Second Semester MCA Degree Examination, June/July 2023
Software engineering

Q1-a. Describe software engineering code of ethics and professional practices as described by IEEE/ACM

Ans. The IEEE (Institute of Electrical and Electronics Engineers) and ACM (Association for Computing Machinery) have jointly established a code of ethics and professional practices for software engineers. This code provides guidelines for ethical behavior and professional conduct in the field of software engineering. While the specific details may evolve over time, the general principles include:

1. **PUBLIC:** Software engineers shall act consistently with the public interest. They should consider the safety, health, and welfare of the public and disclose any factors that could endanger it.
2. **CLIENT AND EMPLOYER:** Software engineers shall act in a manner that is in the best interests of their clients and employers and shall ensure that the products of their work meet the highest professional standards.
3. **PRODUCT:** Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. This includes being diligent in the design, testing, and maintenance of software.
4. **JUDGMENT:** Software engineers shall maintain integrity and independence in their professional judgment. They should provide honest assessments of software and system implications, ensuring that decisions are made based on objective analysis.
5. **MANAGEMENT:** Software engineers shall subscribe to fair management practices and not promote any action that is known to be unethical or illegal. They should work to improve their management skills and promote an ethical approach to the management of software development and maintenance.
6. **PROFESSION:** Software engineers shall advance the integrity and reputation of the profession consistent with the public interest. They should foster professional development, mentor colleagues, and contribute to the community to enhance the understanding and appreciation of the field.
7. **COLLEAGUES:** Software engineers shall be fair to and supportive of their colleagues. They should avoid malicious or subversive behavior and strive to foster a positive and collaborative working environment.
8. **SELF:** Software engineers shall participate in lifelong learning and consistently enhance their professional skills. They should promote ethical approaches to the practice of software engineering and maintain high standards of professional conduct.

Adherence to this code of ethics is crucial for maintaining the trust of clients, employers, and the public, and for ensuring the responsible and ethical development of software and technology.

1-b. Why Software engineering is important? List the reasons. Brief the attributes of good software.

The importance of software engineering lies in the fact that a specific piece of Software is required in almost every industry, every business, and purpose. As time goes on, it becomes more important for the following reasons.

1. Reduces Complexity

Dealing with big Software is very complicated and challenging. Thus, to reduce the complications of projects, software engineering simplifies complex problems and solves those issues one by one.

2. Handling Big Projects

Big projects need lots of patience, planning, and management, which you never get from any company. It is only possible if the company uses software engineering to deal with big projects without problems.

3. To Minimize Software Costs

Software engineers are paid highly as Software needs a lot of hard work and workforce development. These are developed with the help of a large number of codes. But programmers in software engineering project all things and reduce the things which are not needed. As a result of the production of Software, costs become less and more affordable for Software that does not use this method.

4. To Decrease Time

If things are not made according to the procedures, it becomes a huge loss of time. Accordingly, complex Software must run much code to get definitive running code. So, it takes lots of time if not handled properly. And if you follow the prescribed software engineering methods, it will save your precious time by decreasing it.

5. Effectiveness

Making standards decides the effectiveness of things. Therefore, a company always targets the software standard to make it more effective. And Software becomes more effective only with the help of software engineering.

6. Reliable Software

The Software will be reliable if software engineering, testing, and maintenance are given. As a software developer, you must ensure that the Software is secure and will work for the period or subscription you have agreed upon.

The essential attributes of good software are:

Acceptability - Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.

Dependability and security - Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Software has to be secure so that malicious users cannot access or damage the system.

Efficiency - Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, resource utilization, etc.

Maintainability - Software should be written in such a way that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.

OR

2.a Describe the waterfall and incremental software process models with suitable diagrams.

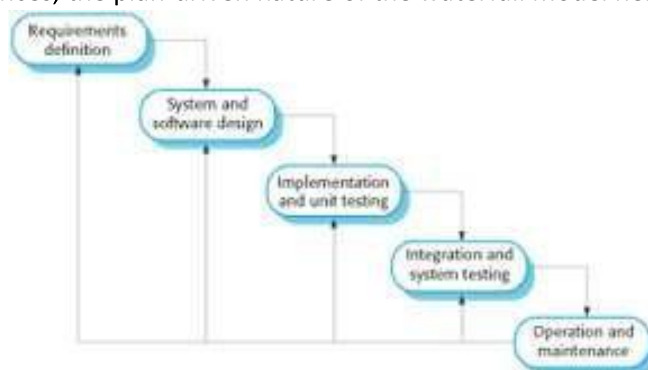
Ans. Waterfall Model:

- There are separate identified phases in the waterfall model:
    - Requirements analysis and definition
    - System and software design
    - Implementation and unit testing
    - Integration and system testing
    - Operation and maintenance

The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.
Problems:

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
    - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
    - Few business systems have stable requirements.

The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.



Incremental Development Model:

- The cost of accommodating changing customer requirements is reduced.
    - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.

It is easier to get customer feedback on the development work that has been done.

- Customers can comment on demonstrations of the software and see how much has been implemented.

More rapid delivery and deployment of useful software to the customer is possible.

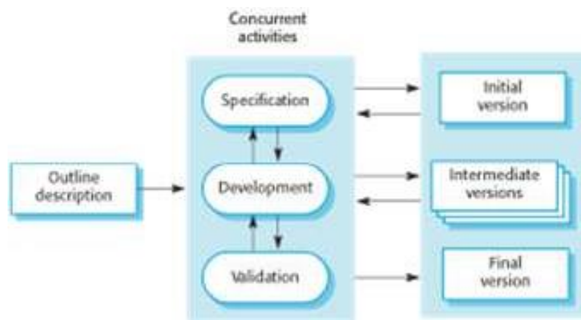
- Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Problems:

- The process is not visible.
    - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.

System structure tends to degrade as new increments are added.

- Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.



- Changes are usually incorporated in documents without following any standard procedure. Thus, verification of all such changes often becomes difficult.
- The development of high-quality and reliable software requires the software to be thoroughly tested. Though thorough testing of software consumes the majority of resources, underestimating it because of any reasons deteriorates the software quality.

2.b. Describe the principles of Agile Methods.

Ans. Agile is an iterative and incremental approach to software development that emphasizes flexibility, collaboration, and customer satisfaction. Here are the twelve Agile principles:

1. **Customer Satisfaction through Early and Continuous Delivery of Valuable Software:**
   - Deliver working software frequently, with a preference for shorter timescales, to provide tangible value to the customer.
2. **Welcome Changing Requirements, Even Late in Development:**
   - Embrace changes in requirements, even if they occur late in the development process, to provide the customer with a competitive advantage.
3. **Deliver Working Software Frequently:**
   - Aim to deliver a working product as frequently as possible, with a preference for shorter development cycles.
4. **Collaboration between Business Stakeholders and Developers:**
   - Foster a collaborative environment where business stakeholders and developers work together daily throughout the project.
5. **Build Projects around Motivated Individuals:**
   - Give motivated individuals the resources and support they need and trust them to get the job done.
6. **Face-to-Face Communication is Most Effective:**
   - Maximize face-to-face communication within the team as it is the most efficient and effective method of conveying information.
7. **Working Software is the Primary Measure of Progress:**
   - Focus on delivering a working product, as it is the ultimate measure of progress in Agile development.
8. **Maintain a Sustainable Pace of Work:**
   - Encourage a sustainable pace of work to maintain a consistent level of productivity and prevent burnout.
9. **Continuous Attention to Technical Excellence and Good Design:**
   - Prioritize technical excellence and good design to ensure the long-term maintainability and adaptability of the software.
10. **Simplicity—the Art of Maximizing the Amount of Work Not Done:**

- Emphasize simplicity in design and implementation, maximizing the value of work accomplished and minimizing unnecessary complexity.

11. **Self-Organizing Teams:**
    - Allow the team to self-organize, encouraging collaboration and creativity while recognizing the expertise of individual team members.
12. **Regular Reflection and Adaptation:**
    - Regularly reflect on the team's performance and adapt processes accordingly, fostering a culture of continuous improvement.

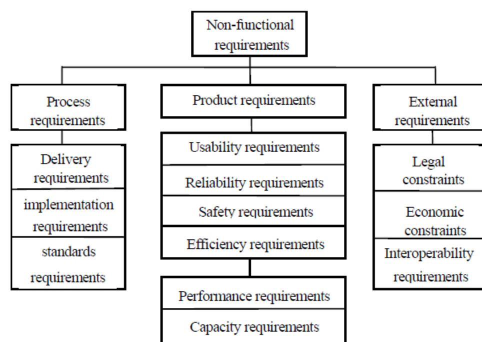
2.c. Explain the extreme programming release cycle.
Ans. Extreme Programming (XP) is an Agile software development methodology that emphasizes flexibility, adaptability, and continuous improvement. The XP release cycle is characterized by its iterative and incremental approach to software development. The process involves a series of short, time-boxed iterations, with each iteration resulting in a potentially shippable product increment. Here's an overview of the XP release cycle:

1. **Exploration:**
    - The development process begins with an initial exploration phase where the project team collaborates with the customer to understand and prioritize the project requirements.
    - User stories, which represent features or functionalities from the user's perspective, are created and prioritized based on customer input.
2. **Planning Game:**
    - During the planning game, the development team and the customer work together to plan the upcoming iteration.
    - The customer defines user stories and sets priorities, while the development team estimates the effort required for each story.
    - Together, they agree on the scope of the iteration, making decisions about what can be accomplished in the upcoming cycle.
3. **Iteration:**
    - The core of the XP release cycle is the iteration, also known as a time-boxed development cycle. Iterations typically last one to three weeks.
    - The development team selects user stories from the prioritized backlog and commits to delivering them by the end of the iteration.
    - The team designs, codes, tests, and integrates the selected user stories to produce a potentially shippable product increment.
4. **Stand-up Meetings:**
    - Daily stand-up meetings are conducted to facilitate communication within the development team. Each team member shares progress, obstacles, and plans for the day.
5. **Continuous Integration:**
    - Throughout the iteration, developers practice continuous integration, merging their code frequently to ensure that the entire codebase is always in a working state.
    - Automated tests are run to verify that new changes do not introduce defects.
6. **Testing:**
    - Testing is an integral part of the XP release cycle. Developers write automated unit tests for their code, and additional testing is performed during the iteration to ensure the quality of the product increment.
7. **Acceptance Testing:**
    - At the end of each iteration, the customer reviews the completed user stories and provides feedback.

- Acceptance testing is performed to validate that the delivered functionality meets the customer's expectations.

8. **Release:**
    - At the end of each iteration, the software is potentially shippable, meaning it could be released to users if the customer decides to do so.
    - The customer has the option to release the product or continue refining and adding features in subsequent iterations.
9. **Feedback and Adaptation:**
    - After each iteration, the team reflects on the process and gathers feedback from the customer.
    - Lessons learned from the iteration are used to adapt and improve the development process in the next iteration.

3. a. Explain the classification of non-functional requirements with neat sketch and example.
Ans.



1. **Product Requirements:**
    - These requirements specify characteristics that the product must have but are not related to any specific function. Examples include:
        - Reliability: The system's ability to perform without failure over a specified period.
        - Performance: Specifies how the system should respond under certain conditions.
        - Security: Requirements related to data protection, access control, and other security aspects.
2. **Organizational Requirements:**
    - These requirements are related to the organization's policies and procedures. Examples include:
        - Software Quality: Specifies the level of quality that must be maintained in the software development process.
        - Compliance: Requirements related to legal and regulatory standards that the system must adhere to.
3. **External Requirements:**
    - These requirements arise from factors external to the system and its development process. Examples include:
        - Interoperability: The ability of the system to operate with other external systems.
        - Ethical: Requirements related to ethical considerations, such as avoiding biased decision-making in algorithms.
4. **Project Requirements:**
    - These requirements are constraints imposed by the project environment and the system's stakeholders. Examples include:

        - Schedule: Specifies the timeframe within which the system must be developed and delivered.
        - Budget: Constraints related to the financial aspects of the project.
5. **Interface Requirements:**
    - These requirements specify how the system interfaces with other software components or hardware. Examples include:
        - User Interfaces: Specifications for how the user interacts with the system.
        - Communication Protocols: Requirements related to data exchange between different parts of the system.
6. **Implementation Requirements:**
    - These requirements are related to the implementation of the system. Examples include:
        - Language and Tools: Specifies the programming language and development tools to be used.
        - Platform Compatibility: Requirements related to the hardware and software platforms on which the system will run.

3.b. Explain the notations used in writing Software Requirement specification.
Natural Language:
Description in Plain Text: Many requirements are written in natural language to describe functionalities, constraints, and user expectations. However, it's important to ensure clarity and avoid ambiguity.
Structured English:
Structured Text: Use of structured English, which is a form of pseudocode, to describe algorithms and processes in a more formalized and readable way. It helps in clarifying the logic behind specific requirements.
Mathematical Notations:
Mathematical Expressions: In cases where precision is crucial, mathematical notations may be used to describe certain aspects of requirements, especially in areas like algorithms or calculations.
Tabular Representations:
Tables and Matrices: Tabular formats are often used to present complex information in a structured way, such as requirements traceability matrices linking requirements to test cases.
Design Description Language:
This approach uses a language like programming language but with more abstract features to specify the requirements by defining an abstract model of the system.

OR

4. a Discuss the various difficulties that the software engineer faces during eliciting and understanding requirements.

1. Ans. **Stakeholder Communication:**
   Projects often involve numerous stakeholders with varying backgrounds, roles, and expectations. Effectively communicating and reconciling different viewpoints can be challenging.
2. **Requirements Volatility:**
   Changing Requirements: Requirements are subject to change, either due to evolving user needs or a better understanding of the system. Handling these changes while maintaining project stability can be challenging.
3. **Incomplete Requirements:**
   Ambiguity: Ambiguous or vague requirements can lead to misunderstandings and misinterpretations among team members.
4. **Conflicting Requirements:**

Different stakeholders may have conflicting needs or priorities. Resolving such conflicts requires negotiation and compromise. Requirements may conflict with technical constraints or limitations, requiring careful trade-offs.

5. **Requirements Traceability:**
   Keeping track of the relationships between requirements, design, and testing can be challenging. Changes in one area may have ripple effects, and ensuring traceability becomes crucial.
6. **Changing Technology:**
   In dynamic technological environments, requirements may become outdated quickly. Keeping up with technological changes and adapting requirements accordingly can be demanding.
7. **Size and Complexity of the System:**
   Larger systems often involve a higher number of requirements, making it challenging to manage and understand the entire scope.
8. **User Involvement:**
   Lack of active involvement from end-users can lead to misunderstandings and result in solutions that do not align with user needs.
9. **Tool Support:**
   Insufficient or ineffective tools for requirements management and documentation can hinder the elicitation and understanding process.
10. **Legal and Ethical Considerations:**
    Complying with legal and ethical standards may introduce additional complexities, especially in industries with strict regulations.

4.b. Discuss the important activities of requirement engineering Process with a neat diagram.

Ans. The requirements engineering process has many potential pitfalls. Strengthening your RE process enables you to avoid many common challenges. The process serves as a compass, guiding you in determining the exact deliverables and doing so with greater accuracy. All important parties are on the same page about what steps need to be taken to achieve the desired outcome. Strengthen your RE process by considering the following:

Elicit requirements. Elicitation is about becoming familiar with all the important details involved with the project. The customer will provide details about their needs and furnish critical background information. You will study those details and also become familiar with similar types of software solutions. This step provides important context for development.

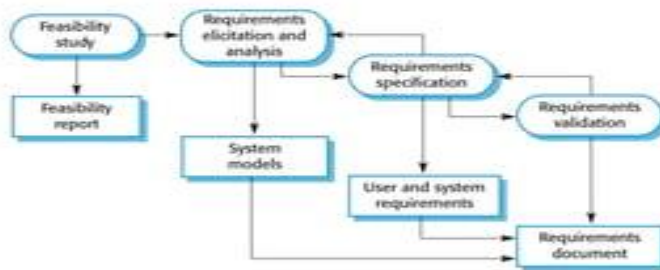
Requirements specification. During the specification phase, you gather functional and nonfunctional project requirements. A variety of tools are used during this stage, including data flow diagrams, to add more clarity to the project goals.

Verification and validation. Verification ensures the software is built to the customer's requirements. In contrast, validation ensures the software is implementing the right functions. If the requirements don't go through the validation stage, there is the potential for time-consuming and expensive reworks.

Requirements management. RM is an ongoing process that runs in parallel to the other three processes just described. In RM, you're matching all the relevant processes to their requirements. You will analyze, document, and prioritize the requirements – and communicate with relevant stakeholders. Any requirements that need modification are handled in an efficient and systematic manner.

As we implement the different components of RE, it also helps to get a sense of what should be excluded from requirements. Understanding this will help us focus more accurately on developing requirements and better meeting client expectations.

The requirements engineering process



5.a Explain Generalization and Inheritance with example.

Ans. **Generalization and Inheritance** are concepts in software engineering related to object-oriented programming (OOP) that allow for the creation of a hierarchical structure in code, promoting code reuse and modularity. Here's an explanation of each concept with an example:

Generalization:

Generalization is the process of extracting common properties or behaviors from a set of entities and forming a more general entity that encompasses these commonalities. It's a way of abstracting common features from multiple classes to create a more generalized class. In generalization, a superclass (or parent class) is created to represent common characteristics, and subclasses (or child classes) inherit from the superclass, inheriting its properties and behaviors.

Inheritance:

Inheritance is a mechanism in OOP where a new class (subclass or derived class) inherits properties and behaviors from an existing class (superclass or base class). The subclass can reuse the code of the superclass and extend or override its functionality as needed. Inheritance promotes code reuse and establishes an "is-a" relationship between classes.

In the previous example, **Circle**, **Square**, and **Triangle** inherit from the generalized **Shape** class, signifying an "is-a" relationship: a circle is a shape, a square is a shape, and a triangle is a shape. The subclasses can reuse the common properties and behaviors defined in the **Shape** class while providing their specific implementations for calculating area.

5.b. Discuss about navigation of class models with suitable diagrams and examples.

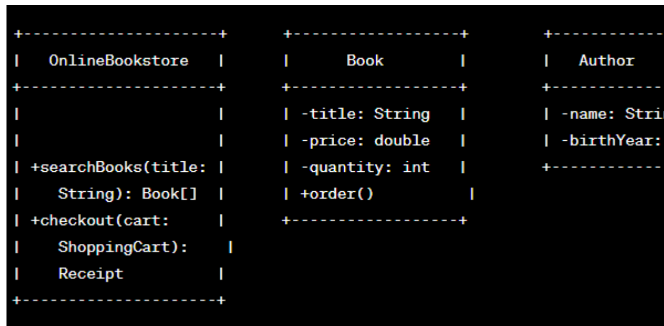
Ans. Navigating class models involves understanding how objects interact and collaborate within a software system. Class diagrams in object-oriented modeling provide a visual representation of the classes, their attributes, methods, and relationships. Let's discuss class model navigation with suitable diagrams and examples.

Class Diagram Overview:

A class diagram consists of classes, attributes, methods, and relationships. It visually represents the structure of a system, showcasing the static aspects of the software.

Example Class Diagram:

Consider a simplified example of an online bookstore:



In this diagram:

- **OnlineBookstore** has associations with **Book** instances and performs operations like searching books and processing orders.
- **Book** has attributes such as **title**, **price**, and a reference to the **Author** class. It also has a method **order()**.
- **Author** has attributes like **name** and **birthYear**.

Navigation Examples:

1. **Searching for Books:**
   - Navigation from **OnlineBookstore** to **Book** instances:

javaCopy code

OnlineBookstore bookstore = new OnlineBookstore(); Book[] searchResults = bookstore.searchBooks("Java Programming");

2. **Ordering a Book:**
   - Navigation from **Book** to a method call (**order()**):

javaCopy code

Book selectedBook = // retrieve a specific book; selectedBook.order();

3. **Accessing Author Information:**
   - Navigation from **Book** to **Author** instance:

javaCopy code

Book someBook = // retrieve a specific book; Author author = someBook.getAuthor();

4. **Checkout Process:**
   - Navigation between different classes during the checkout process:

javaCopy code

OnlineBookstore bookstore = new OnlineBookstore(); ShoppingCart cart = // create a shopping cart; Receipt receipt = bookstore.checkout(cart);

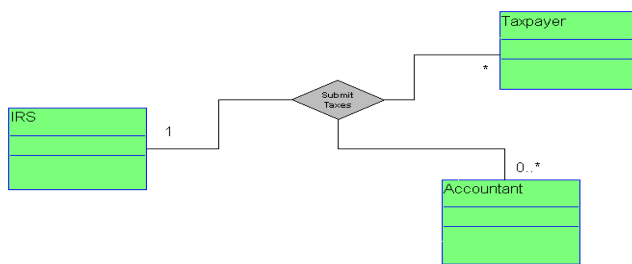
Navigation in Diagram:

The arrows in the diagram represent associations between classes, indicating the direction of the relationship. For example:

- The association between **OnlineBookstore** and **Book** indicates that an online bookstore is associated with multiple books.
- The composition relationship between **ShoppingCart** and **Receipt** suggests that a **ShoppingCart** is part of the checkout process, and a **Receipt** is created during checkout.

6a. What is N array association? Illustrate the aggregation with associations and compositions with suitable examples.

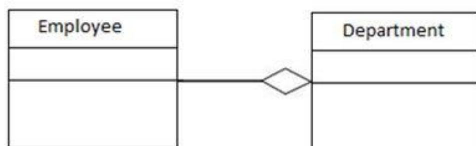
An N-ary association is a relationship that exists between three or more classes. The association cannot be broken down into a regular, binary association without some information being lost. The great majority of associations are binary; very few are ternary (N = 3). Rarer still are associations of order four or more (N = 4 or more). A common example of a ternary association is the one that exists between a taxpayer, an accountant, and the IRS.



Association, Aggregation, and Composition are terms that represent relationships among objects.

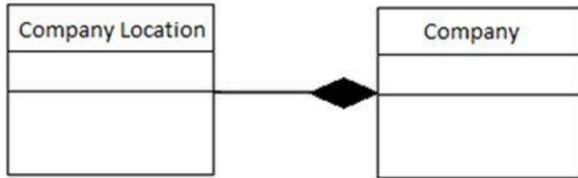
Association
Association is a relationship among the objects. Association is "*a*" relationship among objects. In Association, the relationship among the objects determines what an object instance can cause another to perform an action on its behalf. We can also say that an association defines the multiplicity among the objects. We can define a one-to-one, one-to-many, many-to-one and many-to-many relationship among objects.
Aggregation

Aggregation is a special type of Association. Aggregation is "*the*" relationship among objects. We can say it is a direct association among the objects. In Aggregation, the direction specifies which object contains the other object. There are mutual dependencies among objects. For example, departments and employees, a department has many employees but a single employee is not associated with multiple departments.



Composition

Composition is special type of Aggregation. It is a strong type of Aggregation. In this type of Aggregation the child object does not have their own life cycle. The child object's life depends on the parent's life cycle. Only the parent object has an independent life cycle. If we delete the parent object then the child object(s) will also be deleted. We can define the Composition as a "Part of" relationship.

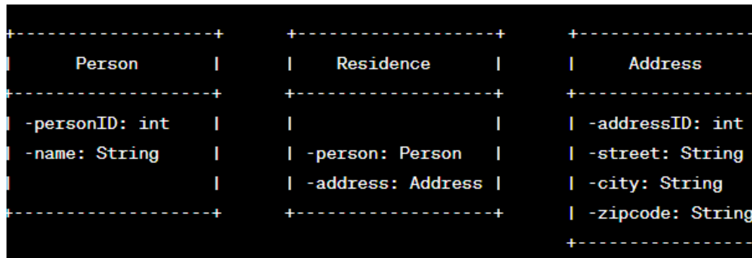


6.b. Explain the concept of reification and constraints with neat diagram and examples.

Reification is the process of representing abstract concepts or relationships as concrete entities in a model. In the context of modeling and software engineering, reification involves turning abstract relationships into tangible objects or classes, making them first-class citizens that can be manipulated and reasoned about like any other entities.

Example:

Consider a simple example where you have an abstract relationship between a **Person** and an **Address** called "Residence." Instead of treating this relationship purely abstractly, you reify it into a distinct class:



In this example, the abstract relationship "Residence" is reified into the **Residence** class, which connects a **Person** to an **Address**. Now, you can associate additional information with a person's residence, such as the type of residence, move-in date, etc.

Constraints are rules or restrictions that define the valid states, behaviors, or relationships within a system. In software modeling, constraints ensure that certain conditions are met or maintained, helping to enforce correctness and integrity.

Example:

Consider a constraint related to the **Person** class, where you want to ensure that a person's age is always greater than or equal to 18:

```
+------------------+
|      Person      |
+------------------+
| -personID: int   |
| -name: String    |
| -age: int        |
+------------------+
| +isAdult(): bool |
+------------------+
```

In this example, the **isAdult()** method represents a constraint. It checks whether the person's age is 18 or older. This constraint ensures that certain conditions are satisfied when dealing with a person's age.

7.a. Explain System models with suitable example.

Ans. System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification.

You may develop different models to represent the system from different perspectives. For example:

1. An external perspective, where you model the context or environment of the system.
2. An interaction perspective where you model the interactions between a system and its environment or between the components of a system.
3. A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

There are three ways in which graphical models are commonly used:

1. As a means of facilitating discussion about an existing or proposed system.
2. As a way of documenting an existing system.
3. As a detailed system description that can be used to generate a system implementation.

There are four important types of system models, namely,

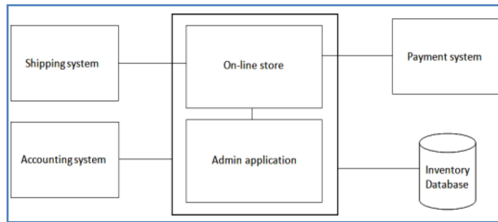
- Context models,
- Interaction models,
- Structural and
- Behavioral models

Context Model

At an early stage in the specification of a system, you should decide on the system boundaries. This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system's environment.

Context models are often depicted as box and line diagrams since such simple diagrams are enough to put the system into context with other systems. The context model for our case study is shown in the following figure.

Figure for Context model of the online store.



Context models normally show that the system's environment contains other systems but the types of relationships between the systems of the environment and the system that is being specified are not shown.

Interactive Modeling

All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs, interaction between the system being developed and other systems or interaction between the components of the system.

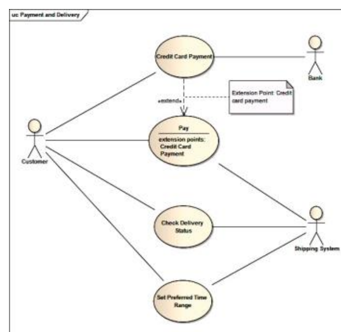
There are two related (complementary) approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external actors (users or other systems).
2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

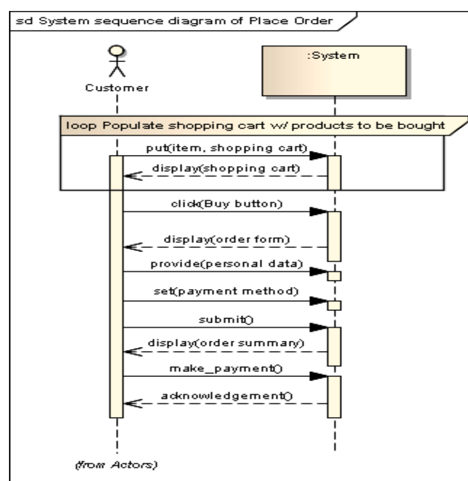
Use case models and sequence diagrams present interaction at different levels of detail and so may be used together.

Actors may represent roles played by human users, external systems or hardware, or other subjects.

Use cases describing interactions to Bank and Shipping system (external systems).



Use cases describing interactions to Accounting system (external systems).



Structural models

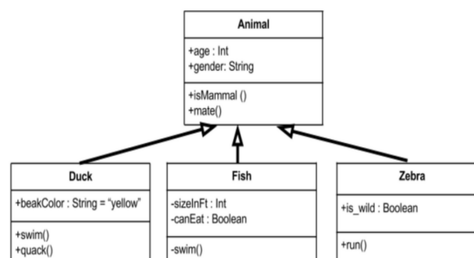
- Identification of domain classes
- Refining the model

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design or dynamic models, which show the organization of the system when it is executing.

We abstract like things and call them ***classes***.

A class diagram depicts the structure of a class by denoting:

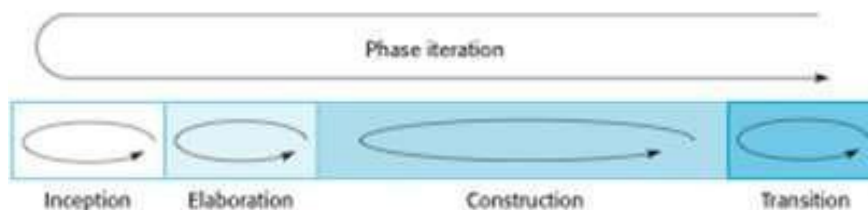
- classes,
- their attributes,
- operations (or methods),



- and the relationships among objects.

7. b. With a neat diagram, explain the working procedure of RUP with its advantages.

Rational Unified Process (RUP) is a software development process for object-oriented models. It is also known as the Unified Process Model. Some characteristics of RUP include use-case driven, Iterative (repetition of the process), and Incremental (increase in value) by nature



The RUP recognizes that conventional process models present a single view of the process.

In contrast, the RUP is normally described from three perspectives:

1. A dynamic perspective, which shows the phases of the model over time.
2. A static perspective, which shows the process activities that are enacted.
3. A practice perspective, which suggests good practices to be used during the process.

Fig shows the phases in the RUP. These are:

1. **Inception:**

   Goal: To establish a business case for the system.It is necessary to identify all external entities (people and systems) that will interact with the system and define these interactions.This information can then be used to assess the contribution that the system makes to the business.

2. **Elaboration:**

   Goal: To develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks.

3. **Construction:**

Involves system design, programming, and testing.

Parts of the system are developed in parallel and integrated during this phase.

On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.

4. **Transition:**

It is concerned with moving the system from development community to the user community and making it work in a real environment.On completion of this phase, you should have a documented software system that is working correctly in its operational environment.

Advantages:
It provides good documentation, it completes the process in itself.
It provides risk-management support.
It reuses the components, and hence total time duration is less.
Good online support is available in the form of tutorials and training.

8. a. Define Design pattern. Explain the essential elements of design pattern.

Ans. The design pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings.

→ The pattern is not a detailed specification.

→ Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.

→ Design patterns are usually associated with object-oriented design.

→ The general principle of encapsulating experience in a pattern is one that is equally applicable to any kind of software design

→ The four essential elements of design patterns were defined by the 'Gang of Four' in their patterns book:

- A name that is a meaningful reference to the pattern.
- A description of the problem area that explains when the pattern may be applied.
- A solution description of the parts of the design solution, their relationships, and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
- A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

8.b. Explain in detail about the implementation issues involved in software engineering.

- Ans. There are some challenges faced by the development team while implementing the software. Some of them are mentioned below:

Programming Language Selection:

Choosing the appropriate programming language is a crucial decision during implementation. Factors such as project requirements, team expertise, performance considerations, and platform compatibility should be taken into account.

Coding Standards and Conventions:

Establishing and adhering to coding standards is essential for maintaining code quality and consistency. Consistent coding styles enhance readability, maintainability, and collaboration among team members.

Code Modularity and Reusability:

Breaking down the software into modular components promotes reusability and maintainability. Well-designed and modular code allows developers to reuse components in different parts of the system or in future projects.

Error Handling and Exception Management:

Implementation must include robust error-handling mechanisms to deal with unexpected situations. Proper handling of errors and exceptions ensures that the software behaves predictably and is more resilient to failures.

Concurrency Control:

For systems that involve concurrent processes or multi-threading, managing concurrency becomes a critical issue. Proper synchronization mechanisms and techniques are required to avoid race conditions and ensure data consistency.

Memory Management:

Efficient memory allocation and deallocation are essential for preventing memory leaks and optimizing the performance of the software. Improper memory management can lead to resource exhaustion and system instability.

Performance Optimization:

Identifying and optimizing performance bottlenecks is crucial for achieving the desired level of efficiency. This may involve profiling the code, optimizing algorithms, and using appropriate data structures.

Testing and Debugging:

Rigorous testing and debugging are integral parts of the implementation process. Testing should cover unit testing, integration testing, system testing, and acceptance testing to ensure the correctness and reliability of the software.

Documentation:

Comprehensive documentation of the code, including comments, helps in understanding the codebase and facilitates maintenance. Documentation is essential for future development, bug fixing, and collaboration among team members.

Version Control and Configuration Management:

Implementing effective version control and configuration management practices ensures that changes to the codebase are tracked, controlled, and documented. This is crucial for collaboration in team-based development.

9. a. Discuss Test Driven Development (TDD) with its process and list out its benefits.

Ans. Test-driven development (TDD) is an approach to program development in which testing and code development are interleaved.

→ Test-driven development was introduced as part of agile methods such as Extreme Programming.

→ The fundamental TDD process is shown in fig 3.9. The steps in the process are asfollows:

1. Start by identifying the increment of functionality that is required. This shouldnormally be small and implementable in a few lines of code.

2. Write a test for this functionality and implement this as an automated test. Thismeans that the test can be executed and will report whether or not it has passedor failed.
3. Run the test, along with all other tests that have been implemented. Initially, the functionality is not implemented, so the new test will fail. This is deliberate as it shows that the test adds something to the test set.
4. You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what's already there.
5. Once all tests run successfully, then move on to implementing the next chunk of functionality.

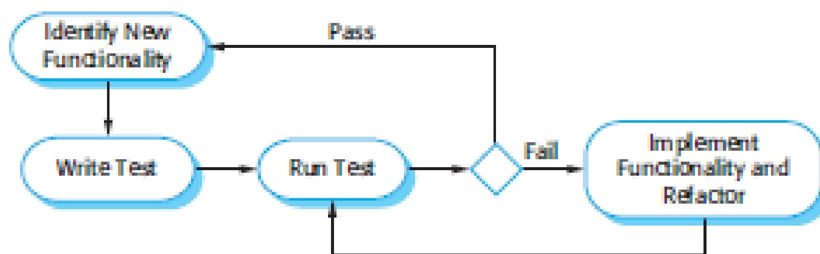


Fig 3.9: Test-driven development

→ Benefits of test-driven development are:

1. **Code coverage:** In principle, every code segment that you write should have at least one associated test. Therefore, you can be confident that all of the codein the system has actually been executed. Code is tested as it is written so defects are discovered early in the development process.
2. **Regression testing**: A test suite is developed incrementally as a program is developed. Regression tests can be run to check that changes to the program have not introduced new bugs.
3. **Simplified debugging:** When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified. Debugging tools need not be used to locate the problem. Reports of the use of test-driven development suggest that it is hardly ever necessary to use an automated debugger in test-driven development
4. **System documentation**: The tests themselves act as a form of documentation that describe what the code should be doing. Reading the tests can make it easier to understand the code.

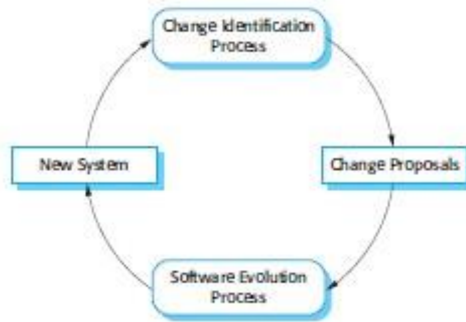
9.b. Explain software evolution process with a neat diagram.

Ans. Software evolution processes vary depending on the type of software being maintained, the development processes used in an organization and the skills of the people involved.

→ In some organizations, evolution may be an informal process where change requests mostly come from conversations between the system users and developers.

→ In other companies, it is a formalized process with structured documentation producedat each stage in the process.

→ The processes of change identification and system evolution are cyclic and continue throughout the lifetime of a system



Change identification and evolution process

→ Fig. shows an overview of the evolution process.

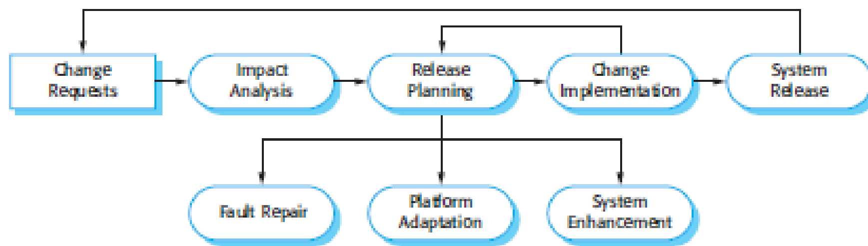


Fig 3.15: The software evolution process

→ The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers.

→ The cost and impact of these changes are assessed to see how much of the system isaffected by the change and how much it might cost to implement the change.

→ If the proposed changes are accepted, a new release of the system is planned.

→ During release planning, all proposed changes (fault repair, adaptation, and newfunctionality) are considered.

→ A decision is then made on which changes to implement in the next version of thesystem.

→ The changes are implemented and validated, and a new version of the system isreleased.

→ The process then iterates with a new set of changes proposed for the next release.

→ Change implementation can be thought of as an iteration of the development process, where the revisions to the system are designed, implemented, and tested.

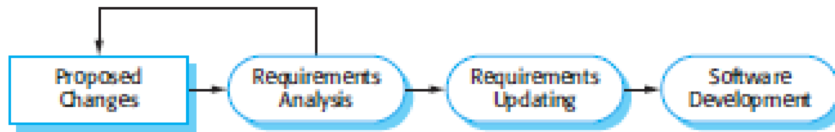
→ However, a critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for change implementation.

→ During this program understanding phase, it becomes necessary to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program.

→ This understanding is required to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

→ The change implementation stage of this process should modify the system specification, design, and implementation to reflect the changes to the system (Fig 3.16).

→ During the evolution process, the requirements are analyzed in detail and implicationsof the changes emerge that were not apparent in the earlier change analysis process.



10.a. Describe the three main types of software maintenance. List some of the difficulties and distinguish between them.

Ans. Software maintenance is the general process of changing a system after it has been delivered.

→ There are three different types of software maintenance:

1. **Fault repairs:** Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve rewriting several program components Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.
2. **Environmental adaptation:** This type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating system, or other support software changes. The application system must be modified to adapt it to cope with these environmental changes.
3. **Functionality addition**: This type of maintenance is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater thanfor the other types of maintenance.

10.b.Explain why problems with support software might mean an organization has to replace legacy systems.

Ans. Support software plays a crucial role in the functioning of any software system, particularly in the context of legacy systems. Legacy systems are older software or hardware systems that remain in use because they continue to fulfill the core functions for which they were originally designed. However, as technology evolves, organizations often face challenges with support software that may necessitate the replacement of legacy systems. Here are several reasons why problems with support software can lead to the replacement of legacy systems in software engineering:

1. **Outdated Technology:**
   - Legacy systems may be built on outdated technologies that are no longer actively supported or maintained by vendors. If the support software used by the legacy system becomes obsolete, it can lead to compatibility issues, security vulnerabilities, and a lack of access to essential updates and patches.
2. **Security Concerns:**
   - As technology advances, new security threats emerge, and outdated support software may not provide adequate protection against modern cyber threats. This can expose the organization to security vulnerabilities, making the legacy system a potential target for malicious activities.
3. **Incompatibility with Modern Software:**
   - Legacy systems may face challenges when integrating with newer software applications, databases, or third-party services. If the support software is not

compatible with the evolving technological landscape, it can hinder the organization's ability to adopt new and efficient solutions.

4. **Lack of Vendor Support:**
   - When the vendor support for essential software components used in a legacy system is discontinued, organizations may struggle to find assistance in case of issues or bugs. The absence of vendor support can lead to prolonged downtime, increased maintenance costs, and the inability to address evolving business needs.
5. **Performance Issues:**
   - Outdated support software may not be optimized for modern hardware or may lack performance enhancements found in newer technologies. This can result in suboptimal system performance, impacting the organization's efficiency and competitiveness.
6. **Compliance and Regulatory Requirements:**
   - Changes in industry regulations and compliance standards may necessitate updates to support software to ensure that the legacy system meets the latest legal and regulatory requirements. Failure to comply with industry standards can lead to legal consequences and reputational damage.
7. **Limited Scalability:**
   - Legacy systems may struggle to scale to accommodate growing business demands. If the support software cannot effectively handle increased workloads or additional users, it can impede organizational growth and hinder the ability to adapt to changing business requirements.
8. **High Maintenance Costs:**
   - Maintaining and supporting legacy systems can become increasingly expensive as the availability of skilled personnel diminishes, and the costs associated with maintaining outdated technologies rise. Transitioning to newer systems with more cost-effective support options may be a prudent financial decision.
9. **User Experience Issues:**
   - Users may face difficulties in interacting with legacy systems due to outdated user interfaces or limited usability features. Improvements in user experience, driven by advancements in support software, can enhance productivity and user satisfaction.
10. **Strategic Alignment:**
    - Organizations may need to align their IT systems with strategic business goals. If legacy systems and their support software do not support the organization's current or future objectives, replacement becomes a strategic necessity for staying competitive in the market.