



You:

Describe elements of the requirement model. Scenario-based ,Class-based, Behavioral, Flow-oriented



ChatGPT:

The requirement model is a crucial component of software engineering that helps capture, document, and manage the specifications and expectations for a software system. Different types of requirement models serve various purposes throughout the software development life cycle. Here's a brief description of four common elements of the requirement model:

1. Scenario-based:

- *Description:* Scenario-based requirements focus on describing how the system will behave in specific situations or contexts. Scenarios are typically narratives that outline a sequence of events involving users, the system, and external entities.
- *Purpose:* They provide a concrete and detailed understanding of the system's expected behavior in real-world situations, helping stakeholders visualize the system's functionality in different contexts.

2. Class-based:

- *Description:* Class-based requirements organize information based on the classification of entities or objects within the system. These entities can include classes, objects, and their relationships, forming the basis for object-oriented design and implementation.
- *Purpose:* Class-based models facilitate a clear understanding of the data structure and relationships within the system, aiding in the development of a robust and maintainable software architecture.

3. Behavioral:

- *Description:* Behavioral requirements focus on describing the dynamic aspects of the system, emphasizing how it responds to stimuli or events. This includes specifications for system behavior in response to user input, external events, and internal processes.
- *Purpose:* Behavioral models help stakeholders comprehend the system's functionality from an interactive and dynamic perspective, providing insights into how the system should respond to various stimuli.

4. Flow-oriented:

- *Description:* Flow-oriented requirements emphasize the flow of data, control, and events within the system. These models often use diagrams, such as flowcharts or activity diagrams, to illustrate the sequence of actions or steps in a particular process.
- *Purpose:* Flow-oriented models help stakeholders understand the logical flow of activities and processes within the system, making it easier to identify potential bottlenecks, dependencies, and areas for optimization.

Each type of requirement model serves a specific purpose, and often a combination of these models is used to provide a comprehensive and detailed representation of the software system's requirements. Tailoring the choice of models to the specific needs of the project and its stakeholders is crucial for effective communication and successful software development.



Explain flow oriented software modeling.



Flow-oriented software modeling involves representing the logical flow of activities, events, or processes within a software system using visual diagrams. This type of modeling is particularly useful for understanding the sequential order of operations and the overall structure of a system. Flow-oriented models often use diagrams such as flowcharts or activity diagrams to illustrate the flow of control, data, and events. Here are key aspects of flow-oriented software modeling:

1. Flowcharts:

- *Description:* Flowcharts are diagrams that use different shapes and arrows to represent various steps, decisions, and processes within a system. Each shape in a flowchart represents a specific type of action or process, and arrows indicate the flow of control or data between them.
- *Purpose:* Flowcharts provide a visual representation of the sequential flow of activities in a system, making it easier for stakeholders to understand the order of execution and decision points.

2. Activity Diagrams:

- *Description:* Activity diagrams, part of the Unified Modeling Language (UML), are a type of flow-oriented diagram that depicts the flow of activities and actions within a system. Nodes represent activities, and arrows represent the transitions between them.
- *Purpose:* Activity diagrams are especially useful for modeling the dynamic aspects of a system, showing the sequence of actions and the conditions under which they occur. They are particularly beneficial for representing business processes.

3. Process Flow:

- *Description:* Process flow diagrams illustrate the flow of data or information through different stages or processes in a system. They often use symbols to represent different types of processes, decision points, and data storage.
- *Purpose:* Process flow diagrams are valuable for understanding the overall structure and flow of data within a system, aiding in the identification of dependencies and potential areas for optimization.

4. Data Flow Diagrams (DFD):

- *Description:* DFDs depict the flow of data within a system and how it is processed. They use circles to represent processes, arrows to represent data flow, and rectangles to represent data stores or external entities.
- *Purpose:* DFDs help in understanding how data moves through different processes within a system, providing insights into data transformation and storage.

5. Sequential Flow of Events:

- *Description:* This aspect of flow-oriented modeling involves documenting the sequential flow of events or actions that occur in response to external stimuli, user interactions, or system triggers.
- *Purpose:* By visualizing the sequence of events, stakeholders can gain a clear understanding of how the system behaves over time and in different scenarios.

Flow-oriented software modeling is valuable for communication among stakeholders, including developers, designers, and business analysts. It aids in visualizing the system's behavior, identifying potential issues, and ensuring a common understanding of the software's functionality.



You:

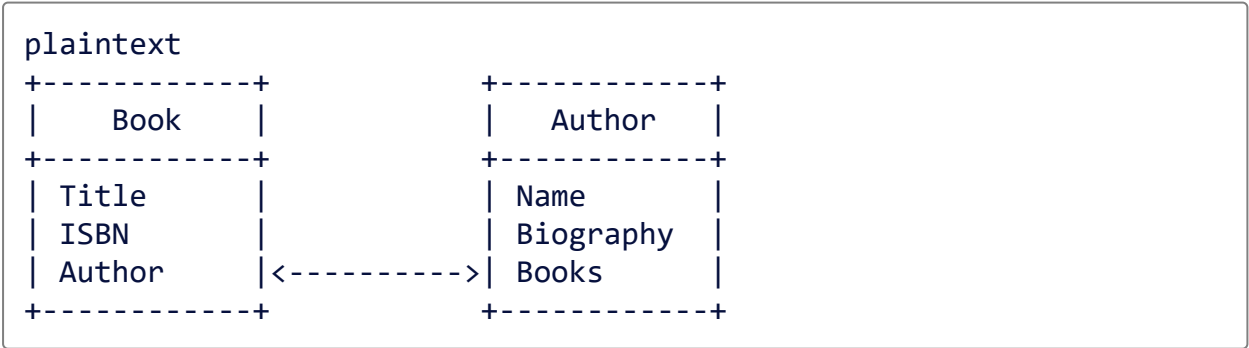
User Explain association and dependencies with an example.in software engineering



In software engineering, association and dependencies are concepts used to describe the relationships between different elements or components in a system. These concepts are often represented in various diagrams, such as class diagrams in the Unified Modeling Language (UML). Let's explore each concept with examples:

1. **Association:**

- *Definition:* Association represents a relationship between two or more classes, indicating that objects of one class are somehow connected to objects of another class.
- *Example:* Consider a library management system. There may be two classes: Book and Author. An association between these classes signifies that a book is associated with an author, and an author can be associated with multiple books. This association is typically represented by a line connecting the two classes, and it may include multiplicity annotations to indicate the cardinality of the relationship (e.g., one-to-one, one-to-many, many-to-many).



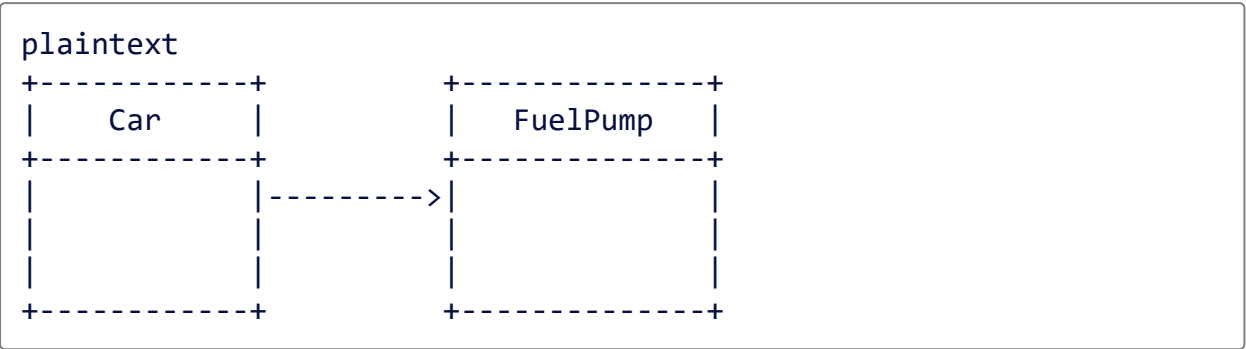
In this example, the association between the Book and Author classes implies that each book has an association with an author, and an author can be associated with multiple books.

2. **Dependencies:**

- *Definition:* Dependency represents a relationship where a change in one element (e.g., class, module, method) may affect another element. Dependencies are categorized into

various types, such as direct dependencies or indirect dependencies.

- *Example:* Consider a scenario where a Car class depends on a FuelPump class. The Car class relies on the FuelPump class to provide the necessary fuel for the car to function.



In this example, there is a dependency between the Car and FuelPump classes. If there are changes to the FuelPump class (e.g., changes in its interface or implementation), it may affect the Car class, and the dependency relationship is illustrated by an arrow pointing from the dependent class (Car) to the class it depends on (FuelPump).

Understanding associations and dependencies is crucial for designing modular and maintainable software systems. Associations define how different components collaborate, while dependencies highlight the relationships that impact the behavior of one component based on changes in another. Both concepts contribute to effective software design, encapsulation, and overall system understanding.



Discuss concepts of data modeling.



Data modeling is a fundamental step in the software development process that involves creating a representation of the data and its relationships within a system. It helps in defining the structure of the database, ensuring data integrity, and providing a blueprint for database design. Several key concepts are integral to the process of data modeling:

1. Entity:

- *Definition:* An entity is a real-world object or concept that has data to be stored in the database. Entities are represented as tables in a relational database, and each row in the table corresponds to an instance of that entity.
- *Example:* In a university database, entities might include Student, Course, and Instructor.

2. Attribute:

- *Definition:* Attributes are the properties or characteristics of an entity. They describe the data that can be stored for each instance of an entity.

- *Example:* For the Student entity, attributes could include StudentID, FirstName, LastName, and Birthdate.

3. Relationship:

- *Definition:* A relationship represents an association between two or more entities. Relationships define how entities are connected to each other and describe the nature of the connections.
- *Example:* In a library database, there might be a relationship between the Book and Author entities, where an author can be associated with multiple books, and a book can have one or more authors.

4. Primary Key:

- *Definition:* A primary key is a unique identifier for a record within a table. It ensures that each row in the table can be uniquely identified.
- *Example:* In a Student table, the StudentID might serve as the primary key.

5. Foreign Key:

- *Definition:* A foreign key is a field in a table that refers to the primary key of another table. It establishes a link between the two tables, enforcing referential integrity.
- *Example:* In a Course table, a foreign key might link to the Instructor table through the InstructorID field.

6. Normalization:

- *Definition:* Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves breaking down large tables into smaller, related tables and using relationships between them.
- *Example:* In a denormalized database, information about a student and their courses might be duplicated across multiple records. Through normalization, this information would be organized into separate tables.

7. Cardinality:

- *Definition:* Cardinality defines the numerical relationship between two entities in a relationship. It describes how many instances of one entity are associated with how many instances of another entity.
- *Example:* In a one-to-many relationship between Department and Employee, one department may have many employees, but each employee belongs to only one department.

8. ERD (Entity-Relationship Diagram):

- *Definition:* An ERD is a graphical representation of entities, their attributes, and the relationships between them. It is a visual tool used to model the structure of a database.
- *Example:* An ERD might include entities such as Customer, Order, and Product, along with their attributes and relationships.

Data modeling is an essential part of database design, providing a clear and structured way to represent and organize data within a system. It helps ensure data accuracy, consistency, and efficient retrieval.



The dictionary defines the word principle as “an important underlying law or assumption required in a system of thought.” Throughout this book I’ll discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., communication), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason. David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:

The First Principle: The Reason It All Exists A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: “Does this add real value to the system?” If the answer is “no,” don’t do it. All other principles support this one.

The Second Principle: KISS (Keep It Simple, Stupid!) Software design is not a haphazard process. There are many factors to consider in any design effort. All design should be as simple as possible, but no simpler. This facilitates having a more easily understood and easily maintained system. This is

CHAPTER 1 SOFTWARE AND SOFTWARE ENGINEERING 19 13 Reproduced with permission of the author [Hoo96]. Hooker defines patterns for these principles at <http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment>. Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.

pre75977_ch01.qxd 11/27/08 3:11 PM Page 19 not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean “quick and dirty.” In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: Maintain the Vision A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: What You Produce, Others Will Consume Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, always specify, design, and implement knowing someone else will have to understand what you are doing. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: Be Open to the Future A system with a long lifetime has more value. In today’s computing environments, where specifications change on a moment’s notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true “industrial-strength” software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. Never design yourself into a corner. Always ask “what if,” and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.

14 This could very possibly lead to the reuse of an entire system. 20 **CHAPTER 1 SOFTWARE AND SOFTWARE ENGINEERING** If software has value, it will change over its useful life. For that reason, software must be built to be maintainable. 14 This advice can be dangerous if it is taken to extremes. Designing for the “general problem” sometimes requires performance compromises and can make specific solutions inefficient. Note: “There is a certain majesty in simplicity which is far above all the quaintness of wit.” Alexander Pope (1688–1744) pre75977_ch01.qxd 11/27/08 3:11 PM Page 20

The Sixth Principle: Plan Ahead for Reuse Reuse saves time and effort. 15 Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are

many techniques to realize reuse at every level of the system development process. . . . Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated. The Seventh principle: Think! This last principle is probably the most overlooked. Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous. If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated. NOW BY ABOVE DETALIS ANSWER THE QUESTION :Enlist and explain any three principles that guide the software engineering practice. in short



ChatGPT:

Certainly! Here are three principles that guide software engineering practice, based on the provided excerpt:

1. The First Principle: The Reason It All Exists

- *Summary:* A software system exists to provide value to its users. All decisions in software engineering should be made with this central goal in mind. Before specifying requirements or determining functionality, one should ask if it adds real value to the system.
- *Implication:* This principle emphasizes the importance of aligning software development efforts with the end goal of delivering value to users.

2. The Second Principle: KISS (Keep It Simple, Stupid!)

- *Summary:* Software design should be as simple as possible while still meeting requirements. Simplicity facilitates better understanding and maintenance. It doesn't mean sacrificing features but encourages elegant designs through thoughtful iterations.
- *Implication:* Prioritizing simplicity leads to more maintainable and less error-prone software, fostering long-term sustainability.

3. The Third Principle: Maintain the Vision

- *Summary:* A clear vision is essential for project success. Without conceptual integrity, a system risks becoming a patchwork of incompatible designs. Compromising the architectural vision weakens the system. An empowered architect is crucial for enforcing compliance and ensuring success.
- *Implication:* This principle underscores the significance of maintaining a unified vision throughout the software development process to prevent inconsistencies and ensure the success of the project.

These principles, as outlined by David Hooker, provide valuable guidance for software engineers in making decisions that contribute to the creation of effective, valuable, and maintainable software systems.



Discuss requirement analysis with its importance.



Requirement Analysis:

Requirement analysis is a crucial phase in the software development life cycle (SDLC) that involves gathering, documenting, and understanding the needs and expectations of stakeholders for a software system. It is the foundation upon which the entire software development process is built. The goal of requirement analysis is to create a comprehensive and unambiguous specification that serves as a blueprint for designing and implementing the software.

Key Steps in Requirement Analysis:

1. **Elicitation:** Gathering information from stakeholders, including users, customers, and other relevant parties, to understand their needs and expectations.
2. **Documentation:** Capturing and documenting requirements in a clear and unambiguous manner. This may involve creating use cases, user stories, functional and non-functional requirements, and other artifacts.
3. **Analysis:** Examining and refining gathered requirements to ensure clarity, completeness, and consistency. Analyzing potential conflicts or contradictions and resolving them.
4. **Validation:** Ensuring that the documented requirements accurately represent the stakeholders' needs and that they are feasible and aligned with the project's goals.
5. **Management:** Managing changes to requirements throughout the development process and maintaining traceability between requirements and other project artifacts.

Importance of Requirement Analysis:

1. Understanding Stakeholder Needs:

- *Significance:* Requirement analysis helps in identifying and understanding the needs of stakeholders, including end-users, customers, and other parties involved.
- *Impact:* This understanding forms the basis for developing a system that meets the expectations of the intended users and stakeholders.

2. Scope Definition:

- *Significance:* Requirement analysis helps in defining the scope of the software project.
- *Impact:* A well-defined scope ensures that the development team focuses on delivering the functionalities and features that are essential for the success of the project, avoiding scope creep.

3. Risk Management:

- *Significance:* Identifying potential risks and challenges early in the process.
- *Impact:* Early recognition allows for proactive risk mitigation strategies, reducing the likelihood of project delays or failures.

4. Basis for Design and Implementation:

- *Significance*: Providing the foundation for designing and implementing the software system.
- *Impact*: A thorough understanding of requirements guides developers in creating a system that aligns with user expectations and project goals.

5. Cost and Time Estimation:

- *Significance*: Helping in estimating the resources, time, and costs required for the project.
- *Impact*: Accurate estimations contribute to effective project planning and resource allocation, preventing budget overruns and project delays.

6. Quality Assurance:

- *Significance*: Forming the basis for testing and quality assurance activities.
- *Impact*: A well-defined set of requirements enables the creation of test cases and ensures that the final product meets the specified criteria for quality and functionality.

7. Customer Satisfaction:

- *Significance*: Ensuring that the delivered software meets customer expectations.
- *Impact*: Customer satisfaction is critical for the success of any software project. Requirement analysis helps in aligning the development process with customer needs, improving overall satisfaction.

In summary, requirement analysis is a fundamental and foundational activity in software engineering. Its thorough execution significantly contributes to the success of a software project by ensuring a clear understanding of stakeholder needs, effective project planning, risk mitigation, and the delivery of a high-quality, customer-satisfying product.



Elaboration. The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model (Chapters 6 and 7) that identifies various aspects of software function, behavior, and information. Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced. Negotiation. It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs." You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss 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 measure of satisfaction. NOW BY ABOVE DETAILS ANSWER THE QUESTION :Explain elaboration and negotiation in terms of requirement gathering taking an appropriate example.in short



Elaboration in Requirement Gathering:

- **Definition:** Elaboration is the process of expanding and refining the information obtained during inception and elicitation to develop a detailed requirements model. It involves identifying various aspects of software function, behavior, and information.
- **Example:** In the development of a banking software system, during elaboration, user scenarios could be created to describe how customers interact with the system (e.g., making transactions, checking balances). These scenarios lead to the identification of analysis classes (e.g., Account, Transaction), defining their attributes and services.

Negotiation in Requirement Gathering:

- **Definition:** Negotiation is the process of resolving conflicts and managing competing demands from different stakeholders. It involves prioritizing requirements, assessing their cost and risk, and addressing conflicts to achieve a balance that satisfies all parties.
- **Example:** In the same banking software project, stakeholders may have conflicting requirements, such as the marketing team wanting more user-friendly features and the security team prioritizing robust authentication. Through negotiation, stakeholders rank requirements, discuss conflicts, and find compromises to balance user-friendliness and security without compromising the overall success of the project.



The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user. In modern software engineering work, coding may be (1) the direct creation of programming language source code (e.g., Java), (2) the automatic generation of source code using an intermediate design-like representation of the component to be built, or (3) the automatic generation of executable code using a “fourth-generation programming language” (e.g., Visual C). The initial focus of testing is at the component level, often called unit testing. Other levels of testing include (1) integration testing (conducted as the system is constructed), validation testing that assesses whether requirements have been met for the complete system (or software increment), and (3) acceptance testing that is conducted by the customer in an effort to exercise all required features and functions. The following set of fundamental principles and concepts are applicable to coding and testing: Coding Principles. The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated: Preparation principles: Before you write one line of code, be sure you • Understand of the problem you’re trying to solve. • Understand basic design principles and concepts. • Pick a programming language that meets the needs of the software to be built and the environment in which it will operate. • Select a programming environment that provides tools that will make your work easier. • Create a set of unit tests that will be applied once the component you code is completed. Programming principles: As you begin writing code, be sure you • Constrain your algorithms by following structured programming [Boh00] practice. • Consider the use of pair programming. • Select data structures that will meet the needs of the design. • Understand the software architecture and create interfaces that are consistent with it. • Keep conditional logic as simple as possible. • Create nested loops in a way that makes them easily testable. • Select meaningful variable names and follow other local coding standards. CHAPTER 4 PRINCIPLES THAT GUIDE PRACTICE 111 uote: “For much of my life, I have been a software voyeur, peeking furtively at other people’s dirty code. Occasionally, I find a real jewel, a well-structured program written in a consistent style, free of kludges, developed so that each component is simple and

organized, and designed so that the product is easy to change.” David Parnas Avoid developing an elegant program that solves the wrong problem. Pay particular attention to the first preparation principle. pre75977_ch04.qxd 11/27/08 3:27 PM Page 111 • Write code that is self-documenting. • Create a visual layout (e.g., indentation and blank lines) that aids understanding. Validation Principles: After you’ve completed your first coding pass, be sure you • Conduct a code walkthrough when appropriate. • Perform unit tests and correct errors you’ve uncovered. • Refactor the code. More books have been written about programming (coding) and the principles and concepts that guide it than about any other topic in the software process. Books on the subject include early works on programming style [Ker78], practical software construction [McC04], programming pearls [Ben99], the art of programming [Knu98], pragmatic programming issues [Hun99], and many, many other subjects. A comprehensive discussion of these principles and concepts is beyond the scope of this book. If you have further interest, examine one or more of the references noted. Testing Principles. In a classic book on software testing, Glen Myers [Mye79] states a number of rules that can serve well as testing objectives: • Testing is a process of executing a program with the intent of finding an error. • A good test case is one that has a high probability of finding an as-yet- undiscovered error. • A successful test is one that uncovers an as-yet-undiscovered error. These objectives imply a dramatic change in viewpoint for some software develop- ers. They move counter to the commonly held view that a successful test is one in which no errors are found. Your objective is to design tests that systematically un- cover different classes of errors and to do so with a minimum amount of time and effort. If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, and that behavioral and performance requirements appear to have been met. In addition, the data collected as testing is conducted provide a good indication of software reliabil- ity and some indication of software quality as a whole. But testing cannot show the absence of errors and defects; it can show only that software errors and defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted. 112 PART TWO MODELING WebRef A wide variety of links to coding standards can be found at www.literateprogramming.com/fpstyle.html. What are the objectives of software testing? ? In a broader software design context, recall that you begin “in the large” by focusing on software architecture and end “in the small” focusing on compo- nents. For testing, you simply reverse the focus and test your way out. pre75977_ch04.qxd 11/27/08 3:27 PM Page 112 Davis [Dav95b] suggests a set of testing principles⁶ that have been adapted for use in this book: Principle 1. All tests should be traceable to customer requirements.⁷ The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer’s point of view) are those that cause the program to fail to meet its requirements. Principle 2. Tests should be planned long before testing begins. Test planning (Chapter 17) can begin as soon as the requirements model is com- plete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated. Principle 3. The Pareto principle applies to software testing. In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program compo- nents. The problem, of course, is to isolate these suspect components and to thoroughly test them. Principle 4. Testing should begin “in the small” and progress toward testing “in the large.” The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system. Principle 5. Exhaustive testing is not possible. The number of path per- mutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to en- sure that all conditions in the component-level design have been exercised. NOW BY ABOVE DETALIS ANSWER THE QUESTION :Explain and describe construction principles.in short



Construction Principles:

1. Preparation Principles:

- *Explanation:* Before writing code, ensure a clear understanding of the problem, grasp basic design principles, choose a suitable programming language and environment, and create a set of unit tests.
- *Example:* Before developing an online shopping system, understand user requirements, design the system flow, choose a web development framework, and prepare unit tests for core functionalities like product selection and checkout.

2. Programming Principles:

- *Explanation:* Follow structured programming practices, consider pair programming, select appropriate data structures, align code with the software architecture, keep conditional logic simple, and use meaningful variable names.
- *Example:* In coding for a financial application, follow structured programming to ensure clarity, use pair programming for critical modules, and choose data structures that efficiently handle financial data.

3. Validation Principles:

- *Explanation:* After initial coding, conduct code walkthroughs, perform unit tests, correct errors, and refactor code for improved clarity and maintainability.
- *Example:* After implementing an authentication module in a security software project, conduct a code walkthrough to identify potential vulnerabilities, perform unit tests to ensure correct functionality, and refactor the code for improved security.

Testing Principles:

1. Traceability to Customer Requirements:

- *Explanation:* Ensure that all tests are traceable to customer requirements, aiming to uncover errors that may impact the software's ability to meet customer needs.
- *Example:* In testing an e-commerce platform, ensure that tests cover all customer requirements, such as secure payment processing, efficient product search, and order tracking.

2. Early Test Planning:

- *Explanation:* Plan tests long before testing begins, starting as soon as the requirements model is complete and refining as the design model solidifies.
- *Example:* In a healthcare software project, plan tests for data privacy and accuracy as soon as requirements specify patient information handling, refining the test plan during the subsequent design phases.

3. Pareto Principle in Testing:

- *Explanation:* Recognize that a significant portion of errors may originate from a small percentage of program components, emphasizing the need to isolate and thoroughly test these components.
- *Example:* Identify critical components in a communication protocol for a networking application, focusing testing efforts on these components to uncover potential vulnerabilities.

4. Testing "In the Small" to "In the Large":

- *Explanation:* Start testing at the component level, then progress to integrated clusters of components, and finally test the entire system to uncover errors at different levels of granularity.

- *Example:* In testing a social media platform, start by testing individual features (e.g., posting, commenting), then move to integrated testing of feature combinations, and eventually test the entire platform for overall performance and user experience.

5. Exhaustive Testing Is Not Possible:

- *Explanation:* Acknowledge the impracticality of testing every possible path permutation, focusing instead on adequately covering program logic and ensuring all conditions in the component-level design are exercised.
- *Example:* In testing a video streaming application, prioritize testing scenarios that cover various user interactions and system responses, acknowledging that it's impossible to test every possible combination of user actions and network conditions.

These principles guide the coding and testing activities in software construction, emphasizing preparation, disciplined programming practices, thorough validation, and effective testing strategies.



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

6.5.1 Identifying Analysis Classes

If you look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when you “look around” the problem space of a software application, the classes (and objects) may be more difficult to comprehend. We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a “grammatical parse” [Abb83] on the use cases developed for the system to NOW BY ABOVE DETALIS ANSWER THE QUESTION :Explain class based modeling with an appropriate example.in short



Class-Based Modeling:

Explanation: Class-based modeling involves representing the structure and behavior of a system using classes, objects, attributes, operations (methods), relationships, collaborations, and other related elements. It provides a way to organize and understand the components of a system and their interactions.

Key Elements of Class-Based Modeling:

1. Classes and Objects:

- *Definition:* Classes represent a blueprint for creating objects, which are instances of those classes.
- *Example:* In a library system, there could be a Book class with attributes like title and author. Each physical book in the library would be an object instantiated from the Book

class.

2. Attributes:

- *Definition:* Attributes are properties or characteristics of a class that describe its state.
- *Example:* The Book class may have attributes such as Title, Author, and ISBN.

3. Operations (Methods):

- *Definition:* Operations define the behavior or actions that can be performed on objects.
- *Example:* The Book class may have operations like CheckOut() and Return().

4. Class-Responsibility-Collaborator (CRC) Models:

- *Definition:* CRC models help identify classes, their responsibilities, and collaborations between them.
- *Example:* In a CRC model, a Library class may have responsibilities like managing inventory, and collaborations with the Book class for checking in/out.

5. Collaboration Diagrams:

- *Definition:* Collaboration diagrams illustrate the interactions and collaborations between objects.
- *Example:* A collaboration diagram might show how a Member object interacts with a Book object to borrow or return books in the library system.

6. Packages:

- *Definition:* Packages are used to organize and group related classes in a modular way.
- *Example:* In a software system for a university, there could be packages like Student Management and Course Management containing relevant classes.

Example: Consider a simple banking system:

• Classes and Objects:

- Account class with objects representing individual bank accounts.
- Customer class with objects representing bank customers.

• Attributes:

- Account class attributes: AccountNumber, Balance.
- Customer class attributes: CustomerID, Name.

• Operations (Methods):

- Account class operations: Deposit(), Withdraw().
- Customer class operations: OpenAccount(), CloseAccount().

• CRC Model:

- Account class responsibilities: Manage account balance, perform transactions.
- Collaborations: Interact with Customer class for account ownership.

• Collaboration Diagram:

- Illustrates how a Customer object collaborates with an Account object to deposit or withdraw money.

- **Packages:**

- Banking System package containing classes like Account and Customer.

This example demonstrates how class-based modeling helps organize and represent components of a system, fostering a clear understanding of their relationships and interactions.



Specification. In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these. Some suggest that a “standard template” [Som97] should be developed and used for a specification, arguing that this leads to requirements that are presented in a way that establishes a firm base for design. If you work beyond that point, you’re doing design. Elaboration is a good thing, but you have to know when to stop. The key is to describe the problem in a way that establishes a firm base for design. If you work beyond that point, you’re doing design. 4 A service manipulates the data encapsulated by the class. The terms operation and method are also used. If you are unfamiliar with object-oriented concepts, a basic introduction is presented in Appendix 2. There should be no winner and no loser in an effective negotiation. Both sides win, because a “deal” that both can live with is solidified. pre75977_ch05.qxd 11/27/08 3:30 PM Page 122 The formality and format of a specification varies with the size and the complexity of the software to be built. consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments. Validation. The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification5 to ensure that all software requirements have been 5 Recall that the nature of the specification will vary with each project. In some cases, the “specification” is a collection of user scenarios and little else. In others, the specification may be a document that contains scenarios, models, and written descriptions. pre75977_ch05.qxd 11/27/08 3:30 PM Page 123 stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product. The primary requirements validation mechanism is the technical review (Chapter 15). The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements. NOW BY ABOVE DETALIS ANSWER THE QUESTION :Discuss specification and validation of requirements with an appropriate example.in short



Specification and Validation of Requirements:

Specification:

- **Definition:** In the context of computer-based systems and software, a specification refers to a document, graphical models, formal mathematical models, usage scenarios, a prototype, or a combination of these elements.

- **Flexibility:** The formality and format of a specification can vary based on the size and complexity of the software project. It can be a written document with natural language descriptions and graphical models for large systems, or it might consist mainly of usage scenarios for smaller, well-understood projects.

Example: Consider the development of a mobile banking application:

- **Written Document Specification:**
 - For a large-scale banking system, the specification might include detailed written requirements, data flow diagrams, and UML class diagrams.
- **Graphical Models:**
 - The specification could use UML diagrams to illustrate the system's architecture, showing the relationships between components and their interactions.
- **Usage Scenarios:**
 - Additionally, the specification might include specific usage scenarios, describing how a user logs in, transfers money, and checks account balance.

Validation of Requirements:

- **Definition:** Validation is the process of assessing the quality of work products produced during requirements engineering to ensure that requirements are unambiguous, consistent, complete, and conform to established standards.
- **Primary Mechanism: Technical Review:**
 - Technical reviews, involving software engineers, customers, users, and stakeholders, are the primary mechanism for requirements validation.

Example: In the mobile banking application project:

- **Technical Review Team:**
 - A review team consisting of software engineers, bank representatives, and end-users examines the specification for potential issues.
- **Identification of Errors:**
 - During the review, the team identifies inconsistencies, omissions, errors, or conflicting requirements in the specification.
- **Clarifications and Corrections:**
 - Discussions within the review team lead to clarifications and corrections to address identified issues.

This process ensures that the mobile banking application's requirements are well-defined, meet user needs, and align with the project's objectives. The specification undergoes validation to catch errors early in the development process, saving time and resources in the long run.
