This achieves the object-oriented principle called encapsulation. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. Information about one thing should be localized with a single class, not distributed across multiple classes. A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be

shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.

5. Responsibilities should be shared among related classes, when appropriate. There are many cases in which a variety of related objects must all exhibit the same behavior at the same time.

Collaborations:

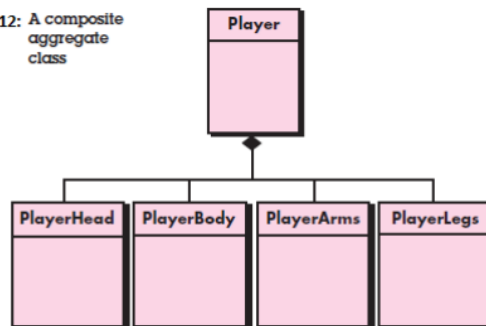
Classes fulfill their responsibilities in one of two ways:

- (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
- (2) A class can collaborate with other classes.

To help in the identification of collaborators, you can examine three different generic relationships between classes

- (1) The is-part-of relationship,
- (2) The has-knowledge-of relationship, and
- (3) The depends-upon relationship.

Fig 6.12: A composite aggregate class



When a complete CRC model has been developed, stakeholders can review the model using the following approach

1. All participants in the review are given a subset of the CRC model index cards. Cards that collaborate should be separated
2. All use-case scenarios (corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately.
4. When the token is passed, the holder of the Sensor card is asked to describe the responsibilities noted on the card.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This continues until the use case is finished. When all use cases have been reviewed, requirements modeling continue.

3.11.5 Associations and Dependencies:

In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another. In UML these relationships are called associations. In some cases, an association may be further defined by indicating multiplicity. Referring to the multiplicity constraints, where “one or more” is represented using 1..*, and “0 or more” by 0..*. In UML, the asterisk indicates an unlimited upper bound on the range.

Fig 6.13 Multiplicity

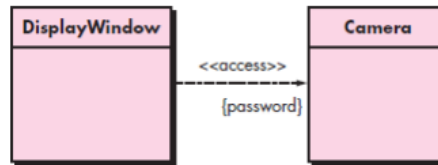
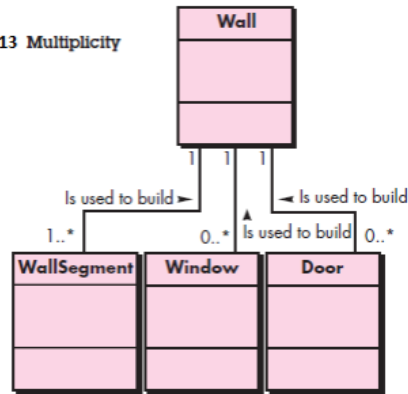


Fig 6.14: Dependencies

3.11.6 Analysis Packages: An important part of analysis modeling is categorization. That is, various elements of the analysis model are categorized in a manner that packages them as a grouping—called an analysis package—that is given a representative name. For example, Classes such as Tree, Landscape, Road, Wall, Bridge, Building, and Visual effect might fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints. These classes can be grouped in analysis packages as shown in Figure 6.15. The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages.

Although they are not shown in the figure, other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package

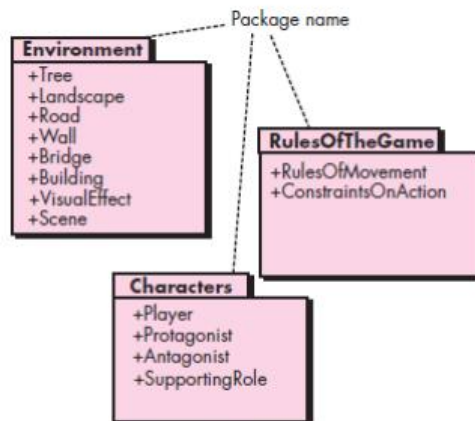


Fig 6.15: Packages

3.12 Functional Modeling

Functional Modeling provides the outline that what the system is supposed to do. It does not describe what is the need of evaluation of data, when they are evaluated and how they are evaluated apart from all it only represents origin of data values. It describes the function of internal processes with the help of DFD (Data Flow Diagram).

Data Flow Diagrams: Function modeling is represented with the help of DFDs. DFD is the graphic representation of data. It shows the input, output and processing of the system. When we are trying to create our own business, website, system, project then there is need to find out how information passes from one process to another so all are done by DFD. There are a number of levels in DFD but up to the third level DFD is sufficient for understanding of any system.

Relationship between Object, Dynamic, and Functional Models

The Object Model, the Dynamic Model, and the Functional Model are complementary to each other for a complete Object-Oriented Analysis.

- Object modeling develops the static structure of the software system in terms of objects. Thus, it shows the “doers” of a system.
- Dynamic Modeling develops the temporal behavior of objects in response to external events. It shows the sequences of operations performed on the objects.
- Functional models give an overview of what the system should do.

MODULE – 4 DESIGN CONCEPTS

Design Concepts: Design within the Context of Software Engineering, the Design Process, Design Concepts, The Design Model.

Introduction to the design process

- The main aim of design engineering is to generate a model which shows firmness, delight and commodity.

- Software design is an iterative process through which requirements are translated into the blueprint for building the software.

Software quality guidelines

- A design is generated using recognizable architectural styles and composes a good design characteristic of components and it is implemented in evolutionary manner for testing.

- The design of the software must be modular i.e the software must be logically partitioned into elements.

- In design, the representation of data, architecture, interface and components should be distinct.

- A design must carry appropriate data structure and recognizable data patterns.

- Design components must show independent functional characteristics.

- A design creates an interface that reduces the complexity of connections between the components.

- A design must be derived using the repeatable method.

- The notations should be used in design which can effectively communicate its meaning.

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need for more specific and detailed requirements in software terms. The output of this process can be used directly into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfil the requirements mentioned in SRS.

Software Design Levels

Software design yields three levels of results:

- **Architectural Design** - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design**- The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes the modular structure of each sub-system and their relation and interaction among each other.
- **Detailed Design**- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines the logical structure of each module and their interfaces to communicate with other modules.

Modularization

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantages of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

Concurrency

Back in time, all software was meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.

In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides the capability to the software to execute more than one part of code in parallel to each other. It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

Example

The spell check feature in word processor is a module of software, which runs alongside the word processor itself.

Design Verification

The output of software design process is design documentation, pseudo codes, detailed logic diagrams, process diagrams, and detailed description of all functional or non-functional requirements. The next phase, which is the implementation of software, depends on all outputs mentioned above.

It is then necessary to verify the output before proceeding to the next phase. The earlier any mistake is detected, the better it is or it might not be detected until testing of the product. If the outputs of design phase are in formal notation form, then their associated tools for

verification should be used otherwise a thorough design review can be used for verification and validation.

By structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is important for good software design, accuracy and quality.

S. No.	Software Design	Software Architecture
01.	Software design is about designing individual modules/components.	Software architecture is about the complete architecture of the overall system.
02.	Software design defines the detailed properties.	Software architecture defines the fundamental properties.
03.	In general, it refers to the process of creating a specification of software artifact which will help developers to implement the software.	In general, it refers to the process of creating high level structure of a software system.
04.	It helps to implement the software.	It helps to define the high-level infrastructure of the software.
05.	Software design avoids uncertainty.	Software architecture manages uncertainty.
06.	Software design is more about an individual module/component.	Software architecture is more about the design of entire system.
07.	It is considered as one initial phase of Software Development Cycle (SSDLC) and it gives detailed idea to developers to implement consistent software.	It is a plan which constrains software design to avoid known mistakes and it achieves one organization business and technology strategy.
08.	Some of software design patterns are creational, structural and behavioral.	Some of software architecture patterns are micro service, server less and event driven.
09.	In one word the level of software design is implementation.	In one word the level of software architecture is structure.
10.	How we are building software design.	What we are building is software architecture.

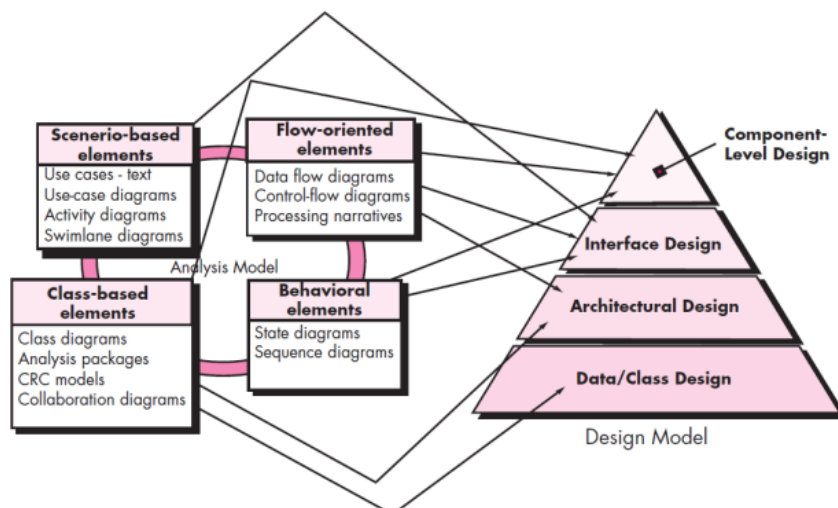
4.1 DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task. Using design notation and design methods, design produces a data/class design, an architectural design, an interface design, and a component design. The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented [Sha96]. The architectural design representation—the framework of a computer-based system—is derived from the requirements model. The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

FIGURE 8.1 Translating the requirements model into the design model



The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

The importance of software design can be stated with a single word—quality. Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder ‘s requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

4.2 DESIGN PROCESS

Software Quality Guidelines and Attributes

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.

Quality Guidelines

1. A designer should exhibit architecture that
 - (a) has been created using recognizable architectural styles or patterns,
 - (b) is composed of components that exhibit good design characteristics
 - (c) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
4. A design should lead to components that exhibit independent functional characteristics.
5. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
6. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
7. A design should be represented using a notation that effectively communicates its meaning.

Quality attributes

The attributes of the design name 'FURPS' are as follows:

i. **Functionality:**

It evaluates the feature set and capabilities of the program.

ii. **Usability:**

It is accessed by considering factors such as human factor, overall aesthetics, consistency and documentation.

iii. **Reliability:**

It is evaluated by measuring parameters like frequency and security of failure, output result accuracy, the mean-time-to-failure (MTTF), recovery from failure and the program predictability.

iv. **Performance:**

It is measured by considering processing speed, response time, resource consumption, throughput and efficiency.

v. **Supportability:**

- It combines the ability to extend the program, adaptability, serviceability. These three terms define maintainability.
- Testability, compatibility and configurability are the terms using which a system can be easily installed and found the problem easily.
- Supportability also consists of more attributes such as compatibility, extensibility, fault tolerance, modularity, reusability, robustness, security, portability, scalability.

4.3 DESIGN CONCEPTS

The set of fundamental software design concepts are as follows:

1. Abstraction

When considering a modular solution to any problem, many levels of abstraction can be posed -

- A solution is stated in broad terms using the language of the problem environment at the highest-level abstraction.
- The lower level of abstraction provides a more detailed description of the solution.
- A sequence of instructions that contain a specific and limited function refers to a procedural abstraction.
- A collection of data that describes a data object is a data abstraction.
- **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. An **example** of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).
- **A data abstraction** is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

2. Architecture

- The overall structure of the software is known as software architecture.
- Structure provides conceptual integrity for a system in a number of ways.
- The architecture is the structure of program modules where they interact with each other in a specialized way.
- The components use the structure of data.
- The aim of the software design is to obtain an architectural framework of a system.
- The more detailed design activities are conducted from the framework.

Set of properties that should be specified as part of an architectural design:

Structural properties:

. Defines the components of a system like modules, objects and filters and the manner in which those components are packaged and interact with one another

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:

Defines how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability and other system characteristics.

Families of related systems:

The design should have the ability to reuse architectural building blocks. Given the specification of these properties, the architectural design can be represented using different models

i. Structural models: represent architecture as an organized collection of program components.

ii. Framework models:

increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

iii. 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.

iv. Process Models: focus on the design of the business or technical process that the system must accommodate.

v. Functional Models: Used to represent the functional hierarchy of a system.

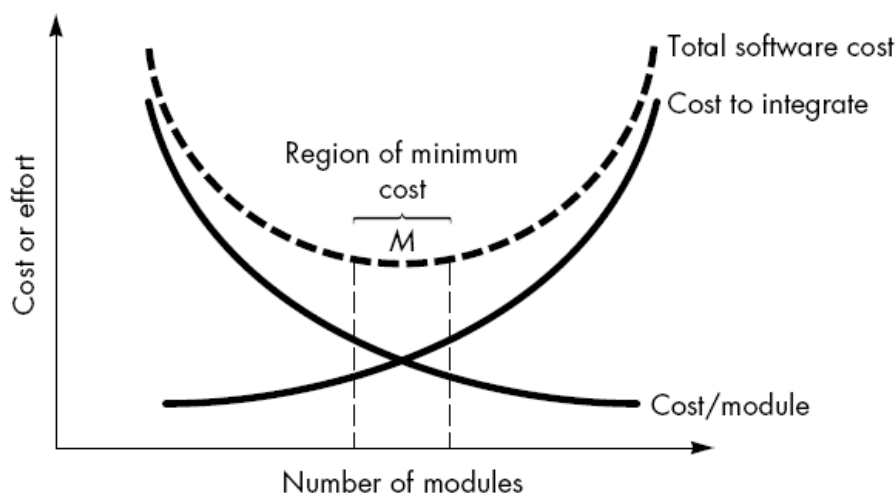
3. Patterns

A design pattern describes a design structure and that structure solves a particular design problem in a specified content.

4. 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.

5. Modularity

- Software is separately divided into name and addressable components. Sometime they are called modules which integrate to satisfy the problem requirements.
- Modularity is the single attribute of software that permits a program to be managed easily.



5. **Information** **hiding**

Modules must be specified and designed so that the information like algorithm and data presented in a module is not accessible for other modules not requiring that information.

6. **Functional independence**

Functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.

The functional independence is accessed using two criteria i.e Cohesion and coupling.

COUPLING AND COHESION

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

a. Cohesion

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

- Cohesion is an extension of the information hiding concept.
- A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.

There are **seven types** of cohesion, namely –

- **Co-incident cohesion** - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- **Logical cohesion** - When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion** - When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion** - When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.
- **Communicational cohesion** - When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.
- **Sequential cohesion** - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion** - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

B.

COUPLING

Coupling is an indication of interconnection between modules in a structure of software. Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely -

- **Content coupling** - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling**- When multiple modules have read and write access to some global data, it is called common or global coupling.
- **Control coupling**- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling**- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- **Data coupling**- Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

7. Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement is reached

Aspects:

A module that enables the concern to be implemented across all other concerns that it crosscuts.

In an ideal context, an aspect is implemented as a separate module 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.

8. Refactoring

- It is a reorganization technique which simplifies the design of components without changing its function behavior.
- Refactoring is the process of changing the software system in a way that it does not change the external behavior of the code and still improves its internal structure.

9. OO design concept

- Object Oriented is a popular design approach for analyzing and designing an application.
- Most of the languages like C++, Java, .NET use object-oriented design concept.
- Object-oriented concepts are used in design methods such as classes, objects, polymorphism, encapsulation, inheritance, dynamic binding, information hiding, interface, constructor, and destructor.

- The main advantage of object-oriented design is that it improves the software development and maintainability.
- Another advantage is that it is faster and has low-cost development, and it creates high quality software.
- The disadvantage of the object-oriented design is that it has a larger program size and it is not suitable for all types of programs.

10. Design classes

- The model of software is defined as a set of design classes.
- Every class describes the elements of problem domain and that focus on features of the problem which are user visible.
- A set of design classes refined the analysis class by providing design details.

There are five different types of design classes and each type represents the layer of the design architecture as follows:

1. User interface classes

- These classes are designed for Human Computer Interaction (HCI).
- These interface classes define all abstractions which are required for Human Computer Interaction (HCI).

2. Business domain classes

- These classes are commonly refinements of the analysis classes.
- These classes are recognized as attributes and methods which are required to implement the elements of the business domain.

3. Process classes

It implements the lower-level business abstraction which is needed to completely manage the business domain class.

4. Persistence classes

It shows data stores that will persist behind the execution of the software.

5. System Classes

System classes implement software management and control functions that allow us to operate and communicate in computing environment and outside world.

Design class characteristics

The characteristics of well-formed designed class are as follows:

1. Complete and sufficient

A design class must be the total encapsulation of all attributes and methods which are required to exist for the class.

2. Primitiveness

- The method in the design class should fulfil one service for the class.
- If service is implemented with a method, then the class should not provide another way to fulfill same thing.

3. High cohesion

- A cohesion design class has a small and focused set of responsibilities.
- For implementing the set of responsibilities, the design classes are applied single-mindedly to the methods and attributes.

4. Low-coupling

- All the design classes should collaborate with each other in a design model.
- The minimum acceptable of collaboration must be kept in this model.
- If a design model is highly coupled then the system is difficult to implement, to test and to maintain over time.

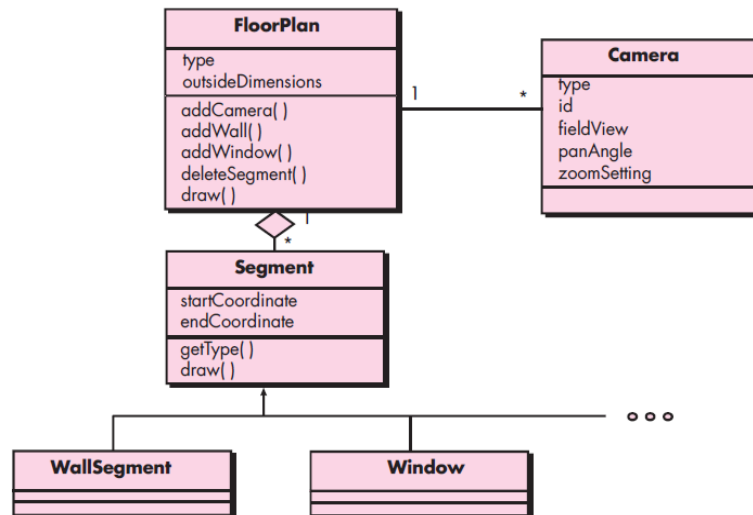
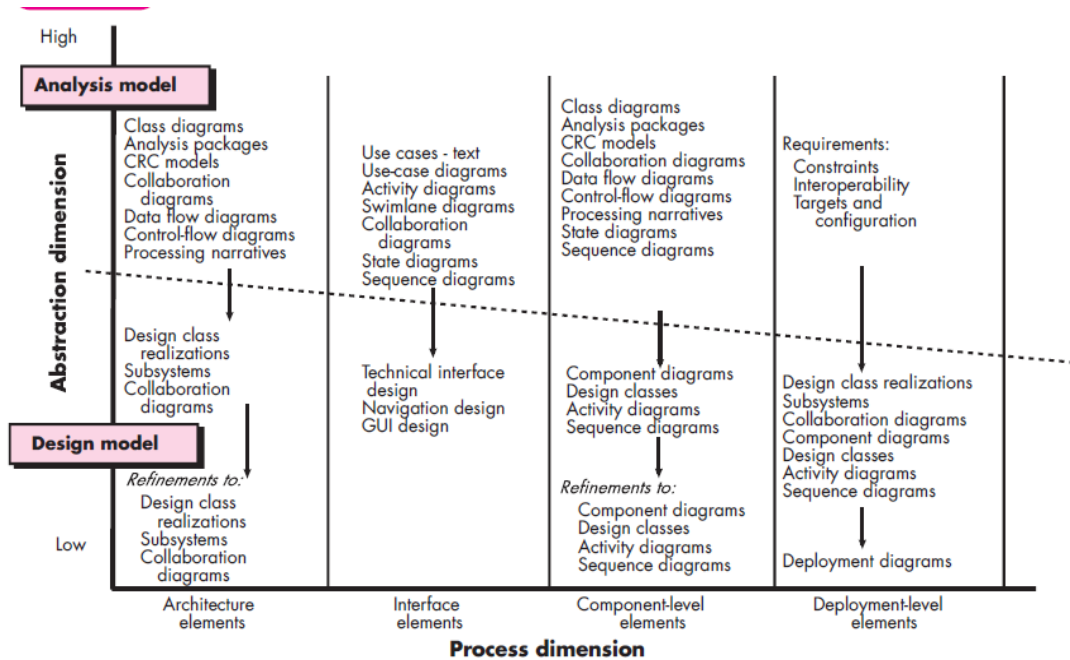


Fig: Design class for Floor Plan and composite Aggregation for the class

4.4 THE DESIGN MODEL:



The following are the types of design elements:

1. Data design elements

- The data design element produced a model of data that represents a high level of abstraction.
- This model is then more refined into more implementation specific representation which is processed by the computer-based system.
- The structure of data is the most important part of the software design.

2. Architectural design elements

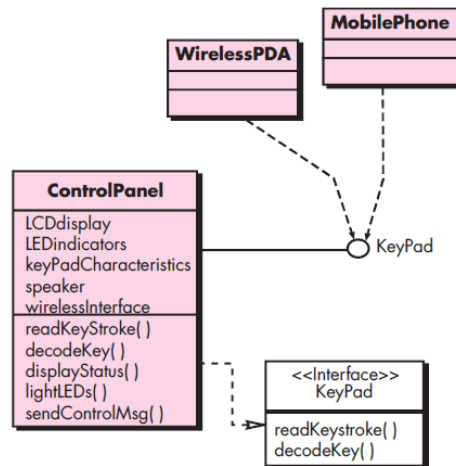
- The architecture design elements provide us with an overall view of the system.
- The architectural design element is generally represented as a set of interconnected subsystems that is derived from analysis packages in the requirement model.

The architecture model is derived from the following sources:

- The information about the application domain to build the software.
- Requirement model elements like data flow diagram or analysis classes, relationship and collaboration between them.
- The architectural style and pattern as per availability.

3. Interface design elements

- The interface design elements for software represent the information flow within it and out of the system.
- They communicate between the components defined as part of architecture.



The following are the important elements of the interface design:

1. The user interface
2. The external interface to the other systems, networks etc.
3. The internal interface between various components.
4. Component level diagram elements
 - The component level design for software is similar to the set of detailed specifications of each room in a house.
 - The component level design for the software completely describes the internal details of each software component.
 - The processing of data structure occurs in a component and an interface which allows all the component operations.
 - In the context of object-oriented software engineering, a component shown in a UML diagram.
 - The UML diagram is used to represent the processing logic.

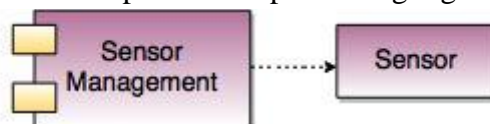


Fig. - UML component diagram for sensor management

5. Deployment level design elements

- The deployment level design element shows the software functionality and subsystem that allocated in the physical computing environment which support the software.
- Following figure shows three computing environment as shown. These are the personal computer, the CPI server and the Control panel.

MODULE 5

ARCHITECTURAL DESIGN

Architectural Design - A Recommended Approach: Software Architecture, Agility and Architecture, Architectural Styles, Architectural Considerations, Architectural Decisions, Architectural Design, Assessing Alternative Architectural Designs.

Software Architecture and Design — Introduction

QUICK LOOK

What is it? 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.

Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data

warehouse designer creates the data architecture for a system. The “system architect” selects an appropriate architectural style from the requirements derived during software requirements analysis.

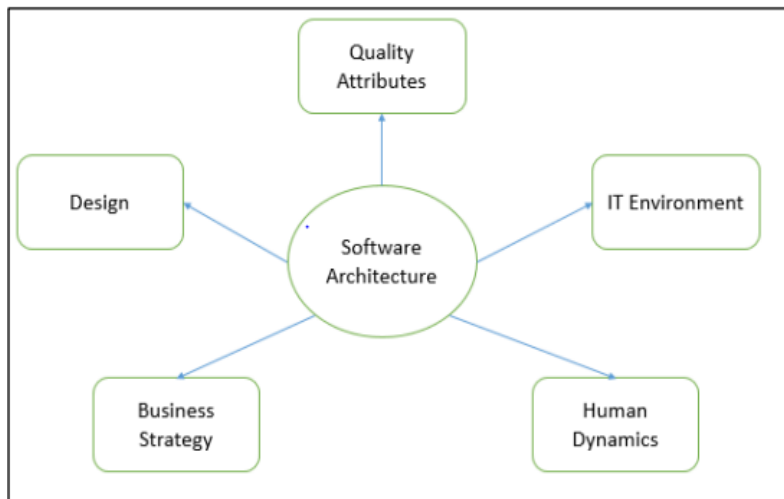
Why is it important? You wouldn’t attempt to build a house without a blueprint, would you? You also wouldn’t begin drawing blueprints by sketching the plumbing layout for the house. You’d need to look at the big picture—the house itself—before you worry about details. That’s what architectural design does—it provides you with the big picture and ensures that you’ve got it right.

What are the steps? 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.

What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

How do I ensure that I’ve done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design is a process that includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



Architecture and Design into two distinct phases: Software Architecture and Software Design. In Architecture, nonfunctional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

5.1 Software Architecture

Architecture serves as a blueprint for a system. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

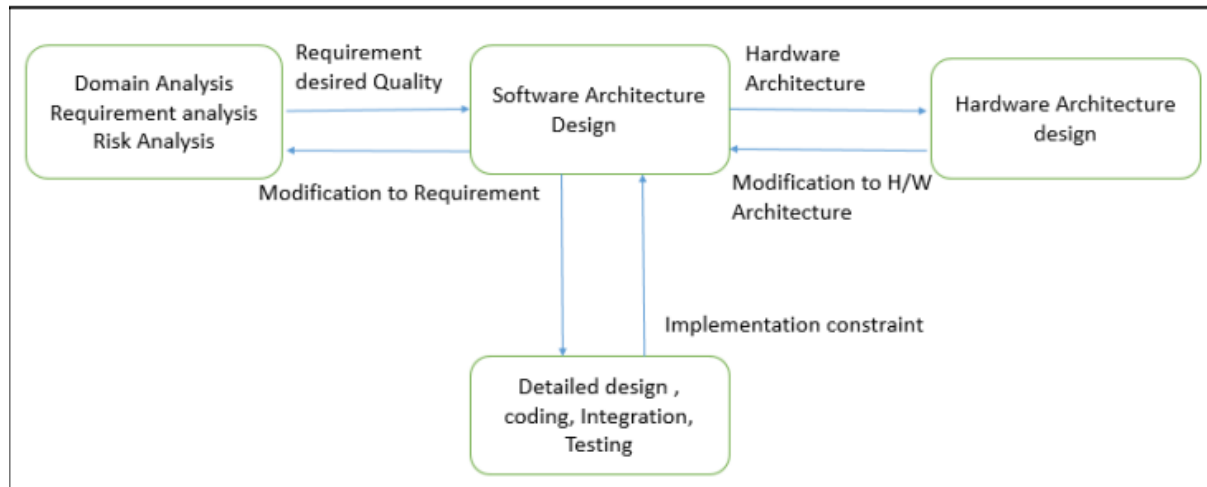
- It defines a structured solution to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
- It involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of:
 - ✓ Selection of structural elements and their interfaces by which the system is composed.
 - ✓ Behavior as specified in collaborations among those elements.
 - ✓ Composition of these structural and behavioral elements into large subsystem.
 - ✓ Architectural decisions align with business objectives.
 - ✓ Architectural styles that guide the organization.

Software Design

Software design provides a design plan that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows:

- To negotiate system requirements, and to set expectations with customers, marketing and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.

It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.



Goals of Architecture

The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements. Some of the other goals are as follows:

- Expose the structure of the system, but hide its implementation details.
- Realize all the use-cases and scenarios.
- Try to address the requirements of various stakeholders.
- Handle both functional and quality requirements.
- Reduce the goal of ownership and improve the organization's market position.
- Improve quality and functionality offered by the system.
- Improve external confidence in either the organization or system.

Limitations

Software architecture is still an emerging discipline within software engineering. It has the following limitations:

- Lack of tools and standardized ways to represent architecture
- Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
- Lack of awareness of the importance of architectural design to software development
- Lack of understanding of the role of software architect and poor communication among stakeholders.
- Lack of understanding of the design process, design experience and evaluation of design

Role of Software Architect

A Software Architect provides a solution that the technical team can create and design for the entire application. A software architect should have expertise in the following areas:

Design Expertise

- Expert in software design, including diverse methods and approaches such as object-oriented design, event-driven design, etc.
- Lead the development team and coordinate the development efforts for the integrity of the design.
- Should be able to review design proposals and tradeoffs among them.

Hidden Role of Software Architect

- Facilitates the technical work among team members and reinforcing the trust relationship in the team.
- Information specialist who shares knowledge and has vast experience.
- Protect the team members from external forces that would distract them and bring less value to the project.

5.2 Agility and Architecture

The view of some agile developers is that architectural design is equated with “big design upfront.” In their view, this leads to unnecessary documentation and the implementation of unnecessary features. However, most agile developers would agree that it is important to focus on software architecture when a system is complex (i.e., when a product has a large number of requirements, lots of stakeholders, or a large number of global users). For this reason, it is important to integrate new architectural design practices into agile process models.

To make early architectural decisions and avoid the rework required to correct the quality problems encountered when the wrong architecture is chosen, agile developers need to anticipate architectural elements² and implied structure that emerges from the collection of user stories gathered. By creating an architectural prototype (e.g., a *walking skeleton*) and developing explicit architectural work products to communicate the right information to the necessary stakeholders, an agile team can satisfy the need for architectural design.

Using a technique called *storyboarding*, the architect contributes architectural user stories to the project and works with the product owner to prioritize the architectural stories with the business user stories as “sprints” (work units) are planned. The architect works with the team during the sprint to ensure that the evolving software continues to show high architectural quality as defined by the nonfunctional product requirements. If quality is high, the team is left alone to continue development on its own. If not, the architect joins the team for the duration of the sprint. After the sprint is completed, the architect reviews the working prototype for quality before the team presents it to the stakeholders in a formal sprint review. Well-run agile projects make use of iterative work product delivery (including architectural documentation) with each sprint. Reviewing the work products and code as it emerges from each sprint is a useful form of architectural review.

Responsibility-driven architecture (RDA) is a process that focuses on when, how, and who should make the architectural decisions on a project team. This approach also emphasizes the role of architect as being a servant-leader rather than an autocratic decision maker and is consistent with the agile philosophy. The architect acts as facilitator and focuses on how the development team works to accommodate stakeholder’s nontechnical concerns (e.g., business, security, usability). Agile teams usually have the

freedom to make system changes as new requirements emerge. Architects want to make sure that the important parts of the architecture were carefully considered and that developers have consulted the appropriate stakeholders. Both concerns may be satisfied by making use of a practice called *progressive sign-off* in which the evolving product is documented and approved as each successive prototype is completed. Using a process that is compatible with the agile philosophy provides verifiable sign-off for regulators and auditors, without preventing the empowered agile teams from making the decisions needed. At the end of the project, the team has a complete set of work products and the architecture has been reviewed for quality as it evolved.

5.3 ARCHITECTURAL STYLES

When a builder uses the phrase — center hall colonial to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an architectural style as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a template for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the style—a —center hall colonial—guides the builder in his work. 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.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered, the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components. An architectural pattern, like an architectural style, imposes a transformation on the design of architecture.

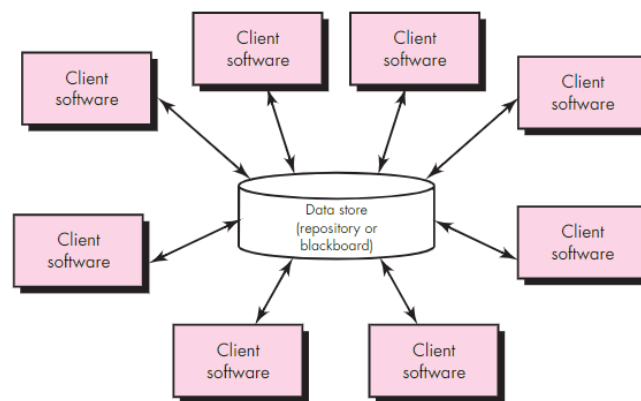
A pattern differs from a style in a number of fundamental ways:

- (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). Patterns can be used in conjunction with an

architectural style to shape the overall structure of a system. A Brief Taxonomy of Architectural Styles 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:

1. 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. Client software accesses a central repository. In some cases, the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a —blackboard that sends notifications to client software when data of interest to the client changes. Data-centered architectures promote integrability. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

FIGURE 9.1
Data-centered
architecture

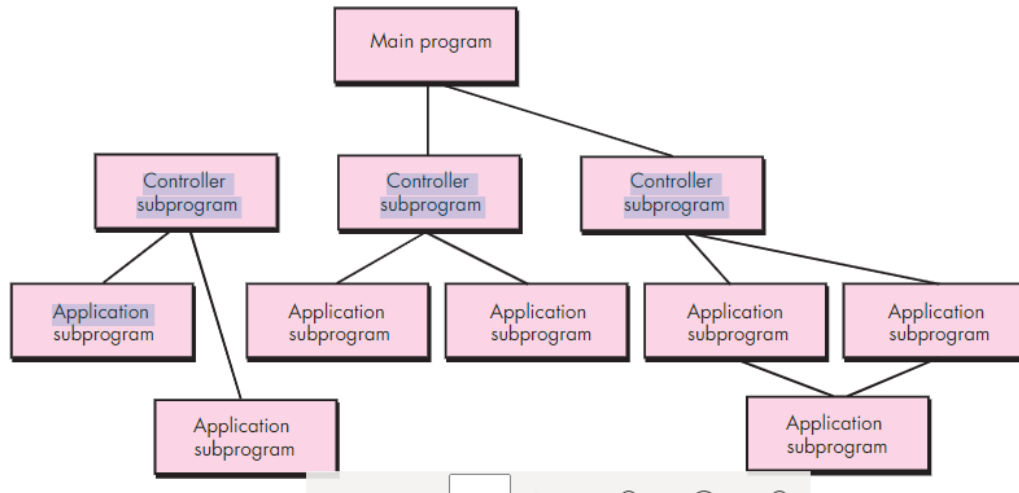


2. Data-flow architectures: This architecture is applied when input data is to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 9.2) has a set of components, called filters, connected by pipes that transmit data from one component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

3. Call and return architectures: This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles 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.

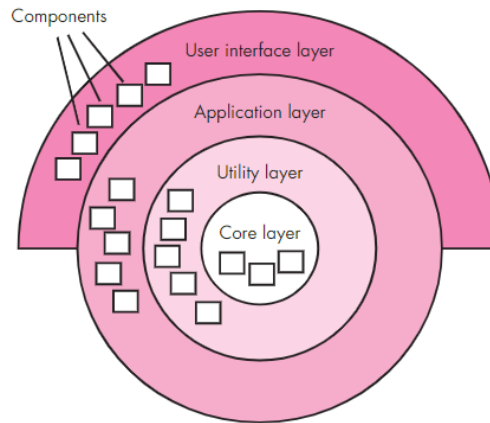
FIGURE 9.3 Main program/subprogram architecture



Remote procedures are called architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.

- 4. Object-oriented architecture:** 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.
- 5. 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. These architectural styles are only a small subset of those available. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. **For example,** a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

FIGURE 9.4
Layered
architecture



5.1.2 Architectural Patterns

As the requirements model is developed, we ‘ll notice that the software must address a number of broad problems that span the entire application. **For example**, the requirements model for virtually every e-commerce application is faced with the following problem:

How do we offer a broad array of goods to a broad array of customers and allow those customers to purchase our goods online? The requirements model also defines a context in which this question must be answered.

For example, an ecommerce business that sells golf equipment to consumers will operate in a different context than an ecommerce business that sells high-priced industrial equipment to medium and large corporations. In addition, a set of limitations and constraints may affect the way in which you address the problem to be solved. Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design. **For example**, the overall architectural style for an application might be call-and return or object-oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns. Some of these problems and a more complete discussion of architectural patterns.

5.1.3 Organization and Refinement Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions [Bas03] provide insight into an architectural style:

Control:

- How is control managed within architecture?
- Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
- How do components transfer control within the system? How is control shared among components?
- What is the control topology (i.e., the geometric form that the control takes)?
- Is control synchronized or do components operate asynchronously?

Data:

- How is data communicated between components?
- Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)?
- Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?
- How do functional components interact with data components?
- Are data components passive or active (i.e., does the data component actively interact with other components in the system)?
- How do data and control interact within the system?
- These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

5.3 ARCHITECTURAL CONSIDERATIONS

Buschmann and Henny suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made.

- **Economy.** The best software is uncluttered and relies on abstraction to reduce unnecessary detail. It avoids complexity due to unnecessary functions and features.
- **Visibility:** As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model later. Important design and domain concepts must be communicated effectively.
- **Spacing:** Separation of concerns in a design is sometimes referred to as *spacing*. Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility.
- **Symmetry:** Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate. As an example of architectural symmetry, consider a **customer account** object whose life cycle is modeled directly by a software architecture that requires both *open* () and *close* () methods. Architectural symmetry can be both structural and behavioral.
- **Emergence:** Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures. For example, many real-time software applications are event driven. The sequence and duration of these events that define the system's behavior is an emergent quality. Because it is very difficult to plan for every possible sequence of events, a system architect should create a flexible system that accommodates this emergent behavior.

These considerations do not exist in isolation. They interact with each other and are moderated by each other. For example, spacing can be both reinforced and reduced by economy. Visibility can be balanced by spacing.

The architectural description for a software product is not explicitly visible in the source code used to implement it. As a consequence, code modifications made over time (e.g., software maintenance activities) can cause slow erosion of the software architecture. The challenge for a designer is to find suitable abstractions for architectural information. These abstractions have the potential to add structuring that improves readability and maintainability of the source code

5.5 Assessing Alternative Architectural Designs:

The big question for a software architect and the software engineers who will work to build a system is simple: Will the architectural bet pay off? To help answer this question, architectural design should result in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method (ATAM) that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. Collect scenarios. A set of use cases is developed to represent the system from the user's point of view.

2. Elicit requirements, constraints, and environment description. This information is required as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.

3. Describe the architectural styles and patterns that have been chosen to address the scenarios and requirements. The architectural style(s) should be described using one of the following architectural views:

- **Module** view for analysis of work assignments with components and the degree to which information hiding has been achieved.
- **Process** view for analysis of system performance.
- **Data flow** view for analysis of the degree to which the architecture meets functional requirements.

4. Evaluate quality attributes by considering each attribute in isolation. The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.

5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed sensitivity points.

6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5. The SEI describes this approach in the following manner: Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive.

For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). The number of servers, then, is a trade-off point with respect to this architecture. These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.